Class Libraries User's Guide

Microsoft® C/C++

Microsoft® C/C++

Version 7.0

Class Libraries User's Guide

For MS-DOS® and Windows™ Operating Systems

Microsoft Corporation

Document No. LN28113-1191

10 9 8 7 6 5 4 3 2 1

Contents Overview

Contents

Figures and Tables

Figures

Tables

Introduction

This *Class Libraries User's Guide* introduction discusses C++ class libraries in general terms and then summarizes the the two class libraries that are included with Microsoft® C/C++ Version 7.0. These libraries include:

- The Microsoft Foundation Class Library

  This library contains a full-featured set of C++ classes for Microsoft Windows™. It includes not only Windows classes but also general-purpose classes for collections, files, persistent storage, exceptions, diagnostics, memory management, strings, and time.

- The Microsoft iostream Class Library

  The iostream classes can be used for most input/output, but they are particularly useful for text-mode output. These classes are popular because they have a programming interface that is compatible with C++.

Following the class library summary is a section that tells you how to use the library documention.

To start using this guide, you will need a basic knowledge of the C++ programming language. To use the Windows Foundation classes, you should be familiar with the C-language application programming interface to Microsoft Windows.

The Promise of C++: Reusable Class Libraries

C++ is a powerful language in its own right, but its real value lies in its ability to be extended with class libraries. These already-written libraries of C++ classes appear as though they were part of the language.

As an example, consider the familiar C and C++ data type **int**:

```
int i = 5;
i += 3;  // i = 8
```

Now suppose you had a class library that included a "string" data type called **CString**. You could write expressions such as:

```
CString s = "very";
s += " easy"; // s = "very easy"
```

Notice how the **CString** data type, called a "class," appears to be part of the language.

As another example, consider the **cout** object introduced in the *C++ Tutorial*. An expression such as:

```
cout << "i = " << i << "\n";
```

depends on the Microsoft iostream Class Library because the << operator, normally used for shifting left, has been overloaded to insert values into the output stream **cout**.

C++ Class Libraries vs. C Function Libraries

C function libraries have been available for years. Some, such as the standard runtime libraries, are bundled with the compiler; others, such as database and user interface libraries, are sold by independent software vendors.

This section compares C function libraries with C++ class libraries. The advantages of the C++ language, such as inheritance and polymorphism, are discussed in the *C++ Tutorial*.

Advantages of C++ Class Libraries

Class libraries have several advantages when compared with function libraries:

- Classes encapsulate code and data.

  Ordinary C libraries consist of one or more discrete program modules, which manage data in nonsystematic ways. It is generally difficult to isolate and control a particular library's data. By contrast, in C++ the data is made an integral part of the class through "encapsulation." The class designer can control access to the data through the class's member functions.

- New classes appear to be language extensions.

  You can create objects of a library class the same way you create instances of the C++ built-in types. Thus classes, with their special constructors and overloaded operators, provide a natural programming interface.

- Inheritance eliminates "code cloning."

  If you need a new C function that is "similar to but different from" a library C function, you must copy the original (assuming you have the source) and then change its name. A C++ class allows you to add functionality through derivation without disturbing the original code. Indeed, class derivation does not require you to have the base class source code.

- Variable and function name collisions are minimized.

  If two classes have identical data member names or member function names, there is no conflict. Classes must not have the same name, however, unless they are nested.

- Tools such as class browsers enhance source code control.

  C++ adds a level of structure not possible with C function libraries. The Source Browser in the Microsoft Programmer's WorkBench is a tool that allows you to view your source code in the order of class hierarchy.

How to Use a Class Library

There are several ways to use a C++ class library:

- Construct objects directly from the classes provided
- Derive new classes
- Modify the class source code

Direct Use of Classes and Objects

Many class libraries provide classes that support the direct construction of useful objects. Some even construct the objects for you prior to the execution of your **main** program.

The iostream Class Library includes the predefined objects **cout** and **cin**. Many developers use those objects directly, and others construct their own objects from the classes such as **ifstream** and **ostrstream**.

In the Microsoft Foundation Class Library, some classes, such as those dealing with strings, time, and some low-level Windows functions, are most often used directly for the construction of objects.

Derivation of New Classes

Certain classes are designed for the purpose of derivation. **CObject**, the root class for most Microsoft Foundation Classes, is an example of a class that is not meant to be used directly. Likewise, the **CWnd** window class (for Microsoft Windows) is generally used as a base class for customized windows. A **CWnd** object can display itself and handle basic messages, but it doesn't show or accept data. A derived class can provide the mouse and keyboard notification message functions that make the window part of an application.

Other classes can be used directly or they can be derived from in order to add new functionality. In the Microsoft Foundation Classes tutorial, you will see how a special-purpose `CPersonList` class can be derived from the generic Microsoft Foundation Library **CObList** class. The derived class adds new data members and member functions.

Modification of Class Source Code

Sometimes you can't achieve all your customization objectives by class derivation. You may, for example, need access to a private data member. If you have the library source code, you can modify the class directly. Even if you don't modify the class, the source code is a useful learning and debugging resource.

The source code for the Microsoft Foundation Class Library is provided. The source code for the Microsoft iostream Class Library is available separately.

The Microsoft Foundation Class Library

The Microsoft Foundation Class Library is a C++ class library primarily designed for developing applications for the Microsoft Windows (version 3.x) graphical user interface. The Foundation classes, together with all necessary libraries, are included with Microsoft C/C++ Version 7.0.

In addition to special-purpose Windows classes, the Microsoft Foundation Class Library contains general-purpose classes that are useful both inside and outside of the Windows environment. These general-purpose classes use functions from standard run-time libraries, but they do not depend on Windows functions.

Windows Classes

Both Windows and C++ are *object oriented*, and it is natural to use an object-oriented language to interface with an object-oriented graphical user interface. The Windows classes in the Microsoft Foundation Class Library provide the link. These classes offer the following features:

- Close coupling to the C-language Windows library

  The Microsoft Foundation Windows classes are really a direct C++ wrapping of the familiar C functions for Windows. This feature provides maximum speed and storage efficiency, and it offers total programming flexibility. The resulting programs are as fast and small as C programs, and they can incorporate C-language function calls (including Windows calls) anywhere.

- Significant reduction of programming "surface area" from C

  C++ source programs for Windows are smaller and easier to understand because many complex functions are encapsulated in the classes. You derive your own application classes from the library's base classes. These derived classes give you access to all the base class functionality without code duplication.

- Wide array of useful window classes

  The library contains ready-made, derivable classes for ordinary frame windows, Multiple Document Interface (MDI) frame and child windows, edit controls, list boxes, combo boxes, buttons, and so forth.

- Efficient processing of Windows messages

  The Microsoft Foundation Windows classes replace error-prone **case** statements with C++ member functions. You specify, by means of a special syntax called a "message map," which Windows "notification" messages you expect; then you write the necessary member functions. No space-consuming virtual functions are necessary. C++ member functions are thus effectively reconciled with Windows queued messages.

- Ability to derive "midlevel" window classes

  If your application needs a feature, such as scrolling, repeated in many different windows, you can write an abstract window class that contains this functionality by deriving it from one of the Microsoft Foundation Classes. Then you can further derive special-purpose classes that share the desired feature.

- Useful utility classes

  There are useful classes for common Windows objects such as display contexts, pens, brushes, menus, points, and rectangles. These classes permit you to maximize your use of the C++ language.

General-Purpose Classes

The general-purpose classes are useful both with and without Windows. All classes are grouped in the same library because most of them share a common base class, **CObject**. The general-purpose classes offer support for the following:

- Collections

  The library provides efficient collection classes for ordered lists, indexed arrays, and keyed maps (dictionaries). Sixteen collection variations accommodate strings, void pointers, object pointers, bytes, words, and double words. A template expansion tool, provided in the sample code, permits creation of customized collection classes.

- Strings

  The **CString** class adds dynamically allocated strings to C++. These strings can be manipulated with a Basic-like syntax that includes concatenation operators and functions such as **Mid**, **Left**, and **Right**. They can be printed to diagnostic output and written to (and read from) disk.

- Time and date

  A time class, together with a companion time-difference class, offers date-time arithmetic and automatic formatting of binary time values into human-readable dates and times.

- Files

  File classes offer a C++ interface to both the low-level and "stdio" input/output files. In-memory files are also supported. The file class hierarchy allows all three file types to be accessed polymorphically through the **CFile** base class.

- Exception processing

  Errors can be systematically trapped using a syntax that models the proposed ANSI C++ exception processing standard. This feature eliminates the need for error check logic after every function call.

- Persistent objects

  The object archiving feature allows objects of specified **CObject**-derived classes to be stored to and loaded from a "persistent" storage medium such as a disk. If a collection is archived, then all the members of the collection will, in turn, be archived.

- Debugging support and diagnostics

  Individual objects of selected **CObject**-derived classes may be printed in human-readable form. Memory allocation statistics are available, and it is possible to print the contents of a range of memory. Such a memory dump will display not only object information but also the line number and source module name where each memory block was allocated. Special "guard bytes", inserted before and after allocated memory, allow corruption to be detected. All these diagnostic features are disabled in the Release versions of the library.

The Microsoft iostream Class Library

The iostream Class Library provided with the Microsoft C/C++ Compiler is based on the AT&T C++ version 2.1 specification and thus conforms to the descriptions in the more recent C++ textbooks. This library offers a complete input and output capability for binary and text data and can operate in buffered or unbuffered mode.

The iostream classes are most useful for formatted text output. You are probably familiar with the **cout** predefined output stream used mostly for debugging output. The iostream classes are compatible with the Microsoft Foundation Classes.

The Microsoft iostream Class Library documentation emphasizes the following points:

- Using the formatted stream output features
- Using stream member functions for file manipulation
- Using the input stream extractors
- Overloading << and >> operators for your own classes
- Writing custom "manipulators" for special formatting
- Deriving from the **streambuf** class for custom processing

How to Use the Class Library Documentation

This *Class Library User's Guide* is divided into three parts:

Part 1	The Microsoft Foundation Class Library Tutorial
Part 2	The Microsoft Foundation Class Library Cookbook
Part 3	The Microsoft iostream Class Library Tutorial

The *Class Library Reference* is divided into two parts:

Part 1	Overview of the Microsoft Foundation Class Library
Part 2	The Microsoft Foundation Class Reference
Part 3	The Microsoft iostream Class Reference

If you want to learn about the Microsoft Foundation classes, read the overview chapters in the *Class Library Reference*, Part 1. Then, after you have installed the software, work through the examples in this *Class Libraries User's Guide* tutorial. After you are familiar with the Foundation class basics, you can study the *Class Libraries User's Guide* cookbook.

The alphabetical class reference and global function reference (*Microsoft Class Libraries Reference*, Part 2) are useful during the software development process. Remember that the reference material is also available in Help.

For input/output streams, read the Microsoft iostream Class Library tutorial in Part 3 of the *Class Libraries User's Guide*, then refer to Part 3 of the *Class Libraries Reference.*

Document Conventions

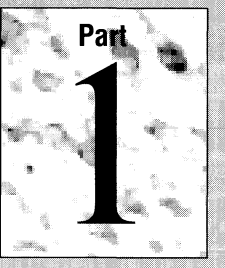
This book uses the following typographic conventions:

Example	Description
STDIO.H	Uppercase letters indicate filenames, segment names, 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 functions and constants begin with either a single or double underscore. These are part of the name and are mandatory. For example, to have the **__cplusplus** manifest constant be recognized by the compiler, you must enter the leading double underscore.
expression	Words in italics indicate placeholders for information you must supply, such as a filename.
[[*option*]]	Items inside double square brackets are optional.
#pragma pack {**1** | **2**}	Braces and a vertical bar indicate a choice among two or more items. You must choose one of these items unless double square brackets ([[ ]]) surround the braces.
`#include <io.h>`	This font is used for examples, user input, program output, and error messages in text. It is also used for names of user-derived classes and members.
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.

Example	Description
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 ' '.
Color Graphics Adapter (CGA)	The first time an acronym is used, it is usually spelled out.

The Microsoft Foundation Class Library Tutorial



The Microsoft Foundation Class Library Tutorial

Part 1 of the Microsoft Foundation Class Libraries User's Guide contains a tutorial designed to teach you the elements of using the Microsoft Foundation Class Library. Basic techniques for using a class library are explored through a tutorial on the fundamental, non-Windows classes. This tutorial features collection classes, exception handling, and diagnostic facilities. The rest of Part 1 teaches the elements of Windows programming with the Windows classes.

Chapter 1 explains how to prepare for the tutorial and how to build the example programs in the tutorial with NMAKE or Programmer's WorkBench.

Chapter 2 shows how to build a simple DOS phone number database application with collection classes, classes for time and strings, exception handling, and diagnostics.

Chapter 3 shows how to write a simple Windows "Hello, World!" program with the Windows classes.

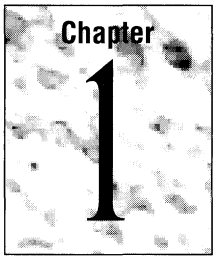
Chapters 4 through 6 explain more about Windows programming through a more capable Windows version of the phone number database presented in Chapter 2.

Chapter 4 shows how to wrap the phone number data model from Chapter 2 in your own C++ class. Class `CDataBase` provides a clean interface to the data and simplifies the final Windows program.

Chapter 5 explains how to use dialog objects to put Windows dialog boxes on the screen and interact with the user.

Chapter 6 shows how to write handler functions for the phone book application's menu commands and how to handle mouse and keyboard input.

Using the Microsoft Foundation Class Library Tutorial



The first six chapters of this book (Part 1) make up the Microsoft Foundation Class Library tutorial. Use these chapters to get a quick introduction to the Microsoft Foundation Classes or to take a step-by-step tour through some of their fundamentals.

1.1 What's in the Tutorial

The tutorial uses several example programs to introduce you to programming with the Microsoft Foundation Class Library. For additional information about how to accomplish particular programming tasks, see the cookbook (Part 2) in Chapters 7 through 17. As you work through the tutorial and explore the cookbook, you can look up the classes, functions, and other components of the Microsoft Foundation Class Library in the *Class Libraries Reference*. This information is also available in Help.

Topics

This tutorial shows you how to:

- Design with C++ and objects.
- Use collection classes.
- Design and implement persistent objects.
- Use the diagnostic facilities of the class library.
- Use objects in character-based programs.
- Use objects to program Microsoft Windows.

If you want to study a more specific topic, use the cookbook chapters. If you want to learn to build complete object-oriented programs—both character-based and Windows—follow the tutorial.

Programs

The tutorial presents four sample programs to demonstrate many of the classes and facilities of the Microsoft Foundation Class Library:

- DMTEST, Chapter 2

  This simple, noninteractive DOS program demonstrates the use of collection classes, object serialization, diagnostics, and other features of the Microsoft Foundation Class Library.

- HELLO, Chapter 3

  This simple Windows program demonstrates the fundamentals of writing a Windows program with the Microsoft Foundation Classes.

- PHBOOK, Chapters 4 through 6

  This more ambitious Windows program demonstrates more complicated dialog boxes, standard Windows version 3.1 open, save, and print dialogs, Windows menus and menu-handler functions, basic uses of the keyboard and the mouse, and more.

- CMDBOOK, Chapter 4

  This program parallels PHBOOK but without Windows. It presents a character-based command interface to the data model developed in Chapter 1 and used in PHBOOK. Because of its great similarity to PHBOOK, CMDBOOK is not presented in detail but is provided if you wish to pursue character-based programming with the Microsoft Foundation Class Library and C/C++.

1.2 How to Use the Tutorial

There are two ways to use this tutorial. If you learn best by typing the code yourself, you can follow along step by step through the example programs. On the other hand, if you prefer, you can read the discussion sections in each chapter and build the examples from the code files provided on the distribution disks.

What You Need to Know

To make effective use of this tutorial, you should have some experience programming in C and some familiarity with Microsoft Windows programming. Because the Microsoft Foundation Class Library uses C++, the more you know about the C++ language the better.

You can improve your understanding of Windows programming with the Microsoft *Windows SDK* documentation and with books like *Programming Windows, Version 3*, second edition, by Charles Petzold, and *Peter Norton's Windows 3.0 Power Programming Techniques*, by Peter Norton and Paul Yao.

You can improve your understanding of C++ programming with the C++ tutorial included in your Microsoft C/C++ package or with *C++ Primer*, second edition, by Stanley B. Lippman.

Chapters 3 through 6 cover Windows programming with the Microsoft Foundation Class Library. Chapter 2 and part of Chapter 4 cover non-Windows programming.

Work Along with the Tutorial

To work along with the tutorial, follow the steps presented in the next five chapters. Each chapter instructs you in putting together the necessary code files. You'll add class declarations, member functions, and other pieces of code step by step. Each chapter also includes complete listings of the code, which you can use to check your work.

If you choose to follow the tutorial, make your own working directory for each example. This keeps the files you create separate from files of the same names in the example directories. Copy the appropriate makefile to your new directory (see "Makefile Locations" on page 8).

At frequent intervals, you'll find discussion sections that sum up the code you've just added to your files and explain what that code is doing. If you wish, you can skip past any of the discussion sections to continue creating your example code files, but you'll probably find the discussion sections worth pausing for.

Once you complete your files, you'll find instructions for building the program. Later sections of this chapter will show you how to build your program using the Programmer's WorkBench (PWB) or using the NMAKE utility.

Get Right to Your Own Code

If you prefer to skip the step-by-step tutorial, you should read the rest of this chapter, then skim the remaining five chapters, focusing on the discussion sections. In each chapter, you'll probably want to read the introduction and the discussion sections and look at the complete code listings.

On your Microsoft C/C++ distribution disks you'll find code files for all the tutorial examples, including makefiles for use with NMAKE and project files for use with PWB. You can build the examples, modify them to try new techniques and other classes, and then move on to your own programs.

1.3 How to Build Microsoft Foundation Programs

This section explains how to build the example programs using either PWB or NMAKE.

Necessary Setup

If you chose to install the Microsoft Foundation Class Library when you installed Microsoft C/C++, you have all of the files and directories you need. Your PATH, INCLUDE, and LIB environment variables should be set up to compile programs that use the Microsoft Foundation Classes. The paths given below are relative to where you installed Microsoft C/C++. If you installed Microsoft C/C++ into the C700 directory, for example, the INCLUDE path given below is C700\MFC\INCLUDE.

Your INCLUDE variable should include normal C 7.0 includes and the path to the MFC\INCLUDE directory, which contains Microsoft Foundation Class Library include files.

Your LIB variable should include normal C run-time libraries and the path to the MFC\LIB directory, which contains Microsoft Foundation Class Library run-time library files.

Your INCLUDE and LIB paths do not need to be in any particular order. If you write your own makefile for NMAKE, be sure to list the "afx" library appropriate to your chosen memory model first in the list of libraries you link with.

Makefiles and Build Directories

This section describes where to find the makefiles for the tutorial example programs and explains how those makefiles are set up.

Makefile Locations

The makefiles for the tutorial examples are in the TUTORIAL directory except those for HELLO.EXE, which is in the HELLO directory.

Note All of the tutorial programs except HELLO are placed in one directory, MFC\SAMPLES\TUTORIAL. Because of this, you can't simply use the default makefile as you can with HELLO. For these programs, you must give NMAKE a specific makefile name.

The makefiles for building with PWB have the same base name as the example program file, plus the .MAK extension. The makefiles for building with NMAKE have the same base name as the example program file but no extension.

Makefiles for the DMTEST, CMDBOOK, and PHBOOK examples are in the MFC\SAMPLES\TUTORIAL directory. Makefiles for the HELLO example are in the MFC\SAMPLES\HELLO directory.

Makefile Defaults

By default, the tutorial makefiles all build release mode programs.

If you want to build debug mode, see "How to Switch from Release to Debug Builds" on page 12. The paths listed here and throughout the chapter are relative paths. You'll probably want to write your own code in separate directories. In particular, if you follow the tutorial step by step and create your own code files to match the code listings given in the chapters, you'll need to write your versions in your own directories to avoid naming clashes with the same files provided on the distribution disks.

After a build, you'll find the .EXE and .OBJ files for the build in these directories.

How to Build with PWB

This section outlines the basic steps needed to build a program using PWB. For further information about using PWB, see the PWB Tutorial in *Environment and Tools*.

► **To build your example program with PWB, do the following:**

1. Run PWB.
   - If you are running Windows, start PWB from the Program Manager.
   - If you are running PWB from DOS or from a DOS command shell in Windows, type `PWB` at the command line.
2. From the Project menu, choose the Open Project command.

   This command displays a dialog box of the same name in which you can name your project and select a makefile for the project.

3. Select the .MAK file for your program and click the OK button.

   Use the following makefiles for the tutorial examples:

   - For DMTEST, Chapter 2, use DMTEST.MAK.
   - For HELLO, Chapter 3, use HELLO.MAK.
   - For PHBOOK, Chapters 4 through 6, use PHBOOK.MAK.
   - For CMDBOOK, Chapter 4, use CMDBOOK.MAK.

   For the locations of these makefiles, see "Makefile Locations" on page 8.

4. From the Options menu, choose the Build Options command. The Build Options dialog box appears. Confirm that "Use Release Options" is set. This is the default provided in the tutorial's makefiles.

   Note The makefiles for the tutorial examples build release versions by default. To see how this default setup looks in PWB, choose some of the options commands in the PWB Options menu and examine the dialog boxes to see which options are selected.

5. To compile, from the Project menu, choose the Build command.

   The Build command on the menu now includes your project name, for example, "Build: DMTEST.exe."

6. When the build finishes, the Build Results dialog box will appear, showing how many errors and warnings occurred. If there were no errors, click the Cancel button in the dialog box.

   If you're compiling a DOS program and running PWB under DOS or Windows, you can instead click the Run Program button in the dialog box. This executes the program from PWB. If you're compiling for Windows, see "How to Run Your Program" on page 13.

7. If you get errors, do one of the following:
   - After building a DOS program from the Build menu, choose Debug to locate the source of the error. For more details, see your documentation for PWB in *Environment and Tools*.
   - After building a Windows program, switch to the Windows Program Manager and run the Codeview debugger. See your Windows documentation for more information.

How to Build with NMAKE

NMAKE is the command-line project-management facility provided with Microsoft C/C++. This section explains how to build the tutorial programs from the DOS command line with NMAKE. Although NMAKE isn't required for the tutorial examples, if you are unfamiliar with NMAKE and would like more explanation, see the documentation for NMAKE in *Environment and Tools*.

► **To build your program with NMAKE, do the following:**

1. At the DOS command line (or in a DOS command shell from Windows), type

   ```
   NMAKE  <makefile name>
   ```

   This command is not case sensitive.

   Use the appropriate makefile name for your program. The names for the tutorial examples are shown below.
   - For DMTEST, Chapter 2, use DMTEST
   - For HELLO, Chapter 3, no makefile name is required (default of MAKEFILE is used)
   - For PHBOOK, Chapters 4 through 6, use PHBOOK
   - For CMDBOOK, Chapter 4, use CMDBOOK
2. If you supply additional arguments to NMAKE, you must add the /F compiler option:

   ```
   NMAKE /F <makefile name> [[other arguments]]
   ```

   The /F compiler option is not required when only one argument is given.
3. When the build completes, debug and rebuild the program if necessary. If you make changes to your source files, NMAKE rebuilds the files that have changed and any files that depend on them.
4. After a successful build, run the program to test it. For information about running your programs, see "How to Run Your Program" on page 13.

How to Switch from Release to Debug Builds

If you want to build the examples, or your own programs, for debugging, follow the instructions in this section. When you build for debugging, CodeView information is generated to help you use the CodeView debugger. For more information about CodeView, see *Environment and Tools*.

When you switch to debug mode, you set the **_DEBUG** flag so that the debugging facilities built into the example programs are enabled. The **ASSERT** and **TRACE** macros, explained in Chapter 2, will provide diagnostic information. Diagnostic messages are displayed to the debugger.

Debug Mode for PWB

► **To switch from release builds to debugging builds with PWB:**

1. Open the project file for the program.
2. From the Build menu, choose the Build Options command.
3. Select "Use Debug Options."

   You will need to change the "Build directory" field in the same dialog box to specify the correct directory to build into.
4. Then build the program as explained in "How to Build with PWB" on page 9.

Debug Mode for NMAKE

► **To switch from release builds to debugging builds with NMAKE:**

- Add the `DEBUG=1` option to your NMAKE command line, as illustrated here for the PHBOOK program:

```
NMAKE /F PHBOOK DEBUG=1
```

Note Case is significant for the `DEBUG=1` option. It must be uppercase.

1.4 How to Run Your Program

This section explains how to run your tutorial example programs. The process differs depending on whether the program was built as a DOS program or a Windows program.

The makefiles for the tutorial programs are set up for DOS or Windows. The DMTEST program in Chapter 2 and the CMDBOOK program discussed at the end of Chapter 4 run under DOS. The other programs, HELLO and PHBOOK, must be run under Windows.

How to Run Your DOS Program

If you build a DOS program with PWB and you run PWB under DOS, you can run the program directly from PWB.

► **To run a DOS program from PWB:**

- When the PWB build-completion dialog box appears, click the Run Program button.

If you build your DOS program with NMAKE, run the program from the DOS command line.

► **To run a DOS program from the command line:**

- At the DOS command line, type the name of your program's .EXE file and press ENTER.

  If necessary, supply a pathname to the .EXE file.

  For example, to run the executable file for DMTEST, type the following at the command line (assuming that DMTEST.EXE is in the \TUTORIAL directory):

```
DMTEST
```

Remember that the default makefiles for most of the tutorial programs build into subdirectories of the MFC\SAMPLES\TUTORIAL directory, as explained in "Makefile Locations" on page 8. Run the program from that directory, or supply a path to that directory.

How to Run Your Windows Program

The example programs for Chapters 3 through 6 are Windows programs. They must be run from within the Microsoft Windows environment.

Note You cannot use the Run or Debug menu commands in PWB with a Windows program. To run your Windows program, switch to the Windows Program Manager. To debug a Windows program, switch to the Program Manager and execute the CodeView debugger from there.

► **To run your program in Windows:**

1. From the Windows Program Manager File menu, choose the Run command.

   The Run dialog box appears.

2. Type the path and program name just as you would from the DOS command line.

   For example, to run the HELLO program (assuming that HELLO.EXE is in the C700\MFC\SAMPLES\HELLO directory), type

   ```
   C:\C700\MFC\SAMPLES\HELLO\HELLO.EXE
   ```

   If you prefer, you can use the New command from the Program Manager's File menu to assign an icon to your program and tell Windows where to locate the executable file. Then you can run the program by double-clicking its icon. For information on this process, see your Microsoft Windows documentation.

1.5 Summary

This chapter introduced the Microsoft Foundation Class Library tutorial, shows you how to use it, and explained how to build and run the example programs.

The next chapter explores the Microsoft Foundation Class Library's collection classes and introduces the fundamentals of designing with objects. Even if you're anxious to get to the Windows chapters, this chapter is worth reading.

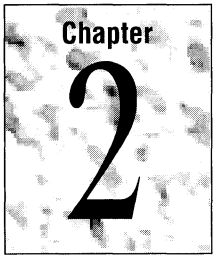
Chapter 3 demonstrates the fundamentals of Windows programming with the Microsoft Foundation Class Library. You'll build a simple Windows version of the familiar "Hello, World!" program.

Chapters 4 through 6 take you deeper into Windows programming with the Microsoft Foundation Class Library. You'll build a small but complete Windows application called PHBOOK, which uses the data model designed in Chapter 2 to implement a simple personal phone list program.

Chapter 4 also briefly discusses the CMDBOOK program, which provides a character-based parallel to PHBOOK.

In addition to the tutorial and its example programs, your distribution disks contain many other examples that use the Microsoft Foundation Classes. For non-Windows applications, see TEMPLDEF and RESTOOL. For Windows applications, see CHART, MULTIPAD, and SHOWFONT, among others. A README file explains what these applications do and what programming techniques they demonstrate.

Creating a Data Model with the Microsoft Foundation Classes



The previous chapter introduced you to the steps necessary to build programs that use the Microsoft Foundation Class Library. This chapter assumes that you have successfully installed the Microsoft Foundation Class Library on your system and have read the previous chapter.

In this and the rest of the tutorial chapters, you will learn how to use the components of the library in the design of your programs. You will see how the built-in functionality of the Microsoft Foundation Classes can reduce the code that you have to write to realize the goals of your programs. This chapter emphasizes the Microsoft Foundation Class Library's "collection" classes in particular.

This chapter describes how to use Microsoft Foundation Classes to create a data model for a simple name and phone number database program. The purpose of the chapter is to demonstrate how the component classes from the Microsoft Foundation Class Library can help you design at a high level of abstraction and greatly simplify the implementation of your design.Use of the Microsoft Foundation Class Library for Microsoft Windows programming is covered in the next four chapters. This chapter explains the non-Windows classes of the Microsoft Foundation Class Library.

2.1 In This Chapter

Follow this tutorial to write a simple program that uses an object-oriented database. The database and the data objects stored in it are based on classes from the Microsoft Foundation Class Library. The process can be summarized as follows:

1. Design the data items.
2. Design a list to store the data.
3. Test the data model.

The rest of this section describes the example program.

The Data Model Program

In this chapter, you will develop a data model and a simple program to test it. The example is called DMTEST.

This section is an overview of what the program does and what you will be learning about the Microsoft Foundation Class Library.

What Is a Data Model?

A data model is an abstraction that represents the structure of the data that a program manages. The data model for this chapter, for example, consists of person objects and a list object to contain them.

The data model is completely independent of the user interface. The data model knows nothing about how the data will be displayed to the user, nor does it know how the user will communicate with the program. The data model communicates with the user interface of the program through a well-defined set of member functions.

One way to think of this relationship is that the data model is a server, and the user interface is a client. The user interface interacts with the user and translates user input into requests that the interface sends to the data model. The data model responds to the requests and sends information back to the user interface, which the user interface then displays to the user.

Figure 2.1 shows the relationship between the user, the user interface, and the data model. You can see that the user never directly interacts with the data model.

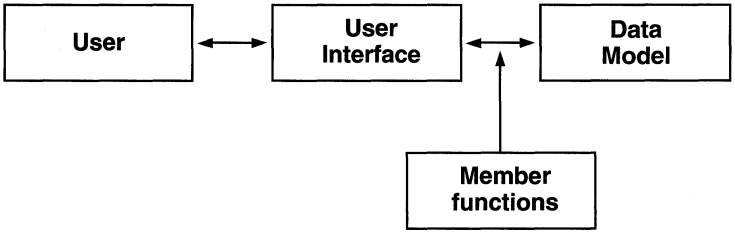


Figure 2.1 The Data Model and the User Interface

The data model's independence from the user interface is a very important concept in the design of resusable programs. This independence enhances the reusability of the data model. Thus, the data model for names and phone numbers developed in this chapter can be used with a text-only interface or with a Microsoft Windows interface without any changes to the data model. The next chapters show how to

develop a Windows user interface that works with the data model to create a complete interactive program.

What the Example Does

The purpose of the example program is to manage a list of names and phone numbers. Each data item represents a single person and contains that person's name and phone number. The user can add persons to the list, find all matches for a specified name, and save and restore the data to and from a disk file. The C++ objects constituting the data model provide all of these capabilities.

The DMTEST program demonstrates these capabilities by:

- Creating a database and adding names to it.
- Serializing the database (writing it to disk).
- Deserializing the database (reading it from disk).
- Searching the database for a person.
- Disposing of the objects.

Code for the Data Model

To view the complete code for the DMTEST program, see Listings 1, 2, and 3 at the end of the chapter.

The code shown is available on the distribution disks in files PERSON.H, PERSON.CPP, and DMTEST.CPP.

Microsoft Foundation Classes Used in the Data Model

This chapter demonstrates the use of six classes from the Microsoft Foundation Class Library:

- Class **CObject**

  Each record in the database is represented by an object of the class `CPerson`, which is derived from the Microsoft Foundation Class **CObject**.The `CPerson` class builds upon the functionality of **CObject**, adding member variables representing the name and phone number of a person. In addition, the `CPerson` class overrides functions from **CObject** that are related to serialization so that the name and phone number can be saved to and restored from disk.

- Class **CObList**

  Collection classes are designed to contain collections of similar objects. The Microsoft Foundation Class Library provides three kinds of collections: lists, arrays, and maps (or dictionaries). In the example, a list collection is used to contain all the `CPerson` objects in the database. The Microsoft Foundation

Classes include several useful list classes, but because we need some specialized list functionality, the list used in this chapter will be derived from the Microsoft Foundation **CObList** class. The specialized list class makes use of all of **CObList**'s capabilities, but also adds some new functions, including one that can find all elements of the list that match a specified last name.

- Class **CString**

  CString objects represent the name and phone number member variables of a `CPerson` object.

- Class **CTime**

  A **CTime** object represents the last modification time and date of a `CPerson` object.

- Classes **CFile** and **CArchive**

  A **CFile** object identifies and opens the file used to serialize the database. Serialization in the Microsoft Foundation Class Library is done with a **CArchive** object, which uses an opened **CFile** object to perform the serialization.

Other Capabilities Demonstrated

The following list describes other capabilities that your data objects can use. Some are available because your objects are derived from class **CObject** and some simply because you are using the Microsoft Foundation Class Library. These capabilities are demonstrated in the DMTEST program.

- Serialization

  Serialization is the act of saving an object to a disk file or reading it back in (sometimes called "deserialization"). Objects of the class `CPerson` can serialize themselves to and from a disk file. Likewise, the collection of `CPerson` objects can serialize itself and all its elements. Because a collection can automatically serialize all its elements, the act of serializing the database is reduced to a single function call to serialize the collection. This cuts down the amount of code you have to write.

- Exceptions

  The Microsoft Foundation Class Library's exception-handling mechanisms "catch" exceptions that are "thrown" by the Microsoft Foundation Class Library functions as those functions encounter errors. Exceptions provide a way for you to respond to errors, especially by the file-handling classes. This, along with the **TRACE** macro for printing messages, gives you a convenient, structured way to process errors, both during development and in the finished program.

- Diagnostics

  The **TRACE** macro is used throughout the code in this chapter to provide diagnostic output to track program progress. The Microsoft Foundation Class Library also provides an ability to dump the contents of objects to assist in

debugging your program and facilities for testing the validity of your assumptions, such as whether a pointer points to a valid area of memory.

2.2 How to Write the DMTEST Program

This section gives an overview of the steps in writing the DMTEST program. As you work through the steps, you will learn what files to prepare, where to put the code in them, and how to compile the program.

To write the DMTEST program with the Microsoft Foundation Classes:

1. Design the `CPerson` data object.

   Derive the `CPerson` data class from the Microsoft Foundation Class **CObject**. Figure 2.2 shows the class hierarchy for `CPerson`. For more about this step, see "Design the CPerson Data Object" on page 22.

2. Design the `CPersonList` object.

   Derive the `CPersonList` object from the Microsoft Foundation Class **CObList**. Figure 2.2 shows the class hierarchy for `CPersonList`. For more about this step, see "Design the CPersonList Object" on page 36.

3. Test the data model.

   Write a small test program to demonstrate the capabilities of the data model. For more about this step, see "Test the Data Model" on page 49.

4. Build the program.

   Compile and link the data model test program. For more about this step, see "Build the Program" on page 65.

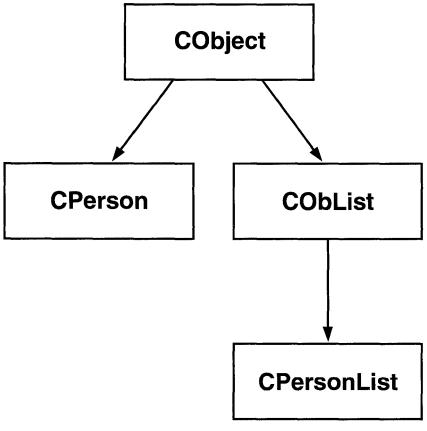


Figure 2.2 Object Class Hierarchies for Data Model Objects

2.3 Design the CPerson Data Object

This section explains the first step in writing the DMTEST program: design the data and prepare two code files. You'll create an interface file for the data model called PERSON.H and an implementation file called PERSON.CPP. The process will be described in two main steps with several substeps:

1. Create the interface file.
   a. Create a file PERSON.H and add directives.
   b. Add a class declaration.
   c. Add an **#endif** directive.
2. Create the implementation file.
   a. Create a file PERSON.CPP and add directives.
   b. Add macro invocations.
   c. Add member function definitions.

Create the Interface File

PERSON.H contains a list of preprocessor directives and two C++ class declarations. Class `CPerson` defines a class of "person objects." Class `CPersonList` defines a class of list objects capable of containing `CPerson` objects.

► To create the PERSON.H interface file:

1. Create a file called PERSON.H and add the following directives at the top of the file:

```
#ifndef __PERSON_H__
#define __PERSON_H__
#ifdef _DOS
    #include <afx.h>
#else
    #include <afxwin.h>
#endif
#include <afxcoll.h>
```

   The directives above prevent any implementation code in PERSON.H from being included twice, in case two files include PERSON.H and one of them includes the other. This is a common safety measure used in all Microsoft Foundation **#include** files. If the code were included twice, you'd get linker errors.

2. Add the following class declaration for `CPerson` to PERSON.H:

```
// class CPerson:
// Represents one person in the phone database.  This class is
// derived from CObject (mostly to get access to the serialization
```

```
// protocol).
//
class CPerson : public CObject
{
    DECLARE_SERIAL( CPerson );

public:
//Construction
    // For serializable classes, declare a constructor with no
arguments.
    CPerson()
    { m_modTime = CTime::GetCurrentTime(); }

    CPerson( const CPerson& a );

    // For our convenience, also declare a constructor with arguments.
    CPerson( const char* pszLastName,
    const char* pszFirstName,
    const char* pszPhoneNum );

//Attributes
    // Member functions to modify the protected member variables.
    void SetLastName( const char* pszName )
    { ASSERT_VALID( this );
        ASSERT( pszName != NULL);
        m_LastName = pszName;
        m_modTime = CTime::GetCurrentTime(); }

    const CString& GetLastName() const
    { ASSERT_VALID( this );
        return m_LastName; }

    void SetFirstName( const char* pszName )
    { ASSERT_VALID( this );
        ASSERT( pszName != NULL );
        m_FirstName = pszName;
        m_modTime = CTime::GetCurrentTime(); }

    const CString& GetFirstName() const
    { ASSERT_VALID( this );
        return m_FirstName; }

    void SetPhoneNumber( const char* pszNumber )
    { ASSERT_VALID( this );
        ASSERT( pszNumber != NULL );
        m_PhoneNumber = pszNumber;
        m_modTime = CTime::GetCurrentTime(); }

    const CString& GetPhoneNumber() const
    { ASSERT_VALID( this );
        return m_PhoneNumber; }

    const CTime GetModTime() const
```

```
    { ASSERT_VALID( this );
        return m_modTime; }

//Operations
    CPerson& operator=( const CPerson& b );

//Implementation
protected:
    // Member variables that hold data for person
    CString         m_LastName;
    CString         m_FirstName;
    CString         m_PhoneNumber;
    CTime           m_modTime;

public:
    // Override the Serialize function
    virtual void Serialize( CArchive& archive );

#ifdef _DEBUG
    // Override Dump for debugging support
    virtual void Dump( CDumpContext& dc ) const;
    virtual void AssertValid() const;
#endif
};
```

C++ techniques are used to derive class `CPerson` from the Microsoft Foundation Class **CObject**. Notice that the class declares several constructors, an overloaded assignment operator, several member functions for getting and setting the attributes of a person object, and several member variables for storing information about a person. The class also takes advantage of **CObject**'s ability to write an object's contents to disk by invoking the **DECLARE_SERIAL** macro and overriding the **Serialize** member function. In addition, it overrides several of **CObject**'s member functions to provide diagnostics during program development. The class is discussed in detail in "Discussion: The CPerson Class" on page 28.

3. Add the following directive as the last line of code in PERSON.H:

```
#endif // __PERSON_H__
```

Later you'll add the declaration for class `CPersonList` to PERSON.H. Be sure to keep this **#endif** directive as the last line of code in the file.

Create the Implementation File

PERSON.CPP contains several preprocessor directives, two macro invocations to support object serialization, and definitions for several of the member functions of classes `CPerson` and `CPersonList`. Some of the member functions were defined inline as part of the class declarations in file PERSON.H, but the longer ones were left for definition in PERSON.CPP.

► To create the PERSON.CPP implementation file:

1. Create a file called PERSON.CPP and add the following directives at the top of the file:

```
#include "person.h"
#include <string.h>

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif
```

   Besides **#include** directives, these directive lines provide support for debugging when the **_DEBUG** flag is defined. The directives help identify which file an error occurred in.

2. Add the following macro invocations to PERSON.CPP below the preprocessor directives:

```
// Call 'IMPLEMENT_SERIAL' macro for all the
//  classes declared in person.h

IMPLEMENT_SERIAL( CPerson, CObject, 0 )
```

   This line adds code to support object serialization so that `CPerson` and `CPersonList` objects can write themselves to a disk file and read themselves in from a file. Serialization is discussed in detail in "How to Serialize a CPerson Object" on page 33. Later you'll add a similar line for the `CPersonList` class.

3. Add the following member function definitions for class `CPerson`:

```
// CPerson::CPerson
// Copy Constructor for CPerson class
//
CPerson::CPerson( const CPerson& a )
{
    ASSERT_VALID( this );
    ASSERT_VALID( &a );
    m_LastName = a.m_LastName;
    m_FirstName = a.m_FirstName;
    m_PhoneNumber = a.m_PhoneNumber;
    m_modTime = a.m_modTime;
}


// CPerson::CPerson
// Memberwise Constructor for CPerson class
//
CPerson::CPerson( const char* pszLastName,
     const char* pszFirstName,
     const char* pszPhoneNum )
```

```
{
    ASSERT_VALID( this );
    m_LastName = pszLastName;
    m_FirstName = pszFirstName;
    m_PhoneNumber = pszPhoneNum;
    m_modTime = CTime::GetCurrentTime();
}


// CPerson::operator=
// Overloaded operator= to perform assignments
//
CPerson& CPerson::operator=( const CPerson& b )
{
    ASSERT_VALID( this );
    ASSERT_VALID( &b );
    m_LastName = b.m_LastName;
    m_FirstName = b.m_FirstName;
    m_PhoneNumber = b.m_PhoneNumber;
    m_modTime = b.m_modTime;
    return *this;
}


// CPerson::Dump
// Write the contents of the object to a
// diagnostic context
//
// The overloaded '<<' operator does all the hard work
//
#ifdef _DEBUG

void CPerson::Dump( CDumpContext& dc ) const
{
    ASSERT_VALID( this );
    // Call base class function first
    CObject::Dump( dc );

    // Now dump data for this class
    dc  << "\n"
    << "Last Name: " << m_LastName << "\n"
    << "First Name: " << m_FirstName << "\n"
    << "Phone #: " << m_PhoneNumber << "\n"
    << "Modification date: " << m_modTime << "\n";
}


void CPerson::AssertValid() const
{
    CObject::AssertValid();
}

#endif
```

```
// CPerson::Serialize
// Read or write the contents of the object
// to an archive
//
void CPerson::Serialize( CArchive& archive )
{
    ASSERT_VALID( this );
    // Call base class function first
    CObject::Serialize( archive );

    // Now dump data for this class
    if ( archive.IsStoring() )
    {
                TRACE( "Serializing a CPerson out.\n" );
    archive << m_LastName << m_FirstName
              << m_PhoneNumber << m_modTime;
    }
    else
    {
                TRACE( "Serializing a CPerson in.\n" );
    archive >> m_LastName >> m_FirstName
               >> m_PhoneNumber >> m_modTime;
    }
}
```

These definitions complete the member functions declared as part of the `CPerson` class declaration. They define the following member functions:

- A copy constructor—to make copies of a `CPerson` object
- A constructor—to create new `CPerson` objects with initializing information passed as parameters
- An overloaded assignment operator—to assign one `CPerson` object to another
- A `Dump` member function— to dump diagnostic information about `CPerson` objects
- An `AssertValid` member function—to test the validity of a `CPerson` object
- A `Serialize` member function—to write a `CPerson` object's data to a file or read data from a file into a `CPerson` object

Later you'll add member function definitions for class `CPersonList` to PERSON.CPP.

At this point, you've added all of the code for class `CPerson` to both files.

To continue the tutorial, see "Design the CPersonList Object" on page 36. For more information about the steps you just completed, see "Discussion: The CPerson Class," which follows.

Discussion: The CPerson Class

This discussion does not instruct you to add any new code to your files. Code is sometimes repeated to illustrate a point, but you do not need to add it.

A `CPerson` object is designed to manage the name and phone number of one person. A `CPerson` object is constructed from the `CPerson` class. `CPerson` is derived publicly from class **CObject**.

In effect, a `CPerson` object is created from a stock of existing components: a string class, a time class, and a general object class (**CObject**, from which `CPerson` is derived). The new `CPerson` class automatically inherits a great deal of functionality from **CObject** and then adds to its inheritance. `CPerson` also relies heavily on the built-in capabilities of **CString** and **CTime**. These component classes encapsulate specialized kinds of data storage, control access to that data, and cooperate in the ability of `CPerson` to serialize itself and to provide diagnostic information.

The result of creating `CPerson` from library components is that you write less code, and the code encapsulated by the component objects comes fully tested. This leaves you more time to focus on high-level design issues and reduces debugging and maintenance time and costs.

The `CPerson` class declaration given above requires some explanation. The discussion that follows explains how to construct `CPerson` objects, how `CPerson` data is stored and accessed, how to test a new object for validity, how to serialize a `CPerson` object, and how to get a diagnostic dump of a `CPerson` object during debugging.

Class CPerson Constructors

A `CPerson` object is constructed when one of its constructors is invoked. `CPerson` has two constructors, one with parameters and one without. It also has a copy constructor and an overloaded assignment operator.

Constructor with Parameters You can use a constructor with parameters to construct `CPerson` objects in your program. To initialize objects created this way, the public constructor for `CPerson` takes initialization arguments, which you supply at construction time.

Constructor Without Parameters The parameterless constructor of class `CPerson` is used internally by the class to support serialization, but you must supply it in your class declaration.

How to Construct CPerson Objects You can construct a `CPerson` object in two ways:

- You can construct a `CPerson` object on the frame of a function (as a local variable) as follows:

```
void f()
{
    CPerson thePerson( "Smith", "Mary", "435-8159" );

    // Other function code

}
```

- You can construct a `CPerson` object dynamically on the heap, using the C++ **new** operator, as follows:

```
CPerson* pPerson = new CPerson( "Smith", "Mary", "435-8159");
```

The Copy Constructor The copy constructor is a special constructor that takes a C++ reference to a `CPerson` object as its argument. The copy constructor copies the data members of the person object whose copy constructor has been invoked into the object passed as an argument. This allows you to make duplicates of `CPerson` objects if you need to. For an important discussion, see the shaded box "Copy Constructors" on page 30.

The Overloaded Assignment Operator Class `CPerson` overloads the C++ assignment operator (=) to provide correct assignment of one person object to another. For an important discussion, see the shaded box "Copy Constructors" on page 30.

About the Constructors For any class derived from **CObject** that will be serialized, the Microsoft Foundation Class Library requires that you define a constructor with no arguments. This constructor must at least put the object into a valid state so that it can be safely deleted. Usually this means setting all the member variables to some default null state. If you forget to define a constructor with no arguments for a serializable class, you will get a compiler error at the line that contains the **IMPLEMENT_SERIAL** macro.

For serializable classes, you must define a constructor with no arguments.

The constructor with no arguments is used only internally for serialization. The declaration inside class `CPerson` looks like this:

```
CPerson();
```

In addition to the required constructor with no arguments, you may also declare a constructor that takes arguments to initialize the member variables of the object, as in `CPerson`:

```
CPerson( const char* pszLastName,
     const char* pszFirstName,
     const char* pszPhoneNum );
```

This practice of defining several variations on the constructor is common in C++ programming. You must declare at least one public constructor.

Objects constructed on the frame of a function are allocated when the function is called. At the time of allocation, the constructor is invoked and the object initialized. When the function completes, the destructors of any objects allocated on the frame are invoked automatically to destroy the objects.

Copy Constructors

In general, it is good C++ practice to also supply a copy constructor and an overloaded assignment operator with your data classes. A copy constructor creates a duplicate of the current object. An overloaded assignment operator allows you to assign one object to another with the C++ assignment operator. You'll often want to duplicate an object or assign one to another.

However, in C++ the semantics of copying can be quite subtle. There is one case in which the constructors provided by a class are not invoked to initialize a new object—when an object is initialized with another object of its class. By default in this case, C++ performs a "memberwise" copy of an object's data members. This may not be what you want, since, for example, copying a pointer to a null-terminated C string or array copies an address, not the string's data. This can lead to errors if the original object's destructor frees the memory that the copied object points to. For more information, see the *C++ Tutorial* in your Microsoft C/C++ package.

`CPerson` is an example of a class that does provide copy and assignment services. The data members of `CPerson` are **CString** objects, which themselves provide a copy constructor and an overloaded assignment operator. Thus, you can safely copy and assign `CPerson` objects.

You can construct objects dynamically on the heap at any time. Use the **new** operator to allocate the space. When you call **new**, the object's constructor is invoked automatically and the object is initialized. The **new** operator returns a pointer to the object. However, unlike allocation on the frame, allocation on the heap requires that the programmer explicitly deallocate the object with the C++ **delete** operator.

In the case of classes such as `CPerson`, it is good practice for created objects to live beyond the scope of the function where they are created, so objects are most often created with the **new** operator.

How CPerson Data Is Stored and Accessed

`CPerson` uses two Microsoft Foundation Classes to store its data in member variables. This section explains the use of classes **CString** and **CTime**.

Class CString **CString** is used to store the first and last names and the phone number. The declarations of these member variables looks like this:

```
CString     m_pszLastName;
CString     m_pszFirstName;
CString     m_pszPhoneNumber;
```

`CPerson` uses the Microsoft Foundation Class **CString** to store its data in the `m_pszLastName`, `m_pszFirstName`, and `m_pszPhoneNumber` member variables. The names of these variables follow the Microsoft Foundation Class Library convention of prefixing member variable names with "m_".

The **CString** class is used because **CString** objects are dynamic. A **CString** object encapsulates a string that can automatically grow up to approximately 32,000 characters. **CString**s also have the ability to serialize themselves, so when it is time to serialize a `CPerson` object, you can simply rely upon each **CString** in the object to serialize itself without needing to know about the internal structure of the **CString**. This is a considerable advantage.

Class CTime **CTime** is used to store date and time information. The modification time variable in `CPerson` is declared like this:

```
CTime       m_modTime;
```

The `CPerson` class also uses a **CTime** member variable to represent the date and time of the last modification of each `CPerson` object. The time and date are set

when the object is created, and modified whenever any of the other member variables are changed. For example, the `SetLastName` member function looks like this:

```
void CPerson::SetLastName( const char* pszName )
{
    m_pszLastName = pszName;
    m_modTime = CTime::GetCurrentTime();
}
```

In the body of this function, the first line sets the last name value. The second line sets the modification time. This operation is transparent to the user of the class. It demonstrates one of the virtues of providing a controlled interface to a class, as discussed in the next section.

Like the name and phone number information, the modification date is serialized with the `CPerson` object.

How to Access CPerson Data

Given a `CPerson` object, how do you examine and update its data? The class provides four pairs of member functions to set and get the values of a `CPerson` object's member variables. For example, use the `SetLastName` member function to set a new value for a person object's last name member. Typically, the values are set all at once by the public constructor, which takes arguments and loads them into the member variables. But you can also modify the object's data at any time with the "Set" and "Get" member functions.

It's common in object-oriented programming to define such data-access functions. Notice that the member functions are defined as public to invite use, while the member variables are defined as protected to prevent outside use. Such controlled access to protected member variables helps to ensure the data's integrity.

For example, in the `CPerson` class, all the member functions that set the member variables also set the modification-date member variable to reflect the time of the change. If it were possible to access member variables directly from outside the object, a person object could be updated without updating its modification date. The data could become invalid without this kind of secure encapsulation.

Validity Testing for Objects

Class `CPerson` demonstrates some of the facilities provided by the Microsoft Foundation Classes for testing the validity of objects. The class uses the **ASSERT_VALID** macro and overrides the **AssertValid** member function of class **CObject**.

Along with the **ASSERT** macro, which is discussed in Chapter 4, these facilites allow you to test your assumptions. Before you use an object, it's wise to test the

validity of its internal state. For example, if you have an object that represents a stack data structure, you can confirm that the top and bottom of the stack are in a valid relationship: the top is either "above" the bottom or equal to it (in the case of an empty stack). Similarly, a pointer to an object must point to a valid area of memory.

During debugging, you can use these assumption-testing facilities freely. If an assumption fails the test, the program asserts, prints a diagnostic message, and halts. When you build the program for release, the assumption testing code is not compiled.

`CPerson` demonstrates the form of these facilities, but you'll need to wait until Chapter 4 for a more meaningful example. In `CPerson`, the **ASSERT_VALID** macro typically tests whether the **this** pointer is **NULL** within a member function. The override of the **AssertValid** member function simply calls its base class. In class `CDataBase` in Chapter 4, you'll see some more serious testing of assumptions.

For more information about assumption testing, see Chapter 11 of the cookbook.

How to Serialize a CPerson Object

The ability to serialize data to and from the disk is probably the most important attribute of the `CPerson` class. To enable serialization, you can derive your class from the **CObject** class, use the **DECLARE_SERIAL** and **IMPLEMENT_SERIAL** macros, and override the virtual **Serialize** member function. The version of **Serialize** that is defined for **CObject** can work with data in the **CObject** class only. When you override **Serialize** for your class, you extend the capability of the function so that it can handle the data in your class as well as the data in **CObject**.

The Archive Object The **Serialize** function takes a **CArchive** object as its argument. **CArchive** is a Microsoft Foundation Class that provides a context for reading and writing object data to and from a disk file. An archive uses a class's overloaded insertion and extraction operators (<< and >>) to write and read object data to and from the storage media. Notice that even though an archive uses the same overloaded operators as the general-purpose I/O stream objects (such as **cin** and **cout**) provided with Microsoft C, a **CArchive** object is different from an I/O stream:

- A **CArchive** object handles data in binary form, which the computer can process efficiently.
- General-purpose I/O streams handle data in textual form, which makes it easy for humans to interpret.

An individual **CArchive** object can be created for reading or for writing, but not for both at the same time. Thus, each **CArchive** object maintains internal status information that indicates whether it is for loading (reading) or for storing (writing) data. The **Serialize** function checks that status in the **CArchive** object passed to it as an argument to determine whether to read or write the object data.

The TRACE Macro The code for the `Serialize` member function in `CPerson` was shown on page 27. Notice that it uses the **TRACE** macro to print out a debugging message indicating that the function has been called. The **TRACE** macro is designed so that it is activated when you build a debug version of your program, but deactivated when you build a release version. Thus, you can sprinkle **TRACE** messages liberally throughout your code to monitor program execution during development, and they will be deactivated automatically when you build a version of your program to ship. This means that you don't have to go back and comment the messages out or bracket them with **#ifdef _DEBUG** and **#endif** statements.

What Serialize Does When a `CPerson` object is serialized, the following actions occur:

1. The `CPerson` object's `Serialize` member function is called.

   In the example program in this chapter, the `CPersonList` object that contains a database of `CPerson` objects calls `Serialize` for each object in the list.

2. The `Serialize` member function immediately calls **Serialize** for its base class, which in this example is **CObject**. The base class's data is thus written to disk.

   By calling the base class version of **Serialize** first, you ensure that all the contents of the base class portion of your object are correctly serialized. If the base class is itself a derived class, the **Serialize** function for the base class of `CPerson` is also called. Thus **Serialize** is called for all classes in the hierarchy above your class. Figure 2.3 shows this sequence for `CPerson`.

3. The `CPerson` object's `Serialize` member next writes its own data to disk.

   To prepare, the `Serialize` function calls the **IsStoring** member function for the **CArchive** object. If the archive is for storing data, then each member variable of the `CPerson` object is written with the << insertion operator. If, on the other hand, the archive is for reading, the >> extraction operator is used to read each member variable. The insertion and extraction operators perform all the operations necessary to make sure that the member variables are correctly written or read.

   Notice that the member variables are extracted in the same order that they were inserted. This ensures that each member variable is matched with the correct data.

Any serializable object can be written to disk with a single line of code. As you'll see later, a collection or list of serializable objects can also be serialized simply

and with minimal code. Figure 2.3 shows the steps taken as a `CPerson` object is serialized.

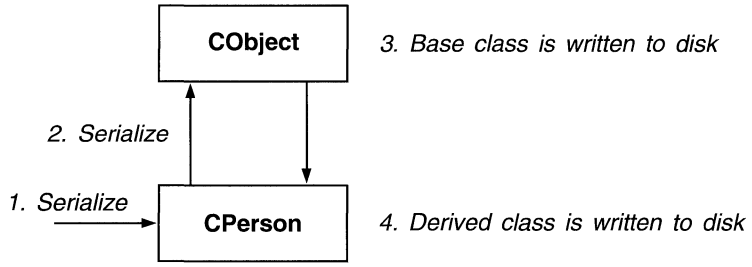


Figure 2.3 Steps in Serializing a Person Object

How to Dump a CPerson Object's Data

The previous section described how to override the **Serialize** member function to read and write object contents to and from a **CArchive** object. The **Dump** member function performs a similar function, but instead of writing out binary data to a **CArchive** object, **Dump** writes a textual representation of the object data to a **CDumpContext** object. A **CDumpContext** object is typically used for debugging output during program development. It is similar to the general I/O streams in that it is often directed to the screen or a log file, but a **CDumpContext** object can be used only for output, not for input. A **CDumpContext** object's output cannot be formatted.

The **Dump** function writes the contents of an object, including descriptive labels, to a diagnostic context. If you compare it to the **Serialize** function, you will see three main differences:

- **Serialize** operates on a **CArchive** object and **Dump** uses a **CDumpContext** object.
- **Dump** writes out descriptive text labels along with the textual representation of the value of each member variable, while **Serialize** reads and writes only the binary value of the member variables.
- **Dump** is a write-only operation.

The code for the `Dump` member function of `CPerson` is shown on page 26. Notice that it is bracketed with an **#ifdef _DEBUG/#endif** block so that it will not be included in a release version of your program.

Notice also that, like `Serialize`, the first statement of `Dump` calls the base class's version of the function. This ensures that the contents of the base class portion of the object get dumped first. (That is, if the base class has any member variables, they're written out before member variables of the derived class.) Then the rest of

the `Dump` function uses the insertion operator to send descriptive labels and the contents of each member variable of the `CPerson` class. Once again, the insertion operator does all the hard work.

The Microsoft Foundation Class Library provides a predefined **CDumpContext** object named **afxDump**. You can use this object as the argument to the `Dump` function. The **afxDump** dump context object writes the dump information to standard output. For a Windows program, **afxDump** uses the Windows function **OutputDebugString** to route the dump information to the debugger if present, or to the auxiliary (**AUX**) device if not. This occurs only if tracing is enabled. For more information, see Technical Note 7 in file TN007.TXT in your distribution disks. The **afxDump** object is available only in debug mode, but class **CDumpContext** can be used for programs in release mode.

For example, if you had a `CPerson` object, you could dump it to the predefined dump context with the following code. The DMTEST code also calls the **SetDepth** member function of class **CDumpContext** to specify that all data of all objects is to be dumped. This call is discussed again in step 1 of "Test the Data Model" on page 49 and "How FindPerson Is Tested" on page 58.

```
CPerson myPerson( "Smith", "Mary", "223-9175" );
myPerson.AssertValid();   // See if object contains valid data
myPerson.Dump( afxDump );
```

The output looks like this:

```
Last Name: Smith
First Name: Mary
Phone #: 223-9175
Modification date: Fri Jul 19 13:36:30 1999
```

2.4 Design the CPersonList Object

This section explains the second step in writing the DMTEST program: design a collection object to hold the `CPerson` data. You'll be adding more code to your PERSON.H and PERSON.CPP files. This process will be described in two main steps:

1. Add a class declaration to file PERSON.H.
2. Add code to PERSON.CPP.
   a. Add a macro invocation.
   b. Add code for member functions.

Each `CPerson` object represents one person in the database of names and phone numbers that you're building. You can use one of the collection classes from

the Microsoft Foundation Class Library to derive your own list class to manage a list of `CPerson` objects. The list class designed in this section is called `CPersonList`.

► To add the CPersonList object code to PERSON.H:

Add the following class declaration for `CPersonList` to file PERSON.H after the `CPerson` declaration:

```
// class CPersonList:
// This represents a list of all persons in a phone database.  This
// class is derived from CObList, a list of pointers to CObject-type
// objects.

class CPersonList : public CObList
{
    DECLARE_SERIAL( CPersonList )

public:
//Construction

    CPersonList()
    { m_bIsDirty = FALSE; }

    // Add new functions
    CPersonList* FindPerson( const char * szTarget );

    // SetDirty/GetDirty
    // Mark the person list as "dirty" (meaning "modified"). This
    // flag can be checked later to see if the database
    //  needs to be saved.
    //
    void SetDirty( BOOL bDirty )
    { ASSERT_VALID( this );
        m_bIsDirty = bDirty; }

    BOOL GetDirty()
    { ASSERT_VALID( this );
        return m_bIsDirty; }

    // Delete All will delete the Person objects as well as the
    // pointers.
    void DeleteAll();

protected:
    BOOL  m_bIsDirty;
};
```

C++ techniques are used to derive class `CPersonList` from the Microsoft Foundation Class **CObList**. The class declares a constructor, a member function for searching the list, member functions for flagging changes to the list and for testing

that flag, and a member function for deleting the list. The class also invokes the **DECLARE_SERIAL** macro to do its part in serializing the data to and from the disk.

Note that `CPersonList` requires only a single constructor with no arguments. This constructor is defined inline, so you don't have to add a function definition for it to your PERSON.CPP file. The definition is taken care of as part of the declaration. For more information about the `CPersonList` constructor, see "Discussion: The CPersonList Class" on page 39.

This completes your PERSON.H file.

► To add the CPersonList object code to PERSON.CPP:

1. Add the following macro invocation to PERSON.CPP just below the similar **IMPLEMENT_SERIAL** macro for `CPerson`:

```
IMPLEMENT_SERIAL( CPersonList, CObList, 0 )
```

2. Add code for the member functions of each `CPersonList` at the end of PERSON.CPP:

```
// CPersonList::FindPerson
//
CPersonList* CPersonList::FindPerson( const char * szTarget )
{
    ASSERT_VALID( this );

    CPersonList* pNewList = new CPersonList;
    CPerson* pNext = NULL;

    // Start at front of list
    POSITION pos = GetHeadPosition();

    // Iterate over whole list
    while( pos != NULL )
    {
       // Get next element (note cast)
       pNext = (CPerson*)GetNext( pos );

       // Add current element to new list if it matches
       if ( _strnicmp( pNext -> GetLastName(), szTarget,
         strlen( szTarget ) ) == 0 )
           pNewList -> AddTail(pNext);
    }

    if ( pNewList -> IsEmpty() )
    {
       delete pNewList;
       pNewList = NULL;
    }
```

```
    return pNewList;
}


// CPersonList::DeleteAll
// This will delete the objects in the list as the pointers.
//
void CPersonList::DeleteAll()
{
    ASSERT_VALID( this );
    POSITION pos = GetHeadPosition();

    while( pos != NULL )
    {
       delete GetNext(pos);
    }
    RemoveAll();
}
```

`CPersonList` declares four member functions to manipulate the data stored in the list and in the member variable of `CPersonList`. The four member functions are `FindPerson`, `SetDirty`, `GetDirty`, and `DeleteAll`. `SetDirty` and `GetDirty` are declared inline, so you've already added their definitions with the class declaration in PERSON.H.

This completes your PERSON.CPP file.

At this point, your PERSON.H and PERSON.CPP files are complete. Check to be sure that you have added all the **#include** and other compiler directives. Compare your code to the full code listings presented in Listings 1 and 2 on page 66 and 69. In the continuation of the tutorial, you will create a third file containing a main program to test the data model.

To continue the tutorial, see "Test the Data Model" on page 49. For more information about the steps you just completed, see "Discussion: The CPersonList Class," which follows.

Discussion: The CPersonList Class

This discussion does not instruct you to add any new code to your files. Code is sometimes repeated to illustrate a point, but you do not need to add it.

A `CPersonList` object is a "collection" of `CPerson` objects. You construct a `CPersonList` object from the `CPersonList` class declared above. `CPersonList` is derived publicly from the Microsoft Foundation Class **CObList**.

Figure 2.4 shows a collection schematically.

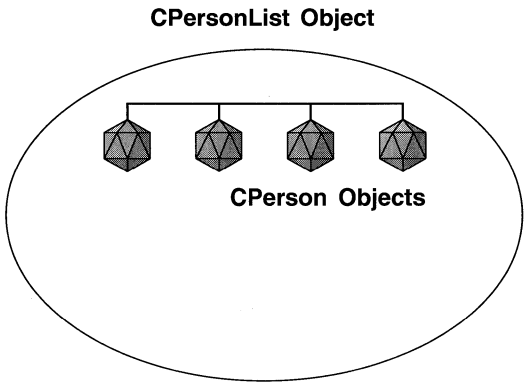


Figure 2.4 A Person List Object and the Data Objects It Contains

The discussion that follows explains how to construct a `CPersonList` object, how to add person objects to it, how to serialize the whole list, and how to search the list for all persons with a given last name.

Collection Classes

The Microsoft Foundation Class Library supplies fourteen collection classes in three categories:

Classes in the first category encapsulate arrays. The Microsoft Foundation Class Library provides arrays of bytes, words, double words, pointers to objects, objects, and strings.

Classes in the second category encapsulate linked lists. The Microsoft Foundation Class Library provides lists of pointers, objects, and strings. For example, the class **CObList** can contain **CObject**s or objects of any class derived from **CObject**, such as `CPerson`. `CPersonList` is derived from **CObList**.

Classes in the third category encapsulate maps. A map is a collection that maps one kind of data to another. For example, the Microsoft Foundation Class Library includes class **CMapStringToOb**. This class contains mappings from strings to **CObject**s. This class lets you store objects and look them up by string-type key values. It is a good alternative for implementing `CPersonList` if you want to experiment.

For more information about the Microsoft Foundation Class Library's collection classes, see the *Class Libraries Reference* and Chapter 9.

How to Construct a CPersonList Object

The constructor for this class is declared publicly because you must be able to invoke it from outside the class. A list requires no initialization values, so you don't need a second constructor that takes arguments. The constructor of `CPersonList` simply initializes the list's **m_bIsDirty** member variable to **FALSE**, signifying that the list currently has no unsaved changes.

▶ To construct a `CPersonList`:

Use one of the two ways you construct a `CPerson` object:

- On the frame of a function

  When a function executes, a list declared as a local variable is constructed. When the function returns, the list's destructor is called to destroy the list. For example, to construct a local list in a function:

```
void AFunction()
{
    CPersonList  myList;  // List is constructed

    // Operations on the list

    // List is destroyed as function exits
}
```

- In the heap

  A list constructed dynamically with the C++ **new** operator exists until you explicitly destroy it by invoking the C++ **delete** operator. To create a new `CPersonList` in the heap:

```
CPersonList* pMyList = new CPersonList;
```

  This invokes the `CPersonList` constructor for `pMyList`.

How to Add Persons to the List

This section shows how to create and add person objects to a list object. You do not need to add this code to your files.

▶ To add a person object to the list:

1. Create a person object:

```
CPerson* pNewPerson = new CPerson( "Smith",
                "Mary",
                "435-8159" );
```

2. Add the person to the list:

```
pMyList -> AddHead( pNewPerson );
```

This code calls the **AddHead** member function of `CPersonList`, which the class inherits from **CObList** without overriding. Because the list was created as a pointer to a `CPersonList` object, the **->** operator is used to access the member function.

How to Serialize the List

This section explains how to serialize the `CPerson` database; that is, to write the data members of all `CPerson` objects in the list to disk. It also explains the reverse process: deserializing (reading) the database back from disk. In both cases, serialization uses a **CFile** object and a **CArchive** object. These objects are discussed in the next section below.

The process also optionally uses a **CFileException** object. The **CFileException** object is used to return information about any file errors that may have occurred during the attempt to open the file. For more information about serialization, see "More on Serialization" on page 60. For more information about exceptions, see "Exception Handling" on page 61.

▶ To serialize a CPersonList:

1. Create a **CFile** object and call its **Open** member function.

   In the call to **Open**, specify the open permissions. Because the permissions are defined inside class **CFile**, you need to qualify their identifiers with the **CFile** class name:

```
theFile.Open( pszFileName, CFile::modeCreate | CFile::modeWrite )
```

   The arguments to **Open** specify the filename, the access mode (**CFile::modeRead** or **CFile::modeWrite**), and, optionally, a preconstructed **CFileException** object (not used here).

   Note The filename argument is a C++ reference to a **CString** object. You can pass an ordinary null-terminated C-language string in a **CString** argument, as is done with the `szFileName` string in DMTEST.CPP. For more information about **CString**, see the *Class Libraries Reference* and Chapter 7.

2. Use the **CFile** object to create a **CArchive** object:

```
CArchive theOutArchive( &theOutFile, CArchive::store );
```

This example shows an archive created for writing. The second argument specifies whether the archive is for loading (reading) or storing (writing). For more details, see "More on Serialization" on page 60.

3. Use the **CArchive** object as you would a C++ iostream, such as **cin** or **cout**:

```
theOutArchive << pList;
```

You use the overloaded insertion (**<<**) or extraction (**>>**) operator to pass data through the archive object. The one you use depends on whether you are writing or reading the list.

4. Close the archive object and then the file object, in that order:

```
theOutArchive.Close();
theOutFile.Close();
```

If you close them out of order, an exception is thrown.

The default serialization behavior of a collection is to serialize all its elements. Because all the elements of a `CPersonList` are `CPerson` objects, and because `CPerson` objects know how to serialize themselves, you can rely on the default behavior for a correctly serialized list. This means that you can serialize a `CPersonList` and all its elements to or from a **CArchive** with a single statement. The code fragments below show how easily this can be done (don't add this code to your files):

```
CPersonList* pList = new CPersonList;

// Add CPerson elements
.
.
.
// To serialize the collection out to disk
// Create a file object
CFile theOutFile;
// Open the file
if( !theOutFile.Open( pszFileName, CFile::modeCreate ||
CFile::modeWrite ), NULL )
{
    // Error handling
    TRACE( "Unable to open a file for serialization\n" );
    return FALSE;
}
// Create an archive object from the file object
CArchive theOutArchive( &theOutFile, CArchive::store );

// Serialize
theOutArchive << pList;
```

```
// Close the archive and the file, in that order
theOutArchive.Close();
theOutFile.Close();
```

And to deserialize the collection back in from disk:

```
CPersonList* pOtherList = NULL;
CFile theInFile;
if( !theInFile.Open( pszFileName, CFile::modeRead ) )
{
    // Error handling
}

// Create an archive object for reading
CArchive theInArchive( &theInFile, CArchive::load );

// Deserialize
theInArchive >> pOtherList;
// Close the archive and the file, in that order
```

As the code shows, a single insertion or extraction statement is sufficient to completely serialize the entire collection. Type-safety is maintained during the serialization operation. Thus, the serialization mechanism checks the type of each object in the file before it is added to the list and throws an exception if it encounters an incorrect object type. Figure 2.5 shows the steps in serializing a list of objects.

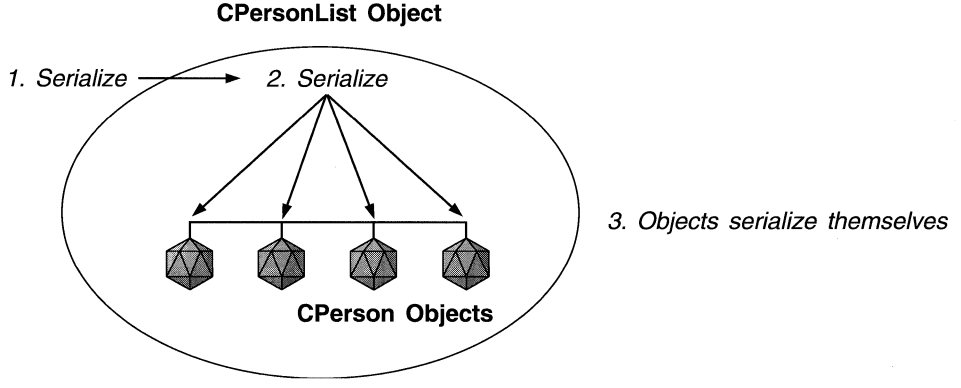


Figure 2.5 Steps in Serializing a List of Person Objects

Note You must use the **DECLARE_SERIAL** macro in the declaration of a class for the class to be serializable. Look at the declarations of `CPerson` and `CPersonList` discussed previously. You must also use the **IMPLEMENT_SERIAL** macro in the .CPP file that defines the member functions declared in your .H file.

The Microsoft Foundation Class Library provides class **CFile** and class **CArchive** for working with disk files. You can still use standard C I/O routines, but these classes are a good alternative because they encapsulate file handling in objects. For more information about **CFile**, **CArchive**, and serialization, see "More on Serialization" on page 60.

How to Search the List

This section explains how to search the `CPersonList` for a particular person. In the example program, the last name is used as the key for the search. The steps are described below:

► **To search a CPersonList:**

1. Call the target list's `FindPerson` member function:

```
CPersonList* pFound = pDataBase -> FindPerson( szLastName );
```

`FindPerson` takes an argument of type **CString**. You can pass an initialized **CString** object or a null-terminated string containing the last name to find. For more information on the properties of a **CString** object, see Chapter 7.

`FindPerson` returns a new `CPersonList` object, `pFound`. If there were any finds, `pFound` contains pointers to the objects found in the target list, `pDataBase`.

Note The pointers in the new list point to the original objects, which are still in the original target list.

2. Examine the returned list to see if it contains objects:

```
if( !pFound -> IsEmpty() )
{
    // Do something with the found list
}
```

`CPersonList` inherits an **IsEmpty** member function from **CObList**. It returns **TRUE** if the list is empty.

3. Delete the found list when you finish with it:

```
delete pFound;
```

The found list is allocated dynamically by `FindPerson`, which returns a pointer to the found list. Because `pFound` is allocated in the heap, you must deallocate its storage with the **delete** operator.

Remember that the found list's elements are pointers to `CPerson` objects still in the original target list. Don't call the `DeleteAll` member function of `pFound`. That would destroy the objects in the target list. All you want to do is delete the found list, which leaves the target list intact.

In the code for `FindPerson`, you can see a number of objects and their member functions in use. To iterate over the list in search of the key last name, `FindPerson` uses the **GetHeadPosition** and **GetNext** member functions that `CPersonList` inherits from **CObList**.

GetHeadPosition is used to start at the beginning of the list, and **GetNext** is used to get access to successive elements of the list. To compare the key string with the last name member variable of each `CPerson` object in the list, `FindPerson` calls the **_strnicmp** run-time function and passes it the last name of the next person in the list.

CString has several member functions for string comparison—these are comparable to the C run-time library functions for string comparison. To obtain the last name of the next person in the list, `FindPerson` calls the person object's **GetLastName** member function, which you wrote as part of class `CPerson`.

If matching last names are found, `FindPerson` returns a list containing pointers to all found objects. The returned list can be useful in its own right, since it is a `CPersonList` object with all the capabilities of the original list from which it was built. You can display the found list, operate on it, add to it, delete from it, and so on.

How to Delete the Entire Database

This section explains how to delete the database and its contents after you finish using it.

▶ **To delete the entire database:**

1. Call the `CPersonList` object's `DeleteAll` member function to delete the contents:

   ```
   pDataBase -> DeleteAll();
   ```

   The `DeleteAll` member function, which you added to `CPersonList`, performs two operations. First, it iterates through the list and invokes the **delete** operator for each contained object in turn. Then it calls the **CObList** member function `RemoveAll`, which frees underlying storage and marks the list as empty. After the `RemoveAll` operation, the list contains no pointers to any objects.

2. Invoke the C++ **delete** operator to delete the list object:

   ```
   delete pDataBase;
   ```

The reason you added a `DeleteAll` member function to `CPersonList` is that deleting objects from the list and removing them from the list are not the same thing. If you delete an object from the list, the list still has a pointer to the object, but this pointer is now invalid because the object it formerly pointed to no longer exists.

On the other hand, if you remove an object from the list, you remove the list's pointer to the object, but the object itself still exists. Thus, if you want to keep a list but empty it without destroying the objects it contained, use `RemoveAll`. If you want to destroy the contents of a list without destroying the list itself, use `DeleteAll`. If you want to destroy both the list and its contents, first call `DeleteAll`, then invoke **delete** on the list object. Figure 2.6 summarizes these deletion processes.

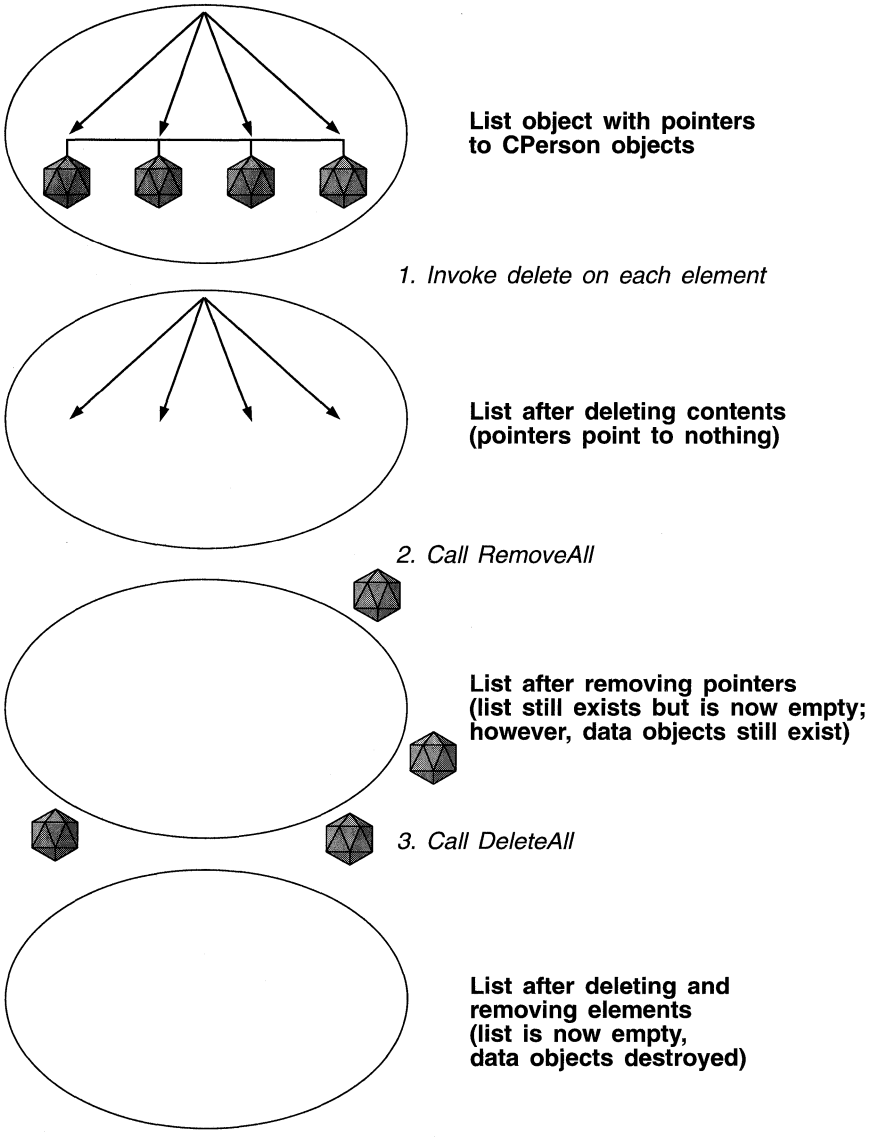


Figure 2.6 Deletion and Removal of Data in a List

The `DeleteAll` member function of class `CPersonList` uses two other useful member functions inherited from class **CObList**. The **GetHeadPosition** function returns a value of type **POSITION**, which provides access to the head element of the list. The **GetNext** function returns the **POSITION** of the next element. You can access the element with this **POSITION** value. To see these member

functions in use, see the code for `DeleteAll` in file PERSON.CPP in Listing 2. For more information about class **CObList**, see the *Class Libraries Reference*.

Summary of Collection Use

The first step in designing a data model is to declare the `CPerson` class. Each `CPerson` object represents the name and phone number of an individual person. Next, use one of the Microsoft Foundation Class Library's list classes to derive a custom list class to hold a collection of `CPerson` objects and to search for a person matching a specified last name.

To customize the list class, you derive from **CObList** and add new functions as necessary to add new functionality. For the most part, however, you can rely on the inherited functionality of the Microsoft Foundation list class, including the ability to serialize the collection and all its elements with a single statement.

2.5 Test the Data Model

This section explains the third step in building the data model: testing it. This process will be described in several steps:

1. Create a file called DMTEST.CPP.
2. Create a person database.
3. Serialize the list (write it to a file).
4. Deserialize the stored data.
5. Test the `FindPerson` function.
6. Clean up and quit.
7. Add the supporting functions.

▶ To create a file called DMTEST.CPP:

1. Add the following **#include** directive:

```
#include "person.h"
```

2. Add the following function prototypes below the **#include** line:

```
// Function prototypes.
CPersonList* MakeDataBase();
CFile* OpenForReading( const CString& rFileName );
CFile* OpenForWriting( const CString& rFileName );
CPersonList* ReadDataBase( CFile* pFile );
BOOL WriteDataBase( CFile* pFile, CPersonList* pDataBase );
```

```
void TestFindPerson( CPersonList* pDataBase );
void ListDataBase( CPersonList* db );
```

3. Add the beginnings of your program's **main** function:

```
void main()
{
    const char szFileName[] = "tutorial.dat";


#ifdef _DEBUG
    // Prepare for display of search results
    const int nDumpChildren = 1;
    afxDump.SetDepth( nDumpChildren );
#endif
```

The first line in the function declares a filename. The other lines prepare the **afxDump** "dump context," predefined by the Microsoft Foundation Class Library. When results of the database search test are displayed later, all objects in the list will dump their contents. The default "depth" of 0 dumps only information about the list, not its contents. For more information about setting the depth, see "How FindPerson Is Tested" on page 58.

▶ To create a person database:

Add the following lines to the **main** function:

```
printf( "Create a person list and fill it with persons\n" );
CPersonList* pDataBase = MakeDataBase();
```

These lines call a function to create and return a database with several person objects in it. You'll add the `MakeDataBase` function later.

▶ To serialize the list (write it to a file):

Add the following lines of code to the **main** function below the lines added in the previous step:

```
printf( "Serialize the person list\n" );
CFile* pFile;  // Declare a file object

TRY
   {
   // Could throw a file exception if can't open file
   pFile = OpenForWriting( szFileName );
```

```
// Could throw an archive exception if can't create
WriteDataBase( pFile, pDataBase );
}
CATCH( CFileException, theException )
{
   printf( "Unable to open file for writing\n" );
   exit( -1 );
}
AND_CATCH( CArchiveException, theException )
{
   printf( "Unable to save the database\n" );
   pFile -> Close();  // Close up
   delete pFile;
   exit( -1 );
}
END_CATCH

// No exceptions, so close up
pFile -> Close();
delete pFile;

ListDataBase( pDataBase );

printf( "Delete the list and all its elements\n" );
pDataBase -> DeleteAll();
ListDataBase( pDataBase );
delete pDataBase;
```

These lines create a file object, use it to create an archive object, use the archive object to serialize the list, and clean up afterward. The same process was covered briefly earlier in "How to Serialize the List" on page 42. For more information about the roles of the **CFile** and **CArchive** objects, see "More on Serialization" on page 60. Once the list has been serialized to disk, the list can be deleted.

Additionally, the code handles exceptions that may occur if a file can't be opened or an archive can't be created successfully or fails while writing. The use of the **TRY**, **CATCH**, **AND_CATCH**, and **END_CATCH** macros for exception handling is discussed in "Exception Handling" on page 61.

► **To deserialize the stored data:**

Add the following lines to the **main** function below those added in the previous step:

```
printf( "Deserialize the data from disk into a new list\n" );
CPersonList* pDataBase2;   // Create a new, empty list

TRY
{
// Could throw a file exception if can't open file
   pFile = OpenForReading( szFileName );
```

```
// Could throw an archive exception if can't create
   pDataBase2 = ReadDataBase( pFile );
}
CATCH( CFileException, theException )
{
   printf( "Unable to open file for reading database\n" );
   exit( -1 );
}
AND_CATCH( CArchiveException, theException )
{
   printf( "Unable to read the database\n" );
   pFile -> Close();  // Close up before exiting
   delete pFile;
   exit( -1 );
}
END_CATCH

// No exceptions, so close up
pFile -> Close();
delete pFile;

ListDataBase( pDataBase2 );
```

As in the previous step, these lines create a file object, then an archive object, and use the archive object to deserialize the list. After it's read in, the list is printed to demonstrate success. Most of the real work of file object creation, file opening, archive creation, and serialization takes place in the supporting functions `OpenForReading` and `ReadDataBase`. You'll add these functions in the last step.

It's particularly interesting that deserialization recreates the person objects as it reads in their data. As objects are created and reinitialized with the data from disk, they are stored in the new, empty list, `pDataBase2`. Thus, the objects formerly serialized are fully reconstructed by the process of deserialization. For more information about deserialization, see "More on Serialization" on page 60.

► To test the FindPerson function:

Add the following lines to the **main** function below the lines added in the previous step:

```
printf( "Test the FindPerson function\n" );
if ( pDataBase2 != NULL )
   TestFindPerson( pDataBase2 );
```

If anything has been read into the new list, these lines call the `TestFindPerson` function, passing it a pointer to the list. You'll add `TestFindPerson` in the last step. For more information about this testing, see "How FindPerson Is Tested" on page 58.

▶ To clean up and quit:

Add the following lines to the **main** function after the lines added in the previous step:

```
    printf( "Delete the list and all its elements\n" );
    pDataBase2 -> DeleteAll();
    delete pDataBase2;

    TRACE( "End of program\n" );
}
```

These lines first delete all elements of the list, then delete the list object. Deleting dynamically constructed objects such as the list and file objects prevents memory leaks. These lines conclude the code for the **main** function. Your **main** function should now match the one in Listing 3 on page 73.

▶ To add the supporting functions:

1. Add the `MakeDataBase` function, which creates a new list object, fills it with person objects, and returns the list:

```
// MakeDataBase - Create a database and add some persons
//
CPersonList* MakeDataBase()
{
    TRACE( "  Make a new person list on the heap\n" );
    CPersonList* pDataBase = new CPersonList;

    TRACE( "  Add several new persons to the list\n" );
    CPerson* pNewPerson1 = new CPerson( "Smith", "Mary", "435-8159" );
    pDataBase -> AddHead( pNewPerson1 );

    CPerson* pNewPerson2 = new CPerson( "Smith", "Al", "435-4505" );
    pDataBase -> AddHead( pNewPerson2 );

    CPerson* pNewPerson3 = new CPerson( "Jones", "Steve",
            "344-9865" );
    pDataBase -> AddHead( pNewPerson3 );

    CPerson* pNewPerson4 = new CPerson( "Hart", "Mary", "287-0987" );
    pDataBase -> AddHead( pNewPerson4 );

    CPerson* pNewPerson5 = new CPerson( "Meyers", "Brian",
            "236-1234" );
    pDataBase -> AddHead( pNewPerson5 );

    TRACE( "  Return the completed database to main\n" );
    return pDataBase;
}
```

2. Add the `OpenForReading` function, which creates a file object and uses it to open the file specified by **rFileName** for reading:

```
// OpenForReading - open a file for reading
//
CFile* OpenForReading( const CString& rFileName )
{
    CFile* pFile = new CFile;
    CFileException* theException = new CFileException;
    if ( !pFile -> Open( rFileName, CFile::modeRead, theException ) )
    {
       delete pFile;
       TRACE( "  Threw file exception in OpenForReading\n" );
       THROW( theException );
    }

    // Exit here if no exceptions
    return pFile;
}
```

3. Add the `OpenForWriting` function, which creates a file object and uses it to open the file specified by **rFileName** for writing:

```
// OpenForWriting - open a file for writing
//
CFile* OpenForWriting( const CString& rFileName )
{
    CFile* pFile = new CFile;
    CFileStatus status;
    UINT nAccess = CFile::modeWrite;

    // GetStatus will return TRUE if file exists,
    // or FALSE if it doesn't exist
    if ( !CFile::GetStatus( rFileName, status ) )
       nAccess |= CFile::modeCreate;

    CFileException* theException = new CFileException;
    if ( !pFile -> Open( rFileName, nAccess, theException ) )
    {
       delete pFile;
       TRACE( "  Threw a file exception in OpenForWriting\n" );
       THROW( theException );
    }

    // Exit here if no errors or exceptions
    TRACE( "  Opened file for writing OK\n" );
    return pFile;
}
```

4. Add the `ReadDataBase` function, which creates an archive object and uses it to deserialize data from the disk, creating a new list:

```
// ReadDataBase - read data into a person list
//
CPersonList* ReadDataBase( CFile* pFile )
{
    CPersonList* pNewDataBase = NULL;

    // Create an archive from pFile for reading
    CArchive archive( pFile, CArchive::load );

    TRY
    {
       // and deserialize the new database from the archive
       archive >> pNewDataBase;
    }
    CATCH( CArchiveException, theException )
    {
       TRACE( "  Caught an archive exception in ReadDataBase\n" );
#ifdef _DEBUG
    theException -> Dump( afxDump );
#endif
       archive.Close();

       // If we got part of the database then delete it so we don't
       // have any Memory leaks
       if ( pNewDataBase != NULL )
       {
           pNewDataBase -> DeleteAll();
           delete pNewDataBase;
       }
       THROW_LAST();
    }
    END_CATCH

    // Exit here if no errors or exceptions
    archive.Close();
    return pNewDataBase;
}
```

5. Add the `WriteDataBase` function, which creates an archive object and uses it to serialize the list to disk:

```
// WriteDataBase - write data from a person list to disk
//
BOOL WriteDataBase( CFile* pFile, CPersonList* pDataBase )
{
    // Create an archive from pFile for writing
    CArchive archive( pFile, CArchive::store );

    TRY
```

```
    {
        // and serialize the data base to the archive
        archive << pDataBase;
    }
    CATCH( CArchiveException, theException )
    {
        TRACE( "  Caught an archive exception in WriteDataBase\n" );
#ifdef _DEBUG
    theException -> Dump( afxDump );
#endif
        archive.Close();
        THROW_LAST();
    }
    END_CATCH

    // Exit here if no errors or exceptions
    archive.Close();
    return TRUE;
}
```

6. Add the `TestFindPerson` function, which demonstrates successful and unsuccessful searches of the list, using the list's `FindPerson` function:

```
// TestFindPerson - test CPersonList::FindPerson
//
void TestFindPerson( CPersonList* pDataBase )
{
    printf( "  Looking for the name Banipuli\n" );
    CPersonList* pFound = pDataBase -> FindPerson( "Banipuli" );
    if ( pFound -> IsEmpty() )
    {
        printf( "  No matching persons\n" );
    }
    else
    {
    printf( "  Found matching persons\n" );
#ifdef _DEBUG
    pFound -> Dump( afxDump );
#endif
    }
    delete pFound;

    printf( "  Looking for the name Smith\n" );
    pFound = pDataBase -> FindPerson( "Smith" );
    if ( pFound -> IsEmpty() )
    {
        printf( "  No matching persons\n" );
    }
    else
    {
        printf( "  Found matching persons\n" );
#ifdef _DEBUG
```

```
    pFound -> Dump( afxDump );
#endif
    }

    // Don't DeleteAll the found list since it
    // shares CPerson objects with database
    delete pFound; // Deletes only the list object
}
```

7. Add the `ListDataBase` member function, which writes out the contents of the database:

```
void ListDataBase( CPersonList* db )
{
    CPerson* pCurrent;
    POSITION pos;

    if ( db -> GetCount() == 0 )
       printf( "  List is Empty\n" );
    else
    {
       printf( "  List contains:\n" );
       pos = db -> GetHeadPosition();
       while ( pos != NULL )
       {
           pCurrent = (CPerson*)db -> GetNext( pos );
           printf( "\t%s, %s\t%s\n", (const char*)pCurrent ->
               GetLastName(),
               (const char*)pCurrent -> GetFirstName(),
               (const char*)pCurrent -> GetPhoneNumber() );
    }
    }
}
```

   Your code for the supporting functions should match that given in Listing 3 on page 73.

Your DMTEST.CPP file is now complete. It contains one **#include** directive, six function prototypes, the **main** function, and seven supporting functions.

To continue the tutorial and compile the program, see "Build the Program" on page 65. For more information about the steps you just completed, see "Discussion: Testing the Data Model," which follows.

Discussion: Testing the Data Model

This discussion does not instruct you to add any new code to your files. Code is sometimes repeated to illustrate a point, but you do not need to add it.

The DMTEST program has no real user interface. The **main** function in file DMTEST.CPP simply creates a database, fills it with person objects, and demonstrates its capabilities. In a later chapter of the tutorial, the data model will be integrated with a Microsoft Windows user interface. A sample program which provides a character-based user interface to the database is also provided on your distribution disks.

Previously, in the discussion of the `CPersonList` class, you saw briefly how to add person objects to a list, how to search a list for a person, and how to delete the list and its contents when you finished with it. The following discussion recaps and adds to the previous discussion.

How the Database Is Created and Destroyed

`MakeDataBase` is interesting primarily for how it makes and returns a filled database. It makes the database by constructing a `CPersonList` object dynamically in the heap, using the **new** operator. After filling the list with `CPerson` objects, also constructed in the heap with **new**, `MakeDataBase` returns a pointer to the list object. This lets the database object and the objects stored in it persist after the function returns.

Because the database and its contents were created as dynamic objects in the heap, you must explicitly destroy them when you finish. Thus the last two lines in the **main** function call the `CPersonList` member function `DeleteAll` to delete the `CPerson` objects in the list and then use the **delete** operator to delete the list object itself:

```
pDataBase2 -> DeleteAll();  // Delete the contents
delete pDataBase2;  // Delete the list object
```

How FindPerson Is Tested

The `FindPerson` member function of class `CPersonList` searches the list for a given last name and builds a second list containing pointers to any found `CPerson` objects. The function returns a pointer to the list of found objects.

The `TestFindPerson` function in DMTEST.CPP simply creates a new list and searches it for two different last names.

The first search looks for the name "Banipuli," which is not in the database. When `FindPerson` returns an empty list, `TestFindPerson` uses the `CPersonList` member function **IsEmpty**, inherited from the base class of `CPersonList`, **CObList**, to detect the list's empty condition.

The second search looks for "Smith," a name that is in the database several times. This time `FindPerson` returns a list containing pointers to two `CPerson` objects. These contain the names "Mary Smith" and "Al Smith."

Because the search was successful, `TestFindPerson` uses the predefined "dump context" **afxDump** to dump the contents of the found list to the standard output. Recall that, at the beginning of the **main** function, the **afxDump** object's **SetDepth** member function was called to specify that not only the list object be dumped but the list's content objects as well.

The **afxDump** object is used here as a simple way to display the contents of the found list.. Note that **afxDump** only works if the **_DEBUG** flag is defined, so you only get the dump during debugging.

The output of this dump is shown with the full program output below (for a release version of the program).

```
Create a person list and fill it with persons
Serialize the person list
  List contains:
    Meyers, Brian    236-1234
    Hart, Mary    287-0987
    Jones, Steve    344-9865
    Smith, Al    435-4505
    Smith, Mary    435-8159

Delete the list and all its elements
  List is Empty
Deserialize the data from disk into a new list
  List contains:
    Hart, Mary    287-0987
    Jones, Steve    344-9865
    Smith, Al    435-4505
    Smith, Mary    435-8159

Test the FindPerson function
  Looking for the name Banipuli
  No matching persons
  Looking for the name Smith
  Found matching persons

Delete the list and all its elements
End of program
```

More on Serialization

You've already seen the code to serialize the database to disk and to deserialize it from disk in the steps on pages 42-44. (This topic was also discussed earlier, in "How to Serialize the List" on page 42.) This section shows how to use **CFile** and **CArchive** objects. The next section shows how to handle exceptions.

Recall that to serialize a `CPersonList` you:

1. Create a **CFile** object and call its **Open** member function.
2. Create a **CArchive** object, passing the **CFile** to it as an argument.
3. Use the overloaded insertion (<<) or extraction (>>) operator to pass data through the archive object. The one you use depends on whether you are writing or reading the list.
4. When you finish, close the archive object and then the file object.

The objects used in this process are a **CFile** object, a **CArchive** object, and, optionally, a **CFileException** object. The **CFileException** object is used to return information about any file errors that may have occurred during the attempt to open the file. For more information on exceptions, see "Exception Handling" on page 61.

The CFile Object The Microsoft Foundation Class Library provides class **CFile** and two derived classes, **CStdioFile** and **CMemFile**. **CFile** is for binary files, **CStdioFile** is for buffered text files as defined in STDIO.H, and **CMemFile** is for memory-based files. You can use these classes or your own classes derived from them.

A file object, of any of these classes, handles file opening, closing, and other file operations. The DMTEST program uses class **CFile** and its **Open**, **Close**, and **GetStatus** member functions. For additional **CFile** capabilities, see the *Class Libraries Reference*.

You can create a **CFile** object on the frame of a function or in the heap. The code in DMTEST.CPP typically declares pointers to **CFile** objects and allocates the objects dynamically with **new**. See the `OpenForReading` function in step 2.

Before you open the **CFile** object, you can check that the file exists by calling the **CFile** object's **GetStatus** member function. Pass the filename and a preconstructed **CFileStatus** object. **CFileStatus** is a Microsoft Foundation Class that contains status information about the file when **GetStatus** returns. If the call to **GetStatus** fails, it returns **FALSE**. You can examine the contents of the **CFileStatus** object to see why. For more information about **CFileStatus**, see the *Class Libraries Reference*.

In DMTEST.CPP, if **GetStatus** returns **FALSE**, it means the file doesn't exist. The inclusive **OR** assignment operator, |=, is used to add the mode **CFile::modeCreate** to the existing access parameter, *nAccess*, before calling **Open**.

You need an opened **CFile** object to pass to the constructor of your **CArchive** object, as in `ReadDataBase`. The arguments to the **Open** member function of class **CFile** are a filename, the access mode, and an optional **CFileException** object. The filename argument is a null-terminated string. The access mode argument is one of the access modes defined in class **CFile**, such as **CFile::modeRead** or **CFile::modeWrite**. The optional **CFileException** argument is an object that you construct before you pass it to **Open**. When **Open** returns, this object contains information about any exception that is thrown. For more information on this argument, see "Exception Handling" on this page.

Be sure to close your **CArchive** object before you call **CFile::Close**.

The CArchive Object A **CArchive** object uses a **CFile** object to establish a connection with a disk file. Once the connection is made, you can use the **CArchive** object as a stream, much as you would use the standard C++ iostreams, **cin** and **cout**.

Pass an already-opened **CFile** object to the **CArchive** constructor when your archive object is created. A **CArchive** object must be initialized for loading (reading) or storing (writing). It can't be used for both. To initialize the archive for reading, pass **TRUE** as the second argument to the constructor. For writing, pass **FALSE**. For example, in `ReadDataBase`, the archive object is constructed like this:

```
CArchive archive( pFile, TRUE );
```

Class **CArchive** overloads the insertion (<<) and extraction (>>) operators. To write data with a **CArchive** opened for writing, do this:

```
archive << pDataBase;
```

The overloaded insertion operator causes the **Serialize** member functions of the `CPersonList` object and its `CPerson` elements to write the appropriate data to the file specified when the archive was opened.The serialization process could throw a **CArchiveException**. For an explanation of exceptions and how they're handled, see the next section.

Exception Handling

This section explains the use of the Microsoft Foundation Class Library's exception-handling mechanism as used in DMTEST.CPP. For more information about exception handling, see Chapter 12, and see the *Class Libraries Reference* under class **CException**.

Some of the code in the Microsoft Foundation Class Library generates exceptions when errors occur. The Microsoft Foundation Classes provide a mechanism for handling exceptions in an object-oriented fashion. Certain error-prone operations, such as memory allocation, file processing, and archive processing, detect errors and "throw" exceptions. When an exception is thrown, an exception object of the appropriate class is constructed. To process exceptions, you set up "exception frames" with a set of predefined macros. (For more information about how to set up exception frames, see "The Exception Frames in the Main Function" on page 63.)

You can then "catch" the exception at any level of the hierarchy of exception frames in your program. For example, suppose that, in your program, function A calls function B, which calls function C. Suppose further that you set up exception frames in functions A and C. If an exception is thrown in function C, you can choose to catch it in C, or you can catch it in A. To view this mechanism diagrammatically, see Figure 2.7.

You can also explicitly throw exceptions yourself, as a way of selecting the level at which your program responds to an error.

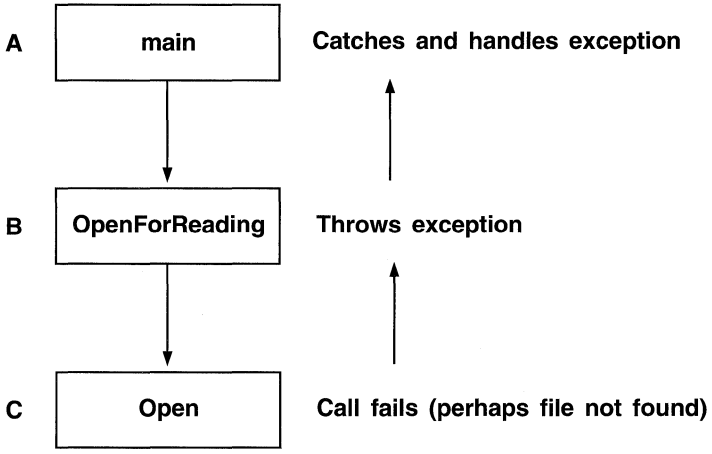


Figure 2.7 How an Exception is Handled by a Calling Function

The best guideline for handling exceptions is to catch them only if there is something useful to do about them. For example, if you can recover from the error that generated an exception, or if you can at least clean up, by all means catch the exception. But if, for example, your program throws a memory-allocation exception and there is no way you can free enough memory to make the allocation attempt succeed if tried again, there is little point in catching the exception. Sometimes, as in DMTEST.CPP, the only useful response to an exception is to alert the user before the program terminates.

The following sections explain how exception handling is used in DMTEST.CPP.

The Exception Frames in the Main Function To process an exception, set up an "exception frame" around the code that can throw an exception. An exception frame specifies a region of code in which you wish to catch exceptions. It also specifies blocks of actions to take if an exception is caught. Use the **TRY**, **CATCH**, **AND_CATCH**, and **END_CATCH** macros, provided by the Microsoft Foundation Class Library.

Enclose the code that can throw an exception in a **TRY** block. For example, in DMTEST.CPP, **TRY** blocks enclose the calls that open files and create archives. The code that actually throws the exception could be deep in the function call chain. For instance, **main** calls `OpenForWriting`, which calls the **CFile** member function **Open**, which could throw an exception. Those calls, and any exceptions they throw, are enclosed in the **TRY** block in **main**. For example, in DMTEST.CPP:

```
TRY
{
    pFile = OpenForWriting( szFileName );

    WriteDataBase( pFile, pDataBase );
}
```

Enclose code to handle, or respond to, a particular exception type in a **CATCH** block. In DMTEST.CPP, code to handle a file exception is enclosed like this:

```
CATCH( CFileException, theException )
{
    // Code to handle the exception
}
```

The **CATCH** macro specifies two arguments:

- The exception type (in this case class **CFileException**)
- An exception object (`e`, the actual exception thrown)

For more information about the exception classes—**CException**, **CFileException**, **CArchiveException**, and so on—see the exception classes in the *Class Libraries Reference* and see Chapter 12.

If, as in DMTEST.CPP, a **TRY** block encloses code that could throw more than one exception type, you can use the **AND_CATCH** macro to set up additional **CATCH** blocks for the **TRY** block. In DMTEST.CPP, a **CATCH** block is used to catch file exceptions and an **AND_CATCH** block is used to catch archive exceptions. The **AND_CATCH** macro takes the same kinds of arguments as **CATCH**.

Be sure to end the **TRY/CATCH** exception frame with the **END_CATCH** macro.

The **main** function in DMTEST.CPP uses two exception frames, one for serialization and one for deserialization.

The Exception Object Passed to CFile's Open Member Function The code for `OpenForWriting` and `OpenForReading` shows the **Open** member function of class **CFile** being called. **Open** takes three arguments (although the third is optional). The third argument is an exception object, created previously, used to pass back exception information for the call. The exception object argument is a pointer to a **CFileException** object, created dynamically in the heap with the call

```
CFileException* theException = new CFileException;
```

When **Open** returns, you can examine the member variables of object `e`. You can access public member variables and member functions of class **CFileException** to examine the cause of the exception or to convert it to a standard error code—for example:

```
TRACE( "  Cause: %d\n", theException -> m_cause );
```

The **m_cause** member variable of a **CFileException** object contains a code identifying the error type. These error types are defined in an **enum** declaration in class **CFileException**. **Open** also returns a Boolean value that you can examine in the usual way to test the results of the function call.

The THROW and THROW_LAST Macros In `OpenForReading`, the call to **Open** looks like this:

```
if( !pFile -> Open( rFileName, CFile::modeRead, theException ) )
{
    delete pFile;
    TRACE( "  Threw file exception in OpenForReading\n" );
    THROW( theException );
}
```

Note that the **THROW** macro is invoked here inside a nested scope, which is delineated by the braces of the **if** block. The **if** block, executed if the **Open** call fails, cleans up by deleting the **CFile** object, `pFile`. Otherwise, it defers further processing of the error condition for handling by the caller of `OpenForReading`. However, because the code is in a nested scope, not the scope of `OpenForReading`, it's necessary to throw the exception out of that scope, where it can be available to **main** when `OpenForReading` returns.

You can also use the **THROW** macro to throw your own exceptions. These can be of predefined types, such as **CFileException**, or of types you derive from **CException** or any of its derived classes.

The **THROW_LAST** macro is used in `ReadDataBase`. In that function, a **TRY/CATCH** exception frame does some local handling of archive exceptions, then uses **THROW_LAST** to pass the exceptions on for further handling by **main**.

If deserialization of the data fails in midstream, a partially complete person list could be left over, which results in memory leakage. The local exception frame catches the **CArchiveException**, deletes any partially completed data, and then invokes the **THROW_LAST** macro to throw the same exception again so it can be caught again later.

You can use the exception-handling mechanism provided by the Microsoft Foundation Class Library to considerable advantage. For more information about exceptions, see Chapter 12.

Now that your files are complete, the next section shows you how to compile the DMTEST program.

2.6 Build the Program

To build your program, follow the instructions given in Chapter 1 of the tutorial. The required files are PERSON.H, PERSON.CPP, and DMTEST.CPP. All are available in the \C7\MFC\SAMPLE\TUTORIAL directory.

The Programmer's WorkBench (PWB) makefile for DMTEST is called DMTEST.MAK. The NMAKE makefile is called DMTEST with no extension.

DMTEST builds as a DOS program, so you must run it from the DOS command line.

2.7 Summary of the DMTEST Program

This chapter has introduced you to the Microsoft Foundation Class Library by building a data model for a name and phone number program. You have seen how to derive a data class from **CObject** and to override functions for serialization and debugging output. You have also seen how to derive a collection class to manage a list of `CPerson` objects.

This chapter also showed how to write a simple program to test the features of the data model classes. This test program showed how to use the Microsoft Foundation file and archive classes.

Because `CPerson` and `CPersonList` are data model classes, they cannot display themselves or interact with the user. `CPerson` and `CPersonList` are designed to be used by a user-interface class that can get input from a user and send output back to the user. How this interaction is implemented depends on the target operating environment of the final program. For example, the user-interface implementations will be very different for a program designed to run in a text-only environment versus a program designed to run under Microsoft Windows.

After a chapter devoted to the fundamentals of Windows programming with the Microsoft Foundation Class Library, the tutorial continues with three chapters that show you how to put a Windows user interface on the data model.

2.8 File Listings

The code shown in listings 1-3 is available on your distribution disks as PERSON.H, PERSON.CPP, and DMTEST.CPP.

Listing 1

```
// person.h : Defines the class interfaces for CPerson, CPersonList.
//
// This is a part of the Microsoft Foundation Classes C++ library.
// Copyright (C) 1991 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Microsoft Foundation Classes Reference and Microsoft
// QuickHelp documentation provided with the library.
// See these sources for detailed information regarding the
// Microsoft Foundation Classes product.

#ifndef __PERSON_H__
#define __PERSON_H__
#ifdef _DOS
    #include <afx.h>
#else
    #include <afxwin.h>
#endif
#include <afxcoll.h>


/////////////////////////////////////////////////////////////////////////////
// class CPerson:
// Represents one person in the phone database.  This class is derived from
// CObject (mostly to get access to the serialization protocol).

class CPerson : public CObject
{
    DECLARE_SERIAL( CPerson );
```

```
public:
//Construction
    // For serializable classes, declare a constructor with no arguments.
    CPerson()
    { m_modTime = CTime::GetCurrentTime(); }

    CPerson( const CPerson& a );

// For our convenience, also declare a constructor with arguments.
    CPerson( const char* pszLastName,
    const char* pszFirstName,
    const char* pszPhoneNum );

//Attributes
    // Member functions to modify the protected member variables.
    void SetLastName( const char* pszName )
    { ASSERT_VALID( this );
        ASSERT( pszName != NULL);
        m_LastName = pszName;
        m_modTime = CTime::GetCurrentTime(); }

    const CString& GetLastName() const
    { ASSERT_VALID( this );
        return m_LastName; }

void SetFirstName( const char* pszName )
    { ASSERT_VALID( this );
        ASSERT( pszName != NULL );
        m_FirstName = pszName;
        m_modTime = CTime::GetCurrentTime(); }

    const CString& GetFirstName() const
    { ASSERT_VALID( this );
        return m_FirstName; }

void SetPhoneNumber( const char* pszNumber )
    { ASSERT_VALID( this );
        ASSERT( pszNumber != NULL );
        m_PhoneNumber = pszNumber;
        m_modTime = CTime::GetCurrentTime(); }

    const CString& GetPhoneNumber() const
    { ASSERT_VALID( this );
        return m_PhoneNumber; }

const CTime GetModTime() const
    { ASSERT_VALID( this );
        return m_modTime; }

//Operations
    CPerson& operator=( const CPerson& b );
```

```
//Implementation
protected:
    // Member variables that hold data for person
    CString         m_LastName;
    CString         m_FirstName;
    CString         m_PhoneNumber;
    CTime           m_modTime;

public:
    // Override the Serialize function
    virtual void Serialize( CArchive& archive );

#ifdef _DEBUG
    // Override Dump for debugging support
    virtual void Dump( CDumpContext& dc ) const;
    virtual void AssertValid() const;
#endif
};

////////////////////////////////////////////////////////////////////////////
// class CPersonList:
// This represents a list of all persons in a phone database.  This class is
// derived from CObList, a list of pointers to CObject-type objects.

class CPersonList : public CObList
{
    DECLARE_SERIAL( CPersonList )

public:
//Construction

CPersonList()
    { m_bIsDirty = FALSE; }

// Add new functions
    CPersonList* FindPerson( const char * szTarget );

    // SetDirty/GetDirty
    // Mark the person list as "dirty" (meaning "modified").  This flag can be
    // checked later to see if the database needs to be saved.
    //
    void SetDirty( BOOL bDirty )
    { ASSERT_VALID( this );
        m_bIsDirty = bDirty; }

    BOOL GetDirty()
    { ASSERT_VALID( this );
        return m_bIsDirty; }
```

```
// Delete All will delete the Person objects as well as the pointers.
    void DeleteAll();

protected:
    BOOL  m_bIsDirty;
};

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

#endif // __PERSON_H__
```

Listing 2

```
// person.cpp : Defines the class behaviors for CPerson, CPersonList.
//
// This is a part of the Microsoft Foundation Classes C++ library.
// Copyright (C) 1991 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Microsoft Foundation Classes Reference and Microsoft
// QuickHelp documentation provided with the library.
// See these sources for detailed information regarding the
// Microsoft Foundation Classes product.

#include "person.h"
#include <string.h>

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////
//
// Call 'IMPLEMENT_SERIAL' macro for all the
//  classes declared in person.h

IMPLEMENT_SERIAL( CPerson, CObject, 0 )
IMPLEMENT_SERIAL( CPersonList, CObList, 0 )

/////////////////////////////////////////////
// Define member functions for classes
//  declared in person.h
//
```

```
//////////////////////////////////////////////
// CPerson::CPerson
// Copy Constructor for CPerson class
//
CPerson::CPerson( const CPerson& a )
{
    m_LastName = a.m_LastName;
    m_FirstName = a.m_FirstName;
    m_PhoneNumber = a.m_PhoneNumber;
    m_modTime = a.m_modTime;
}

//////////////////////////////////////////////
// CPerson::CPerson
// Memberwise Constructor for CPerson class
//
CPerson::CPerson( const char* pszLastName,
     const char* pszFirstName,
     const char* pszPhoneNum )
{
    ASSERT_VALID( this );
    m_LastName = pszLastName;
    m_FirstName = pszFirstName;
    m_PhoneNumber = pszPhoneNum;
    m_modTime = CTime::GetCurrentTime();
}

//////////////////////////////////////////////
// CPerson::operator=
// Overloaded operator= to perform assignments.
//
CPerson& CPerson::operator=( const CPerson& b )
{
    ASSERT_VALID( this );
    ASSERT_VALID( &b );
    m_LastName = b.m_LastName;
    m_FirstName = b.m_FirstName;
    m_PhoneNumber = b.m_PhoneNumber;
    m_modTime = b.m_modTime;
    return *this;
}

//////////////////////////////////////////////
// CPerson::Dump
// Write the contents of the object to a
//  diagnostic context
//
// The overloaded '<<' operator does all the hard work
//
#ifdef _DEBUG
```

```
void CPerson::Dump( CDumpContext& dc ) const
{
    ASSERT_VALID( this );
    // Call base class function first
    CObject::Dump( dc );

// Now dump data for this class
    dc   << "\n"
    << "Last Name: " << m_LastName << "\n"
    << "First Name: " << m_FirstName << "\n"
    << "Phone #: " << m_PhoneNumber << "\n"
    << "Modification date: " << m_modTime << "\n";
}

void CPerson::AssertValid() const
{
    CObject::AssertValid();
}

#endif

////////////////////////////////////////////
// CPerson::Serialize
// Read or write the contents of the object
//  to an archive
//
void CPerson::Serialize( CArchive& archive )
{
    ASSERT_VALID( this );
    // Call base class function first
    CObject::Serialize( archive );

// Now dump data for this class
    if ( archive.IsStoring() )
    {
    archive << m_LastName << m_FirstName << m_PhoneNumber
        << m_modTime;
    }
    else
    {
    archive >> m_LastName >> m_FirstName >> m_PhoneNumber
        >> m_modTime;
    }
}

////////////////////////////////////////////////////////////////////////////
// CPersonList
// We inherit most of the functionality from CObList.  These are only
// functions that we added to form our own list type.
//
```

```
//////////////////////////////////////////////
// CPersonList::FindPerson
//
CPersonList* CPersonList::FindPerson( const char * szTarget )
{
    ASSERT_VALID( this );

    CPersonList* pNewList = new CPersonList;
    CPerson* pNext = NULL;

    // Start at front of list
    POSITION pos = GetHeadPosition();

// Iterate over whole list
    while( pos != NULL )
    {
    // Get next element (note cast)
    pNext = (CPerson*)GetNext(pos);

    // Add current element to new list if it matches
    if ( _strnicmp( pNext -> GetLastName(), szTarget, strlen( szTarget ) )
         == 0 )
        pNewList -> AddTail(pNext);
    }

    if ( pNewList -> IsEmpty() )
    {
    delete pNewList;
    pNewList = NULL;
    }

    return pNewList;
}

//////////////////////////////////////////
// CPersonList::DeleteAll
// This will delete the objects in the list as the pointers.
//
void CPersonList::DeleteAll()
{
    ASSERT_VALID( this );
    POSITION pos = GetHeadPosition();

while( pos != NULL )
    {
    delete GetNext(pos);
    }
    RemoveAll();
}
```

Listing 3

```
// dmtest.cpp : A program to test the CPerson and CPersonList classes.
//
// This is a part of the Microsoft Foundation Classes C++ library.
// Copyright (C) 1991 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Microsoft Foundation Classes Reference and Microsoft
// QuickHelp documentation provided with the library.
// See these sources for detailed information regarding the
// Microsoft Foundation Classes product.

#include "person.h"

// Function prototypes.
CPersonList* MakeDataBase();
CFile* OpenForReading( const CString& rFileName );
CFile* OpenForWriting( const CString& rFileName );
CPersonList* ReadDataBase( CFile* pFile );
BOOL WriteDataBase( CFile* pFile, CPersonList* pDataBase );
void TestFindPerson( CPersonList* pDataBase );
void ListDataBase( CPersonList* db );

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

void main()
{
    const char szFileName[] = "tutorial.dat";

#ifdef _DEBUG
    // Prepare for display of search results
    const int nDumpChildren = 1;
    afxDump.SetDepth( nDumpChildren );
#endif

printf( "Create a person list and fill it with persons\n" );
    CPersonList* pDataBase = MakeDataBase();

printf( "Serialize the person list\n" );
    CFile* pFile;  // Declare a file object

TRY
    {
        // Could throw a file exception if can't open file
        pFile = OpenForWriting( szFileName );
```

```
// Could throw an archive exception if can't create
        WriteDataBase( pFile, pDataBase );
    }
    CATCH( CFileException, theException )
    {
        printf( "Unable to open file for writing\n" );
        exit( -1 );
    }
    AND_CATCH( CArchiveException, theException )
    {
        printf( "Unable to save the database\n" );
        pFile -> Close();  // Close up
        delete pFile;
        exit( -1 );
    }
    END_CATCH

    // No exceptions, so close up
    pFile -> Close();
    delete pFile;

ListDataBase( pDataBase );

    printf( "Delete the list and all its elements\n" );
    pDataBase -> DeleteAll();
    ListDataBase( pDataBase );
    delete pDataBase;

    printf( "Deserialize the data from disk into a new list\n" );
    CPersonList* pDataBase2;   // Create a new, empty list

TRY
    {
        // Could throw a file exception if can't open file
        pFile = OpenForReading( szFileName );

        // Could throw an archive exception if can't create
        pDataBase2 = ReadDataBase( pFile );
    }
    CATCH( CFileException, theException )
    {
        printf( "Unable to open file for reading database\n" );
        exit( -1 );
    }
    AND_CATCH( CArchiveException, theException )
    {
        printf( "Unable to read the database\n" );
        pFile -> Close();  // Close up before exiting
        delete pFile;
        exit( -1 );
    }
    END_CATCH
```

```
// No exceptions, so close up
    pFile -> Close();
    delete pFile;

ListDataBase( pDataBase2 );

    printf( "Test the FindPerson function\n" );
    if ( pDataBase2 != NULL )
    TestFindPerson( pDataBase2 );

printf( "Delete the list and all its elements\n" );
    pDataBase2 -> DeleteAll();
    delete pDataBase2;

TRACE( "End of program\n" );
}

// MakeDataBase - Create a database and add some persons
//
CPersonList* MakeDataBase()
{
    TRACE( "  Make a new person list on the heap\n" );
    CPersonList* pDataBase = new CPersonList;

TRACE( "  Add several new persons to the list\n" );
    CPerson* pNewPerson1 = new CPerson( "Smith", "Mary", "435-8159" );
    pDataBase -> AddHead( pNewPerson1 );

CPerson* pNewPerson2 = new CPerson( "Smith", "Al", "435-4505" );
    pDataBase -> AddHead( pNewPerson2 );

CPerson* pNewPerson3 = new CPerson( "Jones", "Steve", "344-9865" );
    pDataBase -> AddHead( pNewPerson3 );

CPerson* pNewPerson4 = new CPerson( "Hart", "Mary", "287-0987" );
    pDataBase -> AddHead( pNewPerson4 );

CPerson* pNewPerson5 = new CPerson( "Meyers", "Brian", "236-1234" );
    pDataBase -> AddHead( pNewPerson5 );

TRACE( "  Return the completed database to main\n" );
    return pDataBase;
}
```

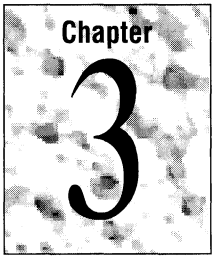
```
// OpenForReading - open a file for reading
//
CFile* OpenForReading( const CString& rFileName )
{
    CFile* pFile = new CFile;
    CFileException* theException = new CFileException;
    if ( !pFile -> Open( rFileName, CFile::modeRead, theException ) )
    {
        delete pFile;
        TRACE( "  Threw file exception in OpenForReading\n" );
        THROW( theException );
    }
    delete theException;

// Exit here if no exceptions
    return pFile;
}

// OpenForWriting - open a file for writing
//
CFile* OpenForWriting( const CString& rFileName )
{
    CFile* pFile = new CFile;
    CFileStatus status;
    UINT nAccess = CFile::modeWrite;

// GetStatus will return TRUE if file exists,
    // or FALSE if it doesn't exist
    if ( !CFile::GetStatus( rFileName, status ) )
        nAccess |= CFile::modeCreate;

    CFileException* theException = new CFileException;
    if ( !pFile -> Open( rFileName, nAccess, theException ) )
    {
        delete pFile;
        TRACE( "  Threw a file exception in OpenForWriting\n" );
        THROW( theException );
    }
    delete theException;

    // Exit here if no errors or exceptions
    TRACE( "  Opened file for writing OK\n" );
    return pFile;
}

// ReadDataBase - read data into a person list
//
CPersonList* ReadDataBase( CFile* pFile )
{
    CPersonList* pNewDataBase = NULL;
```

```
// Create an archive from pFile for reading
    CArchive archive( pFile, CArchive::load );

TRY
    {
        // and deserialize the new data base from the archive
        archive > pNewDataBase;
    }
    CATCH( CArchiveException, theException )
    {
        TRACE( "  Caught an archive exception in ReadDataBase\n" );
#ifdef _DEBUG
        theException -> Dump( afxDump );
#endif
        archive.Close();

        // If we got part of the database then delete it so we don't
        // have any Memory leaks
        if ( pNewDataBase != NULL )
        {
            pNewDataBase -> DeleteAll();
            delete pNewDataBase;
        }
        THROW_LAST();
    }
    END_CATCH

// Exit here if no errors or exceptions
    archive.Close();
    return pNewDataBase;
}

// WriteDataBase - write data from a person list to disk
//
BOOL WriteDataBase( CFile* pFile, CPersonList* pDataBase )
{
    // Create an archive from pFile for writing
    CArchive archive( pFile, CArchive::store );
```

```
    TRY
        {
            // and serialize the data base to the archive
            archive < pDataBase;
        }
        CATCH( CArchiveException, theException )
        {
            TRACE( "  Caught an archive exception in WriteDataBase\n" );
    #ifdef _DEBUG
            theException -> Dump( afxDump );
    #endif
            archive.Close();
            THROW_LAST();
        }
        END_CATCH

    // Exit here if no errors or exceptions
        archive.Close();
        return TRUE;
    }

    // TestFindPerson - test CPersonList::FindPerson
    //
    void TestFindPerson( CPersonList* pDataBase )
    {
        printf( "  Looking for the name Banipuli\n" );
        CPersonList* pFound = pDataBase -> FindPerson( "Banipuli" );
        if ( pFound -> IsEmpty() )
        {
            printf( "  No matching persons\n" );
        }
        else
        {
            printf( "  Found matching persons\n" );
    #ifdef _DEBUG
            pFound -> Dump( afxDump );
    #endif
        }
        delete pFound;
```

```
printf( "  Looking for the name Smith\n" );
    pFound = pDataBase -> FindPerson( "Smith" );
    if ( pFound -> IsEmpty() )
    {
        printf( "  No matching persons\n" );
    }
    else
    {
        printf( "  Found matching persons\n" );
#ifdef _DEBUG
        pFound -> Dump( afxDump );
#endif
    }

// Don't DeleteAll the found list since it
    //  shares CPerson objects with dataBase
    delete pFound; // Deletes only the list object
}

void ListDataBase( CPersonList* db )
{
    CPerson* pCurrent;
    POSITION pos;

if ( db -> GetCount() == 0 )
        printf( "  List is Empty\n" );
    else
    {
        printf( "  List contains:\n" );
        pos = db -> GetHeadPosition();
        while ( pos != NULL )
        {
            pCurrent = (CPerson*)db -> GetNext( pos );
            printf( "\t%s, %s\t%s\n", (const char*)pCurrent -> GetLastName(),
                    (const char*)pCurrent -> GetFirstName(),
                    (const char*)pCurrent -> GetPhoneNumber() );
        }
    }
}
```

Windows Programming with the Microsoft Foundation Classes



The previous chapter showed how to use the Microsoft Foundation Class Library to create a data model: a cooperating set of C++ objects that implements a simple database for storing `CPerson` objects.

This chapter shows how you can use the Microsoft Foundation Classes to create the key elements of a Microsoft Windows user interface. In order to use this chapter, and the next, you need to know something about Windows programming. Good sources for becoming familiar with Windows programming include the *Programming Windows, Version 3.0*, by Charles Petzold, and *Microsoft Windows SDK Guide to Programming.*

The purpose of this chapter is to show you how to use the Windows classes of the Microsoft Foundation Class Library to build applications that have a complete Windows user interface. The Microsoft Foundation Classes help you in two principal ways.

First, the Microsoft Foundation Class Library provides classes from which you can make objects that already have much of the Windows functionality you need. These classes include classes of windows, controls, dialogs, and graphics objects. The Microsoft Class Library also supplies class **CWinApp**, which provides most of the essential application-level processing your program needs. To create a Windows application, you use C++ techniques to derive both your own application class from class **CWinApp** and your own main window class from one of the class library's window classes.

Second, the Microsoft Foundation Class Library simplifies the message-handling apparatus of Windows. To process messages, you add a member function to your derived window class for each Windows message you want to handle. Then you place an entry in a "message map" for each message-processing member function.

The chapter demonstrates these techniques by taking you step by step through the development of a simple Windows application called Hello. As you go, you will see how to write the necessary C++ classes and what they do. After the examples have been developed, the chapter explains in greater detail how the Microsoft Foundation Classes work with Windows and your objects.

3.1 In This Chapter

Follow this tutorial to write a simple but complete Windows application using the Microsoft Foundation Class Library. The resulting program appears smaller and simpler than the equivalent traditional Windows program. The process of writing the program is essentially this:

1. Write an application class that is derived from one of the Microsoft Foundation Classes.
2. Write a window class that is also derived from a Microsoft Foundation class.

The rest of this section describes the example program.

The Hello Program

In the first part of this chapter, you will develop a simple Windows application using the Microsoft Foundation Classes. The example is a "Hello, World" program—the simple starter program familiar to all C programmers.

This section is an overview of what the program does and what you will be learning about the Microsoft Foundation Class Library.

What the Example Does

The example program displays the text "Hello, Windows!" centered in a window on the screen. The window has a menu bar and a set of window controls. If you choose the About command from the Help menu, a dialog box displays information about the program. You can drag the window around the screen or resize the window. If you resize the window, the text is recentered. You can minimize the window to an icon on the Windows desktop or maximize it to fill most of the screen.

Figure 3.1 shows the screen as it appears when Hello runs.

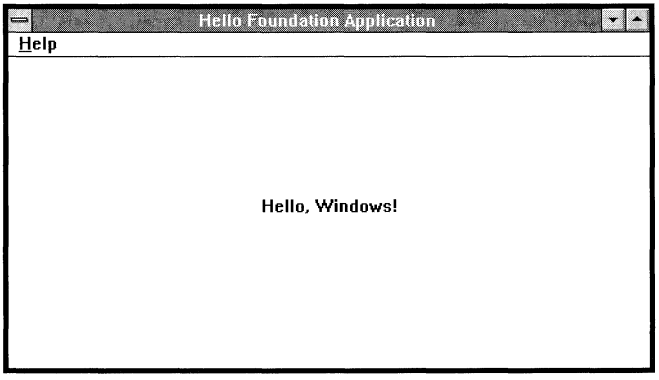


Figure 3.1 The Output of Hello

Although the example program does very little, it has a considerable amount of functionality with very little code. Furthermore, Hello makes a good template or skeleton on which to build more capable applications.

The Code for Hello

To view the complete code for Hello, see Listings 1 and 2. The code shown in these listings is available on the distribution disks in files HELLO.H and HELLO.CPP.

Microsoft Foundation Classes Used in This Chapter

This chapter demonstrates the use of five classes from the Microsoft Foundation Class Library.

CWinApp
You derive Hello's application class from this class.

CFrameWnd
You derive Hello's main window class from this class.

CRect
You pass a **CRect** object as an argument to a window-creation function. The argument specifies the rectangle in which the window is to be displayed.

CPaintDC
You construct an object of this class to create a Windows device context.

CModalDialog
You construct an object of this class to put a dialog box on the screen.

For more information about these classes, see the *Class Libraries Reference* and the discussion in this chapter.

The sections that follow take you through the components of Hello and explain how to write them, what they consist of, and in a general way how they work. A more detailed explanation of how they all work together is described in "How Hello Works" on page 110.

3.2 How to Write the Hello Program

This section gives an overview of the steps in writing Hello. As you work through the steps, you will learn what files to prepare and how to compile the program.

To write Hello with the Microsoft Foundation Classes, you must:

1. Create an application object.

   The application object represents your application and is responsible for application-level initialization. Its most important task is to construct a main window object. For more information on this step, see "Create an Application Object" on page 85.

2. Put a window on the screen.

   It takes two steps to display a window with the Microsoft Foundation Classes. First, construct a main window object. Second, during that construction, create a window for display. For more information on this step, see "Put a Window on the Screen" on page 90.

3. Arrange for communication with Windows.

   Once a window has been created, it must respond to pertinent Windows messages, such as **WM_PAINT** or **WM_COMMAND**. To arrange message processing in your window object, add a "message map" and appropriate "message-handler" functions. For more information on this step, see "Arrange for Communication with Windows" on page 94.

4. Paint the window.

   With a window showing, paint the contents of its client area. For Hello, paint the text "Hello, Windows!" For more information on this step, see "Paint the Window" on page 101.

5. Add an About dialog box.

   As a final touch, add a dialog box that displays information about the program. The user activates this "About" dialog box by choosing the About command from the Help menu. For more information on this step, see "Add an About Dialog Box" on page 105. For more information on menus and other resources, see the next step.

6. Prepare supporting files.

   Windows programs require some supporting files. Add a module-definition file, a resource file, a resource include file, and a makefile. For more information on this step, see "Prepare Supporting Files" on page 107.

7. Build the program.

   Once you have prepared all the files, compile and link the program to produce the executable file. Remember that you must run the program from Windows. For more information on this step, see "Build the Program" on page 109.

You will also use these same steps as a foundation for writing your own Windows programs with the Microsoft Foundation Classes.

The sections that follow break these steps into smaller steps. At each step, the related code is presented. After the steps, a Discussion section explains what the code does and why it was done that way. Where appropriate, additional advanced discussion in a special box elaborates upon the code and the basic discussion.

3.3 Create an Application Object

This section explains the first step in writing Hello: the creation of the application object. You'll create parts of two files in this section: HELLO.H and HELLO.CPP.

This process consists of two steps:

1. Derive an application class.
2. Write a Windows function.

► To derive an application class:

1. Create a HELLO.H interface file.

   a. Add the following preprocessor directives:

   ```
   #ifndef __HELLO_H__
   #define __HELLO_H__
   ```

   These directives are similar to the ones you used in PERSON.H in the previous chapter. They prevent multiple inclusion of any code in the HELLO.H file.

   b. Add the following class declaration to your HELLO.H file:

   ```
   class CTheApp : public CWinApp
   {
   public:
       // An override of InitInstance
       BOOL InitInstance();
   };
   ```

You derive Hello's application class, `CTheApp`, from the Microsoft Foundation class CWinApp. The name of the derived class is your choice. For Hello, it's called `CTheApp`.

Class `CTheApp` inherits member variables and member functions from its base class but overrides the **InitInstance** member function of class **CWinApp**. See the discussion under "Discussion: Hello's Application Class" on page 87.

c. Add the following preprocessor directive to the bottom of file HELLO.H:

```
#endif // __HELLO_H__
```

As you add more declarations to the file later, keep this directive as the last item in the file.

2. Create a HELLO.CPP implementation file.

a. Add the following preprocessor directives at the top:

```
#include <afxwin.h>
#include "resource.h"
#include "hello.h"
```

When you program with the Microsoft Foundation Class Library, always include the file AFXWIN.H.

b. Add the following variable declaration to HELLO.CPP below the **#include** directives:

```
CTheApp   theApp;
```

This declares a variable of type `CTheApp` in order to construct the program's one and only application object. A suitable name for this variable is `theApp`.

After the application object is constructed from this variable, its member functions are called to initialize the application and to create a window object. See the following discussion.

► To create a window object:

1. Write your overriding `InitInstance` member function, which is where you create a main window object.
2. Add the following function definition for `InitInstance` to the end of your HELLO.CPP file:

```
BOOL CTheApp::InitInstance()
{
    // Construct a window object in the heap
    m_pMainWnd = new CMainWindow();

    // Show the window
    m_pMainWnd -> ShowWindow( m_nCmdShow );
```

```
    // Paint the window
    m_pMainWnd -> UpdateWindow();

    return TRUE;
};
```

Since the HELLO.CPP file will eventually contain functions belonging to two different classes, set up a section for each class.

`InitInstance` is an appropriate place to construct a main window object, which will be responsible for putting a window on the screen. Note that putting a window on the screen is a three-step process: construct the main window object (which creates a window), call two of its member functions to make the new window visible, and cause its client area to be painted. This process is explained further in the following discussion and in "Paint the Window" on page 101.

To continue with the tutorial, see "Put a Window on the Screen" on page 90. For more information about the code you just added, see the following discussion.

Discussion: Hello's Application Class

The primary purpose of Hello's application object is to construct a main window object. This section discusses that process further.

The Application Class and the Application Object

Under the Microsoft Foundation Classes, the application class is used to construct an application object. Member functions of that object are called to perform initialization tasks and to run the program's Windows message loop. Typically, your application object creates other objects, such as window objects, that do the application's specific work.

As a class derived publicly from the Microsoft Foundation class **CWinApp**, your application object inherits the member variables and functions needed to do application-level initialization and to run the message loop. Most of the functions are called automatically when your program runs. You can override any of **CWinApp**'s member functions to customize the program, including **InitApplication**, **ExitInstance**, and **InitInstance**. Typically, however, you'll override only the **InitInstance** and perhaps **ExitInstance** member functions of class **CWinApp**. In Hello, your overriding `InitInstance` is where you put the code that creates and displays a window.

For more information about **CWinApp**, see the *Class Libraries Reference* and Chapter 13 in this book. For more information about what the application object does, see the following sections, especially "How Hello Works" on page 110.

Figure 3.2 shows the class hierarchy for the application class.

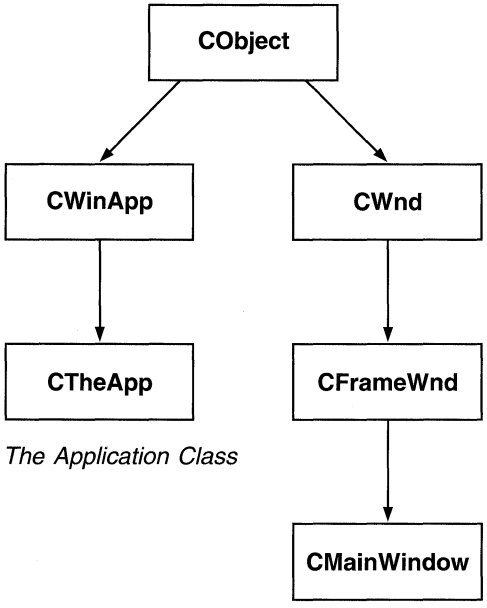


Figure 3.2 Object Class Hierarchies for Hello

InitInstance and the Window Object

When you override **InitInstance** for Hello, you use the C++ **new** operator to allocate and initialize a main window object. When the constructor for this object is invoked, it creates the window data structures that Windows needs to display a window on the screen. `InitInstance` completes the window-creation task by making the window visible and causing it to be painted.

After the `CMainWindow` constructor returns, `InitInstance` calls two member functions of the window object:

ShowWindow.
Makes the new window visible.

UpdateWindow.
Causes Windows to send a **WM_PAINT** message to the window, whose `OnPaint` member function then paints the contents of its client area. These calls are discussed further in the discussion section following "Put a Window on the Screen" on page 90.

The **m_pMainWnd** identifier used in `InitInstance` is a member variable that `CTheApp` inherits from the **CWinApp** classs. It stores a pointer to the application's main window object.

Its name reflects a convention of the Microsoft Foundation Class Library for naming member variables. All member variables begin with the "m_" prefix. The rest of the variable's name reflects its purpose and its type, in familiar Windows Hungarian variable-naming fashion. The "p" prefix in **m_pMainWnd** signifies a pointer. When writing Microsoft Foundation programs, it's useful to follow these conventions.

Once the application is initialized, it must put a window on the screen and cause its initial contents, if any, to be displayed. The following sections continue that process.

More About the Application Object

It's possible to create your window object in some other part of your program, but `InitInstance` is a logical place to do it because each running copy of your program (recall that Windows can run multiple copies at once) calls `InitInstance` once to perform initialization for that copy. Your application object is a global object, so like all C++ global objects, it is constructed early in the program initialization process. By the time Windows calls your application's **WinMain** function, a fully initialized application object exists and its member functions can be called. For more information about this sequence of events, see "How Hello Works" on page 110.

Where is **WinMain**? Experienced Windows programmers are used to writing this essential function themselves, but the Microsoft Foundation Class Library provides a globally defined **WinMain** function for you. You don't have to use this version of **WinMain**, but you'll find that it does what you want most of the time and can be customized as well. For more information about substituting your own version of **WinMain** or customizing the Microsoft Foundation's version, see "How You Can Customize Your Windows Application" on page 112 and see the cookbook.

Notice that class `CTheApp` does not declare a constructor. Because there is nothing to do in such a constructor, you can simply rely on the constructor inherited from the base class, **CWinApp**, to construct your application object. Of course, you could add a constructor to your program's application class if you had something for it to do.

Note that you can't create any windows or make any Windows function calls in the constructor because **WinMain** hasn't been called yet.

3.4 Put a Window on the Screen

This section explains the second step in writing Hello: create a window. In this section you'll add more code to the HELLO.H and HELLO.CPP files that you started in the previous section, in order to create Hello's main window class.

► **To write Hello's main window class:**

1. Derive a main window class from the Microsoft Foundation class **CFrameWnd**. Add the following declaration of class `CMainWindow` to your HELLO.H file:

```
class CMainWindow : public CFrameWnd
{
public:
    // Constructor
    //
    CMainWindow();

    // Handler function for painting messages
    //
    afx_msg void OnPaint();      // For WM_PAINT message

    // Handler function for About dialog
    //
    afx_msg void OnAbout();

    // Macro to declare a message map
    //
    DECLARE_MESSAGE_MAP()
};
```

   Put this declaration just below the preprocessor directives at the top of the file.

   The **afx_msg** modifier is similar to the **virtual** keyword of C++. Member functions prefixed with **afx_msg** are prototyped in class **CWnd** and can be overridden in derived window classes. They tie into the message map associated with the Microsoft Foundation window class.

   At this point, your HELLO.H file is complete. You can check it against Listing 1.

2. Add the following `CMainWindow` constructor definition to your HELLO.CPP file:

```
CMainWindow::CMainWindow()
{
    LoadAccelTable( "MainAccelTable" );
    Create( NULL,, "Hello Foundation Application",
```

```
        WS_OVERLAPPEDWINDOW, rectDefault,
                NULL, "MainMenu" );
}
```

Put the constructor in a section of the file for `CMainWindow` member functions. Keep this section separate from the section that contains the `InitInstance` definition. The HELLO.CPP file in Listing 2 puts all `CMainWindow` code above all `CTheApp` code.

To continue the tutorial, see "Arrange for Communication with Windows" on page 94. For more information about the code you just added, see the discussion below.

Discussion: Creating Windows

The following sections explain how Hello puts a window on the screen.

Hello's Main Window Class

Previously, you saw the process of constructing a main window object in the `InitInstance` member function of the application object ("Create an Application Object" on page 85). The next step is to have that window object create a window on the screen—a window with a caption bar, a frame, and various Windows controls.

Part of this work is done in the application object's `InitInstance` function. After creating the window object with **new**, `InitInstance` calls two of the window object's functions to display it. The rest of the work is done by the window object itself.

The `CMainWindow` class is derived publicly from **CFrameWnd**. The Microsoft Foundation Class Library provides several other window classes from which you might choose to derive your own window classes, depending on your needs. For information about the choices, see the *Class Libraries Reference*. **CFrameWnd** is commonly chosen to represent an application's main window. Figure 3.2, on page 88, shows the class hierarchy for Hello's main window class.

`CMainWindow` has the following components:

- A constructor
- Two message-handler member functions
- A macro invocation

The CMainWindow Constructor When `InitInstance` constructs a main window object, the `CMainWindow` constructor is invoked. You saw the code for the constructor of `CMainWindow` above. There are many things you could do in your

constructor. Hello uses the constructor to create a window for display. The window is created by a call to the **Create** member function that `CMainWindow` inherits from its base class, **CFrameWnd**, and which **CFrameWnd** inherits in turn from its own base class, **CWnd**.

The call to **Create** creates the window but doesn't make it visible. **Create** takes the following arguments:

- A window class name.

  Hello's first argument is **NULL**. If you pass **NULL** for this argument, the Microsoft Foundation Classes select an appropriate window class and prepare its data structures. Traditional Windows programmers are accustomed to registering their own window classes with Windows. With the Microsoft Foundation, you can still register classes if you need to, but the most commonly used window classes are preregistered, and the Microsoft Foundation chooses the most appropriate one. For more information about registering window classes, see "How Hello Works." Note that a "window class" in traditional Windows is not the same thing as a C++ window class, such as **CFrameWnd**, in the Microsoft Foundation Class Library.

- A string specifying caption text for the window.

  Hello's second argument is a null-terminated string containing the text to be displayed as a caption in the window's title bar.

- A window style.

  Hello's third argument is a constant specifying the window style as **WS_OVERLAPPEDWINDOW**. The **WS_OVERLAPPEDWINDOW** style specifies an overlapping window with a caption, a thick-frame border, a system menu, and minimize and maximize boxes. You'll typically pass **WS_OVERLAPPEDWINDOW** for a main program window, but you can pass any value for this argument that you pass for the corresponding argument in the Windows **CreateWindow** function. For more information about window styles, see the *Windows SDK Reference.*

- A rectangle specifying where to display the window on the screen.

  Hello's fourth argument is **rectDefault**. If you pass this predefined value instead of your own **CRect** object, the Microsoft Foundation Classes use a default rectangle specified by Windows. The default position and dimensions of the window depend on the system and on how many other applications have been started. Class **CRect** is a Microsoft Foundation class designed to represent a two-dimensional rectangle similar to the Windows **RECT** data type. The class provides member functions to manipulate rectangles in a variety of ways. For more information about **CRect**, see the *Class Libraries Reference.*

- A pointer to the parent window (of type **CWnd**), if any.

  Hello's fifth argument is **NULL**. Because this is a main window, not a child window, it has no parent window.

- The name of the window's menu resource.

  Hello's sixth argument is "MainMenu," a string that specifies the menu template name used in file HELLO.RC to define Hello's menu.

The `CMainWindow` constructor also calls the LoadAccelTable member function of class **CWnd**. Class `CMainWindow` inherits this member function from **CWnd**. The call to **LoadAccelTable** loads a Windows accelerator table, which defines the shortcut keys (also known as accelerator keys) that the program can respond to. The Hello program defines one shortcut key: the `F1` key calls up the About dialog box. For more about resources such as accelerator tables, see the *Windows SDK Guide to Programming*.

CMainWindow's Member Functions and Message-Map Macro Besides its constructor, class `CMainWindow` overrides two message-handler member functions, **OnAbout** and **OnPaint**, and invokes the **DECLARE_MESSAGE_MAP** macro. These functions and the macro are discussed further in "Arrange for Communication with Windows" on page 94, in "Paint the Window" on page 101, and in "Add an About Dialog Box" on page 105.

How InitInstance Displays the Window

The main window constructor is invoked when Hello's `InitInstance` member function allocates a `CMainWindow` object with **new**. Once the constructor completes, control returns to `InitInstance`. At this point, the window is ready for display but still is not visible on the screen.

To display the window, `InitInstance` calls the newly created window object's **ShowWindow** member function. You'll recall that class `CMainWindow` didn't declare a **ShowWindow** function. Instead, it inherits **ShowWindow** from **CFrameWnd**.

ShowWindow makes the window visible, but nothing has yet been painted in the new window's client area. To accomplish that, `InitInstance` calls the main window object's **UpdateWindow** member function. `CMainWindow` inherits this function, like **ShowWindow**, from **CFrameWnd**. **UpdateWindow** causes Windows to send a **WM_PAINT** message to the window. When the window responds to that message, it paints the text "Hello, Windows!" How the window responds to **WM_PAINT** is explained in "Paint the Window" on page 101. Figure 3.3 shows the process schematically.

After calling the **ShowWindow** and **UpdateWindow** member functions, `InitInstance` is finished. At that point, Hello begins its message loop and is ready to receive messages from Windows.

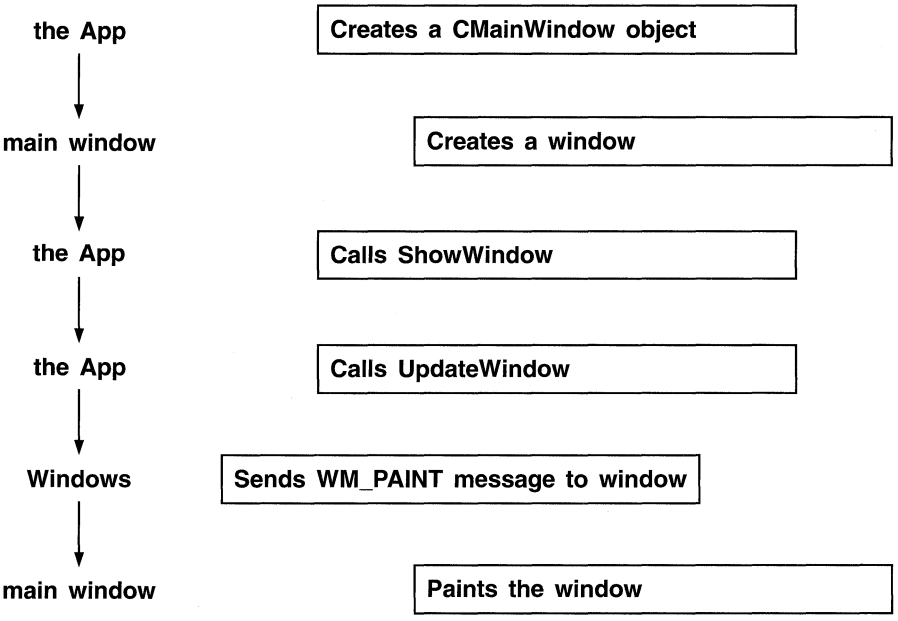


Figure 3.3 Window Display

3.5 Arrange for Communication with Windows

This section explains the third step in writing Hello: set up the mechanism by which Hello responds to Windows messages.

► **To set up window message hooks for the main window:**

1. Add the following message map for the new window class to your HELLO.CPP file:

```
BEGIN_MESSAGE_MAP(CMainWindow, CFrameWnd)
    ON_WM_PAINT()
    ON_COMMAND(IDM_ABOUT, OnAbout)
END_MESSAGE_MAP()
```

You can put the message map with your code for the main window class. The message map connects specific Windows messages with member functions of your window class provided as message handlers. The message map is really a part of your window class, so writing it is part of writing the class. It's important to put the message map in the HELLO.CPP file rather than the HELLO.H file to ensure that the macros comprising the map are not invoked more than once. The macros create code and therefore allocate memory. Thus they must not be included in more than one module.

You already added the **DECLARE_MESSAGE_MAP** macro to your `CMainWindow` class declaration. Any window class you write requires this macro as part of its declaration. The message map in HELLO.CPP is the implementation corresponding to the message-map declaration.

2. Add message-handler functions for the Windows messages you need to process.

   For Hello, you'll add two message-handler member functions to the `CMainWindow` class. The code for this step is given later in the tutorial, in "Paint the Window" on page 101 and "Add an About Dialog Box" on page 105.

   The Microsoft Foundation Class Library provides default behavior for all messages, but you must provide `CMainWindow` member functions to override the default behavior and handle the menu command for the About dialog box and to handle the **WM_PAINT** message, which signals your window to paint its contents.

To continue the tutorial, see "Paint the Window" on page 101. For more information about the code you just added, see the following discussion.

Discussion: Communication with Windows

The Microsoft Foundation Class Library provides a communications mechanism for connecting Windows messages to the message-handler member functions of your windows classes. This section describes the mechanism, emphasizing the message map.

This discussion does not instruct you to add any code to your files.

Message Maps

You write a Microsoft Foundation message map using macros from a set of predefined macros. The following sections discuss the parts of a message map, where the macros go, and how the message map connects your handlers to the Windows messages they handle.

To write the message map, use the **BEGIN_MESSAGE_MAP** and **END_MESSAGE_MAP** macros. Between them, add an entry for each Windows message your main window (or dialog) will handle. The Microsoft Foundation Class Library predefines a set of macros to use for the entries, as discussed in the following sections.

You write the message-handler functions corresponding to your message map entries. The Microsoft Foundation Class Library specifies some rules for naming your message handlers and for specifying their argument signatures. The rules are discussed in the following sections.

What the Message Map Is For Windows programs are spoken of as "event driven." The user interacts with the windows, menus, and controls in the Windows user interface of your program. User-generated events, such as mouse clicks and keystrokes, place "messages," each based on the **MSG** structure defined for Windows, in your application's message queue. Your application uses a message loop to get messages from the queue and send the messages to the appropriate window for handling.

A Windows program must have a mechanism for selecting the appropriate code to respond to Windows messages, such as **WM_CREATE**, **WM_PAINT**, or **WM_COMMAND**.

Traditional Windows programs define a window procedure for each registered "window class." The window procedure, often called **WndProc,** typically contains a switch statement which uses the message information passed to the window procedure to select appropriate code to respond to the message.

The Microsoft Foundation Classes provide a mechanism for the same purpose called a message map. These message maps are similar to C++ "v-tables" but are more space efficient. The message map defines linkages between particular Windows messages and corresponding member functions of the window object.

Message Map Macros

Hello uses a message map with two entries, as described previously. **ON_COMMAND**, which is used to respond to commands such as **WM_COMMAND** messages, is only one of the macros available for associating messages with functions. The Microsoft Foundation Class Library provides many such macros, including, for example, **ON_WM_PAINT**, **ON_WM_CREATE**, and **ON_WM_SIZE**. For more information about the macros associated with message map entries, see the cookbook and the *Class Libraries Reference.* The **ON_WM_PAINT** macro is discussed further, in "Paint the Window" on page 101.

For instance, the example code for Hello defines a message map with an entry providing a connection between the constant **IDM_ABOUT** and the `OnAbout` member function. **IDM_ABOUT** represents the menu ID of the About menu command, as defined in the resource file associated with Hello.

When a window procedure receives a **WM_COMMAND** message for a menu command, the window procedure's *wParam* parameter contains a menu ID to identify which menu command was chosen. Hello's message map uses the **ON_COMMAND** macro to associate **IDM_ABOUT** with the `OnAbout` member function.

Logically, the message map is part of the main window class, so a good place to put it is with the `CMainWindow` code.

What Hello's Message Map Does The macros create entries in a table. The message-processing mechanisms in the Microsoft Foundation Classes use the table to locate and call the function associated with a message. Figure 3.4 shows the process of routing messages to handlers via message maps.

What happens if there is no entry in the message map of your window class for a given message? Each window class in the class hierarchy (**CWnd**, **CFrameWnd**, and `CMainWindow` in the example) has its own message map. The message processing mechanism can move up the hierarchy of message maps in search of a message-to-function mapping.

Because default handlers are declared in class **CWnd**, a default handler will be found at the **CWnd** level of the hierarchy if nowhere else. If the default behavior for a given handler has been overridden at some level, the mechanism finds a function to execute and calls it. Because of this structure, it's a good idea to call the base class's version of your handler function, much as you call **DefWindowProc** in traditional Windows programming.

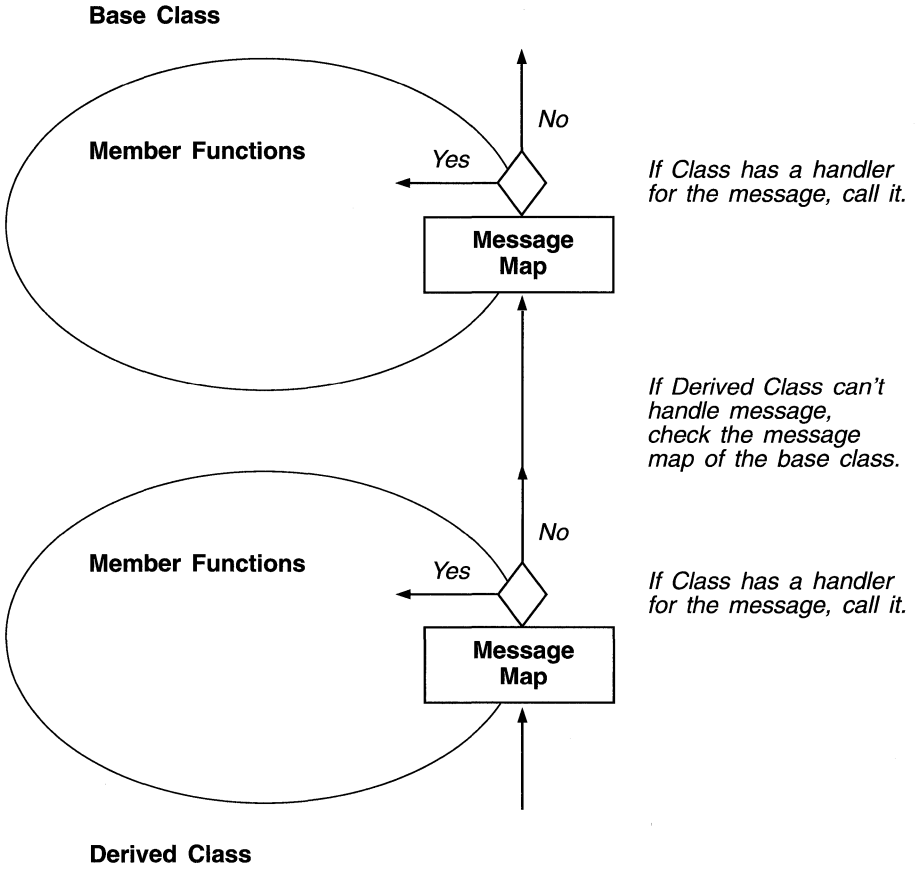


Figure 3.4 How Message Maps Route Messages to Handlers

Rules for Message-Handler Functions If you use the message-handling apparatus of the Microsoft Foundation Classes in your Windows program, there are some guidelines and requirements for the names and parameter signatures of your message handler functions, such as `OnAbout` and `OnPaint`. There are three main categories of messages that a window receives:

- **WM_COMMAND** messages generated by user menu selections or menu-accelerator keys.
- Notification messages from child windows, such as a message from a button to its parent window indicating that the button has been clicked: **BN_CLICKED**.

A notification message is a **WM_COMMAND** message in which the *wParam* parameter contains the control ID for the child window and the *lParam* parameter contains a notification control code in the high-order word and the control handle in the low-order word.

- Other **WM_XXX** messages, such as **WM_PAINT** or **WM_SIZE**, generated by the system or by user input.

Note You can always get to the raw Windows **MSG** structure by calling the **GetCurrentMsg** function.

Your handler functions for menu messages and child window notification messages (the first two preceding categories) take no arguments and return no value. No arguments are needed because your main window object stores the message information needed to process the message. The two categories of messages use the information that Windows passes with the message differently, but the conventions for naming your handler functions are the same.

Your handler functions for messages in the third category above, however, do require various arguments, depending on the message, and can return a value. Because the Windows messages are all standard, the names and argument signatures required for these handler functions are predefined by the Microsoft Foundation Classes.

For example, your handler for the **WM_PAINT** message must be named `OnPaint`. It takes no arguments and returns no value.

Other message handlers do require arguments, and some return values. The `OnSize` handler function for a **WM_SIZE** message, for example, requires two arguments, one of type **UINT** and one of type **CPoint**. `OnSize` returns no value. An `OnEraseBkgnd` handler for the **WM_ERASEBKGND** message requires one argument, a pointer to a device context class object, and returns a **BOOL**.

To determine the correct argument signature and return type for your message-handler functions, see the *Class Libraries Reference*. The signatures are also listed as function prototypes in the **CWnd** class declaration in file AFXWIN.H. Each prototype is preceded by the **afx_msg** identifier. You should copy and paste these prototypes into your own code as needed. See the additional information in the box "Default Message Handlers" on page 100.

For more information about message maps, see the cookbook. For more examples, see the next chapter.

The next several sections examine how to write the message handler member functions for Hello's main window class. These functions correspond to the two entries in Hello's message map above. They handle:

- Responding to an application request to paint or repaint a window's contents.
- Displaying an About dialog box.

Default Message Handlers

The Microsoft Foundation Classes supply default message handlers, as function prototypes, for all standard Windows messages. All of the Microsoft Foundation's window classes also have their own message maps. Because of this, the Microsoft Foundation provides good default message handling equivalent to the use of **DefWindowProc** and its relatives in traditional Windows programming.

The default message handler functions work through the message maps and are not actually virtual functions, although their behavior appears to be like that of virtual functions. To override the default behavior for any Windows message, you must provide your own handler function for the message in your window class, and you must make an entry in your message map for the function. The function must follow the prototype for its message, given in class **CWnd**. For example, in **CWnd**, the prototype for **OnPaint** is

```
afx_msg void OnPaint()
```

Your function declaration for `CMainWindow's OnPaint` member function looks like this:

```
afx_msg void OnPaint();
```

If you do override the default handler by supplying your own, it is a good practice to call your base class's version of the handler. You can do this at any appropriate point in the body of your handler. For example, in a function that draws a graphic, you might call the base class version to get the default processing, then add your own processing. In another case, you might do your processing first, then call the base class version to add the default processing. Calling the base class version of your function is optional. You will certainly want to do it for any handler that augments rather than replaces the behavior of the base class's version of the handler.

3.6 Paint the Window

This section explains the fourth step in writing Hello: paint text in the window.

► **To paint text in Hello's window:**

1. Add the `OnPaint` handler function for processing **WM_PAINT** messages to HELLO.CPP:

```
void CMainWindow::OnPaint()
{
    CString s = "Hello, Windows!";
    CPaintDC dc( this );
    CRect rect;

    GetClientRect( rect );
    dc.SetTextAlign( TA_BASELINE | TA_CENTER );
    dc.SetBkMode( TRANSPARENT );
    dc.TextOut( rect.right / 2, rect.bottom / 2, s, s.GetLength() );
}
```

   Put the `OnPaint` member function in your HELLO.CPP file with other `CMainWindow` member functions.

2. You already added the **ON_WM_PAINT** macro to Hello's message map. Its line in the message map looks like this:

```
ON_WM_PAINT()
```

   Requirements for naming the `OnPaint` member function and for the macro name are discussed below.

To continue the tutorial, see "Add an About Dialog Box" on page 105. For more information on the code you just added, see the following discussion.

Discussion: Painting Text

What happens inside your `OnPaint` member function? This discussion explains the statements in the `OnPaint` member function and the requirements for naming the function and for making its message map entry.

The OnPaint Name

OnPaint, as a message-handler function, is predefined in the Microsoft Foundation class **CWnd**, where its prototype is given. **OnPaint** is designed to handle the **WM_PAINT** message, and **WM_PAINT** is one of the messages requiring a

specific parameter signature. The **OnPaint** function happens to take no arguments and to return no value. Thus, its declaration in class `CMainWindow` looks like this:

```
afx_msg void OnPaint();
```

Its corresponding macro entry, **ON_WM_PAINT**, is also predefined.

Inside OnPaint

The code in Hello's `OnPaint` member function has three fundamental components. To paint the text, you do these three things:

1. Create a device context.

   `OnPaint` constructs a **CPaintDC** object. The class name stands for class "Paint Device Context."

2. Determine the area in which to paint.

   `OnPaint` uses the window object's **GetClientRect** member function to get a rectangle corresponding to the client area of the window. `CMainWindow` inherits **GetClientRect**.

3. Paint the text.

   `OnPaint` uses three **CPaintDC** member functions to align and paint the text.

In the Windows graphical user interface, it is common to speak of anything you display in a window, even text, as being "drawn" or "painted." In Hello, the text "Hello, Windows!" is painted in the window, through the use of graphics functions.

Your program is responsible for painting its window contents when requested by Windows. Typically, a window is painted when the program starts, when the window is first displayed, and when the window changes size or is uncovered after being covered by another window, when new drawing takes place. Windows sends **WM_PAINT** when anything happens that requires updating the window.

The `OnPaint` member function handles the **WM_PAINT** message.

When the contents of your window's client area change or when the client area must be redrawn ("updated"), Windows sends a **WM_PAINT** message to the window. What your `OnPaint` message-handler function does in response to that message depends on your needs. No matter what your program displays in its windows, the `OnPaint` handler is the place to do the drawing.

What OnPaint Does

Except for its use of a "device context object," the code for `OnPaint` is straightforward Windows programming. The syntax looks cleaner, but the calls made should be familiar to Windows programmers. Figure 3.5 shows the sequence of steps in Hello's `OnPaint` member function.

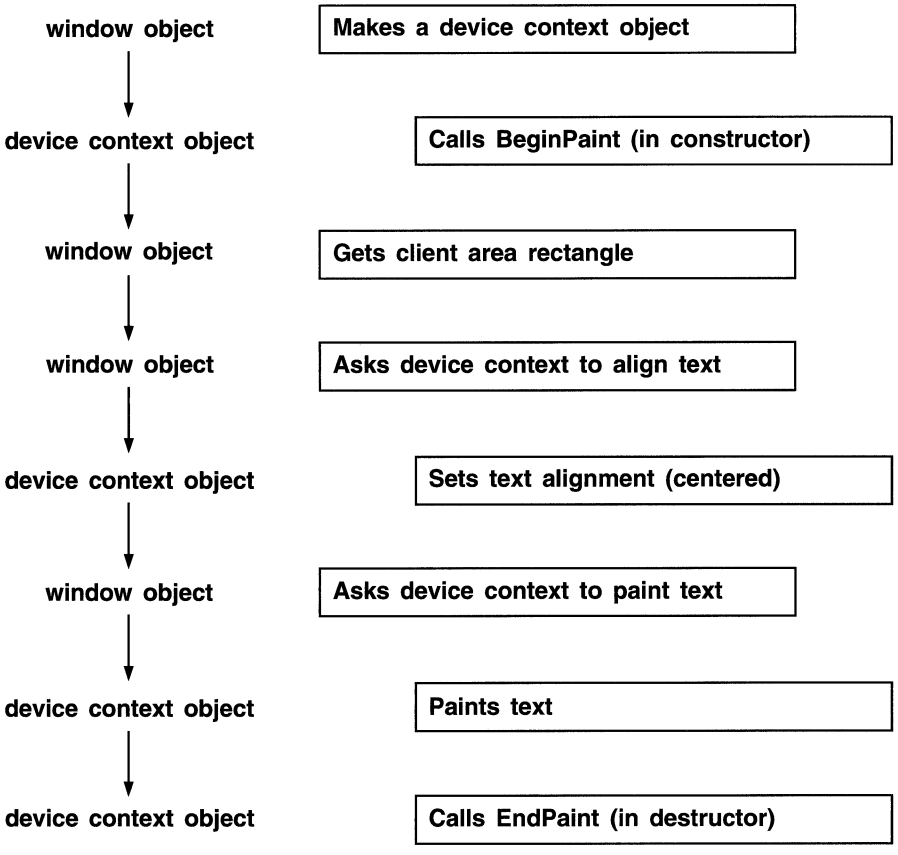


Figure 3.5 Sequence of Events in Hello's OnPaint Function

The `OnPaint` member function of class `CMainWindow` uses three local variables:

- A **CString** object to contain the text to paint.
- A device context object of class **CPaintDC**.
- A **CRect** object.

The device context object, *dc*, is the Microsoft Foundation Class Library way to set up a Windows device context for painting on the screen. When the **CPaintDC** object is constructed on the stack frame, it gets a pointer to the window object that owns it, passed with the C++ **this** keyword. The **CPaintDC** object needs this information to associate the device context with the window. During its initialization, the **CPaintDC** object calls the Windows **BeginPaint** function. When the **CPaintDC** object is destroyed as OnPaint exits, its destructor calls the Windows **EndPaint** function.

OnPaint calls the main window object's **GetClientRect** member function, inherited from **CWnd** through **CFrameWnd**. This function returns the coordinates of the window's client area in the *rect* argument.

Armed with the client area information, OnPaint then calls two member functions of the device context object to align and paint the text. The call to the device context's **SetTextAlign** member function (of class **CPaintDC**) specifies that the coordinates given are to be considered the center of the text. Because Hello's `OnPaint` member function gives the center of the window, the text is centered in the window.

The call to the device context's **SetBkMode**, with an argument of **TRANSPARENT**, specifies that the window background remains as it is rather than being filled with the current background color before painting. If you compile and run the Hello program, you can see that if the window changes size, the text is redrawn so it is always centered.

The call to the device context's **TextOut** member function paints the text in the window. To center the starting point of the text in the client area, the code divides the right-side and bottom coordinates by 2, defining a point at the horizontal and vertical center of the client area. The arguments passed previously to **SetTextAlign** cause the output to use this point as the baseline for drawing, and the text is also centered horizontally on the point.

All **CWnd** and **CPaintDC** member functions called in `OnPaint` closely parallel functions of the same name in the Windows API. You could, of course, simply use the Windows calls in your `OnPaint` member function, but using the Microsoft Foundation Classes adds simplicity and flexibility.

The primary functionality of Hello is now in place. But Hello will have more of the Windows look if you add an About box. The next section shows how.

3.7 Add an About Dialog Box

This section explains the fifth step in writing Hello: add an About dialog box.

► **To add an About dialog box:**

1. Add the `OnAbout` member function to handle **WM_COMMAND** messages. Put the following function definition in your HELLO.CPP file with other `CMainWindow` member function definitions:

```
void CMainWindow::OnAbout()
{
    CModalDialog about( "AboutBox", this );
    about.DoModal();
}
```

   As a message handler, `OnAbout` is similar in most respects to the `OnPaint` member function.

2. You already added the **ON_COMMAND** macro for the IDM_ABOUT menu ID to the message map. Its line in the message map looks like this:

```
ON_COMMAND( IDM_ABOUT, OnAbout )
```

   The dialog template for Hello's About dialog box was created with a dialog editor. The template is in file HELLO.DLG, which is referenced in the program's resource script file, HELLO.RC, with a line like this:

```
rcinclude hello.dlg
```

   This line points the resource compiler to the file containing the dialog template. Note that in the dialog template created by the dialog editor the template name "ABOUTBOX" is uppercase. However, the name is not case sensitive, as the code for `OnAbout` shows.

   At this point, your HELLO.H and HELLO.CPP files are complete if you have followed all of the directions in this tutorial. You can check them against Listings 1 and 2, respectively.

   Requirements for naming the handler function and the macro were discussed in "Rules for Message-Handler Functions" on page 98. To continue the tutorial, see "Prepare Supporting Files" on page 107.

Discussion: The About Dialog Box

A program's About dialog box displays information about the program, usually including the program's name and a copyright notice.

Figure 3.6 shows what Hello's About dialog box looks like.

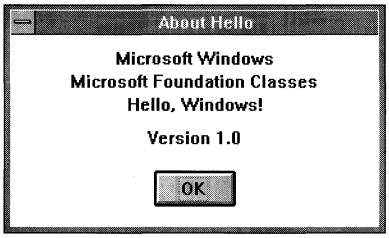


Figure 3.6 Hello's About Dialog Box

To put a dialog box on the screen, `OnAbout` uses an object of the **CModalDialog** class. **CModalDialog** is a Microsoft Foundation class that displays and operates a modal dialog box that contains controls such as buttons and text fields. **CModalDialog** provides a member function, **DoModal**, that operates the dialog box. As long as **DoModal** has control, the user must deal with the dialog box. The user can use any of the controls contained in the dialog box. The user can either dismiss the dialog box, usually with a **CANCEL** button, or accept any changes made, usually with an **OK** button. Hello's dialog box has only one control, an **OK** button.

What OnAbout Does

The `OnAbout` member function constructs a **CModalDialog** object called "about." The first argument to the constructor is a string that identifies a resource in an associated resource file. The second argument is **this**, a C++ keyword that contains a pointer to the current object. Because this code is inside a `CMainWindow` member function, **this** refers to the `CMainWindow` object. Passing this pointer to the dialog enables it to identify its parent window.

When to Use CModalDialog

The Microsoft Foundation's dialog box classes are derived from class **CWnd**, so they inherit **CWnd**'s members and then add new functionality specific to dialogs.

The `OnAbout` member function of `CMainWindow` uses a technique suitable for very simple dialog boxes. The function simply constructs an object of class **CModalDialog**. (This dialog is so simple that you could implement it with a message box.) If you need a more complex dialog box, you must derive your own dialog class from **CModalDialog** (or from **CDialog**) and construct an object of the derived class. For more information on creating dialog boxes, see Chapter 5 of the tutorial and Chapter 15 of the cookbook.

Once the **CModalDialog** object has been constructed, `OnAbout` calls its **DoModal** member function to display the dialog and interact with the user. In Hello, the About dialog box closes when the user clicks the **OK** button.

Summary of the Hello Program's Code

At this point, you have derived the two necessary classes from Microsoft Foundation Classes. You have written the code for their member functions. And you have written a message map to connect Windows messages to your main window class's handler functions.

After you set up some additional files common to Windows programming, as detailed in the next section, you can compile Hello and then run it. If you prefer, you can compile the version of the program already provided on the distribution disks.

3.8 Prepare Supporting Files

Besides the two source code files, HELLO.H and HELLO.CPP, you will need several additional files in order to compile Hello. The files will look like those shown in Listings 3 - 5.

► **To prepare these files, do the following:**

1. Create a module-definition file, with the .DEF extension.

   All Windows programs require a module-definition file, with the .DEF extension. The module definition file for Hello is shown in Listing 3 and is available on the distribution disks as HELLO.DEF.

2. Create a resource file with an .RC extension.

   If you use custom resources, such as icons, you need a resource script file, with the .RC extension, to define the resources. The resource file for Hello is shown in listing 4 and is available on the distribution disks as HELLO.RC. For your own icons, you also need files created with a resource editor, such as SDKPaint. The icon resource file HELLO.ICO is furnished on the distribution disks also.

3. Create a resource include file with an .H extension.

   Hello uses an .H file to define resource ID numbers for the resources listed in its resource file. The resource include file for Hello is shown in Listing 4 and is available on the distribution disks as RESOURCE.H.

Discussion: The Supporting Files

The supporting files for Hello are given in Listings 3, 4, and 5. Listing 3 is the module-definition file, required for all Windows programs. Listing 4 is the resource script file, required of Windows programs that define their own resources, such as menus, dialogs, icons, and accelerators. Listing 5 is the resource include file, used to declare ID numbers associated with the resources. These files are provided on the distribution disks for your use, or you can create your own files based on the listings. The next sections explain each file.

The Module-Definition File

The module-definition file for Hello appears in Listing 3.

This file is typical for a Windows program. For more information on module-definition files, see the *Windows SDK Guide to Programming* and the *Windows SDK Reference, Volume 2*.

For programs that use the Microsoft Foundation Classes, always use **FIXED** for the **CODE** and **DATA** segments. C++ code can't be moved by the real-mode memory manager, so you must specify **FIXED**. Under protected mode, the protected-mode page manager will automatically take care of swapping the **FIXED** code in and out of memory.

You can simply use the HELLO.DEF file supplied on the distribution disks and modify it as needed for future programs. The values specified here are fairly standard for Windows programs, but you can adjust the **HEAPSIZE** or **STACKSIZE** values, for example, to suit your needs.

The Resource Script File

The resource script for Hello is shown in Listing 4.

This file specifies four resources: a custom icon, a menu, an accelerator table, and a dialog resource file created with a dialog editor.

The icon entry specifies an ID number, **AFX_IDI_STD_FRAME**, with which the icon resource can be loaded. The ID number is defined in the Microsoft Foundation Class Library. The icon entry uses an **ICON** statement to associate the ID number with a file that contains the icon data. If you supply an icon resource, you also must supply the file that contains that resource: for example, HELLO.ICO.

The menu entry defines a pop-up-style menu labeled "Help" with one menu item, labeled "About Hello...F1" The About menu command is associated with the ID number **IDM_ABOUT**, defined in the RESOURCE.H file (see the next section).

The accelerator table entry defines an accelerator table resource to associate keys with menu items. For Hello, the F1 function key displays the About dialog box by causing Windows to generate a **WM_COMMAND** message for the menu item. You saw previously how the `OnAbout` function handles that message.

The dialog resource file, which was created with a dialog editor, refers to a .DLG file containing a resource template. The template defines a dialog box resource to be used for the About dialog box. The dialog box has the caption, or window title, "About Hello" and contains several lines of text identifying the program. Hello's About dialog template is in the file HELLO.DLG on the distribution disks.

You can simply use the resource script provided on the distribution disks in file HELLO.RC. If you do not want to provide custom icons, you can remove the appropriate lines from the resource script. You will also need to remove them from the makefile.

The Resource Include File

Hello uses a file with the .H extension to define its application-specific resource ID numbers. The file appears in Listing 5. Hello's file defines only one resource ID.

This definition is used to match the About dialog resource in the .RC file with the **CModalDialog** object that uses the resource to create the dialog.

You can simply use RESOURCE.H on the distribution disks.

3.9 Build the Program

To build your program, follow the instructions given in Chapter 1 of the tutorial. The required files are HELLO.H, HELLO.CPP, HELLO.DEF, HELLO.RC, HELLO.ICO, HELLO.DLG, and RESOURCE.H. All are available in the MFC\SAMPLE\HELLO directory in your Microsoft C/C++ installation.

The Programmer's WorkBench (PWB) makefile for HELLO is called HELLO.MAK. The NMAKE makefile is called MAKEFILE with no extension.

HELLO builds as a Windows application, so you must run it from Microsoft Windows.

3.10 How Hello Works

Without a visible function called "main," as in C, or "**WinMain**," as in Windows, it is hard to see how Hello does anything. Where is its entry point? What is its sequence of execution?

This section shows Hello's sequence of execution at two levels. First, the sequence is described at a general level to give you an overview. Second, the sequence is described again at a more detailed level. You can skip either of these sections, depending on your knowledge of C++ and Windows.

A General View

When Hello runs, the C++ code creates the `theApp` object, an object of your application class, for example `CTheApp`.

After the application object is constructed, Windows calls the **WinMain** function. **WinMain** performs some initialization chores. Then it calls your `InitInstance`. Finally, it starts the message loop.

Typically, as in Hello, you'll use the call to your application object's overriding `InitInstance` member function to construct your main window. You accomplish this by constructing a main window object of your main window class, such as `CMainWindow`. Then you call three member functions of class **CWnd**, such as **CreateWindow**, **UpdateWindow**, and **ShowWindow**. **CreateWindow** is called from the constructor of your main window object. The other functions are called from `InitInstance`.

The call to **CreateWindow** creates the Windows data structures for a window. The call to **UpdateWindow** causes Windows to send a **WM_PAINT** message to the window procedure associated with your window as soon as the message loop starts. The call to **ShowWindow** causes the window to appear on the screen.

When the message loop starts, the window object's message map is used to call its OnPaint member function, which paints the string "Hello, Windows!" in the window. Then the message loop continues until a **WM_QUIT** message causes the loop to end and the program to terminate.

Figure 3.7 shows the sequence of events when a Microsoft Foundation Windows application runs.

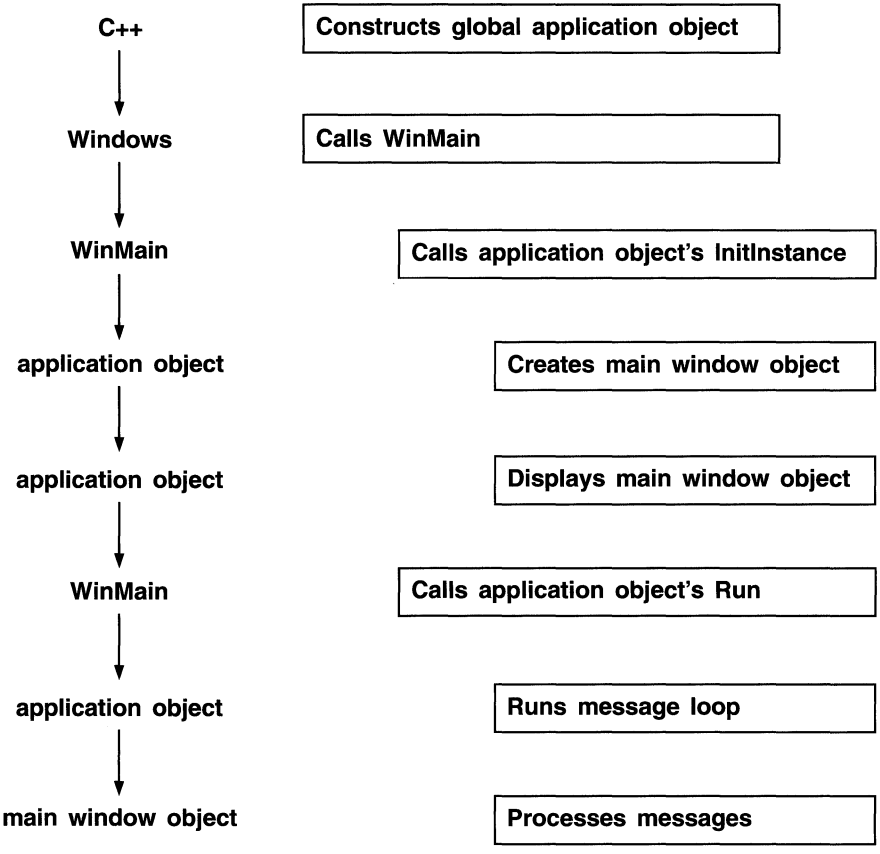


Figure 3.7 Sequence of Events When a Foundation Windows Application Runs

A More Detailed View

When Hello executes, two key actions occur. First, C++ static and global objects, such as Hello's `theApp` global application object, are constructed. A global variable in the class library is set to point to the application object. Second, when Windows initialization completes, Windows calls the **WinMain** function. The Microsoft Foundation Classes supply a **WinMain** function that does the traditional tasks of any Windows **WinMain** function. The function is responsible for application and instance initialization and for running the application's message loop.

The WinMain Function

As part of its own initialization code, **WinMain** checks the *hPrevInstance* argument passed to it by Windows. If *hPrevInstance* is **NULL**, **WinMain** calls the application object's `InitApplication` member function to perform first-time initialization.

Then, regardless of the value of *hPrevInstance*, **WinMain** calls the application object's `InitInstance` member function to perform extra initialization for this particular program instance. In Hello, the version of `InitInstance` that **WinMain** calls is the version defined in class `CTheApp` as an override of **InitInstance**.

After initialization, **WinMain** calls the application object's **Run** member function to begin the message loop. As Windows interacts with the user, it detects mouse clicks, keystrokes, and other events. It places messages corresponding to these events in an application message queue. The message loop retrieves messages from the application's message queue.

Window Class Registration

In a traditional Windows **WinMain** function, one important initialization task is to register one or more "window classes." Windows uses the registration information when it creates specific windows to display on the screen.

Note that a window class in Windows is not a C++ object class, as are the window classes derived from class **CWnd** in the Microsoft Foundation Classes.

A Windows application written with the Microsoft Foundation Classes registers several standard window classes for you. You can then simply create windows based on the registered classes. However, it is also possible to register your own custom window classes if you need something special.

Hello works nicely with the default window class registrations supplied by the Microsoft Foundation Class Library, so you do not need to do your own window registration.

How You Can Customize Your Windows Application

As shown above, the Microsoft Foundation Classes supply a **WinMain** function that provides several standard Windows actions for you:

- Windows application and instance initialization.
- Registration of several standard window classes.
- A message loop.

The Microsoft Foundation Classes also supply several ways to customize or override the standard facilities and behavior. You can:

- Write your own `WinMain` function and substitute it at link time for the **WinMain** provided by the Microsoft Foundation Classes.
- Call a global **AfxRegisterWndClass** function to register your own window classes.
- Override member functions such as **InitApplication**, **InitInstance**, **OnIdle**, **Run**, and **ExitInstance** in your derived application class.
- Make your program a Multiple Document Interface (MDI) application.
- Add dialog boxes, menus, and accelerators.

3.11 Summary

This chapter demonstrated the fundamental techniques for using the Microsoft Foundation Class Library to write Microsoft Windows programs.

The class library helps you think about Windows programming in a more object-oriented way. It also promotes more reusable code.

The next three chapters develop a larger Windows application, building on the classes and techniques explored in this chapter.

3.12 File Listings

The code shown in listings 1-5 is available on your distribution disks as HELLO.H, HELLO.CPP, HELLO.DEF, HELLO.RC, and RESOURCE.H.

Listing 1

```
// HELLO.H - Declares the class interfaces for the Hello application.

#ifndef __HELLO_H__
#define __HELLO_H__

// CMainWindow:
// See hello.cpp for the code to the member functions
// and the message map.
//
class CMainWindow : public CFrameWnd
{
public:
    CMainWindow();
```

```
afx_msg void OnPaint();
    afx_msg void OnAbout();

DECLARE_MESSAGE_MAP()
};

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

// CTheApp:
// See hello.cpp for the code to the InitInstance member function.
//
class CTheApp : public CWinApp
{
public:
    BOOL InitInstance();
};
```

Listing 2

```
// HELLO.CPP - Defines the class behaviors for the Hello application.

#include <afxwin.h>
#include "resource.h"
#include "hello.h"

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

// theApp:
// Just creating this application object runs the whole application.
//
CTheApp theApp;

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

// CMainWindow constructor:
// Create the window with the appropriate style, size, menu, etc.
//
CMainWindow::CMainWindow()
{
    LoadAccelTable("MainAccelTable");
    Create(NULL, "Hello Foundation Application",,
                 WS_OVERLAPPEDWINDOW, rectDefault, NULL, "MainMenu");
}
```

```
// OnPaint:
//
void CMainWindow::OnPaint()
{
    CString s = "Hello, Windows!";
    CPaintDC dc( this );
    CRect rect;

GetClientRect( rect );
    dc.SetTextAlign( TA_BASELINE | TA_CENTER );
    dc.SetBkMode( TRANSPARENT );
    dc.TextOut( (rect.right / 2), (rect.bottom / 2),,
                          s, s.GetLength() );
}

// OnAbout:
//
void CMainWindow::OnAbout()
{
    CModalDialog about( "AboutBox", this );
    about.DoModal();
}

// CMainWindow message map:
//
BEGIN_MESSAGE_MAP(CMainWindow, CFrameWnd)
    ON_WM_PAINT()
    ON_COMMAND(IDM_ABOUT, OnAbout)
END_MESSAGE_MAP()

////////////////////////////////////////////////////////////////////////////
// CTheApp

// InitInstance:
//
BOOL CTheApp::InitInstance()
{
    TRACE( "HELLO WORLD\n" );

    m_pMainWnd = new CMainWindow();
    m_pMainWnd -> ShowWindow( m_nCmdShow );
    m_pMainWnd -> UpdateWindow();

return TRUE;
}
```

Listing 3

```
NAME                    Hello

DESCRIPTION   'Hello Microsoft Foundation Classes Windows Application'

EXETYPE             WINDOWS

STUB      'WINSTUB.EXE'

CODE  PRELOAD FIXED DISCARDABLE

;than once concurrently
DATA  PRELOAD FIXED MULTIPLE

HEAPSIZE     1024
STACKSIZE    4096
```

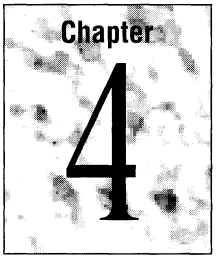
Listing 4

```
/* HELLO.RC : Defines the resources for the Hello application. */

#include <windows.h>
#include <afxres.h>
#include "resource.h"

AFX_IDI_STD_FRAME   ICON    hello.ico

MainMenu MENU
{

POPUP         "&Help"
    {
        MENUITEM "&About Hello...\tF1", IDM_ABOUT
    }
}

MainAccelTable ACCELERATORS
{
    VK_F1,      IDM_ABOUT,  VIRTKEY
}

rcinclude hello.dlg
```

Listing 5

```
/* RESOURCE.H - Defines resource constants for Hello

#define IDM_ABOUT   100
```

Phone Book: A Simple Windows Database



Chapter 2 showed how to use the Microsoft Foundation Class Library to build a data model for a simple name and phone number database. Chapter 3 showed how to use the Microsoft Foundation Classes to build a simple Microsoft Windows user interface.

This chapter and the two chapters following it show how to integrate the data model from Chapter 2 and the Windows user interface from Chapter 3. In these chapters, you'll write a larger Windows application to put a Windows interface on the name and phone number database. You'll begin with the Hello program from Chapter 3 as a template and build new functionality on top of it by adding more menus, by creating more complicated dialog boxes, and by providing keyboard and mouse interaction.

The three chapters cover the following topics:

- Chapter 4: How to use Hello as a template. How to create a simplified interface to the `CPerson` data.
- Chapter 5: How to add dialog boxes for editing data and for file opening, saving, and printing.
- Chapter 6: How to write the main window class and message map. How to add message-handler functions, including handlers for menus, the keyboard, and the mouse. How to prepare the supporting files. How to build the Phone Book program.

4.1 In This Chapter

Follow the tutorial in this chapter to create a class of "database objects." You can use one of these objects to provide a clean interface to the actual data. The interface can be used either in a character-based program or a Windows program. At the end of the chapter, you'll see how to use the interface for a non-Windows database program.

The Phone Book Program

The Phone Book program developed in the next three chapters is a simple name and phone number database. It stores information about people: first name, last name, and phone number. Using the Windows interface, you can create a new database, fill it with information, and save it to a disk file. You can also open an existing database from a file and add names, delete names, and edit information. You can find all entries with the same last name and display the list of found entries. And you can print your database files.

What the Program Does

The Phone Book program displays a window with a menu bar. The menu bar contains three menus, one for file operations, one for database operations, and one for Help.

If you create a new, empty database, the window title changes from "Phone Book" to "Phone Book–Untitled" and you can then add entries with the Add command in the Person menu. When you choose the Save or Save As command from the File menu, a dialog box prompts you to enter a filename, the database is serialized to the file, and the new filename becomes the new window title.

If you open an existing database, you are first prompted to save the existing one if it has unsaved changes. Then a standard Windows dialog box prompts you for a filename, the persistent data stored in the file is deserialized into an automatically created `CPersonList` object, and the new filename replaces the old one in the title bar. The data is displayed in the window, one line per person. If there's more data than the window can display, scrollbars are added to the window, and you can use them to scroll to data that isn't currently displayed. You can also scroll with the RIGHT ARROW, LEFT ARROW, PAGE UP, PAGE DOWN, HOME, and END keys.

To delete or edit a person's information, you must first select the person's line of information in the window. You can choose a person in the database by selecting the person's line of information in the display, either with the mouse or with the UP ARROW and DOWN ARROW keys.

If you add, delete, or edit the information in a database, the database is flagged internally to indicate that it has unsaved changes. When you add, delete, or edit a person's information, the display changes to reflect the new information.

If you search the database for a name, a list of matching list elements is displayed in the window, replacing the display of the full database. Changes to this list are reflected in the main list. To abandon the list of search results and display the full database again, you choose the Find All command in the Person menu.

If you choose the Print command in the File menu, a standard Windows print dialog box is displayed so you can select printing options.

If your file is unsaved and you choose the Save or Save As commands in the File menu, a standard Windows Save dialog box is displayed so you can name the file and select a directory for it. If the file has already been saved, you can choose the Save command to save recent changes.

Figure 4.1 shows the screen as it appears with an open database on display.

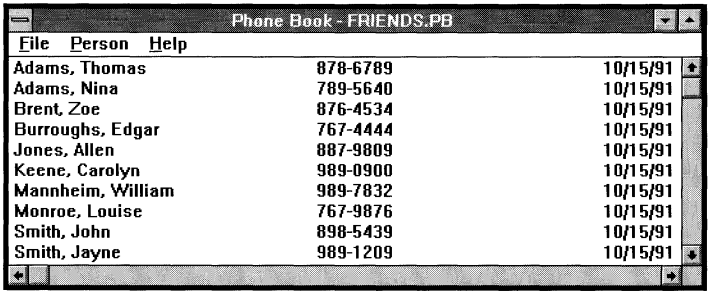


Figure 4.1 The Output of Phone Book

This example program does considerably more than Hello and provides you with a larger model on which to pattern your own Windows programs written with the Microsoft Foundation Class Library.

The Code for Phone Book

To view the complete code for Phone Book, see Listings 1 and 2 in this chapter and Listings 1 and 2 in Chapter 5. The listings in the current chapter give the code in files DATABASE.H and DATABASE.CPP. The listings in Chapter 5 give the code in files VIEW.H and VIEW.CPP. To review the code for the Data Model, see Windows Listings 1 and 2 in Chapter 2.

The code shown in Listings 1 and 2 is available on the distribution disks in files DATABASE.H and DATABASE.CPP.

Microsoft Foundation Classes Used in This Chapter

This chapter and the two following chapters use the classes employed in Chapters 2 and 3. Many of these classes are used in the same way, since Phone Book is built from the foundations of Hello and the Data Model program. But one class in particular, **CModalDialog**, is used more extensively in Chapter 5, and the **CMenu** class is also employed. **CMenu** provides access to the menus in the menu bar and is used for updating menus to suit the context. Menu commands unavailable in the current context are dimmed.

The sections that follow take you through the components of Phone Book, explaining how to write them, what they consist of, and how they work.

4.2 How to Write the Phone Book Program

This section gives an overview of the steps in writing Phone Book. As you work through the steps, you will learn what files to prepare, where to put the code in them, and how to compile the program. You'll build Phone Book by using Hello as a template.

The Steps in Writing Phone Book with the Microsoft Foundation Classes

You'll write the Phone Book program with the Microsoft Foundation Classes in 14 steps, spread over three chapters of the tutorial. The 14 steps can be summarized as follows:

1. Create a simplified data interface (Chapter 4).

   Phone Book uses two `CPersonList` objects, one for the database and one for any person objects found with its search facilities. The simplest way to manage these lists and the mechanisms needed to use them is to encapsulate the lists and mechanisms in another object. Class `CDataBase` lets the program access data through a single clean interface, regardless of which of the two lists is current. Add class `CDataBase`. For more information about this step, see "Create a Simplified Data Interface" on page 122.

2. Copy and modify the Hello files (Chapter 5).

   To use Hello as a template, you need to copy HELLO.H to VIEW.H and HELLO.CPP to VIEW.CPP. Then you need to modify some of the items in the new copies and add other items. For more information about this step, see "Work from a Template" on page 152.

3. Add two kinds of dialog boxes (Chapter 5).

   Phone Book needs two dialog boxes: one for entering a string, used as the name to search for, and one for entering or editing the data for a person. Add classes for these dialog boxes. For more information about this step, see "Add Dialog Boxes" on page 153 in Chapter 5.

4. Determine what messages will be handled (Chapter 6).

   Design the application's menus and list the Windows messages your code needs to handle. When the user chooses a command from the menu, a **WM_COMMAND** message is sent to the appropriate window along with information identifying which menu command is being generated. The application also needs to handle other Windows messages, such as **WM_PAINT**.

Phone Book has more than a dozen menu commands and responds to almost as many other Windows messages. For more information about this step, see "Determine What Messages Will Be Handled" on page 197 in Chapter 6.

5. Add message-handler functions for File menu commands (Chapter 6).

   Phone Book has several File menu commands. Each command needs a handler defined as a member function of class `CMainWindow` and a corresponding entry in the message map of `CMainWindow`. Add these functions. For more information about this step, see "Add Message Handlers for File Menu Commands" on page 205 in Chapter 6.

6. Add message-handler functions for Person menu commands (Chapter 6).

   Phone Book has several Person menu commands. Each command needs a handler defined as a member function of class `CMainWindow` and a corresponding entry in the message map of `CMainWindow`. Add these functions. For more information about this step, see "Add Message Handlers for Person Menu Commands" on page 216 in Chapter 6.

7. Add message-handler functions for Help menu commands (Chapter 6).

   Phone Book has two Help menu commands. Each command needs a handler defined as a member function of class `CMainWindow` and a corresponding entry in the message map of `CMainWindow`. You already added the handler function for the About menu command when you copied the Hello files to start Phone Book. Now add a function for the Help menu command. For more information about this step, see "Add Message Handlers for Help Menu Commands" on page 222 in Chapter 6.

8. Add message-handler functions for creation and sizing (Chapter 6).

   Phone Book responds to a number of commonly handled Windows messages. These include **WM_PAINT**, **WM_CREATE**, and **WM_SIZE**. Add message-handler member functions to the `CMainWindow` class for these messages. For more information about this step, see "Add Message Handlers for Creation and Sizing" on page 224 in Chapter 6.

9. Add scrolling member functions (Chapter 6).

   Phone Book handles both vertical and horizontal scrolling so the user can scroll through an entire database. For more information about this step, see "Add Scrolling Member Functions" on page 227 in Chapter 6.

10. Add a keyboard and mouse interface (Chapter 6).

    Phone Book uses certain keystrokes and mouse clicks to set or change the selection in the window. You can use the UP ARROW and DOWN ARROW keys or the mouse to change the selection. You can also use the DELETE key to delete a selected person or the ENTER key to edit a selected person. In addition, you can click the mouse in the scroll bars to scroll the list of persons. These actions require handler functions as well. Add these handlers. For more information about this step, see "Add a Keyboard and Mouse Interface" on page 230 in Chapter 6.

11. Add a member function to handle the **WM_PAINT** message (Chapter 6).

    The window that displays the database responds to this message to repaint its client area when it becomes invalid. For more information about this step, see "Add a Member Function to Handle the WM_PAINT Message" on page 235 in Chapter 6.

12. Add utility member functions (Chapter 6).

    Phone Book uses several utility functions. These are main window member functions called by the main window object's message-handler functions. For more information on this step, see "Add Utility Member Functions" on page 238 in Chapter 6.

13. Prepare supporting files (Chapter 6).

    As a Windows program, Phone Book requires the same kinds of supporting files as Hello. Add a module definition file, a resource script file, and a resource include file. For more information about this step, see "Prepare Supporting Files" on page 242 in Chapter 6.

14. Build the program (Chapter 6).

    With all the files prepared, compile and link the program. Remember that you must run the program in Windows. For more information about this step, see "Build the Program" on page 243 in Chapter 6.

The sections and chapters that follow detail the procedures involved in each of these 14 steps. Any code related to a procedure is given within the text. Each section (except the following one) concludes with a discussion of what the code does and why it does it that way. Where appropriate, additional advanced discussion in a special box elaborates on the code and the basic discussion.

4.3 Create a Simplified Data Interface

This section explains the first step in writing Phone Book: simplify the data interface with a new class. This process will require several steps:

1. Create interface and implementation files for class `CPerson`.
2. Create an interface file for class `CDataBase`.
3. Design class `CDatabase`.
4. Create an implementation file for class `CDataBase`.
5. Write the member functions of class `CDataBase`.

▶ To create data object files:

- Copy the PERSON.H and PERSON.CPP files that you made when designing the data object to your working directory for the Phone Book program.

  The simplified data interface object that you'll create uses these files to implement the database that it manages.

▶ To create an interface file for class CDataBase:

1. Create a file called DATABASE.H and add the following lines:

```
// database.h - Declares the interface for the CDataBase class.
//
#ifndef __DATABASE_H__
#define __DATABASE_H__

#include "person.h"

// String const for untitled database
extern const char szUntitled[];
```

   The **#define** statements ensure that the code in DATABASE.H is not included more than once. The **#include** statement includes the PERSON.H file developed in Chapter 2. PERSON.H declares classes `CPerson` and `CPersonList`. These classes are used by class `CDataBase`. The string constant declaration refers to a variable used by `CDataBase` and defined in the VIEW.CPP file. You'll create that file in Chapter 5.

2. At the bottom of the DATABASE.H file add the line:

```
#endif  // __DATABASE_H__
```

   Always keep this line at the bottom of the file, after all other code.

▶ To design class CDataBase:

Class `CDataBase` encapsulates two `CPersonList` objects and manages their use. The interface to `CDataBase` provides member functions to aid in accessing the data as Phone Book must do.

- Add the following class declaration for `CDataBase` to your DATABASE.H file after the lines added previously:

```
class CDataBase: public CObject
{
public:
    // constructor
    CDataBase::CDataBase()
```

```
        {
            m_pDataList = NULL;
            m_pFindList = NULL;
            m_szFileName = "";
            m_szFileTitle = "";
        }

    // Create/Destroy CPersonLists
    BOOL New();
    void Terminate();

    // File handling
    BOOL DoOpen( const char* pszFileName );
    BOOL DoSave( const char* pszFileName = NULL );
    BOOL DoFind( const char* pszLastName = NULL );

// Person Handling
    void AddPerson( CPerson* pNewPerson );
    void ReplacePerson( CPerson* pOldPerson,
                        const CPerson& rNewPerson );
    void DeletePerson( int nIndex );
    CPerson* GetPerson( int nIndex );

    // Database Attributes
    int GetCount()
        {
            ASSERT_VALID( this );
            if ( m_pFindList != NULL )
                return m_pFindList -> GetCount();
            if ( m_pDataList != NULL )
                return m_pDataList -> GetCount();
            return 0;
        }

    BOOL IsDirty()
        { ASSERT_VALID( this );
            return ( m_pDataList != NULL ) ? m_pDataList ->
                    GetDirty() : FALSE; }

    BOOL IsNamed()
        { ASSERT_VALID( this );
            return m_szFileName != szUntitled; }

    const char* GetName()
        { ASSERT_VALID( this );
            return m_szFileName; }

    CString GetTitle()
        { ASSERT_VALID( this );
            return  "Phone Book - " + m_szFileTitle; }
    void SetTitle( const char* pszTitle )
        { ASSERT_VALID( this );
            m_szFileTitle = pszTitle; }
```

```
    BOOL IsPresent()
        { ASSERT_VALID( this );
            return m_pDataList != NULL; }

protected:
    CPersonList* m_pDataList;
    CPersonList* m_pFindList;
    CString m_szFileName;
    CString m_szFileTitle;

private:
    CPersonList* ReadDataBase( CFile* pFile );
    BOOL WriteDataBase( CFile* pFile );

#ifdef _DEBUG
public:
    void AssertValid() const;
#endif
};
```

► To create an implementation file for class CDataBase:

- Create a file called DATABASE.CPP and add the following lines to it:

```
// DATABASE.CPP - Definitions for class CDataBase
//
#include "database.h"
#include <string.h>

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

const char szUntitled[] = "Untitled";
```

The **#include** directives make the class declaration in DATABASE.H and the declarations in STRING.H available to the code in the new file. The **#ifdef _DEBUG** lines support diagnostic reporting if you build the program in debug mode. The "Untitled" string is defined as a constant. This is the string referred to earlier from DATABASE.H as an externally-defined variable. It's used for new databases that have not been named yet.

The member functions of `CDataBase` include a constructor and functions that map fairly directly to some of Phone Book's menu commands as well as utility functions designed to make it easier to use the database. The constructor and functions `GetCount`, `IsDirty`, `IsNamed`, `GetName`, and `IsPresent` are all defined inline in the `CDataBase` class declaration. You will add function definitions to your DATABASE.CPP file.

► To write the member functions of CDataBase:

1. Add the `New` member function.

```
// CDataBase::New
// Initializes the database.
//
BOOL CDataBase::New()
{
    ASSERT_VALID( this );

    // Clean up any old data.
    Terminate();

    m_pDataList = new CPersonList;

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

`New` creates a new, empty database, to which the user can add persons. In the Windows program developed in the next two chapters, `New` supports the New command in the File menu.

Note the use of the **ASSERT_VALID** macro to test the assumption that there is a valid database object for which a new `CPersonList` data member can be created. For more information about the **ASSERT_VALID** macro, see "The AssertValid Member Function" on page 137.

2. Add the `Terminate` member function:

```
// CDataBase::Terminate
// Cleans up the database.
//
void CDataBase::Terminate()
{
    ASSERT_VALID( this );

    if ( m_pDataList != NULL )
        m_pDataList -> DeleteAll();

    delete m_pDataList;
    delete m_pFindList;

    m_pDataList = NULL;
    m_pFindList = NULL;

    m_szFileName = szUntitled;
    m_szFileTitle = szUntitled;
}
```

`Terminate` cleans up when the user ends the program or opens a new database file while an old one is still open. In the Windows version of Phone Book, `Terminate` is used to support the Exit, New, and Open commands in the File menu.

3. Add the `AddPerson` member function:

```
// CDataBase::AddPerson
// Inserts a person in the appropriate position (alphabetically by
// last name) in the database.
//
void CDataBase::AddPerson( CPerson* pNewPerson )
{
    ASSERT_VALID( this );
    ASSERT_VALID( pNewPerson );
    ASSERT( pNewPerson != NULL );
    ASSERT( m_pDataList != NULL );

    POSITION pos = m_pDataList -> GetHeadPosition();
    while ( pos != NULL &&
            _stricmp( ((CPerson*)m_pDataList -> GetAt(pos)) ->
            GetLastName(),
                    pNewPerson -> GetLastName() ) <= 0 )
        m_pDataList -> GetNext( pos );

    if ( pos == NULL )
        m_pDataList -> AddTail( pNewPerson );
    else
        m_pDataList -> InsertBefore( pos, pNewPerson );

    m_pDataList -> SetDirty( TRUE );
}
```

`AddPerson` adds a given new person object to the database. In the Windows version of Phone Book, `Add` is used to support the Add command in the Person menu.

Again note the use of the **ASSERT_VALID** macro to test assumptions about the validity of the current database object and about the person object passed as an argument. Also note the use of the related **ASSERT** macro. During debugging, this macro asserts that the expression passed to it is **TRUE**; if not, the program halts with a diagnostic message that tells where the error occurred.

`AddPerson` calls several member functions inherited by class `CPersonList` from its base class, **CObList**, to search the list for the place to add.

4. Add the `GetPerson` member function:

```
// CDataBase::GetPerson
// Look up someone by index.
//
CPerson* CDataBase::GetPerson( int nIndex )
```

```
{
    ASSERT_VALID( this );
    ASSERT( m_pDataList != NULL );

    if ( m_pFindList != NULL )
        return (CPerson*)m_pFindList -> GetAt( m_pFindList ->
        FindIndex( nIndex ) );
    else
        return (CPerson*)m_pDataList -> GetAt( m_pDataList ->
        FindIndex( nIndex ) );

}
```

`GetPerson` retrieves a person from the database by index. In the Windows program, `GetPerson` is used to support several menu commands.

`GetPerson` calls inherited **CObList** member functions to find the specified index. The code in effect requests the database object's `m_pFindList` or its `m_pDataList` to use its **GetAt** member function to retrieve a `CPerson` object in a `CPersonList` object. The specified index into the list is converted to a pointer to the object at that index by a call to the list object's `FindIndex` member function.

5. Add the `DeletePerson` member function:

```
// CDatabase::DeletePerson
// Removes record of person from database.
//
void CDataBase::DeletePerson( int nIndex )
{
    ASSERT_VALID( this );
    ASSERT( m_pDataList != NULL );

    POSITION el = m_pDataList -> FindIndex( nIndex );
    delete m_pDataList -> GetAt( el );
    m_pDataList -> RemoveAt( el );
    m_pDataList -> SetDirty( TRUE ); }
```

`DeletePerson` deletes the person object at a specified index. In the Windows program, `DeletePerson` is used to support the Delete command in the Person menu.

The logic of this member function is similar to that of `GetPerson` above.

6. Add the `ReplacePerson` member function:

```
// CDatabase::ReplacePerson
// Replaces an object in the list with the new object.
//

void CDataBase::ReplacePerson( CPerson* pOldPerson, const CPerson&
rNewPerson )
```

```
{
    ASSERT_VALID( this );

    ASSERT( pOldPerson != NULL );
    ASSERT( m_pDataList != NULL );

    // Using the overloaded operator= for CPerson
    *pOldPerson = rNewPerson;
    m_pDataList->SetDirty( TRUE );
}
```

`ReplacePerson` is a utility member function that replaces an existing person object in the database with a newly edited person object. In the Windows program, `ReplacePerson` is used to support the Edit command in the Person menu. The code takes advantage of the overloaded assignment operator supplied with the `CPerson` class.

7. Add the `DoFind` member function:

```
// CDataBase::DoFind
// Does a FindPerson call, or clears the find data.
//
BOOL CDataBase::DoFind( const char* pszLastName /* = NULL */ )
{
    ASSERT_VALID( this );
    ASSERT( m_pDataList != NULL );

    if ( pszLastName == NULL )
    {
        delete m_pFindList;
        m_pFindList = NULL;
        return FALSE;
    }

    ASSERT( m_pFindList == NULL );
    return ( ( m_pFindList = m_pDataList ->
    FindPerson( pszLastName ) ) != NULL );
}
```

`DoFind` searches for all person objects in the database whose last name data members match a search string. `DoFind` calls the `CPersonList` member function `FindPerson`, which returns a `CPersonList` object containing pointers to all of the found person objects. In the Windows program, `DoFind` is used to support the Find command in the Person menu.

8. Add the `DoOpen` member function:

```
// CDataBase::DoOpen
// Reads a database from the given filename.
//
BOOL CDataBase::DoOpen( const char* pszFileName )
{
    ASSERT_VALID( this );
    ASSERT( pszFileName != NULL );

    CFile file( pszFileName, CFile::modeRead );

    // read the object data from file
    CPersonList* pNewDataBase = ReadDataBase( &file );

    file.Close();

    // get rid of current data base if new one is OK
    if ( pNewDataBase != NULL )
    {
        Terminate();
        m_pDataList = pNewDataBase;
        m_pDataList -> SetDirty( FALSE );

        m_szFileName = pszFileName;
        return TRUE;
    }
    else
        return FALSE;
}
```

`DoOpen` takes a filename as its argument, opens the file of that name, and calls `ReadDatabase` (given later) to read its data into a `CPersonList` object for use as the current database. In the Windows program, `DoOpen` is used to support the Open command in the File menu.

Note that the existing database, if any, is only deleted after the new one is successfully opened.

9. Add the `DoSave` member function:

```
// CDataBase::DoSave
// Saves the database to the given file.
//
BOOL CDataBase::DoSave( const char* pszFileName /* = NULL */ )
{
    ASSERT_VALID( this );

    // if we were given a name store it in the object.
    if ( pszFileName != NULL )
        m_szFileName = pszFileName;
```

```
    CFileStatus status;
    int nAccess = CFile::modeWrite;

    // GetStatus will return TRUE if file exists, or FALSE
    // if it doesn't.
    if ( !CFile::GetStatus( m_szFileName, status ) )
        nAccess |= CFile::modeCreate;

    CFile file( m_szFileName, nAccess );

    // write the data base to a file
    // mark it clean if write is successful
    if ( WriteDataBase( &file ) )
    {
        m_pDataList -> SetDirty( FALSE );
        file.Close();
        return TRUE;
    }
    else
    {
        file.Close();
        return FALSE;
    }
}
```

`DoSave` calls `WriteDatabase` (given below) to serialize the current database to a disk file. In the Windows program, `DoSave` is used to support the Save command in the File menu.

10. Add the `ReadDataBase` member function (you can copy it from the DMTEST.CPP file in Chapter 2 if you like):

```
// CDataBase::ReadDataBase
// Serializes in the database.
//
CPersonList* CDataBase::ReadDataBase( CFile* pFile )
{
    ASSERT_VALID( this );
    CPersonList* pNewDataBase = NULL;

    // Create an archive from pFile for reading.
    CArchive archive( pFile, CArchive::load );

    // Deserialize the new data base from the archive, or catch the
    // exception.
    TRY
    {
        archive >> pNewDataBase;
    }
    CATCH( CArchiveException, e )
```

```
    {
#ifdef _DEBUG
        e -> Dump( afxDump );
#endif
        archive.Close();

        // If we got part of the database, then delete it.
        if ( pNewDataBase != NULL )
        {
            pNewDataBase -> DeleteAll();
            delete pNewDataBase;
        }

        // We caught this exception, but we throw it again so our
        // caller can also catch it.
        THROW_LAST();
    }
    END_CATCH

    // Exit here if no errors or exceptions.
    archive.Close();
    return pNewDataBase;
}
```

`ReadDatabase` serializes the current database to a disk file. In the Windows program, `ReadDatabase` is used along with `DoOpen` to support the Open command in the File menu.

11. Add the `WriteDataBase` member function (you can copy it from the DMTEST.CPP file in Chapter 2 if you like—but delete the second parameter, of type `CPersonList`):

```
// CDataBase::WriteDataBase
// Serializes out the data into the given file.
//
BOOL CDataBase::WriteDataBase( CFile* pFile )
{
    ASSERT_VALID( this );
    ASSERT( m_pDataList != NULL );

    // Create an archive from theFile for writing
    CArchive archive( pFile, CArchive::store );

    // Archive out, or catch the exception.
    TRY
    {
        archive << m_pDataList;
    }
    CATCH( CArchiveException, e )
```

```
    {
#ifdef _DEBUG
        e -> Dump( afxDump );
#endif
        archive.Close();

        // Throw this exception again for the benefit of our caller.
        THROW_LAST();
    }
    END_CATCH

    // Exit here if no errors or exceptions.
    archive.Close();
    return TRUE;
}
```

`WriteDatabase` deserializes a database from a disk file. In the Windows program, `WriteDatabase` is used along with `DoSave` to support the Save and Save As commands in the File menu.

If you copy `WriteDatabase` from Chapter 2, be sure to delete the second parameter, `pDataBase`. Now that `WriteDatabase` has become a member function of a `CDataBase` object, it has direct access to the `CPersonList` object stored in the `m_pDataList` member variable. The parameter is no longer needed.

When you delete the second parameter, change the name `pDataBase` throughout the member function to `m_pDataList` (it occurs once). Also add the following two lines as the first two statements in the member function:

```
ASSERT_VALID( this );
ASSERT( m_pDataList != NULL );
```

12. Add the `AssertValid` member function:

```
#ifdef _DEBUG
void CDataBase::AssertValid() const
{
    if( m_pDataList != NULL )
    {
        ASSERT_VALID( m_pDataList );
        if( m_pFindList != NULL )
            ASSERT_VALID( m_pFindList );
    }
    else
        ASSERT( m_pFindList == NULL );
}
#endif
```

This `AssertValid` member function overrides the **AssertValid** member function of class **CObject**, the base class of `CDataBase`. Note that the definition is bracketed by **#ifdef _DEBUG** and **#endif** directives. The member function is only compiled if the **_DEBUG** flag is defined for the build.

For more information about the purpose in overriding this member function, see "The AssertValid Member Function" on page 137.

Files DATABASE.H and DATABASE.CPP are now complete. You can check them against the files in Listings 1 and 2, given earlier.

To continue the tutorial, see "Add Dialog Boxes" on page 153 in the next chapter. For explanations of the `CDataBase` code just added, see the discussion below, "Discussion: Class CDataBase."

Discussion: Class CDataBase

A `CDataBase` object is used to encapsulate all access to the two `CPersonList` objects that store data for the application. The object provides a clear interface between your Windows code and the database. This section discusses the structure and use of `CDataBase`.

This discussion does not instruct you to add any code to your files.

Purpose and Structure of CDataBase

`CDataBase` and the data model from Chapter 2 make a good model for Windows programs written with C++ and the Microsoft Foundation Class Library. `CDataBase` preserves the independence of the data model from the user interface, but it also tailors access to the data model to the needs of a command-driven user interface, such as the program's Windows user interface.

If `CDataBase` had not been invented, considerably more code would have to be added directly to the main window class. The main window object would have to store two `CPersonList` objects in its member variables (one for the database and one to store persons found by searches), keep track of the filename for the current database, maintain information about which of the two `CPersonList` objects to display, keep track of unsaved changes to the database, manage file opening and I/O, and more. Each of the main window object's message-handler functions would be more complex.

With `CDataBase`, on the other hand, all of that database-related code is neatly encapsulated in a single database object. The main window object is much cleaner and simpler than it otherwise would be. It constructs and stores only a single `CDataBase` object and uses that object for all communication with the data model objects. The interface of `CDataBase`—its list of public member functions—is simple and tailored to the application's needs. Figure 4.2 shows the relationship of `CDataBase` to the Windows user interface and to the `CPersonList` data model.

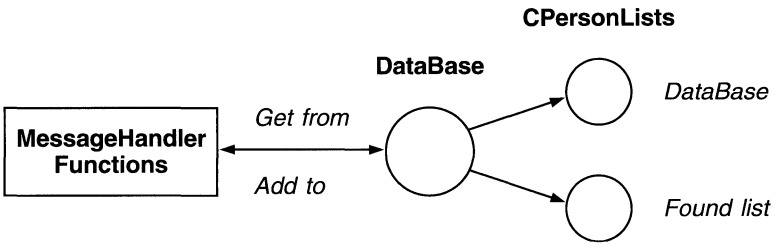


Figure 4.2 Role of the CDataBase Object

CDataBase Member Functions

Class `CDataBase` declares a constructor and a number of member functions. The constructor creates and initializes a `CDataBase` object. As you can see from its inline definition, the `CDataBase` constructor sets the `m_pDataList` and `m_pFindList` member variables of class `CDataBase` to **NULL**, signifying that the database is empty, and sets the `m_szFileName` and `m_szFileTitle` member variables to an empty string value.

Some of the `CDataBase` member functions are directly mapped to menu commands in the Phone Book program. When the user chooses the New, Open, Save, Close, or Exit commands in the File menu, or the Add, Delete, Find, or Edit commands in the Person menu, the message-handler functions in the main window object for those menu commands call corresponding `CDataBase` member functions to do the bulk of the work. You'll see later how these functions are used.

The remaining member functions of `CDataBase` provide utility services, making the database easier to use. Of these, some are database utitlities and some are file-handling utilities.

Database Member Functions

`GetPerson`
Returns a pointer to the `CPerson` object at a given index into the database

Phone Book calls this member when it needs information about a person in the database. The member function is used in repainting the client window.

`GetCount`
Returns the size of the database

Phone Book calls this member when it needs to know how many people are in the database.

`IsDirty`
Returns **TRUE** if the database has unsaved changes

Phone Book calls this member when it must know if the database needs to be saved.

`IsNamed`
Returns **TRUE** if the database has been saved to a file (and thus been given the file's name)

Phone Book calls this member to learn if the database has already been saved or not.

`GetName`
Returns the current name of the database (and its file)

Phone Book calls this member to learn the name of the current database file for use in saving the file. The member contains a full path.

`IsPresent`
Returns **TRUE** if a `CPersonList` is currently allocated for the database

Phone Book calls this function to learn if an existing database is open or not.

`GetTitle`
Returns the title of the current database concatenated to the string "Phone Book -".

Phone Book calls this function to build the window caption.

`SetTitle`
Sets the value of the database title

In Phone Book, this string is set using the information returned by **CommDlg**. **CommDlg** returns the filename and extension all in uppercase letters with no path.

File-Handling Member Functions

The file-handling member functions are familiar. They were used in the Data Model program in Chapter 2, where they were stand-alone functions called by **main**. Here, they have become member functions of the `CDataBase` class. These functions are as follows:

`ReadDataBase`
Deserializes `CPerson` data in a disk file into a new `CPersonList` object

`WriteDataBase`
Serializes the current database `CPersonList` to a disk file

Member Variables

A `CDataBase` object has several member variables. These store:

- The current database, if any, as a `CPersonList` object in `m_pDataList`.

  If no database is currently loaded, the `CDataBase` `m_pDataList` member variable is **NULL**, and the member function `IsPresent` returns **FALSE**.

- The current found list, if any, as another `CPersonList` object in the `CDataBase` member `m_pFindList`.

This list stores pointers to any objects found by a call to the `FindPerson` member function of `CPersonList`. If no found list currently is present, the variable is **NULL**.

- The current filename (and database name) in the `m_szFileName` member variable of class `CDataBase`.

  If no file is currently open, this variable contains an empty string. If a database is open but has not yet been saved, the variable contains the string "Untitled". Otherwise, it contains the filename with a full path.

- The current file title in the `m_szFileTitle` member variable of class `CDataBase`.

  This member variable is used to build a caption string for the window. If no file is currently open, this variable contains an empty string. If a database is open but has not yet been saved, the variable contains the string "Untitled". Otherwise, it contains the filename in uppercase letters with no path. The filename is obtained from the standard Windows open file dialog.

The Phone Book application displays whichever of the two `CPersonList` objects is "current" according to this algorithm:

- If there is a found list (the `m_pFindList` of the `CDataBase` member is not **NULL**), display it.
- Otherwise, if there is a database (`m_pDataList` is not **NULL**), display it.
- If neither list is present, no database is loaded, so display nothing.

Database display is managed by the main window object's `OnPaint` member function, which displays the current list. Because the mechanism for selecting the current list is encapsulated in `CDataBase`, `OnPaint` doesn't have to contain code to make the selection. Other inner workings of the database are handled similarly, which makes the Windows user interface code much simpler than it would be if it had to manage the details of the two lists.

The AssertValid Member Function

Programmers make many assumptions about the validity of their data. One of the more time-consuming tasks a programmer must perform is testing those assumptions at appropriate points to ensure valid processing. The Microsoft Foundation Class Library provides excellent support for this task.

Chapter 2 briefly outlined a mechanism for customizing validity testing of your own objects. During program development, you can use the **ASSERT** and **ASSERT_VALID** macros to test program assumptions. **ASSERT** simply evaluates its argument and, if the result is zero, prints a diagnostic message and halts the program. **ASSERT_VALID** calls the **AssertValid** member function of the object

passed as its argument. Both macros are enabled in debug mode and disabled in release mode. In release mode, they do nothing.

To illustrate, suppose you have an object that represents a linked list with pointers to both the head and tail elements of the list. One reasonable assumption is that the list object itself exists. Another reasonable assumption is that if the list is empty its head and tail pointers should both be **NULL**.

Many of the member functions of class `CDataBase` invoke the **ASSERT_VALID** macro to test the validity of the list objects underlying the database object. **ASSERT_VALID** simply calls the specified object's `AssertValid` member function. Typically, the object you pass to **ASSERT_VALID** is the object pointed to by **this**, but it could be any object.

A class derived from **CObject**, as `CDataBase` is, can simply inherit **CObject**'s version of **AssertValid**.

But a derived class can instead override **AssertValid**, as `CDataBase` does, to implement special testing when the **ASSERT_VALID** macro is invoked. If so, the overridden version is the one called through **ASSERT_VALID**. Hence, the response to the macro is customized.

`CDataBase` uses this mechanism to add validity tests for the objects stored in its data members to the validity test of the `CDataBase` object itself. The overriding `AssertValid` of `CDataBase` invokes the **ASSERT_VALID** macro again for its own `m_pDataList` and `m_pFindList` member variables, which hold `CPersonList` objects.

The chain of validity testing might stop at this level, but in this case class **CObList**, from which class `CPersonList` is derived, overrides **AssertValid**. This override performs additional validity testing on the internal state of the list. If the list is empty, its head pointer and its tail pointer must both be **NULL**. If not, an assertion is performed, diagnostic messages are printed, and the program halts. Similarly, if the list is not empty, both pointers must be non-**NULL** to avoid an assertion.

Thus a validity test on a `CDataBase` object leads to validity tests on the two stored `CPersonList` objects and then to additional validity tests for the internal states of those list objects.

This is quite a powerful mechanism when you build for debugging, and when you subsequently build for release, the mechanism is turned off automatically. The overriding `AssertValid` member function is not compiled in release mode, and the **ASSERT_VALID** macro invocations do nothing. You can put all of this to very good use, as demonstrated by `CPersonList` and `CDataBase`.

How CDataBase Is Used

The program's one `CDataBase` object stores two `CPersonList` objects, as mentioned earlier. It also stores the name of the current file. The program can create new databases and store them in a file, or it can open existing database files to permit operations on the data.

The program can have one of three states:

- No database is currently loaded.

  This is because the user has not yet created a new database with the New command in the File menu, has not opened an existing database with the Open command in the File menu, or has closed a previously open database. There is no database to manipulate. The user can create or open a database or exit the program.

  In this case, the database object's `m_pDataList` and `m_pFindList` member variables are **NULL**. The `m_szFileName` member variable contains an empty string.

- A database exists, but it hasn't been saved to a file.

  The user has chosen the New command in the File menu, which creates a new, empty database. Because the database is unsaved, the string "Untitled" appears in the title bar of the window. The user can add persons to the database, delete persons from it, edit the data for persons already in it, search the database for a particular last name, get information about the program, including help, save the current database to a file, open or create a new database (and be prompted to save the existing one first), or simply exit the program.

  In this case, the `CDataBase` member variable `m_pDataList` contains a `CPersonList` object. If the data has been searched, and the search results not yet deleted, `m_pFindList` contains a second `CPersonList` object. The `m_szFileName` member contains the string "Untitled".

- A database exists, and it has been saved to a file.

  The window caption has been set to the filename. If the user has added, deleted, or edited since the last save, the database is flagged as having unsaved changes. The user can manipulate the database further, close it, open a new database (and be prompted to save the old one if there are unsaved changes), or exit the program.

  In this case, the `CDataBase` object's `m_pDataList` member variable is not **NULL**. The database object's `m_pFindList` member is **NULL** if no search is in progress but contains a `CPersonList` object if the user has chosen the Find command in the Person menu. The database object's `m_szFileName` member contains the name of the file the data was saved to.

4.4 Applications for Class CDataBase

Keep the following image in mind: a `CDataBase` object is a capsule around the complexities of managing two `CPersonList` objects and the files associated with their contents.

Because of this encapsulated design, you can use the class in any programming environment. The next two chapters will build a Microsoft Windows user interface around class `CDataBase`. But you could just as easily use the class in a character-based DOS application.

In fact, that's just what the CMDBOOK example program does. Because CMDBOOK so closely parallels the Phone Book program presented in the remaining chapters of the tutorial, CMDBOOK will not be explored in detail here. But a quick overview is in order.

CMDBOOK presents a command-line user interface. The user types in a command, such as "new" or "add", and a command-line interpreter calls the appropriate function to handle that command. The command functions, in turn, call upon the services of a `CDataBase` object which is absolutely identical to the ones used in Phone Book.

You can study this different use of `CDataBase` in the code provided on the distribution disk. See file CMDBOOK.CPP in the MFC\SAMPLES\TUTORIAL directory.

4.5 What's Next

This chapter got you started. It showed you how to create the beginnings of the Phone Book program using the Hello program from Chapter 3 as a template. And it showed you how to use a C++ object to simplify the interface between Phone Book's Windows code and the data model code from Chapter 2.

With the data model firmly encapsulated, you can continue the tutorial in the next chapter by developing the first parts of the user interface: the dialog boxes by which the program interacts with the user.

4.6 File Listings

The code shown in Listings 1 and 2 is available on your distribution disks in Files DATABASE.H and DATABASE.CPP.

Listing 1

```
// database.h : Declares the interfaces for the CDataBase class.
//
// This is a part of the Microsoft Foundation Classes C++ library.
// Copyright (C) 1991 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Microsoft Foundation Classes Reference and Microsoft
// QuickHelp documentation provided with the library.
// See these sources for detailed information regarding the
// Microsoft Foundation Classes product.

#ifndef __DATABASE_H__
#define __DATABASE_H__

#include "person.h"

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

// string const for untitiled database
extern const char szUntitled[];
/////////////////////////////////////////////////
// CDataBase
// The database object is intended to encapsulate everything
// needed to work the CPersonList into one interface.
// Exception handling, data manipulations, and searching
// are handled on this level freeing the view program, whether it
// is character or windows, to deal with displaying the data.
//
class CDataBase: public CObject
{
public:
    // constructor
    CDataBase()
        {
            m_pDataList = NULL;
            m_pFindList = NULL;
            m_szFileName = "";
            m_szFileTitle = "";
        }

// Create/Destroy CPersonLists
    BOOL New();
    void Terminate();

// File handling
    BOOL DoOpen( const char* pszFileName );
    BOOL DoSave( const char* pszFileName = NULL );
    BOOL DoFind( const char* pszLastName = NULL );
```

```
// Person Handling
    void AddPerson( CPerson* pNewPerson );
    void ReplacePerson( CPerson* pOldPerson, const CPerson& rNewPerson );
    void DeletePerson( UINT nIndex );
    CPerson* GetPerson( UINT nIndex );

// Database Attributes
    UINT GetCount()
        {
            ASSERT_VALID( this );
            if ( m_pFindList != NULL )
                return m_pFindList -> GetCount();
            if ( m_pDataList != NULL )
                return m_pDataList -> GetCount();
            return 0;
        }

    BOOL IsDirty()
        { ASSERT_VALID( this );
            return ( m_pDataList != NULL ) ? m_pDataList -> GetDirty() : FALSE; }

    BOOL IsNamed()
        { ASSERT_VALID( this );
            return m_szFileName != szUntitled; }

    const char* GetName()
        { ASSERT_VALID( this );
            return m_szFileName; }

    CString GetTitle()
        { ASSERT_VALID( this );
            return  "Phone Book - " + m_szFileTitle; }
    void SetTitle( const char* pszTitle )
        { ASSERT_VALID( this );
            m_szFileTitle = pszTitle; }

BOOL IsPresent()
        { ASSERT_VALID( this );
            return m_pDataList != NULL; }

protected:
    CPersonList* m_pDataList;
    CPersonList* m_pFindList;
    CString m_szFileName;
    CString m_szFileTitle;
```

```
private:
    CPersonList* ReadDataBase( CFile* pFile );
    BOOL WriteDataBase( CFile* pFile );

#ifdef _DEBUG
public:
    void AssertValid() const;
#endif
};

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

#endif // __DATABASE_H__
```

Listing 2

```
// database.cpp : Defines the behaviors for the CDataBase class.
//
// This is a part of the Microsoft Foundation Classes C++ library.
// Copyright (C) 1991 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Microsoft Foundation Classes Reference and Microsoft
// QuickHelp documentation provided with the library.
// See these sources for detailed information regarding the
// Microsoft Foundation Classes product.

#include "database.h"
#include <string.h>

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

const char* szUntitled = "Untitled";

/////////////////////////////////////////////////////////////////////////////
// CDataBase
//

/////////////////////////////////////////////////
// CDataBase::New
// Initializes the database.
//
BOOL CDataBase::New()
{
    ASSERT_VALID( this );

    // Clean up any old data.
    Terminate();
```

```
m_pDataList = new CPersonList;

    return ( m_pDataList != NULL );
}

////////////////////////////////////////////////
// CDataBase::Terminate
// Cleans up the database.
//
void CDataBase::Terminate()
{
    ASSERT_VALID( this );

    if ( m_pDataList != NULL )
        m_pDataList -> DeleteAll();

delete m_pDataList;
    delete m_pFindList;

m_pDataList = NULL;
    m_pFindList = NULL;

    m_szFileName = szUntitled;
    m_szFileTitle = szUntitled;
}

////////////////////////////////////////////////
// CDataBase::AddPerson
// Inserts a person in the appropriate position (alphabetically by last
// name) in the database.
//
void CDataBase::AddPerson( CPerson* pNewPerson )
{
    ASSERT_VALID( this );
    ASSERT_VALID( pNewPerson );
    ASSERT( pNewPerson != NULL );
    ASSERT( m_pDataList != NULL );

POSITION pos = m_pDataList -> GetHeadPosition();
    while ( pos != NULL &&
            _stricmp( ((CPerson*)m_pDataList -> GetAt(pos)) -> GetLastName(),
                    pNewPerson -> GetLastName() ) <= 0 )
        m_pDataList -> GetNext( pos );

    if ( pos == NULL )
        m_pDataList -> AddTail( pNewPerson );
    else
        m_pDataList -> InsertBefore( pos, pNewPerson );

m_pDataList -> SetDirty( TRUE );
}
```

```
////////////////////////////////////////////////
// CDataBase::GetPerson
// Look up someone by index.
//
CPerson* CDataBase::GetPerson( UINT nIndex )
{
    ASSERT_VALID( this );
    ASSERT( m_pDataList != NULL );

    if ( m_pFindList != NULL )
        return (CPerson*)m_pFindList -> GetAt( m_pFindList ->
                FindIndex( nIndex ) );
    else
        return (CPerson*)m_pDataList -> GetAt( m_pDataList ->
                FindIndex( nIndex ) );

}

////////////////////////////////////////////////
// CDatabase::DeletePerson
// Removes record of person from database.
//
void CDataBase::DeletePerson( UINT nIndex )
{
    ASSERT_VALID( this );
    ASSERT( m_pDataList != NULL );

    if ( m_pFindList != NULL )
        m_pFindList -> RemoveAt( m_pFindList -> FindIndex( nIndex ) );
    else
        m_pDataList -> RemoveAt( m_pDataList -> FindIndex( nIndex ) );

    m_pDataList -> SetDirty( TRUE );
}

////////////////////////////////////////////////
// CDatabase::ReplacePerson
// Replaces an object in the list with a new object.
//
void CDataBase::ReplacePerson( CPerson* pOldPerson, const CPerson& rNewPerson )
{
    ASSERT_VALID( this );
    ASSERT( pOldPerson != NULL );
    ASSERT( m_pDataList != NULL );

    //Using the overloaded operator= for CPerson
    *pOldPerson = rNewPerson;
    m_pDataList -> SetDirty( TRUE );
}
```

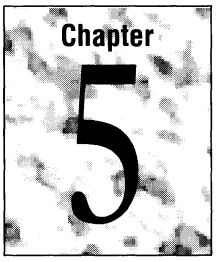
```
////////////////////////////////////////////////
// CDataBase::DoFind
// Does a FindPerson call, or clears the find data.
//
BOOL CDataBase::DoFind( const char* pszLastName /* = NULL */ )
{
    ASSERT_VALID( this );
    ASSERT( m_pDataList != NULL );

    if ( pszLastName == NULL )
    {
        delete m_pFindList;
        m_pFindList = NULL;
        return FALSE;
    }

    // The interface should not allow a second find to occur while
    // we already have one.
    // ASSERT( m_pFindList == NULL );
    // return ( ( m_pFindList = m_pDataList ->
    // FindPerson( pszLastName ) ) != NULL
);
}

////////////////////////////////////////////////
// CDataBase::DoOpen
// Reads a database from the given filename.
//
BOOL CDataBase::DoOpen( const char* pszFileName )
{
    ASSERT_VALID( this );
    ASSERT( pszFileName != NULL );

    CFile file( pszFileName, CFile::modeRead );

// read the object data from file
    CPersonList* pNewDataBase = ReadDataBase( &file );

    file.Close();
```

```
// get rid of current data base if new one is OK
    if ( pNewDataBase != NULL )
    {
        Terminate();
        m_pDataList = pNewDataBase;
        m_pDataList -> SetDirty( FALSE );

        m_szFileName = pszFileName;
        return TRUE;
    }
    else
        return FALSE;
}

////////////////////////////////////////////////
// CDataBase::DoSave
// Saves the database to the given file.
//
BOOL CDataBase::DoSave( const char* pszFileName /* = NULL */ )
{
    ASSERT_VALID( this );

// if we were given a name store it in the object.
    if ( pszFileName != NULL )
        m_szFileName = pszFileName;

CFileStatus status;
    UINT nAccess = CFile::modeWrite;

// GetStatus will return TRUE if file exists, or FALSE if it doesn't.
    if ( !CFile::GetStatus( m_szFileName, status ) )
        nAccess |= CFile::modeCreate;

    CFile file( m_szFileName, nAccess );

// write the data base to a file
    // mark it clean if write is successful
    if ( WriteDataBase( &file ) )
    {
        m_pDataList -> SetDirty( FALSE );
        file.Close();
        return TRUE;
    }
    else
    {
        file.Close();
        return FALSE;
    }
}
```

```
////////////////////////////////////////////////
// CDataBase::ReadDataBase
// Serializes in the database.
//
CPersonList* CDataBase::ReadDataBase( CFile* pFile )
{
    ASSERT_VALID( this );
    CPersonList* pNewDataBase = NULL;

// Create an archive from pFile for reading.
    CArchive archive( pFile, CArchive::load );

// Deserialize the new data base from the archive, or catch the
    // exception.
    TRY
    {
        archive >> pNewDataBase;
    }
    CATCH( CArchiveException, e )
    {
#ifdef _DEBUG
        e -> Dump( afxDump );
#endif
        archive.Close();

        // If we got part of the database, then delete it.
        if ( pNewDataBase != NULL )
        {
            pNewDataBase -> DeleteAll();
            delete pNewDataBase;
        }

        // We caught this exception, but we throw it again so our caller can
        // also catch it.
        THROW_LAST();
    }
    END_CATCH

// Exit here if no errors or exceptions.
    archive.Close();
    return pNewDataBase;
}

////////////////////////////////////////////////
BOOL CDataBase::WriteDataBase( CFile* pFile )
{
    ASSERT_VALID( this );
    ASSERT( m_pDataList != NULL );

    // Create an archive from theFile for writing
    CArchive archive( pFile, CArchive::store );
```

```
// Archive out, or catch the exception.
    TRY
    {
        archive << m_pDataList;
    }
    CATCH( CArchiveException, e )
    {
#ifdef _DEBUG
        e -> Dump( afxDump );
#endif
        archive.Close();

// Throw this exception again for the benefit of our caller.
        THROW_LAST();
    }
    END_CATCH

// Exit here if no errors or exceptions.
    archive.Close();
    return TRUE;
}

#ifdef _DEBUG
void CDataBase::AssertValid() const
{
    if ( m_pDataList != NULL )
    {
        ASSERT_VALID( m_pDataList );
        if ( m_pFindList != NULL )
            ASSERT_VALID( m_pFindList );
    }
    else
        ASSERT( m_pFindList == NULL );
}
#endif
```

Phone Book: Dialog Boxes



The previous chapter described the Phone Book program and got you started on it. This chapter continues development of Phone Book by adding the dialog boxes needed to interact with the user. The next chapter will complete the program.

5.1 In This Chapter

In addition to its simple About dialog box, which is handled the same way as in Hello, the Phone Book program requires five kinds of dialog boxes to get information from the user. Three of these are standard Windows dialog boxes and are not created with Microsoft Foundation Classes. These are discussed later, under "Standard Windows Dialog Boxes" on page 167. The other two, the Find and the Edit dialog boxes discussed below, are derived from one of the Microsoft Foundation dialog classes.

Because the Find and Edit dialog boxes contain additional controls, you can't simply construct a **CModalDialog** object, as in the About dialog box. Instead, you must derive a new dialog class from **CModalDialog** and provide it with code to process the controls. You'll derive two such classes, one for each of the two new kinds of dialog boxes needed for this program.

In this chapter, you'll begin creating two files, VIEW.H and VIEW.CPP. After you complete this chapter and the next, these files will contain the application class and the main window class, including its message map.

The complete code for files VIEW.H and VIEW.CPP is given in Listing 1 and Listing 2 at the end of the chapter.

5.2 Work from a Template

This section explains the second step in developing Phone Book from Hello: borrow most of Hello's code for Phone Book, thus using Hello as a template to get you started.

► **To copy and modify the Hello files:**

1. Make a new directory in your \MFC\SAMPLES\TUTORIAL directory.

   You'll use this directory to put together your own files for the tutorial. You might call the directory CHAPTER5, for instance.

2. Copy HELLO.H with the name VIEW.H in your new directory and change the "Hello" filename in the header comment of VIEW.H to "View."
3. Add or modify the header comments in VIEW.H, replacing references to "Hello" with "View."
4. Change all other references to "Hello" in VIEW.H to "View."
5. Add the following lines to VIEW.H:

```
#ifndef __VIEW_H__
#define __VIEW_H__
```

6. Copy HELLO.CPP to VIEW.CPP. In VIEW.CPP, do the following:
   a. Replace the **#include** directives copied from Hello with the following:

```
#include <afxwin>
#include "resource.h"
#include "database.h"
#include "view.h"

extern "C"
{
    #include <commdlg.h>
}

#define SIZESTRING 256
#define SIZENAME 30
#define SIZEPHONE 26
#define PAGESIZE 8

// A simple way to reduce size of C run times
// Disables the use of getenv and argv/argc
extern "C" void _setargv() {}
extern "C" void _setenvp() {}
```

These declarations specify include files, define preprocessor constants, and invoke a simple technique to reduce the size of the run-time libraries that are incorporated into the program. The **extern "C"** directives specify that C code is to be used in a C++ program. The **#include** statement for file COMMDLG.H causes inclusion of code for the common open, save, and print dialog boxes, which are available in COMMDLG.DLL for Windows, versions 3.0 and 3.1.

b. Delete the contents of the `CMainWindow` constructor (but leave the function itself). You'll replace the constructor code later.

c. Remove the present contents of the `OnPaint` member function of class `CMainWindow` (but leave the function itself). You'll replace the contents later. Also edit the header comment for `OnPaint` to remove references to Hello.

d. Replace all references to Hello in VIEW.CPP with "View."

When you complete the steps above, your files will be ready to receive additional code for the Phone Book program. Notice that nearly all of the code copied from Hello to the VIEW files is still perfectly usable, and it gives you a framework into which you can add your Phone Book code.

To continue the tutorial, see the next section.

5.3 Add Dialog Boxes

This section explains the third step in writing Phone Book: arrange for interaction with the user via dialog boxes. This process requires several steps:

1. Write the dialog classes.
2. Write dialog class member functions.
3. Create a resource script file.
4. Add dialog templates to the resource script file.

► To write the dialog classes:

1. Add the following declaration for class `CFindDialog` to your VIEW.H file (if you want your file to match Listing 2 in this chapter, put `CFindDialog` between the existing declarations for classes `CTheApp` and `CMainWindow`):

```
//  CFindDialog
// This dialog is a one line entry field for getting a search
// string to use as a find filter for the database
//
class CFindDialog : public CModalDialog
{
private:
    CString m_szFindName;
```

```
public:
    CFindDialog( CWnd* pParentWnd = NULL )
        : CModalDialog( "Find", pParentWnd )
          { }

    virtual void OnOK();

    CString& GetFindString() { return m_szFindName; }
};
```

2. Add the following declaration for class `CEditDialog` to your VIEW.H file (after class `CFindDialog`):

```
// CEditDialog
// Used to add or edit a Person object.
//
class CEditDialog : public CModalDialog
{
private:
    CPerson* m_pData;

public:
    CEditDialog( CPerson* person, CWnd* pParentWnd = NULL )
        : CModalDialog( "EditPerson", pParentWnd )
        { m_pData = person; }

    virtual BOOL OnInitDialog();
    virtual void OnOK();

    CPerson* GetData()
        { return m_pData; }
};
```

The first kind of dialog box is used to ask the user for the name of a person to search for in the database. This dialog box has an OK button, a Cancel button, and an editable text field for entering a single string. You saw the declaration for class `CFindDialog` above.

The second kind of dialog box is used to display data about a person to the user and to allow the user to edit a person's data. This dialog has an OK button, a Cancel button, three editable text fields, one static text item to display the data's modification date and time, and four static text fields used to label the editable fields. You saw the declaration for class `CEditDialog` above.

You'll add other declarations to VIEW.H later.

► To write dialog class member functions:

1. Add the following member function definition for class `CFindDialog` to your VIEW.CPP file.

```
// CFindDialog::OnOK
// When the user presses OK get the data entered and store it in this
// object.  Then end the dialog.
//
void CFindDialog::OnOK()
{
    GetDlgItemText( IDC_DATA,
       m_szFindName.GetBuffer( SIZESTRING ), SIZESTRING );
    m_szFindName.ReleaseBuffer();
    EndDialog( IDOK );
}
```

2. Add the following member function definitions for class `CEditDialog` to your VIEW.CPP file:

```
// CEditDialog::OnInitDialog
// Fill in the fields with the data placed in this object
// when it was created.
//
BOOL CEditDialog::OnInitDialog()
{
    SetDlgItemText( IDC_LASTNAME,  m_pData -> GetLastName() );
    SetDlgItemText( IDC_FIRSTNAME, m_pData -> GetFirstName() );
    SetDlgItemText( IDC_PHONE, m_pData -> GetPhoneNumber() );
    SetDlgItemText( IDC_MOD, m_pData -> GetModTime().Format("%m/%d/%y
%H:%M" ) );
    SendDlgItemMessage( IDC_LASTNAME, EM_SETSEL );

    return TRUE;
}


// CEditDialog::OnOK
// When OK is pressed set the data to what the user has entered.
//
void CEditDialog::OnOK()
{
    char szTmp[SIZESTRING];

    GetDlgItemText( IDC_LASTNAME, szTmp, sizeof ( szTmp ) );
    m_pData -> SetLastName( szTmp );

    GetDlgItemText( IDC_FIRSTNAME, szTmp, sizeof ( szTmp ) );
    m_pData -> SetFirstName( szTmp );

    GetDlgItemText( IDC_PHONE, szTmp, sizeof ( szTmp ) );
    m_pData -> SetPhoneNumber( szTmp );

    EndDialog( IDOK );
}
```

You'll add other items to VIEW.CPP later.

► **To create a resource script file:**

1. Create a file called PHBOOK.RC and add the following lines (if you choose to copy the HELLO.RC and HELLO.DLG files, you can modify the HELLO.RC file to reflect these contents):

```
#include <windows.h>
#include <afxres.h>
#include "resource.h"
```

   These lines specify the files to include for building the program.

   Listing 2 on pages 245-6 in Chapter 6 defines the resources for Phone Book. You can use the resource files supplied on the distribution disks (PHBOOK.RC, PHBOOK.ICO, and PHBOOK.DLG) or you can use tools to prepare your own resource script file, icon resource file, and dialog resource file. Icon and dialog editing tools are available as part of the Windows Software Development Kit (SDK) or from other sources.

   If you choose to write your own .RC file, you will add an icon resource, a menu resource template, an accelerator resource template, and six dialog resources to your PHBOOK.RC file.

   As you add resources to PHBOOK.RC, you can also add resource ID declarations to a RESOURCE.H file. You can copy this file from the Hello directory and modify it or create your own file from scratch. To see the full contents of this file when it is complete, see Listing 2 in Chapter 6.

   Note The ID numbers used in Listing 2 and throughout Phone Book begin at 101. This is arbitrary, but if you use different numbers you'll need to change them in the code you add as well.

2. Add the following icon resource entry to PHBOOK.RC after the **#include** lines:

```
AFX_IDI_STD_FRAME ICON phbook.ico
```

   This line specifies the name of a file containing the program's icon resource.

3. Copy the menu resource template from Listing 2 on page 245 in Chapter 6 into your PHBOOK.RC file.

   The template you copy is named "MainMenu." It specifies Phone Book's menus.

4. Copy the accelerator resource template from Listing 2 on page 246 in Chapter 6 into your PHBOOK.RC file.

   The accelerator template you copy is named "MainAccelTable." It specifies the shortcut keys the user can press in Phone Book.

▶ To add dialog templates to PHBOOK.RC:

- You can use a dialog editor to create a .DLG file and then refer to that file from your resource script file with a line like this:

```
rcinclude phbook.dlg
```

The dialogs must have the following dialog template names:

"AboutBox"
Name of the dialog template for an About dialog box

"Find"
Name of the dialog template for a Search dialog box

"EditPerson"
Name of the dialog template for a dialog box in which you edit the information for a person

"NoData"
Name of the dialog template for a Help dialog box, first of three

"NoName"
Name of the dialog template for a Help dialog box, second of three

"Enter"
Name of the dialog template for a Help dialog box, third of three

- Alternatively, you can type resource template specifications into PHBOOK.RC as follows.

The About Box dialog template specifies a dialog window caption, text that appears in the dialog, and an OK push button.

```
ABOUTBOX DIALOG 22, 17, 144, 75
STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
CAPTION "About Phone Book"
BEGIN
    CTEXT           "Microsoft Foundation Classes", -1, 0, 2, 144, 8
    CTEXT           "Phone Book Database", -1, 0, 12, 144, 8
    CTEXT           "Version 1.0", -1, 0, 22, 144, 8
    DEFPUSHBUTTON   "OK", IDOK, 56, 56, 32, 14, WS_GROUP
    ICON            AFX_IDI_STD_FRAME, -1, 10, 30, 16, 16
END
```

Figure 5.1 shows the About dialog box.

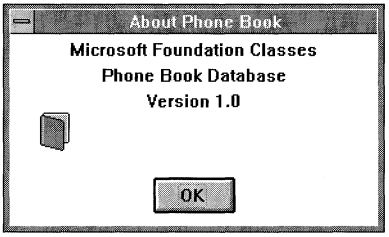


Figure 5.1 Phone Book's About Dialog Box

The Find dialog template specifies a window caption, an editable text field for entering a string, and two push buttons labeled "OK" and "Cancel".

```
FIND DIALOG 22, 17, 90, 50
STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
CAPTION "Enter the last name"
BEGIN
    EDITTEXT        IDC_DATA, 5, 10, 80, 12, ES_AUTOHSCROLL
    DEFPUSHBUTTON   "OK", IDOK, 5, 32, 32, 14, WS_GROUP
    PUSHBUTTON      "CANCEL", IDCANCEL, 42, 32, 32, 14, WS_GROUP
END
```

Figure 5.2 shows the Find dialog box.

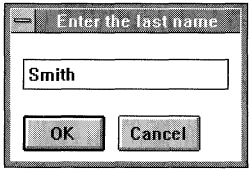


Figure 5.2 Phone Book's Find Dialog Box

The EditPerson dialog template specifies a window caption, three editable text fields for entering or correcting person data, two push buttons, and five static text strings to label the editable fields and to display the last modification date and time for the person being edited.

```
EDITPERSON DIALOG 40, 20, 190, 100
STYLE WS_POPUP | WS_DLGFRAME
BEGIN
    EDITTEXT        IDC_LASTNAME, 55, 10, 100, 12
    EDITTEXT        IDC_FIRSTNAME, 55, 25, 100, 12
    EDITTEXT        IDC_PHONE, 70, 40, 70, 12
```

```
    DEFPUSHBUTTON   "OK", IDOK, 50, 80, 40, 14
    PUSHBUTTON      "Cancel", IDCANCEL, 120, 80, 40, 12
    LTEXT           "Last Name:", IDC_STATICLASTNAME, 5, 11, 41, 12,
                    NOT "First Name:", IDC_STATICFIRSTNAME, 5, 26,
                    41, 12, NOT WS_GROUP
    LTEXT           "Phone Number:", IDC_STATICPHONE, 5, 41, 52, 14,
                    NOT WS_GROUP
    LTEXT           "Last Modified:", IDC_STATICMOD, 10, 60, 50, 14,
                    NOT WS_GROUP
    LTEXT           "", IDC_MOD, 64, 60, 110, 10, NOT WS_GROUP
END
```

Figure 5.3 shows the Edit dialog box.

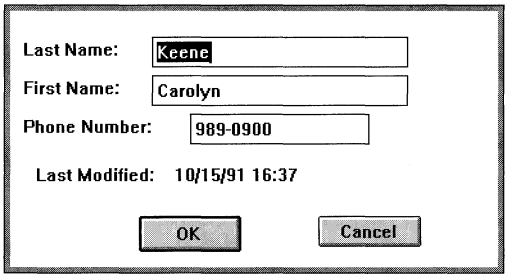


Figure 5.3 Phone Book's Edit Dialog box

The "NoData" dialog template specifies a window caption, one push button, and four lines of static Help text. This dialog template is used when the user has not yet created a database and requests help.

```
NODATA DIALOG 22, 17, 180, 60
STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
CAPTION "Phone Book Help"
BEGIN
    LTEXT           "You have not yet created a database.  In order",
                    -1, 10,5, 180, 8
    LTEXT           "to start using the Phone Book you must first",
                    -1, 10,14, 180, 8
    LTEXT           "either Open an existing database or create",
                    -1, 10, 23,180, 8
    LTEXT           "a New one.", -1, 10, 32, 180, 8
    DEFPUSHBUTTON   "OK", IDOK, 74, 41, 32, 14, WS_GROUP
END
```

Figure 5.4 shows the "No Database" Help dialog box.

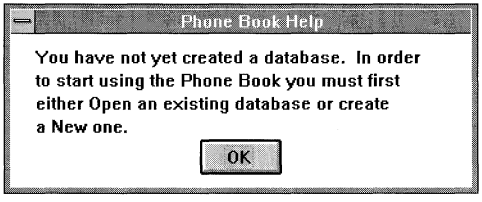


Figure 5.4 Phone Book's "No Database" Help Dialog Box

The NoName dialog template specifies a window caption, two push buttons, and four lines of static Help text. This template is used when the user has an open database and needs instructions for entering data and saving the database.

```
NONAME DIALOG 22, 17, 200, 65
STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
CAPTION "Phone Book Help"
BEGIN
    LTEXT           "You are in data entry mode on an untitled
                    database.", -1, 10, 5, 180, 8
    LTEXT           "Use SaveAs to name the database and save it to
                    the ", -1,10, 14, 180, 8
    LTEXT           "disk.  You can also perform data entry
                    commands.", -1,10, 23, 180, 8
    LTEXT           "Select continue for help on the data entry
                    mode.", -1,10, 32, 180, 8
    DEFPUSHBUTTON   "Continue", IDOK, 44, 43, 32, 14, WS_GROUP
    PUSHBUTTON      "Cancel", IDCANCEL, 120, 43, 32, 14, WS_GROUP
END
```

Figure 5.5 shows the "No Name" Help dialog box.

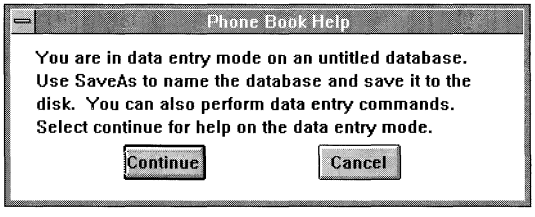


Figure 5.5 Phone Book's "No Name" Help Dialog Box

The Enter dialog template specifies a window caption, one push button, and six lines of static Help text. This template is used when the user needs more information about working with a database. It can be reached by way of the Continue button in the second help dialog.

```
ENTER DIALOG 22, 17, 200, 75
STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
CAPTION "Phone Book Help"
BEGIN
    LTEXT           "Use Person Add to add new people to the list.
                    Select", -1, 10, 5, 180, 8
    LTEXT           "a person with the mouse or arrow keys. With a
                    person", -1, 10, 14, 180, 8
    LTEXT           "selected you can Delete [menu or Delete key] or
                    edit", -1, 10, 23, 180, 8
    LTEXT           "[menu or Enter key]. The Find option will allow
                    you", -1, 10, 32, 180, 8
    LTEXT           "to select a sublist. To display the original
                    list", -1,10, 41, 180, 8
    LTEXT           "select Find All.", -1, 10, 50, 180, 8
    DEFPUSHBUTTON   "OK", IDOK, 84, 59, 32, 14, WS_GROUP
END
```

Figure 5.6 shows the "Enter Data" dialog box.

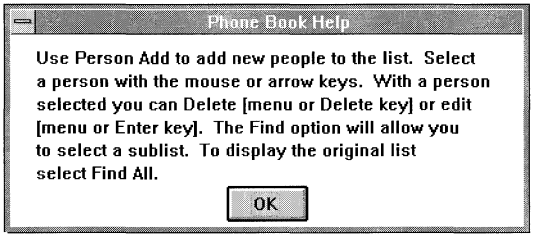


Figure 5.6 Phone Book's "Enter Data" Help Dialog Box

The coding you added is for six dialog resource templates. The resource compiler creates dialog resources from these templates. The first template is for the About dialog box; it's identical to the template in HELLO.RC. The next two templates are for the Find and Edit dialog boxes. The last three templates are for the Help dialogs.

The resource script file is discussed again in "Prepare Supporting Files" on page 242 in Chapter 6. To continue the tutorial, see "Determine What Messages Will be Handled" on page 197 in Chapter 6. For more information about the code you added previously, see "Discussion: Dialog Boxes" on the next page.

Discussion: Dialog Boxes

Windows dialog boxes are used to interact with the user. They convey messages to the user and take user input through various kinds of controls. The Microsoft Foundation Class Library provides two dialog classes that you can use directly or derive your own dialog classes from. This section discusses the dialog classes used in the Phone Book program.

This discussion does not instruct you to add any code to your files.

Modal vs. Modeless Dialog Boxes

Class **CModalDialog** provides "modal" dialog boxes. These are dialog boxes that the user must respond to before he or she can continue work in the underlying application. They put the user in a "mode" that must be dealt with until the dialog box is dismissed, typically with OK or Cancel. It's not a good practice in general to put the user into a limiting mode, but modal dialog boxes are useful in situations that require the user to provide input before the program can continue.

Class **CDialog** provides "modeless" dialogs. When a modeless dialog box is on the screen, the user can treat it like any other application window. The user can continue working in the program, do something in the modeless dialog box, and return to the program without dismissing the dialog box. For many tasks, modeless dialog boxes are preferable to modal dialogs because they don't limit the user's freedom of action.

For more information about class **CDialog**, see Chapter 15 in this manual and the class description in the *Class Libraries Reference*.

The Dialog Classes

The Phone Book program uses two kinds of dialog boxes: standard Windows dialogs and custom-built dialogs based on the Foundation dialog classes.

Some of the dialog boxes used in the Phone Book program are standard Microsoft Windows dialogs. These are displayed and operated by calling a function from COMMDLG.DLL. The Open command in the File menu uses the **GetOpenFileName** function. The Save and Save As commands in the File menu use the **GetSaveFileName** function. The Print command in the File menu uses the **PrintDlg** function. The use of these functions is explained in "Standard Windows Dialog Boxes" on page 167.

Other dialog boxes used in the Phone Book program are based on class **CModalDialog**. You saw the About dialog in Hello, which constructs a dialog object directly from **CModalDialog**. That same dialog code is used for the About dialog box in the Phone Book program. The three Help dialogs in Phone Book also use **CModalDialog** objects. See the discussion of the `OnHelp` member function under "Add Message-Handlers for Help Menu Commands" on page 222 in Chapter 6.

Several other dialog boxes in the Phone Book program are constructed from classes derived from **CModalDialog**. This is because **CModalDialog** doesn't provide member functions to work with any controls you might define in your dialog box, except for the OK and Cancel buttons. Class `CFindDialog` defines a class of dialog box objects designed to get a single string of data from the user. Class `CEditDialog` defines a class of dialog box objects designed to get three pieces of data from the user. These classes add member functions to retrieve any data the user enters in the dialog boxes.

Class CFindDialog

`CFindDialog` is derived from **CModalDialog** to add a constructor, a member variable, and two member functions. Figure 5.2 shows what the `CFindDialog` dialog box typically looks like on the screen.

The member variable of a `CFindDialog` dialog object, `m_szFindName`, stores the data entered in the text box. The `CFindDialog` object's constructor simply invokes the constructor for its base class, **CModalDialog**. The constructor looks like this:

```
CFindDialog( CWnd* pParentWnd = NULL ) :
             CModalDialog( "Find", pParentWnd )
        { };
```

`CFindDialog` inherits the **DoModal** member function from class **CModalDialog**. **DoModal** processes user interactions with the dialog until the user selects either OK or Cancel, then returns the result, either **IDOK** or **IDCANCEL**. You can use the function result to determine which button the user clicked.

If the user clicked the OK button, the `CFindDialog` dialog object's `OnOK` message-handler function is called. In the Phone Book program, the `CFindDialog` object's `OnOK` function overrides **OnOK** from class **CModalDialog**.

The **CModalDialog** version of **OnOK** simply ends the dialog box. The overriding `CFindDialog` extracts the information the user entered in the dialog box and stores it in the dialog object's `m_szFindName` member variable. It does so by calling the **CString** member function **GetBuffer** to allocate a text buffer and then calling the **CDialog** member function **GetDlgItemText** to retrieve the text entered by the user. After retrieving the text, `OnOK` calls the dialog object's **EndDialog** member function, inherited from **CDialog**, to terminate the modal dialog.

If the user clicked the Cancel button, the `CFindDialog` dialog object's `OnCancel` message-handler function is called. In the Phone Book program, **CModalDialog**'s version of **OnCancel** provides default behavior that suits Phone Book's needs. **OnCancel** calls the dialog object's **EndDialog** member function, passing an argument of **FALSE**. This causes the dialog object's **DoModal** member to return **FALSE**.

After the user dismisses your dialog and the **DoModal** member function completes, you can use your own member functions to retrieve the values from the controls in the dialog box. For the `CFindDialog` object, you call the `GetFindString` member function to retrieve the string that the user entered in the dialog box. `GetFindString` returns the string as a CString object.

You build the controls in a dialog box by supplying a resource template in your resource script file. But you must also supply functions in your derived dialog class, such as `GetFindString`, to read the results of those controls. For more about the dialog resource template, see "The Dialog Resources" on page 166.

Class CEditDialog Like `CFindDialog`, `CEditDialog` is derived from **CModalDialog**. The derived class adds member functions to handle dialog controls. These are built into the dialog box via a resource template. Figure 5.3 shows what the `CEditDialog` dialog box looks like on the screen.

`CEditDialog` has one member variable, `m_pData`, to store a pointer to a `CPerson` object. When you construct a `CEditDialog` object, you pass a pointer to a `CPerson` as one of the arguments to the constructor, which installs the `CPerson` in the dialog object's `m_pData` member. If the `CPerson` object passed in has data in its member variables, these are extracted by the `CEditDialog` object's

`OnInitDialog` member function and displayed in the four text fields of the dialog box. If the `CPerson` object is empty, nothing is displayed. You pass a filled `CPerson` object when the dialog box is to be used for editing and an empty `CPerson` object when the dialog box is to be used for data entry.

When the dialog is dismissed, the user has either edited the data and clicked OK or has clicked Cancel. If OK, you can call the `CEditDialog` dialog object's `GetData` member to retrieve the person object and store it in your database.

How Dialog Objects Work

Modal dialog objects, like those in the Phone Book program, are derived from class **CWnd** through class **CDialog**. Some of the functionality of a **CModalDialog** object is inherited from these classes. But **CModalDialog** objects, and objects of classes derived from **CModalDialog**, operate differently from ordinary windows or "modeless" dialog boxes (constructed from class CDialog). For a distinction between modal and modeless dialog boxes, see "Modal vs. Modeless Dialog Boxes" on page 182.

A modal dialog operates when your code calls the dialog object's **DoModal** member function. **DoModal** remains in control until the user dismisses the dialog box by clicking OK, Cancel, or some similar button. The user cannot perform any operations outside the dialog box.

Because a dialog box is a window by inheritance, most Windows messages for the dialog box go through the standard window procedure provided by the Microsoft Foundation Classes for all windows. However, a few special messages, such as **WM_SETFONT** and **WM_INITDIALOG**, instead go through a standard dialog procedure. If a dialog object's dialog procedure receives a **WM_INITDIALOG** message, it calls the dialog object's **OnInitDialog** member function. **CModalDialog** objects inherit this function from class **CDialog**, but classes derived from **CModalDialog** can override it to provide special processing at the time of dialog initialization.

In the Phone Book program, one of the derived **CModalDialog** classes,, `CEditDialog`, overrides **CDialog**'s version of **OnInitDialog** to perform setup chores. Class `CEditDialog` uses its version of `OnInitDialog` to set the initial text that the user sees in the editable text fields of the dialog box and to set the initial selection in the dialog box to the last name field.

The Dialog Resources

When you construct a dialog object, you pass the name of a dialog resource template as an argument. Recall that when you construct an object of a derived class in C++, you append to your constructor call a call to the constructor for your base class. For example, in class `CEditDialog`, the constructor looks like this:

```
CEditDialog( CPerson* person, CWnd* pParentWnd = NULL ) :
                CModalDialog( "EditPerson", pParentWnd )
        { m_pData = person; }
```

In the second line of this code, "EditPerson" is the dialog template name. Windows uses this name as its connection with the dialog template in the resource file, which it uses to display the dialog on the screen and to operate the dialog from the dialog object's **DoModal** member function.

The dialog resource template tells Windows what controls to put in the dialog box, where to put them, what to label them, what styles they have, and what their ID numbers are. This is where, for example, the OK button is defined as a "default push button" placed at coordinates 5, 32 in the dialog window with a height of 32 and a width of 14 units, with the **WS_GROUP** style and the ID number **IDOK**. Some ID numbers, such as **IDOK**, are declared in WINDOWS.H, while others, such as the ID for the editable text field for a person's last name, IDC_LASTNAME, are declared in your own RESOURCE.H file.

For more information about writing and using dialog resource templates, see the Windows SDK documentation.

Message Maps for Dialog Classes

If you look at the hierarchy diagram for the Microsoft Foundation Class Library provided with the *Class Libraries Reference*, you see that **CModalDialog** and its base class, **CDialog**, are derived from class **CWnd**. Thus dialog classes are specialized window classes, and they inherit the member variables and functions of class **CWnd**.

Because dialog classes, including the ones you derive, are window classes, they require message maps to connect Windows messages with the message-handler functions designed to process them. Classes **CDialog** and **CModalDialog**, like all window classes, have their own message maps.

If you provide special message-handler member functions in your derived dialog classes, you must also provide message maps with entries for these handlers.

However, classes `CFindDialog` and `CEditDialog` do not require you to write message maps. This is because you add no message-handler functions to the classes. Each of these classes does override some message-handler functions declared in **CModalDialog**— `OnOK` in both classes and `OnInitDialog` as well in

`CEditDialog`. These are overridden because you need to provide some special processing for these messages. The classes don't override **CModalDialog**'s **OnCancel** member function because the default behavior is sufficient.

Because the classes add no new handlers, they need no new message map entries. Therefore, the message map supplied with **CModalDialog** does the job for them. To save data space in your program, don't supply the message map if you don't have any message handlers.

Standard Windows Dialog Boxes

Phone Book also uses some of the standard Windows dialog boxes. These are the familiar Windows dialog boxes for getting a filename to open or save to and the standard print dialog:

- Standard Open dialog box

  This dialog box lets the user switch directories and select files to open.

  In Phone Book, the `OnOpen` member function of the main window class invokes this dialog box. See the discussion of `OnOpen` under "Discussion: File Menu Message-Handlers" on page 213 in Chapter 6.

- Standard Save dialog box

  This dialog box lets the user switch directories and save a file.

  In Phone Book, the `OnSave` and `OnSaveAs` member functions of the main window class invoke this dialog box. See the discussion of `OnSave` and `OnSaveAs` under "Discussion: File Menu Message-Handlers" on page 213 in Chapter 6.

- Standard Print dialog box

  This dialog lets the user select printing options and print a file.

  In Phone Book, the `OnPrint` member function of the main window class invokes this dialog box. See the discussion of `OnPrint` under "Discussion: File Menu Message-Handlers" on page 213 in Chapter 6.

Note You must include the file COMMDLG.H with an **#include** directive in your VIEW.CPP file. Because you're including a standard C file in a C++ program, place the **#include** directive in an **"extern C"** block. The file COMMDLG.DLL must also be in your PATH when you run the program. COMDLG.DLL is a Windows dynamic link library supplied with the Windows 3.1 SDK and with your Microsoft C/C++, Version 7.0 package. The declaration looks like this:

```
extern "C"
{
    #include <commdlg.h>
}
```

5.4 What's Next

This chapter instructed you to add the dialog boxes that Phone Book uses to communicate with the user. The next chapter instructs you to add the main window class, the message map, and the message-handler member functions that process Windows messages.

5.5 File Listings

The code shown in listing 1 and 2 is available on the distribution disks in files VIEW.H and VIEW.CPP.

Listing 1

```
// view.h : Declares the interfaces to the application and frame window.
//
// This is a part of the Microsoft Foundation Classes C++ library.
// Copyright (C) 1991 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Microsoft Foundation Classes Reference and Microsoft
// QuickHelp documentation provided with the library.
// See these sources for detailed information regarding the
// Microsoft Foundation Classes product.

#ifndef __VIEW_H__
#define __VIEW_H__

/////////////////////////////////////////////////////////////////////////////
// CTheApp
// Derived from CWinApp in order to allow us to override
// the InitInstance member function to create our own window.
//
class CTheApp : public CWinApp
{
public:
    virtual BOOL InitInstance();
};
```

```
//////////////////////////////////////////////////////////////////////////
//  CFindDialog
// This dialog is a one line entry field for getting a search
// string to use as a find filter for the database
//
class CFindDialog : public CModalDialog
{
private:
    CString m_szFindName;

public:
    CFindDialog( CWnd* pParentWnd = NULL )
        : CModalDialog( "Find", pParentWnd )
          { }

virtual void OnOK();

    CString& GetFindString() { return m_szFindName; }
};

//////////////////////////////////////////////////////////////////////////
// CEditDialog
// Used to add or edit a Person object.
//
class CEditDialog : public CModalDialog
{
private:
    CPerson* m_pData;

public:
    CEditDialog( CPerson* person, CWnd* pParentWnd = NULL )
        : CModalDialog( "EditPerson", pParentWnd )
        { m_pData = person; }

    virtual BOOL OnInitDialog();
    virtual void OnOK();

    CPerson* GetData()
        { return m_pData; }
};
```

```
////////////////////////////////////////////////////////////////////////////
// CMainWindow
// The window object that WinApp creates. In this program we
// only use one window class. In that sense this object does
// all the work that makes our window a CPersonList viewer.
//
class CMainWindow : public CFrameWnd
{
private:
    // Variables that contain the window size, font size and scroll
    // position.
    int m_cxChar;
    int m_cyChar;
    int m_nHscrollPos;
    int m_nVscrollPos;
    int m_cxCaps;
    int m_nMaxWidth;
    int m_cxClient;
    int m_cyClient;
    int m_nVscrollMax;
    int m_nHscrollMax;
    int m_nSelectLine;
    CDataBase m_people;

// Private helpers for the other routines.
    void SetMenu();
    BOOL Save( BOOL bNamed=FALSE );
    BOOL FileDlg( BOOL bOpen, int nMaxFile, LPSTR szFile,,
            int nMaxFileTitle, LPSTR szFileTitle );
    BOOL CheckForSave( const char* pszTitle, const char* pszMessage );
    void InvalidateLine();

public:
    // The CMainWindow constructor
    CMainWindow();

// These routines are all overrides of CWnd.  Windows messages
    // cause these to be called.
    afx_msg int  OnCreate( LPCREATESTRUCT cs );
    afx_msg void OnClose();
    afx_msg void OnSize( UINT type, int x, int y );
    afx_msg void OnHScroll( UINT nSBCode, UINT pos, CWnd* control );
    afx_msg void OnVScroll( UINT nSBCode, UINT pos, CWnd* control );
    afx_msg void OnLButtonDown( UINT wParam, CPoint location );
    afx_msg void OnLButtonDblClk( UINT wParam, CPoint location );
    afx_msg void OnKeyDown( UINT wParam, UINT, UINT );
    afx_msg void OnPaint();
```

```
// These routines are all menu items.  User action causes
    // these to be called.
    afx_msg void OnNew();
    afx_msg void OnOpen();
    afx_msg void OnSave();
    afx_msg void OnSaveAs();
    afx_msg void OnDBClose();
    afx_msg void OnPrint();
    afx_msg void OnExit();
    afx_msg void OnAdd();
    afx_msg void OnDelete();
    afx_msg void OnFind();
    afx_msg void OnFindAll();
    afx_msg void OnEdit();
    afx_msg void OnHelp();
    afx_msg void OnAbout();
    afx_msg void OnUp();
    afx_msg void OnDown();

DECLARE_MESSAGE_MAP()
};

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

#endif // __VIEW_H__
```

Listing 2

```
// view.cpp : Defines the behaviors for the application and frame window.
//
// This is a part of the Microsoft Foundation Classes C++ library.
// Copyright (C) 1991 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Microsoft Foundation Classes Reference and Microsoft
// QuickHelp documentation provided with the library.
// See these sources for detailed information regarding the
// Microsoft Foundation Classes product.

#include <afxwin.h>

#include "resource.h"
#include "database.h"
#include "view.h"

extern "C"
{
    #include <commdlg.h>
}
```

```
#define SIZESTRING 256
#define SIZENAME 30
#define SIZEPHONE 26
#define PAGESIZE 8

// a simple way to reduce size of C run times
// disables the use of getenv and argv/argc
extern "C" void _setargv() { }
extern "C" void _setenvp() { }

////////////////////////////////////////////////////////////////////////////
// Create the single global instance of the database viewer app.

CTheApp PersonApp;

////////////////////////////////////////////////////////////////////////////
// CTheApp

/////////////////////////////////////////////////
// CTheApp::InitInstance
// Override InitInstance function to create a CMainWindow object
// and display it on the screen
//
BOOL CTheApp::InitInstance()
{
    m_pMainWnd = new CMainWindow();
    m_pMainWnd -> ShowWindow( m_nCmdShow );
    m_pMainWnd -> UpdateWindow();
    return TRUE;
}

////////////////////////////////////////////////////////////////////////////
// CFindDialog

/////////////////////////////////////////////////
// CFindDialog::OnOK
// When the user hits OK get the data entered and store it in this
// object.  Then end the dialog.
//
void CFindDialog::OnOK()
{
    GetDlgItemText( IDM_PDATA, m_szFindName.GetBuffer( SIZESTRING ), SIZESTRING );
    m_szFindName.ReleaseBuffer();
    EndDialog( IDOK );
}

////////////////////////////////////////////////////////////////////////////
// CEditPerson
```

```
////////////////////////////////////////////////
// CEditDialog::OnInitDialog
// Fill in the fields with the data placed in this object
// when it was created.
//
BOOL CEditDialog::OnInitDialog()
{
    SetDlgItemText( IDC_LASTNAME,  m_pData -> GetLastName() );
    SetDlgItemText( IDC_FIRSTNAME, m_pData -> GetFirstName() );
    SetDlgItemText( IDC_PHONE, m_pData -> GetPhoneNumber() );
    SetDlgItemText( IDC_MOD, m_pData -> GetModTime().Format("%m/%d/%y %H:%M" ) );
    SendDlgItemMessage( IDC_LASTNAME, EM_SETSEL );

return TRUE;
}

////////////////////////////////////////////////
// CEditDialog::OnOK
// When OK is pressed set the data to what the user has entered.
//
void CEditDialog::OnOK()
{
    char szTmp[SIZESTRING];

    GetDlgItemText( IDC_LASTNAME, szTmp, sizeof ( szTmp ) );
    m_pData -> SetLastName( szTmp );

    GetDlgItemText( IDC_FIRSTNAME, szTmp, sizeof ( szTmp ) );
    m_pData -> SetFirstName( szTmp );

    GetDlgItemText( IDC_PHONE, szTmp, sizeof ( szTmp ) );
    m_pData -> SetPhoneNumber( szTmp );

    EndDialog( IDOK );
}

////////////////////////////////////////////////////////////////////////////
// CMainWindow

BEGIN_MESSAGE_MAP( CMainWindow, CFrameWnd )

// File menu commands:
    ON_COMMAND( IDM_NEW, OnNew )
    ON_COMMAND( IDM_OPEN, OnOpen )
    ON_COMMAND( IDM_SAVE, OnSave )
    ON_COMMAND( IDM_SAVEAS, OnSaveAs )
    ON_COMMAND( IDM_CLOSE, OnDBClose )
    ON_COMMAND( IDM_PRINT, OnPrint )
    ON_COMMAND( IDM_EXIT, OnExit )
```

```
// Person menu commands:
    ON_COMMAND( IDM_ADD, OnAdd )
    ON_COMMAND( IDM_DELETE, OnDelete )
    ON_COMMAND( IDM_EDIT, OnEdit )
    ON_COMMAND( IDM_FIND, OnFind )
    ON_COMMAND( IDM_FINDALL, OnFindAll )

// Help menu commands:
    ON_COMMAND( IDM_HELP, OnHelp )
    ON_COMMAND( IDM_ABOUT, OnAbout )

// Selection accelerators:
    ON_COMMAND( VK_UP, OnUp )
    ON_COMMAND( VK_DOWN, OnDown )

// Other Windows messages:
    ON_WM_CREATE()
    ON_WM_CLOSE()
    ON_WM_SIZE()
    ON_WM_VSCROLL()
    ON_WM_HSCROLL()
    ON_WM_LBUTTONDOWN()
    ON_WM_LBUTTONDBLCLK()
    ON_WM_KEYDOWN()
    ON_WM_PAINT()
END_MESSAGE_MAP()

////////////////////////////////////////////////////////////////////////////
// CMainWindow::CMainWindow
// Constructs and initializes a CMainWindow object
//
CMainWindow::CMainWindow()
{
    VERIFY( LoadAccelTable( "MainAccelTable" ) );
    VERIFY( Create( NULL, "Phone Book",,
                    WS_OVERLAPPEDWINDOW, rectDefault, NULL, "MainMenu" ) );
    m_nSelectLine = -1;
}

////////////////////////////////////////////////////////////////////////////
// The Following are CMainWindow Menu Items
```

```
////////////////////////////////////////////////
// CMainWindow::OnNew
// After checking to see if current data needs to be stored, call
// database New and resize/repaint the window.
//
void CMainWindow::OnNew()
{
    if( !CheckForSave( "File New", "Save file before New?" ) )
        return;    m_people.New();
    SetMenu();
    SetWindowText( m_people.GetTitle() );
    OnSize( 0, m_cxClient, m_cyClient );
}

////////////////////////////////////////////////
// CMainWindow::OnOpen
//
void CMainWindow::OnOpen()
{
    if( !CheckForSave( "File Open", "Save file before Open?" ) )
        return;
```

```
// Attempt to open a database file and read it.
    // If a file or archive exception occurs, catch it and
    // present an error message box.
    CString szFileName, szFileTitle;
    TRY
    {
        // Use CommDlg to get the file name and then call DoOpen.
        // Set the Window title and menus.  Resize/Repaint.
        if ( FileDlg( TRUE, SIZESTRING, szFileName.GetBuffer( SIZESTRING ),
                SIZESTRING, szFileTitle.GetBuffer( SIZESTRING ) ) )
        {
            szFileName.ReleaseBuffer();
            szFileTitle.ReleaseBuffer();
            m_people.DoOpen( szFileName );
            m_people.SetTitle( szFileTitle );
            SetWindowText( m_people.GetTitle() );
            SetMenu();
            OnSize( 0, m_cxClient, m_cyClient );
        }
    }
    CATCH( CFileException, e )
    {
        char ErrorMsg[25];
        sprintf( ErrorMsg,"Opening %s returned a 0x%lx.",
                (const char*)szFileTitle, e -> m_lOsError );
        MessageBox( ErrorMsg, "File Open Error" );
    }
    AND_CATCH( CArchiveException, e )
    {
        char ErrorMsg[25];
        sprintf( ErrorMsg,"Reading the %s archive failed.",
                (const char*)szFileTitle );
        MessageBox( ErrorMsg, "File Open Error" );
    }
    END_CATCH
}

////////////////////////////////////////////////
// CMainWindow::OnSave
//
void CMainWindow::OnSave()
{
    Save( m_people.IsNamed() );
}

////////////////////////////////////////////////
// CMainWindow::OnSaveAs
//
void CMainWindow::OnSaveAs()
{
    Save();
}
```

```
////////////////////////////////////////////////
// CMainWindow::OnDBClose
// Closes the current database, checking to see if it should be
// saved first.  Reset the window title and the scroll regions.
// Invalidating the entire screen causes OnPaint to repaint but
// this time without any data.
//
void CMainWindow::OnDBClose()
{
    if( !CheckForSave( "File Close", "Save file before closing?" ) )
        return;
    m_people.Terminate();
    SetWindowText( "Phone Book" );
    SetMenu();
    OnSize( 0, m_cxClient, m_cyClient );
}

////////////////////////////////////////////////
// CMainWindow::OnPrint
// Uses the commdlg print dialog to create a printer dc
// Then it uses code almost identical to the OnPaint code
// to write the data to the printer.
//
void CMainWindow::OnPrint()
{
    PRINTDLG pd;

    pd.lStructSize = sizeof( PRINTDLG );
    pd.hwndOwner=m_hWnd;
    pd.hDevMode=(HANDLE)NULL;
    pd.hDevNames=(HANDLE)NULL;
    pd.Flags=PD_RETURNDC | PD_NOSELECTION | PD_NOPAGENUMS;
    pd.nFromPage=0;
    pd.nToPage=0;
    pd.nMinPage=0;
    pd.nMaxPage=0;
    pd.nCopies=1;
    pd.hInstance=(HANDLE)NULL;

if ( PrintDlg( &pd ) != 0 )
    {
        // CommDlg returned a DC so create a CDC object from it.
        ASSERT( pd.hDC != 0 );
        CDC * dc;
        dc = CDC::FromHandle( pd.hDC );

// Change to hourglass while printing
        SetCursor( AfxGetApp() -> LoadStandardCursor( IDC_WAIT ) );
```

```
// Begin printing the document.
        int rc;
        char szError[50];
        rc = dc -> StartDoc( "Phone Book" );
        if ( rc < 0 )
        {
            sprintf( szError, "Unable to Begin printing - Error[%d]", rc );
            MessageBox( szError, NULL,MB_OK );
            return;
        }

        int x, y;
        CPerson* pCurrent;
        UINT nPerson=0;
        CString szDisplay;
        int nStart, nEnd;

        // Get Height and Width of large character
        CSize extentChar = dc -> GetTextExtent( "M", 1 );
        int nCharHeight = extentChar.cy;
        int nCharWidth = extentChar.cx;

        // Get Page size in # of full lines
        UINT nExtPage = ( dc -> GetDeviceCaps(VERTRES) - nCharHeight )
                / nCharHeight;

        CString pszTitle;
        pszTitle = CString( "Phone Book - " ) + m_people.GetName();

        while ( nPerson != m_people.GetCount() )
        {
        // Print a Page Header
            dc -> StartPage();
            dc -> SetTextAlign ( TA_LEFT | TA_TOP );
            dc -> TextOut( 0, 0, pszTitle, pszTitle.GetLength() );
            dc -> MoveTo( 0, nCharHeight );
            dc -> LineTo( dc -> GetTextExtent( pszTitle, pszTitle.GetLength() ).cx
                    nCharHeight );

        // Print People from start to last person or page size minus
        // 2 ( header size )
            nEnd = min( m_people.GetCount() - nPerson, nExtPage-2 );
            for ( nStart = 0; nStart < nEnd; nStart++, nPerson++ )
            {
                x = 0;
                y = nCharHeight * ( nStart+2 );,

pCurrent = m_people.GetPerson( nPerson );
                szDisplay = " " + pCurrent -> GetLastName() + ", " +
                        pCurrent -> GetFirstName();
                dc -> SetTextAlign( TA_LEFT | TA_TOP );
                dc -> TextOut( x, y, szDisplay, szDisplay.GetLength() );
```

```
szDisplay = pCurrent -> GetPhoneNumber();
                dc -> SetTextAlign( TA_RIGHT | TA_TOP );
                dc -> TextOut( x + SIZENAME * nCharWidth, y, szDisplay,
                        szDisplay.GetLength() );

szDisplay = pCurrent -> GetModTime().Format( "%m/%d/%y %H:%M" );
                dc -> TextOut( x + ( SIZENAME + SIZEPHONE ) * nCharWidth, y,
                    szDisplay, szDisplay.GetLength() );
            }
            dc -> EndPage();
        }
        dc -> EndDoc();
        dc -> DeleteDC();
        SetCursor( AfxGetApp() -> LoadStandardCursor( IDC_ARROW ) );
    }
}

////////////////////////////////////////////////
// CMainWindow::OnExit
//
void CMainWindow::OnExit()
{
    OnClose();
}

////////////////////////////////////////////////
// CMainWindow::OnAdd
// Using the EditDialog fill in a new person object.  If the user
// selects OK then add the person, call OnSize to resize the scroll
// region, and invalidate the screen so it will be redrawn with the
// new person in the correct order.
//
void CMainWindow::OnAdd()
{
    CPerson* person=new CPerson();

    CEditDialog dlgAdd( person, this );
    if ( dlgAdd.DoModal() == IDOK )
    {
        m_people.AddPerson( person );
        OnSize( 0, m_cxClient, m_cyClient );
    }
    else
        delete person;
}
```

```
////////////////////////////////////////////////
// CMainWindow::OnDelete
// Deletes the current selection.  Check to see if the selection is
// now past then end of the list.  Also call OnSize since the list
// length has now changed.
//
void CMainWindow::OnDelete()
{
    if ( m_nSelectLine == -1 )
    {
        MessageBox( "Select a person to delete first" );
        return;
    }
    m_people.DeletePerson( m_nSelectLine );
    if ( m_nSelectLine >= (int)m_people.GetCount() )
        m_nSelectLine--;
    OnSize( 0, m_cxClient, m_cyClient );
}

////////////////////////////////////////////////
// CMainWindow::OnFind
// Gets information from the CFindDialog modal dialog box, then searches for
// matching people.  Note the Add and Delete menu items are disabled after
// a find is made.  Find All is enabled.
//
void CMainWindow::OnFind()
{
    CFindDialog dlgFind( this );
    if ( dlgFind.DoModal() == IDOK &&
                          dlgFind.GetFindString().GetLength() != 0 )
    {
        if ( m_people.DoFind( dlgFind.GetFindString() ) )
        {
            m_nSelectLine = -1;
            CString tmp;
            tmp = m_people.GetTitle() + " Found: "
                    + dlgFind.GetFindString();
            SetWindowText( tmp );
            CMenu* pMenu = GetMenu();
            pMenu -> EnableMenuItem( IDM_FINDALL, MF_ENABLED );
            pMenu -> EnableMenuItem( IDM_DELETE, MF_GRAYED );
            pMenu -> EnableMenuItem( IDM_ADD, MF_GRAYED );
            OnSize( 0, m_cxClient, m_cyClient );
        }
        else
            MessageBox( "No match found in list." );
    }
}
```

```
////////////////////////////////////////////////////
// CMainWindow::OnFindAll
// Returns to view the whole database.  Add, Delete are re-enabled, and
// Find All is again disabled.  OnSize is called because the list
// has changed length.
//
void CMainWindow::OnFindAll()
{
    m_people.DoFind();
    SetWindowText( m_people.GetTitle() );
    CMenu* pMenu = GetMenu();
    pMenu -> EnableMenuItem( IDM_FINDALL, MF_GRAYED );
    pMenu -> EnableMenuItem( IDM_DELETE, MF_ENABLED );
    pMenu -> EnableMenuItem( IDM_ADD, MF_ENABLED );
    OnSize( 0, m_cxClient, m_cyClient );
}

////////////////////////////////////////////////////
// CMainWindow::OnEdit
// Using the member variable m_nSelectLine a CEditDialog is created
// and filled with the selected person.  If the dialog OK button is
// used the dialog saves the changes into the object.
// over any old information.
//
void CMainWindow::OnEdit()
{
    if ( m_nSelectLine == -1 )
    {
        MessageBox( "Select a person to edit first" );
        return;
    }

// Get a pointer to the person in the list.
    CPerson* pPerson = m_people.GetPerson( m_nSelectLine );
    CPerson tmpPerson = *pPerson;

    //Edit the data.
    CEditDialog dlgEdit( &tmpPerson, this );

    //if the ok button is pressed redraw the screen
    if ( dlgEdit.DoModal() == IDOK )
    {
        m_people.ReplacePerson( pPerson, tmpPerson );
        InvalidateLine();
    }
}
```

```
////////////////////////////////////////////////
// CMainWindow::OnHelp
//
void CMainWindow::OnHelp()
{
    if ( !m_people.IsPresent() )
    {
        CModalDialog dlgHelp( "NoData", this );
        dlgHelp.DoModal();
        return;
    }

    if ( !m_people.IsNamed() )
    {
        CModalDialog dlgHelp( "NoName", this );
        if ( dlgHelp.DoModal() == IDCANCEL )
            return;
    }

    CModalDialog dlgHelp( "Enter", this );
    dlgHelp.DoModal();
}

////////////////////////////////////////////////
// CMainWindow::OnAbout
//
void CMainWindow::OnAbout()
{
    CModalDialog dlgAbout( "AboutBox", this );
    dlgAbout.DoModal();
}

////////////////////////////////////////////////////////////////////////////
// The Following are WINDOW messages

////////////////////////////////////////////////
// CMainWindow::OnCreate
// Queries the current text metrics to determine char size.
//
int CMainWindow::OnCreate( LPCREATESTRUCT )
{
    TEXTMETRIC tm;

// Get the text metrics.
    CDC* dc = GetDC();
    dc -> GetTextMetrics( &tm );
    ReleaseDC( dc );
```

```
// Decide the statistics on how many rows, etc., we can display.
    m_cxChar = tm.tmAveCharWidth;
    m_cxCaps = ( (tm.tmPitchAndFamily & 1 )? 3 : 2 ) * m_cxChar / 2;
    m_cyChar = tm.tmHeight + tm.tmExternalLeading;
    m_nMaxWidth = ( SIZENAME + SIZEPHONE + 1 ) * m_cxCaps;
    m_nVscrollPos = m_nHscrollPos = 0;

return 0;
}

////////////////////////////////////////////////
// CMainWindow::OnClose
// Check to see if the current file needs to be saved.  Terminate
// the database and destroy the window.
//
void CMainWindow::OnClose()
{
    if( !CheckForSave( "File Exit", "Save file before exit?" ) )
        return;
    m_people.Terminate();
    DestroyWindow();
}

////////////////////////////////////////////////
// CMainWindow::OnSize
// When resized, we need to recalculate our scrollbar ranges based on what
// part of the database is visible.
//
void CMainWindow::OnSize( UINT, int x, int y )
{
    m_cxClient = x;
    m_cyClient = y;

m_nVscrollMax = max( 0,
            (int)( m_people.GetCount() ) - m_cyClient / m_cyChar );
    m_nVscrollPos = min( m_nVscrollPos, m_nVscrollMax );

SetScrollRange( SB_VERT, 0, m_nVscrollMax, FALSE );
    SetScrollPos( SB_VERT, m_nVscrollPos, TRUE );

m_nHscrollMax = max( 0, ( m_nMaxWidth - m_cxClient ) / m_cxChar );
    m_nHscrollPos = min( m_nHscrollPos, m_nHscrollMax );

SetScrollRange( SB_HORZ, 0, m_nHscrollMax, FALSE );
    SetScrollPos( SB_HORZ, m_nHscrollPos, TRUE );
    Invalidate( TRUE );
}
```

```
////////////////////////////////////////////////
// CMainWindow::OnVScroll
// Translate scroll messages into Scroll increments and then
// checks the current position to determine if scrolling is possible
//
void CMainWindow::OnVScroll( UINT wParam, UINT pos, CWnd* )
{
    short nScrollInc;

    switch ( wParam )
    {
        case SB_TOP:
            nScrollInc = -m_nVscrollPos;
            break;

case SB_BOTTOM:
            nScrollInc = m_nVscrollMax - m_nVscrollPos;
            break;

        case SB_LINEUP:
            nScrollInc = -1;
            break;

case SB_LINEDOWN:
            nScrollInc = 1;
            break;

case SB_PAGEUP:
            nScrollInc = min( -1, -m_cyClient / m_cyChar );
            break;

case SB_PAGEDOWN:
            nScrollInc = max( 1, m_cyClient / m_cyChar );
            break;

case SB_THUMBTRACK:
            nScrollInc = pos - m_nVscrollPos;
            break;

default:
            nScrollInc = 0;
    }

if ( nScrollInc = max( -m_nVscrollPos,
            min( nScrollInc, m_nVscrollMax - m_nVscrollPos ) ) )
    {
        m_nVscrollPos += nScrollInc;
        ScrollWindow( 0, -m_cyChar * nScrollInc );
        SetScrollPos( SB_VERT, m_nVscrollPos );
        UpdateWindow();
    }
}
```

```
////////////////////////////////////////////////
// CMainWindow::OnHScroll
// Translate scroll messages into Scroll increments and then
// checks the current position to determine if scrolling is possible
//
void CMainWindow::OnHScroll( UINT wParam, UINT pos, CWnd* )
{
    int nScrollInc;
    switch ( wParam )
    {
        case SB_LINEUP:
            nScrollInc = -1;
            break;

case SB_LINEDOWN:
            nScrollInc = 1;
            break;

case SB_PAGEUP:
            nScrollInc = -PAGESIZE;
            break;

case SB_PAGEDOWN:
            nScrollInc = PAGESIZE;
            break;

case SB_THUMBPOSITION:
            nScrollInc = pos - m_nHscrollPos;
            break;

default:
            nScrollInc = 0;
    }

if ( nScrollInc = max( -m_nHscrollPos,
             min( nScrollInc, m_nHscrollMax - m_nHscrollPos ) ) )
    {
        m_nHscrollPos += nScrollInc;
        ScrollWindow( -m_cxChar * nScrollInc, 0 );
        SetScrollPos( SB_HORZ, m_nHscrollPos );
       UpdateWindow();
    }
}
```

```
////////////////////////////////////////////////
// CMainWindow::OnUp
// Uses Accelerator tables to link the up arrow key to this
// routine.  Decrements the select line with checking for scrolling
// and wrapping off the top of the list.
//
//
void CMainWindow::OnUp()
{
    InvalidateLine();

    if ( m_nSelectLine <= 0 )
    {
        m_nSelectLine = m_people.GetCount() - 1;
        m_nVscrollPos = max( 0, m_nSelectLine + 1 - ( m_cyClient / m_cyChar ) );
        Invalidate( TRUE );
    }
    else
    {
        m_nSelectLine--;
        if ( m_nSelectLine - m_nVscrollPos < 0 )
            OnVScroll( SB_LINEUP, 0, NULL );

        // Selection is off the screen
        if ( m_nSelectLine - m_nVscrollPos > ( m_cyClient / m_cyChar ) )
        {
            m_nVscrollPos = m_nSelectLine + 1 - ( m_cyClient / m_cyChar );
            SetScrollPos( SB_VERT, m_nVscrollPos, TRUE );
            Invalidate( TRUE );
        }
        if ( m_nSelectLine - m_nVscrollPos < 0 )
        {
            m_nVscrollPos = m_nSelectLine;
            SetScrollPos( SB_VERT, m_nVscrollPos, TRUE );
            Invalidate( TRUE );
        }
    }


InvalidateLine();
}
```

```
////////////////////////////////////////////////
// CMainWindow::OnDown
// Uses Accelerator tables to link the down arrow key to this
// routine.  Inc the select line with checking for scrolling
// and wrapping off the bottom of the list.
//
void CMainWindow::OnDown()
{
    InvalidateLine();

    if ( m_nSelectLine == (int)( m_people.GetCount() - 1 )
            || m_nSelectLine == -1 )
    {
        m_nSelectLine = 0;
        m_nVscrollPos = 0;
        Invalidate( TRUE );
    }
    else
    {
        m_nSelectLine++;
        if ( ( m_nSelectLine - m_nVscrollPos + 1 ) > ( m_cyClient / m_cyChar ) )
            OnVScroll( SB_LINEDOWN, 0, NULL );

        // Selection is off the screen
        if ( ( m_nSelectLine - m_nVscrollPos ) > ( m_cyClient / m_cyChar ) )
        {
            m_nVscrollPos = m_nSelectLine + 1 - ( m_cyClient / m_cyChar );
            SetScrollPos( SB_VERT, m_nVscrollPos, TRUE );
            Invalidate( TRUE );
        }
        if ( ( m_nSelectLine - m_nVscrollPos ) < 0 )
        {
            m_nVscrollPos = m_nSelectLine;
            SetScrollPos( SB_VERT, m_nVscrollPos, TRUE );
            Invalidate( TRUE );
        }
    }

InvalidateLine();
}
```

```
////////////////////////////////////////////////
// CMainWindow::OnLButtonDown
// Turns the location of the mouse pointer into a line number
// and stores that information in m_nSelectLine. Uses
// InvalidateLine to cause OnPaint to change the screen.
//
void CMainWindow::OnLButtonDown( UINT, CPoint location )
{
    InvalidateLine();

    int pos = m_nVscrollPos + location.y / m_cyChar;

    if ( ( m_nSelectLine != pos )  && ( pos < (int)m_people.GetCount() ) )
    {
        m_nSelectLine = pos;
        InvalidateLine();
    }
    else
        m_nSelectLine = -1;
}

////////////////////////////////////////////////
// CMainWindow::OnLButtonDblClk
// Translates mouse left button double click into edit person.
//
void CMainWindow::OnLButtonDblClk( UINT wParam, CPoint location )
{
    if ( m_nSelectLine == -1 )
        OnLButtonDown( wParam, location );
    OnEdit();
}
```

```
////////////////////////////////////////////////
// CMainWindow::OnKeyDown
// Translates keyboard input into scroll messages
//
void CMainWindow::OnKeyDown( UINT wParam, UINT, UINT )
{
    switch ( wParam )
    {
        case VK_HOME:
            OnVScroll( SB_TOP, 0, NULL );
            break;
        case VK_END:
            OnVScroll( SB_BOTTOM, 0, NULL );
            break;
        case VK_PRIOR:
            OnVScroll( SB_PAGEUP, 0, NULL );
            break;
        case VK_NEXT:
            OnVScroll( SB_PAGEDOWN, 0, NULL );
            break;
        case VK_LEFT:
            OnHScroll( SB_PAGEUP, 0, NULL );
            break;
        case VK_RIGHT:
            OnHScroll( SB_PAGEDOWN, 0, NULL );
            break;
    }
}

////////////////////////////////////////////////
// CMainWindow::OnPaint
// This routine does all the painting for the screen.
//
void CMainWindow::OnPaint()
{

    CPaintDC dc( this );

// Set the Text and background colors for the DC; also create a Brush
    CBrush bBack;
    dc.SetTextColor( GetSysColor( COLOR_WINDOWTEXT ) );
    dc.SetBkColor( GetSysColor( COLOR_WINDOW ) );
    bBack.CreateSolidBrush( GetSysColor( COLOR_WINDOW ) );

// Compute the lines that need to be redrawn
    int nStart = max( 0, m_nVscrollPos + dc.m_ps.rcPaint.top / m_cyChar - 1 );
    int nEnd = min( (int)m_people.GetCount(),
                m_nVscrollPos + ( dc.m_ps.rcPaint.bottom / m_cyChar+1 ) );
```

```
// Create a rect the width of the display.
    CRect area( 0, 0, m_cxClient, 0 );

    CString szDisplay;
    CPerson* pCurrent;
    int x,y;
    for ( ;nStart < nEnd; nStart++ )
    {
        // if the current line is the select line then change the
        // colors to the highlight text colors.
        if ( m_nSelectLine == nStart )
        {
            bBack.DeleteObject();
            bBack.CreateSolidBrush( GetSysColor( COLOR_HIGHLIGHT ) );
            dc.SetTextColor( GetSysColor( COLOR_HIGHLIGHTTEXT ) );
            dc.SetBkColor( GetSysColor( COLOR_HIGHLIGHT ) );
        }

// x is the number of characters horz scrolled * the width of
        // char.  y is the current line no. - number of lines scrolled
        // times the height of a line.
        x = m_cxChar * ( -m_nHscrollPos );
        y = m_cyChar * ( nStart - m_nVscrollPos );

// Set the rect to y and y + the height of the line.  Fill the
        // rect with the background color.
        area.top = y;
        area.bottom = y+ m_cyChar;
        dc.FillRect( area, &bBack );

// Get the person and build a string with his name.
        pCurrent = m_people.GetPerson( nStart );
        szDisplay = " " + pCurrent -> GetLastName() + ", " +
                pCurrent -> GetFirstName();

// Set the dc to write using the point as the left top of the
        // character.  Write the name.
        dc.SetTextAlign( TA_LEFT | TA_TOP );
        dc.TextOut ( x, y, szDisplay, szDisplay.GetLength() );

// Write the phone number right aligned.
        szDisplay = pCurrent -> GetPhoneNumber();
        dc.SetTextAlign ( TA_RIGHT | TA_TOP );
        dc.TextOut ( x + SIZENAME * m_cxCaps, y, szDisplay,
            szDisplay.GetLength() );

// Write the time.
        szDisplay = pCurrent -> GetModTime().Format( "%m/%d/%y %H:%M" );
        dc.TextOut ( x + ( SIZENAME + SIZEPHONE ) * m_cxCaps, y,
            szDisplay, szDisplay.GetLength() );
```

```
// If this is the select line then we need to reset the dc
        // colors back to the original colors.
        if ( m_nSelectLine == nStart )
        {
            bBack.DeleteObject();
            bBack.CreateSolidBrush( GetSysColor( COLOR_WINDOW ) );
            dc.SetTextColor( GetSysColor( COLOR_WINDOWTEXT ) );
            dc.SetBkColor( GetSysColor( COLOR_WINDOW ) );
        }
    }
}

////////////////////////////////////////////////////////////////////////
// The following are utility routines

////////////////////////////////////////////////
// CMainWindow::FileDlg
// Call the commdlg routine to display File Open or File Save As
// dialogs. The setup is the same for either. If bOpen is TRUE
// then File Open is displayed otherwise File Save As is displayed.
// The File Name and File Title are stored at the string pointer
// passed in.
//
BOOL CMainWindow::FileDlg( BOOL bOpen, int nMaxFile, LPSTR szFile,
        int nMaxFileTitle, LPSTR szFileTitle )
{

    OPENFILENAME of;

char szDirName[SIZESTRING];
    char * szFilter[] =
        "Phone Book Files (*.pb)\0"
        "*.pb\0"
        "\0";

szDirName[0] = '.';
```

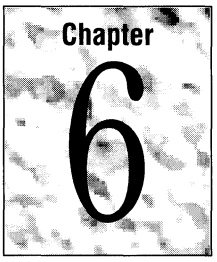
```
of.lStructSize = sizeof( OPENFILENAME );
    of.hwndOwner = m_hWnd;
    of.lpstrFilter = (LPSTR)szFilter;
    of.lpstrCustomFilter = (LPSTR)NULL;
    of.nMaxCustFilter = 0L;
    of.nFilterIndex = 1L;
    of.lpstrFile=szFile;
    of.nMaxFile=nMaxFile;
    of.lpszFileTitle = szFileTitle;
    of.nMaxFileTitle = nMaxFileTitle;
    of.lpstrInitialDir = szDirName;
    of.lpstrTitle = (LPSTR)NULL;
    of.nFileOffset = 0;
    of.nFileExtension = 0;
    of.lpstrDefExt = (LPSTR)"pb";
    if ( bOpen )
    {
        of.Flags = OFN_HIDEREADONLY;
        return GetOpenFileName( &of );
    }
    else
    {
        of.Flags = OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT;
        return GetSaveFileName( &of );
    }
}
```

```
////////////////////////////////////////////////
// CMainWindow::Save
// Handles any time a file needs to be saved to the disk.
// Passing in FALSE for name brings up the file save as dialog
// whether or not the database had a name before.
//
BOOL CMainWindow::Save( BOOL bIsNamed /* = FALSE */ )
{
    CString szFileName, szFileTitle;
    TRY
    {
        if ( bIsNamed )
            m_people.DoSave();
        else
        {
            szFileName = m_people.GetName();
            if ( FileDlg( FALSE, SIZESTRING,
                    szFileName.GetBuffer( SIZESTRING ), SIZESTRING,
                    szFileTitle.GetBuffer( SIZESTRING ) ) )
            {
                szFileName.ReleaseBuffer();
                m_people.DoSave( szFileName );
                m_people.SetTitle( szFileTitle );
                SetWindowText( m_people.GetTitle() );
            }
            else
                return FALSE;
        }
    }
    CATCH( CFileException, e )
    {
        char ErrorMsg[25];
        sprintf( ErrorMsg,"Saving %s returned a 0x%lx.",
                (const char*)szFileTitle, e -> m_lOsError );
        MessageBox( ErrorMsg, "File Open Error" );
    }
    AND_CATCH( CArchiveException, e )
    {
        char ErrorMsg[25];
        sprintf( ErrorMsg, "Reading the %s archive failed.",
                (const char*)szFileTitle );
        MessageBox( ErrorMsg, "File Open Error" );
    }
    END_CATCH
    return TRUE;
}
```

```
////////////////////////////////////////////////
// CMainWindow::CheckForSave
// Whenever a new file is opened this routine will determine if
// there are unsaved changes in the current database. If so it
// will query the user and determine to save or not as appropriate.
//
BOOL CMainWindow::CheckForSave( const char* pszTitle, const char* pszMessage )
{
    if( m_people.IsDirty() )
    {
        UINT nButton = MessageBox( pszMessage, pszTitle, MB_YESNOCANCEL );
        if( nButton == IDYES )
        {
            if( !Save( m_people.IsNamed() )
                return FALSE;
        }
        else if( nButton == IDCANCEL )
            return FALSE;
    }
    return TRUE;
}
```

```
////////////////////////////////////////////////
// CMainWindow::SetMenu
// Whenever the existence of the DataBase is changed this
// routine will reset the menus so only the possible commands
// are accessible.
//
void CMainWindow::SetMenu()
{
    CMenu* pMenu = GetMenu();
    if ( m_people.IsPresent() )
    {
        if ( m_people.IsNamed() )
            pMenu -> EnableMenuItem( IDM_SAVE, MF_ENABLED );
        else
            pMenu -> EnableMenuItem( IDM_SAVE, MF_GRAYED );
        pMenu -> EnableMenuItem( IDM_SAVEAS, MF_ENABLED );
        pMenu -> EnableMenuItem( IDM_CLOSE, MF_ENABLED );
        pMenu -> EnableMenuItem( IDM_PRINT, MF_ENABLED );
        pMenu -> EnableMenuItem( IDM_ADD, MF_ENABLED );
        pMenu -> EnableMenuItem( IDM_DELETE, MF_ENABLED );
        pMenu -> EnableMenuItem( IDM_FIND, MF_ENABLED );
        pMenu -> EnableMenuItem( IDM_EDIT, MF_ENABLED );
    }
    else
    {
        pMenu -> EnableMenuItem( IDM_SAVE, MF_GRAYED );
        pMenu -> EnableMenuItem( IDM_SAVEAS, MF_GRAYED );
        pMenu -> EnableMenuItem( IDM_CLOSE, MF_GRAYED );
        pMenu -> EnableMenuItem( IDM_PRINT, MF_GRAYED );
        pMenu -> EnableMenuItem( IDM_ADD, MF_GRAYED );
        pMenu -> EnableMenuItem( IDM_DELETE, MF_GRAYED );
        pMenu -> EnableMenuItem( IDM_FIND, MF_GRAYED );
        pMenu -> EnableMenuItem( IDM_FINDALL, MF_GRAYED );
        pMenu -> EnableMenuItem( IDM_EDIT, MF_GRAYED );
    }
}

////////////////////////////////////////////////
// CMainWindow::InvalidateLine
// Marks the screen area of the currently selected person as
// invalid causing windows to call OnPaint to redraw the area.
// This is normally used when the selected line is being changed.
//
void CMainWindow::InvalidateLine()
{
    CRect area( 0, ( m_nSelectLine - m_nVscrollPos ) * m_cyChar, m_cxClient,
                ( m_nSelectLine + 1 - m_nVscrollPos ) * m_cyChar );
    InvalidateRect( area );
}
```

Phone Book: Message Handlers



The two previous chapters got you started on the Phone Book program. In Chapter 4, you created the `CDataBase` class to provide a clean interface between the Windows code and the `CPerson` data. In Chapter 5, you added several dialog classes.

6.1 In This Chapter

This chapter completes your development of Phone Book. You'll do the following:

- Add the message map
- Add the main window class
- Add message-handler member functions to handle menu, keyboard, and mouse commands

The next section gets you started with some analysis of what's needed.

6.2 Determine What Messages Will Be Handled

This section explains the fourth step in writing Phone Book: determine what messages will be handled and add the message map for class `CMainWindow`. This will be accomplished in several steps:

1. Plan the message handlers.
2. Add the message map for class `CMainWindow`.
3. Add the class declaration for `CMainWindow`.
4. Add a constructor definition.

► **To plan the message handlers:**

- Determine what message handlers are needed.

The Phone Book program requires about twenty message-handlers, which fall into several categories:

- Menu-command handlers will be needed. These are handler functions for **WM_COMMAND** messages. Windows passes a menu-command ID number with each such message. These handlers will be named after the menu commands they handle.
- The program requires handler functions for Windows creation and sizing messages: **WM_CREATE**, and **WM_SIZE**.
- The program requires handler functions for scrolling: **WM_VSCROLL** and **WM_HSCROLL**.
- The program also requires handler functions for several specific keystrokes (including the F1 function key, which is used to summon help): the UP ARROW, DOWN ARROW, RIGHT ARROW, and LEFT ARROW keys, the PAGE UP, PAGE DOWN, HOME, and END keys. These keys allow the user to move a "selection" from line to line on the display and to scroll the display.
- The program also requires message handlers for mouse clicks. If the user presses the left mouse button in the display area of the program's window, the selection is moved to the clicked line of text. This requires a message-handler function for the **WM_LBUTTONDOWN** message. If the user double-clicks the left mouse button in a line of the display, a **WM_LBUTTONDBLCLK** message invokes the Edit command in the Person menu, displaying data for the double-clicked person.
- The program requires a handler function for the paint message, **WM_PAINT**.

Keys That Don't Need Message Handlers

The DELETE and ENTER keys are handled by mapping the keys to the **IDM_DELETE** and **IDM_EDIT** menu ID numbers. This is done in the accelerator table resource template in PHBOOK.RC. The relevant lines look like this (don't add this code to your files yet):

```
VK_DELETE,    IDM_DELETE, VIRTKEY
VK_RETURN,...IDM_EDIT, VIRTKEY
```

The **VK_DELETE** and **VK_RETURN** lines do the mapping. This mapping causes the message-handler functions for the Delete and Edit commands in the Person menu to be invoked when the user presses the keys. Because DELETE and ENTER can be handled this way, it's unnecessary to provide explicit handlers for these keys.

Your goal in this chapter is to create a VIEW.CPP file that matches the one in Listing 2 in Chapter 5. In that listing, the `CMainWindow` member functions are given in the following groupings:

- Menu-command handlers for all menus

  Phone Book has File, Person, and Help menus.

- Creation and sizing handlers

  These handle the **WM_CREATE** and **WM_SIZE** messages.

- Scrolling handlers

  These handle the vertical and horizontal scrollbars.

- Keyboard and mouse handlers

  These handle keystrokes and mouse clicks.

- The paint handler

  This handles the **WM_PAINT** message.

- Utilities

  These are member functions that support the message-handler functions but are not themselves message-handlers.

Occasionally you'll be asked to add a function out of this order so it can be discussed with related functions. In the few such cases, you'll be told where to add the function so your file maintains the same order as Listing 2 in Chapter 5.

► To add the message map and message-handlers:

- Start adding code to your files by adding message map entries for the message-handler functions. You'll add a message map to VIEW.CPP and a main window class declaration to VIEW.H.

  Replace the old message map for Hello with the following message map for class `CMainWindow` in your VIEW.CPP file:

```
BEGIN_MESSAGE_MAP( CMainWindow, CFrameWnd )

    // File menu commands:
    ON_COMMAND( IDM_NEW, OnNew )
    ON_COMMAND( IDM_OPEN, OnOpen )
    ON_COMMAND( IDM_SAVE, OnSave )
    ON_COMMAND( IDM_SAVEAS, OnSaveAs )
    ON_COMMAND( IDM_CLOSE, OnDBClose )
    ON_COMMAND( IDM_PRINT, OnPrint )
    ON_COMMAND( IDM_EXIT, OnExit )

    // Person menu commands:
    ON_COMMAND( IDM_ADD, OnAdd )
    ON_COMMAND( IDM_DELETE, OnDelete )
```

```
    ON_COMMAND( IDM_EDIT, OnEdit )
    ON_COMMAND( IDM_FIND, OnFind )
    ON_COMMAND( IDM_FINDALL, OnFindAll )

    // Help menu commands:
    ON_COMMAND( IDM_HELP, OnHelp )
    ON_COMMAND( IDM_ABOUT, OnAbout )

    // Selection accelerators:
    ON_COMMAND( VK_UP, OnUp )
    ON_COMMAND( VK_DOWN, OnDown )

    // Other Windows messages:
    ON_WM_CREATE()
    ON_WM_CLOSE()
    ON_WM_SIZE()
    ON_WM_VSCROLL()
    ON_WM_HSCROLL()
    ON_WM_LBUTTONDOWN()
    ON_WM_LBUTTONDBLCLK()
    ON_WM_KEYDOWN()
    ON_WM_PAINT()
END_MESSAGE_MAP()
```

As you saw in the Hello program, this message map connects Windows messages with message-handler member functions defined by the `CMainWindow` class. Notice that the message map also provides the names of the message-handler functions. You'll add the handler functions for these map entries in the next several sections.

► To add the `CMainWindow` class declaration:

- Add the following code to VIEW.H:

```
// CMainWindow
// The window object that WinApp creates. In this program we
// only use one window class. In that sense this object does
// all the work that makes our window a CPersonList viewer.
//
```

```
class CMainWindow : public CFrameWnd
{
private:
    // Variables that contain the window size, font size and scroll
    // position.
    int m_cxChar;
    int m_cyChar;
    int m_nHscrollPos;
    int m_nVscrollPos;
    int m_cxCaps;
    int m_nMaxWidth;
    int m_cxClient;
    int m_cyClient;
    int m_nVscrollMax;
    int m_nHscrollMax;
    int m_nSelectLine;
    CDataBase m_people;

    // Private helpers for the other routines.
    void SetMenu();
    BOOL Save( BOOL bNamed=FALSE );
    BOOL FileDlg( BOOL bOpen, int nMaxFile, LPSTR szFile,
            int nMaxFileTitle, LPSTR szFileTitle );
    BOOL CheckForSave( const char* pszTitle, const char*
            pszMessage );
    void InvalidateLine();

public:
    // The CMainWindow constructor
    CMainWindow();

    // These routines are all overrides of CWnd.  Windows messages
    // cause these to be called.
    afx_msg int  OnCreate( LPCREATESTRUCT cs );
    afx_msg void OnClose();
    afx_msg void OnSize( UINT type, int x, int y );
    afx_msg void OnHScroll( UINT wParam, UINT pos, CWnd* control );
    afx_msg void OnVScroll( UINT wParam, UINT pos, CWnd* control );
    afx_msg void OnLButtonDown( UINT wParam, CPoint location );
    afx_msg void OnLButtonDblClk( UINT wParam, CPoint location );
    afx_msg void OnKeyDown( UINT wParam, UINT, UINT );
    afx_msg void OnPaint();
```

```
    // These routines are all menu items.  User action causes
    // these to be called.
    afx_msg void OnNew();
    afx_msg void OnOpen();
    afx_msg void OnSave();
    afx_msg void OnSaveAs();
    afx_msg void OnDBClose();
    afx_msg void OnPrint();
    afx_msg void OnExit();
    afx_msg void OnAdd();
    afx_msg void OnDelete();
    afx_msg void OnFind();
    afx_msg void OnFindAll();
    afx_msg void OnEdit();
    afx_msg void OnHelp();
    afx_msg void OnAbout();
    afx_msg void OnUp();
    afx_msg void OnDown();

    DECLARE_MESSAGE_MAP()
};
```

This class declaration replaces the `CMainWindow` declaration from Hello. It has the same general form as Hello's version but contains many more member variables and member functions. The declaration also completes your VIEW.H file. Make sure that the final "**#endif** // __VIEW_H__" line is below all other code in the file.

► To add a constructor definition:

- Add the following code to file VIEW.CPP below the message map:

```
CMainWindow()        {
        VERIFY( LoadAccelTable( "MainAccelTable" ) );
        VERIFY( Create( NULL, "Phone Book",
            WS_OVERLAPPEDWINDOW, rectDefault, NULL,
                    "MainMenu" ) );
        m_nSelectLine = -1;
    }
```

You can simply replace the contents of Hello's `CMainWindow` constructor, which you copied when you created VIEW.CPP.

The constructor demonstrates two new features. First, it's used to initialize the `m_nSelectLine` member variable in the `CMainWindow` object, something Hello's constructor didn't do.

Second, the constructor invokes the **VERIFY** macro on the two calls it makes. The result of each call—to **LoadAccelTable** and **Create**—is passed to the **VERIFY** macro. **VERIFY** works differently depending on whether the **_DEBUG** flag is set. If it is, the macro "asserts" if the result is zero (**FALSE**). If the flag is not set, the macro evaluates its argument but does not assert for a zero value. An assertion produces useful diagnostic information. Thus **VERIFY** is a handy way to check the results of functions that return a Boolean or pointer value. You can leave the **VERIFY** macros in place when you build a release version. For more information about the **VERIFY** macro and its companion, the **ASSERT** macro, see Chapter X in this manual.

Note The "afx_msg" prefix on the message-handlers in `CMainWindow` marks a special set of member functions. The message-map code that you provide creates a mechanism similar in effect to a C++ **v-table**. The mechanism maps Windows messages to the message-handlers as if they were virtual functions. Because of this mechanism, you can read "afx_msg" as if it meant "virtual," although the prefix has no effect on compilation. These message-handlers are prototyped in the declaration for class **CWnd**, in file AFXWIN.H. (This file is included in your distribution disks.) To use the message-handlers in your own class declarations, copy the prototypes you need from AFXWIN.H and paste them into your code.

The message-handler functions from class **CWnd** are defaults. When you declare your own message-handler with the same name and parameter signature, the result is equivalent to overriding a virtual function. Some of the message-handler declarations in the `CMainWindow` declaration list parameter types without specifying parameter names. The omitted names mark parameters not used in the Phone Book implementation. If you omit the parameter names, the compiler does not allocate space for those parameters. If you provide names for unused parameters, you'll get compiler warnings.

You'll add the message-handler member functions themselves in subsequent sections of the tutorial. You'll probably want to check your work against Listing 2 in Chapter 5 when you finish, so the following instructions explain where to add each function in the VIEW.CPP file to duplicate the order of functions in that listing. This ordering is a matter of convenience.

To continue the tutorial, see "Add Message-Handlers for File Menu Commands" on page 205. For more information about the steps just completed, see "Discussion: Message-Handler Functions" immediately following.

Discussion: Message-Handler Functions

Once you've grasped the principles of handling Windows messages in your program, the process is straightforward. This discussion reviews the process. Later sections present the message-handler functions.

Provide Message-Handler Functions and Message-Map Entries

Your message-handler functions are declared as member functions of your window class.

For each message-handler function, provide a corresponding macro entry in the message map for your window class.

If you write programs with multiple windows, each different kind of window requires its own set of message-handler functions and corresponding message-map.

Phone Book uses only one kind of window (other than dialogs). Class `CMainWindow` declares a constructor and many message handlers. Its message map, which you implemented in VIEW.CPP, contains matching entries.

Follow the Rules for Naming Message-Handler Functions

As discussed in Chapter 2, the message-map mechanism used in the Microsoft Foundation Class Library to connect Windows messages with your message handlers has certain requirements for names.

- Handler functions for **WM_COMMAND** messages, used primarily for menus and accelerator keys, can be named anything you like, but usually their names reflect their functions. Menu handlers, for example, are typically named after the menu command they handle. These handlers take no arguments and return no values.

  In the message map, use the **ON_COMMAND** macro for each of these commands. The first argument to the macro is the ID number of the menu or accelerator key. The second argument to the macro is the handler function name, such as `OnAbout`.

  Note Message-handler functions in this category are not predeclared in class **CWnd**. The **ON_COMMAND** mechanism provides for your own messages, which cannot be anticipated.

- Handler functions for notification messages from child windows to a parent window follow the same rules as those for **WM_COMMAND** messages. For example, you might call the handler function for a **BN_CLICKED** message `OnBnClicked`.
- Handler functions for other messages, such as **WM_PAINT** and **WM_CREATE**, have strict requirements for names and argument signatures. Class **CWnd** lists prototypes for these message handlers—each prototype is preceded by the identifier **afx_msg**. Class **CWnd** is declared in file AFXWIN.H.

  The handler for the **WM_CREATE** message, for instance, must be named `OnCreate` and must take one argument of type **LPCREATESTRUCT**, a Windows type. The function returns an **int**.

  Note Keep in mind that your message-handler functions in this category are overriding the predefined versions declared in class **CWnd**.

  The macros used in message maps for these functions prefix the message name with **ON_**—for example, **ON_WM_CREATE**. The macros take no arguments.

6.3 Add Message Handlers for File Menu Commands

This section explains the fifth step in writing Phone Book: add message-handler member functions to the `CMainWindow` class for File menu commands. Figure 6.1 shows the Phone Book File menu.

Figure 6.1 Phone Book File Menu

► To add File menu message-handler functions:

1. Add the following `OnNew` message-handler member function to `CMainWindow` section of your VIEW.CPP file. If you want your file to resemble Listing 2 in Chapter 5, add `OnNew` just below the `CMainWindow` message map:

```
// CMainWindow::OnNew
// After checking to see if current data needs to be stored, call
// database New and resize/repaint the window.
//
void CMainWindow::OnNew()
{
    if( !CheckForSave( "File New", "Save file before New?" ) )
        return;    m_people.New();
    SetMenu();
    SetWindowText( m_people.GetTitle() );
    OnSize( 0, m_cxClient, m_cyClient );
}
```

`OnNew` creates a new, empty database, to which the user can add persons. `OnNew` calls a `SetMenu` utility member function. You'll add that function later in the chapter.

2. Add the following `OnOpen` message-handler member function to the `CMainWindow` section of your VIEW.CPP file below the `OnNew` member function:

```
// CMainWindow::OnOpen
//
void CMainWindow::OnOpen()
{
    if( !CheckForSave( "File Open", "Save file before Open?" ) )
        return;
    // Attempt to open a database file and read it.
    // If a file or archive exception occurs, catch it and
    // present an error message box.
    CString szFileName, szFileTitle;
    TRY
    {
        // Use CommDlg to get the filename and then call DoOpen.
        // Set the Window title and menus.  Resize/Repaint.
        if ( FileDlg( TRUE, SIZESTRING, szFileName.GetBuffer
                ( SIZESTRING ), SIZESTRING,
                szFileTitle.GetBuffer( SIZESTRING ) ) )
        {
            szFileName.ReleaseBuffer();
            szFileTitle.ReleaseBuffer();
            m_people.DoOpen( szFileName );
            m_people.SetTitle( szFileTitle );
            SetWindowText( m_people.GetTitle() );
            SetMenu();
            OnSize( 0, m_cxClient, m_cyClient );
```

```
        }
    }
    CATCH( CFileException, e )
    {
        char ErrorMsg[25];
        sprintf( ErrorMsg,"Opening %s returned a 0x%lx.",
                (const char*)szFileTitle, e -> m_lOsError );
        MessageBox( ErrorMsg, "File Open Error" );
    }
    AND_CATCH( CArchiveException, e )
    {
        char ErrorMsg[25];
        sprintf( ErrorMsg,"Reading the %s archive failed.",
                (const char*)szFileTitle );
        MessageBox( ErrorMsg, "File Open Error" );
    }
    END_CATCH
}
```

`OnOpen` opens an existing database file and deserializes the data into the current database. `OnOpen` calls a utility member function `FileDlg` to put up a standard Windows file open dialog box. You'll add that utility function later in the chapter.

3. Add the following `OnSave` message-handler member function to the `CMainWindow` section of your VIEW.CPP file below the `OnOpen` member function:

```
void CMainWindow::OnSave()
{
    Save( m_people.IsNamed() );
}
```

`OnSave` responds to the Save command in the File menu to serialize the current database to a disk file. `Save` is a utility member function called by a number of other member functions. It is not a message-handler function. You'll learn what code to add for `Save` later in the chapter.

4. Add the following `OnSaveAs` message-handler member function to the `CMainWindow` section of your VIEW.CPP file below the `OnSave` member function:

```
void CMainWindow::OnSaveAs()
{
    Save();
}
```

`OnSaveAs` responds to the Save As command in the File menu. It simply calls the `Save` utility member function, which you'll add to the code later.

5. Add the following `Save` member function to the `CMainWindow` section of VIEW.CPP. To keep your file in the same order as Listing 2 in Chapter 5, add `Save` to the end of the file. It's one of the group of utility member functions, which will be kept together at the end of the file.

```
// CMainWindow::Save
// Handles any time a file needs to be saved to the disk.
// Passing in FALSE for name brings up the file save as dialog
// whether or not the database had a name before.
//
BOOL CMainWindow::Save( BOOL bIsNamed /* = FALSE */ )
{
    CString szFileName, szFileTitle;
    TRY
    {
        if ( bIsNamed )
            m_people.DoSave();
        else
        {
            szFileName = m_people.GetName();
            if ( FileDlg( FALSE, SIZESTRING,
                    szFileName.GetBuffer( SIZESTRING ), SIZESTRING,
                    szFileTitle.GetBuffer( SIZESTRING ) ) )
            {
                szFileName.ReleaseBuffer();
                m_people.DoSave( szFileName );
                m_people.SetTitle( szFileTitle );
                SetWindowText( m_people.GetTitle() );
            }
            else
                return FALSE;
        }
    }
    CATCH( CFileException, e )
    {
        char ErrorMsg[25];
        sprintf( ErrorMsg,"Saving %s returned a 0x%lx.",
                (const char*)szFileTitle, e -> m_lOsError );
        MessageBox( ErrorMsg, "File Open Error" );
    }
    AND_CATCH( CArchiveException, e )
    {
        char ErrorMsg[25];
        sprintf( ErrorMsg,"Reading the %s archive failed.",
                (const char*)szFileTitle );
        MessageBox( ErrorMsg, "File Open Error" );
    }
    END_CATCH
    return TRUE;
}
```

Although `Save` is a member function of class `CMainWindow`, it is not a message-handler function. It's a utility member function called by other member functions when the current database needs to be saved. In particular, `Save` is used to implement both `OnSave` and `OnSaveAs`.

6. Add the following `OnDBClose` message-handler member function to the `CMainWindow` section of VIEW.CPP below the `OnSaveAs` member function:

```
// CMainWindow::OnDBClose
// Closes the current database, checking to see if it should be
// saved first.  Reset the window title and the scroll regions.
// Invalidating the entire screen causes OnPaint to repaint but
// this time without any data.
//
void CMainWindow::OnDBClose()
{
    if( !CheckForSave( "File Close", "Save file before closing?" ) )
        return;     m_people.Terminate();
    SetWindowText( "Phone Book" );
    SetMenu();
    OnSize( 0, m_cxClient, m_cyClient );
}
```

`OnDBClose` is called in response to the Close command in the File menu. It handles closing the current database and adjusting the window to reflect that no database is open. The similarly named `OnClose` member function, which you'll add next, is called during the termination process in response to the Exit command in the File menu.

7. Add the following `OnClose` message-handler member function to the `CMainWindow` section of your VIEW.CPP file. To keep your file in the same order as Listing 2 in Chapter 5, add `OnClose` just above the `Save` member function that you added earlier.

```
// CMainWindow::OnClose
// Check to see if the current file needs to be saved. Terminate
// the database and destroy the window.
//
void CMainWindow::OnClose()
{
    if( !CheckForSave( "File Exit", "Save file before exit?" ) )
        return;     m_people.Terminate();
    DestroyWindow();
}
```

Where `OnDBClose` simply closes the current database, `OnClose` also destroys and removes the window. Because of this action, `OnClose` is placed in the file with "Creation and Sizing" member functions, which you'll add later.

8. Add the following `OnExit` message-handler member function to the `CMainWindow` section of your VIEW.CPP file below the `OnDBClose` member function:

```
// CMainWindow::OnExit
//
void CMainWindow::OnExit()
{
    OnClose();
}
```

`OnExit` responds to the Exit command in the File menu. It simply calls the `OnClose` member function to clean up in case there's an open database. `OnClose` also destroys the window and removes it from the screen. The same mechanism is used to process both a system **WM_CLOSE** message and an Exit command from the menu. The same cleanup is required in both instances.

9. Add the following `OnPrint` message-handler member function to the `CMainWindow` section of your VIEW.CPP file below the `OnDBClose` member function and above `OnExit`. This order keeps `OnPrint` in the same sequence as Listing 2 in Chapter 5 and in the sequence of File menu commands.

```
// CMainWindow::OnPrint
// Uses the commdlg print dialog to create a printer dc
// Then it uses code almost identical to the OnPaint code
// to write the data to the printer.
//
void CMainWindow::OnPrint()
{
    PRINTDLG pd;

    pd.lStructSize = sizeof( PRINTDLG );
    pd.hwndOwner=m_hWnd;
    pd.hDevMode=(HANDLE)NULL;
    pd.hDevNames=(HANDLE)NULL;
    pd.Flags=PD_RETURNDC | PD_NOSELECTION | PD_NOPAGENUMS;
    pd.nFromPage=0;
    pd.nToPage=0;
    pd.nMinPage=0;
    pd.nMaxPage=0;
    pd.nCopies=1;
    pd.hInstance=(HANDLE)NULL;

    if ( PrintDlg( &pd ) != 0 )
    {
        // CommDlg returned a DC so create a CDC object from it.
        ASSERT( pd.hDC != 0 );
        CDC * dc;
        dc = CDC::FromHandle( pd.hDC );
```

```
// Change to hourglass while printing
SetCursor( AfxGetApp() -> LoadStandardCursor( IDC_WAIT ) );

// Begin printing the document.
int rc;
char szError[50];
rc = dc -> StartDoc( "Phone Book" );
if ( rc < 0 )
{
    sprintf( szError, "Unable to Begin printing - Error[%d]",
             rc );
    MessageBox( szError, NULL,MB_OK );
    return;
}

int x, y;
CPerson* pCurrent;
UINT nPerson=0;
CString szDisplay;
int nStart, nEnd;

// Get Height and Width of large character
CSize extentChar = dc -> GetTextExtent( "M", 1 );
int nCharHeight = extentChar.cy;
int nCharWidth = extentChar.cx;

// Get Page size in # of full lines
UINT nExtPage = ( dc -> GetDeviceCaps(VERTRES) - nCharHeight )
        / nCharHeight;

CString szTitle;
szTitle = CString( "Phone Book - " ) + m_people.GetName();

while ( nPerson != m_people.GetCount() )
{
    // Print a Page Header
    dc -> StartPage();
    dc -> SetTextAlign ( TA_LEFT | TA_TOP );
    dc -> TextOut( 0, 0, szTitle, szTitle.GetLength() );
    dc -> MoveTo( 0, nCharHeight );
    dc -> LineTo( dc -> GetTextExtent( szTitle,
                  szTitle.GetLength() ).cx, nCharHeight );

    // Print People from start to last person or page size
    // minus 2 ( header size )
    nEnd = min( m_people.GetCount() - nPerson, nExtPage-2 );
    for ( nStart = 0; nStart < nEnd; nStart++, nPerson++ )
```

```
            {
                x = 0;
                y = nCharHeight * ( nStart+2 );

                pCurrent = m_people.GetPerson( nPerson );
                szDisplay = " " + pCurrent -> GetLastName() + ", " +
                        pCurrent -> GetFirstName();
                dc -> SetTextAlign( TA_LEFT | TA_TOP );
                dc -> TextOut( x, y, szDisplay,
                             szDisplay.GetLength() )
                szDisplay = pCurrent -> GetPhoneNumber();
                dc -> SetTextAlign( TA_RIGHT | TA_TOP );
                dc -> TextOut( x + SIZENAME * nCharWidth, y,
                          szDisplay, szDisplay.GetLength() );

                szDisplay = pCurrent -> GetModTime().Format(
                          "%m/%d/%y %H:%M" );
                dc -> TextOut( x + ( SIZENAME + SIZEPHONE ) *
                          nCharWidth, y,
                          szDisplay, szDisplay.GetLength() );
            }
            dc -> EndPage();
        }
        dc -> EndDoc();
        dc -> DeleteDC();
        SetCursor( AfxGetApp() -> LoadStandardCursor( IDC_ARROW ) );
    }
}
```

`OnPrint` prints the current database in response to the Print command in the File menu. It calls the **PrintDlg** function defined in COMMDLG.DLL to put up a standard Windows print dialog.

To continue the tutorial, see "Add Message Handlers for Person Menu Commands" on page 216. For more information on the handlers you just added, see the discussion below, "Discussion: File Menu Message Handlers."

Discussion: File Menu Message Handlers

This discussion explains the File menu message-handler member functions. Figure 6.2 shows schematically how menu commands are processed.

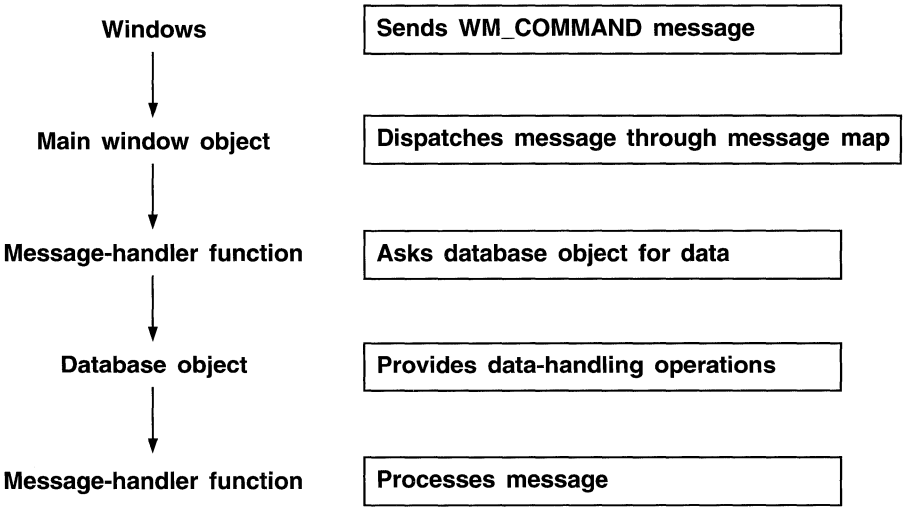


Figure 6.2 How Menu Commands Are Processed

OnNew

When the user chooses the New command in the File menu, a **WM_COMMAND** message causes the main window object's `OnNew` member function to be called through the message map. The function creates a new database if there isn't an existing database.

If there is no open database `OnNew` creates a new, empty database object. To create one, it calls the `New` member function of the `CDataBase` object. This call is passed through the `m_people` member variable of the main window object, a variable of type `CDataBase`. The database object's `New` member function destroys the `CPersonList` object for the old database if there was one (and the `CPersonList` for found items, if there was one), then constructs a new `CPersonList` object, storing it in the database object's `m_pDataList` member.

If there is an existing `CDataBase` object in the window object's `m_people` member variable, `OnNew` checks to see if the database has unsaved changes. If it has, the function displays a Windows message box to ask if the user wants to save the old database before creating a new one. If so, `OnNew` calls the window object's `Save` member function to serialize the old database and make it persistent. In the process, it gets a filename from the user if the database has never been saved

before. Once the old database has been saved, the `New` member function of class `CDataBase` is called as described previously.

After having its `CDataBase` member construct a new `CPersonList`, the window object's `OnNew` member function calls the window object's `SetMenu` member function to adjust the availability of menu commands. Then, because the new database has never been saved to a file, `OnNew` sets the window title to "Untitled," calls the `OnSize` member function because the number of records in the database has changed. `OnSize` calls the window object's **Invalidate** member function, inherited from class **CWnd**, to mark the window's client area invalid. This causes Windows to send a **WM_PAINT** message to the window so it will be redrawn (erased, actually, in the case of a new database).

OnOpen

When the user chooses the Open command in the File menu, the main window object's `OnOpen` member function is called to open an existing database file. If there is an existing database open, `OnOpen`, like `OnNew`, saves the existing database to a file, if the user wishes. Then the function opens the new database.

`OnOpen` calls the `FileDlg` member function to get a filename from the user. `FileDlg` is a utility member that creates and fills up an **OPENFILENAME** data structure and passes it in a call to the Windows common dialog function **GetOpenFileName**. `FileDlg` also sets up a filter string that specifies what file types are to be displayed by the Windows standard file open dialog box. `OnOpen` passes a Boolean value, bOpen, to `FileDlg`. If bOpen is **TRUE**, `FileDlg` invokes the standard open dialog. If it's **FALSE**, `FileDlg` invokes the standard save dialog.

Once `OnOpen` obtains a filename from the user, it calls the database object's `DoOpen` member function to open the file and read it into a `CPersonList` object. The database object's `DoOpen` member calls its `ReadDataBase` member function to do the work. You saw that function in Chapter 2 and again in Chapter 4, under "Discussion: Class CDataBase" on page 134. `ReadDataBase` can throw an exception, so the window object's `OnOpen` member puts its call to

```
m_people.DoOpen
```

inside an exception frame. You saw how exception frames work in Chapter 2.

If a file has been successfully opened and read, the window object's `OnOpen` member sets the window title to the filename, adjusts the menus with a call to the window object's `SetMenu` member, and causes the window to be updated. If there was an exception, `OnOpen` displays a Windows message box with an error message.

OnSave, OnSaveAs, and Save

When a database has unsaved changes, or has never been saved to a file, the user can save it by choosing the Save or Save As command in the File menu. The Save As menu command prompts the user for a filename, then saves the database to that file. The Save command is used when the database already has a filename associated with it. The Save command serializes the `CPersonList` in the database if it has a filename. If not, the Save command also prompts the user for a filename.

Phone Book implements the Save As menu command with the `CMainWindow` class's `OnSaveAs` member function. The main window object's `OnSaveAs` member simply calls the window object's member function, `Save`. `Save` is a utility member function called by several other member functions that need to save an open database. These include the main window object's `OnNew`, `OnOpen`, `OnDBClose`, and `OnClose` member functions.

`Save` first checks to see if the database already has a name. It does this with a call to the database object's `IsNamed` member. If it has a name, `Save` calls the database object's `DoSave` member function through the window object's `m_people` member variable. This call takes no argument because the database already has a name in its `m_FileName` member variable, where `DoSave` can access it. The database object's `DoSave` member calls its `WriteDataBase` member function to do the work.

This call can throw an exception, so the main window object's `Save` member puts an exception frame around the attempt to save the data. (For more information about the database object's `DoSave` member function, see "Discussion: Class CDataBase" on page 134 in Chapter 4.) `Save` attempts to open the file and write the database to it.

If the database has not yet been named, `Save` displays a standard Windows save dialog box asking the user for a filename. The dialog box is invoked by a call to `FileDlg`, the utility member function that was explained under "OnOpen" on page 214. If it gets a filename, it calls the `CDataBase` object's `DoSave` member function through `m_people`, passing the filename as an argument. Then it sets the window title to the file title stored in `m_szFileTitle`. (This is the filename in uppercase letters with no path.)

OnDBClose, OnClose, and OnExit

If the user chooses the Close command in the File menu, the main window object's `OnDBClose` member function is called to close the database. The program continues to run. If the user chooses the Exit command in the File menu, the main window object's `OnExit` member is called. `OnExit` calls `OnClose`. This sequence of calls destroys the database and ends the program. The same cleanup is required for the Exit command in the File menu as for the case in which Windows sends a

WM_CLOSE message in response to the system menu. Both sequences are mapped to the same code.

If the database has unsaved changes, `OnClose` asks the user if the file should be saved. If the user wishes to save the file, the `Save` member function is called to handle writing the file.

If the database doesn't need to be saved, `OnClose` calls the `CDataBase` object's `Terminate` member function, through `m_people`, to destroy the database's `CPersonList` objects (the main list and the found list).

6.4 Add Message Handlers for Person Menu Commands

This section explains the sixth step in writing Phone Book: add message-handler functions for the Person menu. Figure 6.3 shows the Phone Book Person menu.

Figure 6.3 Phone Book Person Menu

► **To add the Person menu message-handlers:**

1. Add the following `OnAdd` message-handler member function to the `CMainWindow` section of your VIEW.CPP file below the `OnExit` member function:

```
// CMainWindow::OnAdd
// Using the EditDialog fill in a new person object.  If the user
// selects OK then add the person, call OnSize to resize the scroll
// region, and invalidate the screen so it will be redrawn with the
// new person in the correct order.
//
void CMainWindow::OnAdd()
{
    CPerson* person=new CPerson();

    CEditDialog dlgAdd( person, this );
    if ( dlgAdd.DoModal() == IDOK )
    {
        m_people.AddPerson( person );
        OnSize( 0, m_cxClient, m_cyClient );
    }
```

```
    else
        delete person;
}
```

`OnAdd` implements the Add command in the Person menu. It constructs a new `CPerson` object, constructs a `CEditDialog` object so the user can fill in the person's name and phone number, and calls member functions of the database object to add the person object to the list.

2. Add the following `OnDelete` message-handler member function to the `CMainWindow` section of your VIEW.CPP file below the `OnAdd` member function:

```
// CMainWindow::OnDelete
// Deletes the current selection.  Check to see if the selection is
// now past the end of the list.  Also call OnSize since the list
// length has now changed.
//
void CMainWindow::OnDelete()
{
    if ( m_nSelectLine == -1 )
    {
        MessageBox( "Select a person to delete first" );
        return;
    }
    m_people.DeletePerson( m_nSelectLine );
    if ( m_nSelectLine >= (int)m_people.GetCount() )
        m_nSelectLine--;
    OnSize( 0, m_cxClient, m_cyClient );
}
```

`OnDelete` works on the currently selected person in the database. It implements the Delete command in the Person menu to delete the selected person.

3. Add the following `OnFind` message-handler member function to the `CMainWindow` section of your VIEW.CPP file below the `OnDelete` member function:

```
// CMainWindow::OnFind
// Gets information from the CFindDialog modal dialog box, then
// searches for matching people. Note the Add and Delete menu
// items are disabled after a find is made. Find All is enabled.
//
void CMainWindow::OnFind()
{
    CFindDialog dlgFind( this );
    if ( dlgFind.DoModal() == IDOK &&
                            dlgFind.GetFindString().GetLength() != 0 )
    {
        if ( m_people.DoFind( dlgFind.GetFindString() ) )
        {
            m_nSelectLine = -1;
```

```
            CString tmp;
            tmp = m_people.GetTitle() + " Found: "
                    + dlgFind.GetFindString();
            SetWindowText( tmp );
            CMenu* pMenu = GetMenu();
            pMenu -> EnableMenuItem( IDM_FINDALL, MF_ENABLED );
            pMenu -> EnableMenuItem( IDM_DELETE, MF_GRAYED );
            pMenu -> EnableMenuItem( IDM_ADD, MF_GRAYED );
            OnSize( 0, m_cxClient, m_cyClient );
        }
        else
            MessageBox( "No match found in list." );
    }
}
```

`OnFind` constructs a dialog object requesting a search string. The string is passed to the database object's `DoFind` member function, which searches the database and returns a list of pointers to found person objects. The window's client area is invalidated so that the next time the window is painted, the found list is displayed instead of the main database list.

4. Add the following `OnFindAll` message-handler member function to the `CMainWindow` section of your VIEW.CPP file below the `OnFind` member function:

```
// CMainWindow::OnFindAll
// Returns to view the whole database. Add, Delete are re-enabled,
// and Find All is again disabled. OnSize is called because the list
// has changed length.
//
void CMainWindow::OnFindAll()
{
    m_people.DoFind();
    SetWindowText( m_people.GetTitle() );
    CMenu* pMenu = GetMenu();
    pMenu -> EnableMenuItem( IDM_FINDALL, MF_GRAYED );
    pMenu -> EnableMenuItem( IDM_DELETE, MF_ENABLED );
    pMenu -> EnableMenuItem( IDM_ADD, MF_ENABLED );
    OnSize( 0, m_cxClient, m_cyClient );
}
```

When a list of finds is on display, `OnFindAll` returns the original database list to the display, replacing the list of found persons.

5. Add the following `OnEdit` message-handler member function to the `CMainWindow` section of your VIEW.CPP file below the `OnFindAll` member function:

```
// CMainWindow::OnEdit
// Using the member variable m_nSelectLine a CEditDialog is created
// and filled with the selected person.  If the dialog OK button is
// used the dialog saves the changes into the object.
```

```
//
void CMainWindow::OnEdit()
{
    if ( m_nSelectLine == -1 )
    {
        MessageBox( "Select a person to edit first" );
        return;
    }

    // Get a pointer to the person in the list.
    CPerson* pPerson = m_people.GetPerson( m_nSelectLine );
    CPerson tmpPerson = *pPerson;

    //Edit the data.
    CEditDialog dlgEdit( &tmpPerson, this );

    //if the ok button is pressed redraw the screen
    if ( dlgEdit.DoModal() == IDOK )
    {
        m_people.ReplacePerson( pPerson, tmpPerson );
        InvalidateLine();
    }
}
```

`OnEdit` implements the Edit command in the Person menu. It constructs a dialog object that displays the information contained in the currently-selected person object. When the user ends the dialog, the selected person object is replaced by a new object containing the new information.

To continue the tutorial, see "Add Message Handlers for Help Menu Commands" on page 222. For more information on the handlers you just added, see the discussion in the next section.

Discussion: Person Menu Message Handlers

This section explains the Person menu message-handler member functions.

OnAdd

With a database open, the user can add new persons to the list. If the user chooses the Add command in the Person menu, the main window object's `OnAdd` member function is called.

`OnAdd` constructs a new `CPerson` object in the heap with **new** and displays a `CEditDialog` box to get information for the person. If the dialog returns **IDOK**, the new person object is added to the database and the window is repainted.

The function uses a `CEditDialog` dialog object to fill the new person object with data. The dialog object is constructed on the frame of the function and its

DoModal member function is called to run the dialog. **DoModal** returns **IDOK** if the user clicked the OK button.

The new person object is added to the database by calling the `AddPerson` member function of the `CDataBase` object, which is stored in the main window object's `m_people` member variable. At this point, the function calls the main window object's `OnSize` member to recalibrate the scroll bars for the new number of persons in the database. (If the size of the data has exceeded the size of the window's client area, Windows automatically adds scroll bars.) Notice this explicit use of `OnSize`, which is normally called in response to a Windows **WM_SIZE** message. For more information on `OnSize`, see page 226.

After adding the new person to the database and recalibrating the scroll bars, `OnAdd` invalidates the entire client area of the window.

OnDelete

In addition to adding new persons, the user can delete existing persons from the database. If the user first selects a line of the display representing a person and then chooses the Delete command in the Person menu, the main window object's `OnDelete` member function is called.

A person must be selected before it's possible to delete that person from the database. So `OnDelete` first checks to see if there is a current selection. If there isn't, the `m_nSelectLine` member variable of the main window object has the value –1. In that case, `OnDelete` displays a Windows message box requesting that the user make a selection first, and returns.

If there is a selection, the function calls its `CDataBase` object's `DeletePerson` member, through the main window object's `m_people` member variable, passing the selection line number as an index to use in finding the correct person in the database. The Delete command can only be selected from the Person menu when the main database is being viewed. Then `OnDelete` locates the correct record in the database, deletes the person object there, and removes the entry from the list.

Once the indicated person has been deleted from the database, `OnDelete` continues by resetting the selection. Unless the person deleted was the last in the database, the function moves the selection to the next person. Otherwise, it moves the selection to the new last person. With the selection reset, `OnDelete` invalidates the client area of the window.

OnEdit

Besides adding and deleting persons, the user can edit persons in the database. If the user selects a person on the display and chooses the Edit command in the Person menu (or presses the ENTER key or double clicks the line with the mouse), the main window object's `OnEdit` member function is called.

As with `OnDelete`, a person must first be selected. If one isn't, `OnEdit` displays a Windows message box asking the user to make a selection first, then returns.

If a person is selected, `OnEdit` calls the database object's `GetPerson` member function, through `m_people`, passing the line number of the selection, which is used as an index into the list. This call returns a pointer to the selected person object.

`OnEdit` constructs a new, empty `CPerson` object, `person`, on the frame of the function and uses the overloaded assignment operator of class `CPerson` to assign the value of the person object retrieved from the list to the new person object. This allows the user to edit a copy rather than the original. Only if the user finishes editing the copy and clicks the OK button in the edit dialog box does the copy replace the original person object in the list.

The function constructs a `CEditDialog` object, passing it a pointer to the copy. While the dialog box runs, the user can edit the person object's first name, last name, or phone number. If the dialog box returns **IDOK**, the `OnEdit` calls the `CDataBase` object's `ReplacePerson` member to replace the original person object with the edited one. `ReplacePerson` copies the edited person's data into the old person object, gets the current date and time, and calls the new `CPerson` object's `SetModTime` member function to record the modification date.

After replacing the original person object in the database with the edited one, `OnEdit` invalidates the line that was changed.

OnFind and OnFindAll

While a database is open, the user can search it for persons with a given last name. If the user chooses the Find command in the Person menu, the main window object's `OnFind` member function is called. After `OnFind` has displayed a list of found persons in the window, the user can choose the FindAll command in the Person menu to redisplay the full database. This menu command calls the main window object's `OnFindAll` member function.

`OnFind` displays a `CFindDialog` box to get a last name to search for. The function then passes this name string as the argument in a call to the `CDataBase` object's `DoFind` member, through `m_people`. The database object's `DoFind` member function operates differently depending on the string passed to it. If the string

argument passed is **NULL**, `DoFind` deletes any existing found list from a previous search.

If a string containing a last name is passed in, the database object's `DoFind` member function calls the `FindPerson` member function of the `CPersonList` object in `m_pDataList` to return a new `CPersonList` object containing pointers to all found objects. This list is assigned to the `CDataBase` object's `m_pFindList` member variable.

While the search is being conducted, the window title changes to the text "Phone Book - [database name] Found: [Search string]".

`OnFind` next updates the menus, using an object of class **CMenu**, to enable the Find All command in the Person menu and disable the Add and Delete commands in the Person menu. The function constructs a **CMenu** object on the frame of the function and calls the Windows function **GetMenu** to obtain a pointer to the program's menus. Then the function calls member functions of the **CMenu** object to adjust the menus.

After updating menus, `OnFind` invalidates the client area so the window will be repainted to display the found list.

The main window object's `OnFindAll` member function works similarly, except that it doesn't need to get a search string from the user. The function calls the `CDataBase` object's `DoFind` member function, through `m_people`. The argument in this call to `DoFind` defaults to **NULL**, which causes `DoFind` to destroy the existing found list. When the window's contents are redisplayed the next time `OnPaint` is called, the full database is displayed.

`OnFindAll` also updates the menus, as in `OnFind`, and invalidates the client area so the window will be repainted to display the full database.

6.5 Add Message Handlers for Help Menu Commands

This section explains the seventh step in writing Phone Book: add message-handler functions for the Help menu. Figure 6.4 shows the Phone Book Help menu. Figures 5.4 through 5.6 in Chapter 5 show the three Help menu dialog boxes in Phone Book.

Figure 6.4 Phone Book Help Menu

► **To add the Help menu message-handlers:**

1. Add the following `OnHelp` message-handler member function to the `CMainWindow` section of your VIEW.CPP file below the `OnEdit` member function:

```
// CMainWindow::OnHelp
//
void CMainWindow::OnHelp()

{
    if ( !m_people.IsPresent() )
    {
        CModalDialog dlgHelp( "NoData", this );
        dlgHelp.DoModal();
        return;
    }

    if ( !m_people.IsNamed() )
    {
        CModalDialog dlgHelp( "NoName", this );
        if ( dlgHelp.DoModal() == IDCANCEL )
            return;
    }

    CModalDialog dlgHelp( "Enter", this );
    dlgHelp.DoModal();
}
```

   You already have an `OnAbout` menu command handler for your main window class. `OnAbout` is the same as it was in the Hello program. You copied it when you copied the Hello files to start Phone Book. To keep the order of definitions in VIEW.CPP consistent with Listing 2 in Chapter 5, you can put the definition of the `OnAbout` member function after `OnHelp`.

To continue the tutorial, see "Add a Keyboard and Mouse Interface" on page 230.
For more information on the handlers you just added, see "Discussion: Help Menu Message-Handlers" below.

Discussion: Help Menu Message Handlers

This section explains `OnHelp`, the Help menu's Help message-handler member functions.

At any time while the Phone Book program is running, the user can choose the Using Phone Book command in the Help menu to get information about how to use the program. This Help is somewhat context-sensitive. That is, different information appears for different situations. This menu command calls the main window object's `OnHelp` member function.

If no database is open, `OnHelp` displays a dialog box explaining that the user can create or open a database.

If a database is open but has not been saved, the function displays a dialog box explaining how to save it. This dialog box also has a Continue button that the user can click to display a second dialog box explaining how to add, delete, and edit persons in the database.

If a database is open and has no unsaved changes, the function displays the dialog box discussed above, which explains how to add, delete, and edit persons in the database.

Each of these dialog boxes uses a **CModalDialog** object. Each uses its own separate dialog resource template in the .RC file.

6.6 Add Message Handlers for Creation and Sizing

This section explains the eighth step in writing Phone Book: add message-handler functions to the `CMainWindow` class for the **WM_SIZE**, and **WM_CREATE** messages, and add the `OnClose` member function.

► **To add these message-handler functions:**

1. Add the following **WM_CREATE** message-handler function to the `CMainWindow` section of your VIEW.CPP file below your `OnAbout` member function and above the `OnClose` member function that you added earlier. This will preserve the same order as in Listing 2 in Chapter 5.

```
// CMainWindow::OnCreate
// Queries the current text metrics to determine char size.
//
int CMainWindow::OnCreate( LPCREATESTRUCT )
{
    TEXTMETRIC tm;

    // Get the text metrics.
    CDC* dc = GetDC();
    dc -> GetTextMetrics( &tm );
    ReleaseDC( dc );

    // Decide the statistics on how many rows, etc., we can display.
    m_cxChar = tm.tmAveCharWidth;
    m_cxCaps = ( (tm.tmPitchAndFamily & 1 )? 3 : 2 ) * m_cxChar / 2;
    m_cyChar = tm.tmHeight + tm.tmExternalLeading;
    m_nMaxWidth = ( SIZENAME + SIZEPHONE + 1 ) * m_cxCaps;
    m_nVscrollPos = m_nHscrollPos = 0;
```

```
    return 0;
}
```

`OnCreate` gets and stores text metric and scroll information to be used later in the program. It's called when Windows sends a **WM_CREATE** message when the window is created.

2. Add the following **WM_SIZE** message-handler function to the `CMainWindow` section of your VIEW.CPP file below `OnCreate` and above `OnClose`:

```
// CMainWindow::OnSize
// When resized, we need to recalculate our scrollbar ranges based on
// what part of the database is visible.
//
void CMainWindow::OnSize( UINT, int x, int y )
{
    m_cxClient = x;
    m_cyClient = y;

    m_nVscrollMax = max( 0,
            (int)( m_people.GetCount() ) - m_cyClient / m_cyChar );
    m_nVscrollPos = min( m_nVscrollPos, m_nVscrollMax );

    SetScrollRange( SB_VERT, 0, m_nVscrollMax, FALSE );
    SetScrollPos( SB_VERT, m_nVscrollPos, TRUE );

    m_nHscrollMax = max( 0, ( m_nMaxWidth - m_cxClient ) / m_cxChar );
    m_nHscrollPos = min( m_nHscrollPos, m_nHscrollMax );

    SetScrollRange( SB_HORZ, 0, m_nHscrollMax, FALSE );
    SetScrollPos( SB_HORZ, m_nHscrollPos, TRUE );
    Invalidate( TRUE );
}
```

`OnSize` recalculates the size of the window's client area, resets the scrollbars, and invalidates the client area to cause repainting.

To continue the tutorial, see "Add Scrolling Member Functions" on page 227. For more information on the handlers you just added, see the discussion in the next section.

Discussion: Creation and Sizing Member Functions

This section discusses the `OnCreate` and `OnSize` message-handler member functions. The `OnClose` member function was discussed earlier in "Add Message Handlers for File Menu Commands" on page 205.

OnCreate

When the window is initially created, Windows sends it a **WM_CREATE** message. This invokes the main window object's `OnCreate` member function. In Hello, there was nothing special to do for a **WM_CREATE** message, but Phone Book uses the message to set its variables for text drawing in the main window's `OnPaint` member function and for scroll-bar initialization. The member function initializes member variables that store the text metrics information and the current scroll size information.

As defined for the `CMainWindow` class, `OnCreate` uses a Windows **TEXTMETRIC** structure and a Foundation device-context object of class **CDC**. The device-context object is constructed on the frame of the function, then initialized by a call to the Windows function **GetDC**. This creates a Windows device context for the screen, which provides information on fonts and text measurements needed for drawing text in the window.

The `OnCreate` member function calls the device-context object's **GetTextMetrics** member function to fill the **TEXTMETRIC** structure with information. Then it releases the device context by calling the Windows function **ReleaseDC**. Using the text information structure, the function sets its variables. These are member variables declared in class `CMainWindow`.

As a local variable, the device-context object is destroyed when the function returns.

OnSize

When Phone Book's window is resized, Windows sends the window a **WM_SIZE** message. The window object's `OnSize` member function responds to this message by recalibrating the scroll bars to fit the resized window. `OnSize` is also called explicitly whenever the size of the data list changes. This occurs, for example, in `OnDelete`, `OnAdd`, `OnFind`, and `OnFindAll`.

Windows passes the new height and width of the window to the `OnSize` function. `OnSize` uses these values to reset its client area size values and to adjust the size of its scroll bars. Two **CWnd** member functions, **SetScrollRange** and **SetScrollPos**, are used to calibrate the scroll bars. Then the window's client area is invalidated to force an update and repaint the window, including its scroll bars.

Note The program doesn't use the Microsoft Foundation Class **CScrollBar** for its scroll bars because, for a main window, Windows automatically supplies scroll bars as needed. Class **CScrollBar** is a control class used to add scroll bars to a child window enclosed by a main window.

6.7 Add Scrolling Member Functions

This section explains the ninth step in writing Phone Book: add message-handler functions for vertical and horizontal scrolling.

▶ To add the scrolling message-handlers:

1. Add the following `OnVScroll` message-handler member function to the `CMainWindow` section of your VIEW.CPP file below the `OnSize` member function:

```
// CMainWindow::OnVScroll
// Translate scroll messages into Scroll increments and then
// checks the current position to determine if scrolling is possible
//
void CMainWindow::OnVScroll( UINT wParam, UINT pos, CWnd* )
{
    short nScrollInc;

    switch ( wParam )
    {
        case SB_TOP:
            nScrollInc = -m_nVscrollPos;
            break;

        case SB_BOTTOM:
            nScrollInc = m_nVscrollMax - m_nVscrollPos;
            break;

        case SB_LINEUP:
            nScrollInc = -1;
            break;

        case SB_LINEDOWN:
            nScrollInc = 1;
            break;

        case SB_PAGEUP:
            nScrollInc = min( -1, -m_cyClient / m_cyChar );
            break;

        case SB_PAGEDOWN:
            nScrollInc = max( 1, m_cyClient / m_cyChar );
            break;

        case SB_THUMBTRACK:
            nScrollInc = pos - m_nVscrollPos;
            break;

        default:
            nScrollInc = 0;
```

```
    }

    if ( nScrollInc = max( -m_nVscrollPos,
            min( nScrollInc, m_nVscrollMax - m_nVscrollPos ) ) )

    {
        m_nVscrollPos += nScrollInc;
        ScrollWindow( 0, -m_cyChar * nScrollInc );
        SetScrollPos( SB_VERT, m_nVscrollPos );
        UpdateWindow();
    }
}
```

`OnVScroll` responds to mouse clicks in the vertical scrollbar. It uses a **switch** statement to determine what part of the scrollbar was clicked. It then translates the location information into scrolling offsets and scrolls the window to match.

2. Add the following `OnHScroll` message-handler member function to the `CMainWindow` section of your VIEW.CPP file below the `OnVScroll` member function:

```
// CMainWindow::OnHScroll
// Translate scroll messages into Scroll increments and then
// checks the current position to determine if scrolling is possible
//
void CMainWindow::OnHScroll( UINT wParam, UINT pos, CWnd* )
{
    int nScrollInc;
    switch ( wParam )
    {
        case SB_LINEUP:
            nScrollInc = -1;
            break;

        case SB_LINEDOWN:
            nScrollInc = 1;
            break;

        case SB_PAGEUP:
            nScrollInc = -PAGESIZE;
            break;

        case SB_PAGEDOWN:
            nScrollInc = PAGESIZE;
            break;

        case SB_THUMBPOSITION:
            nScrollInc = pos - m_nHscrollPos;
            break;

        default:
            nScrollInc = 0;
    }
```

```
    if ( nScrollInc = max( -m_nHscrollPos,
             min( nScrollInc, m_nHscrollMax - m_nHscrollPos ) ) )

    {
        m_nHscrollPos += nScrollInc;
        ScrollWindow( -m_cxChar * nScrollInc, 0 );
        SetScrollPos( SB_HORZ, m_nHscrollPos );
       UpdateWindow();
    }
}
```

`OnHScroll` responds to the horizontal scrollbar.

To continue the tutorial, see "Add a Keyboard and Mouse Interface" on page 230.
For more information about the scrolling message-handlers, see the following discussion.

Discussion: Scrolling Message Handlers

This section explains the message-handlers for vertical and horizontal scrolling.

OnHScroll, OnVScroll, and OnKeyDown

Phone Book provides scrolling support through both the mouse and the keyboard.

When the user clicks the mouse in the vertical scroll bar of the Phone Book window, Windows sends the window a **WM_VSCROLL** message, which invokes the main window object's `OnVScroll` member function. Likewise, a click in the horizontal scroll bar invokes `OnHScroll` for a **WM_HSCROLL** message.

These functions greatly resemble the code normally seen in the **WndProc** function that you write for a traditional Windows program that handles scrolling for a window. Each function uses a C **switch** statement to select an scrolling adjustment based on what part of the scroll bar was clicked. Each **case** in the **switch** statement adjusts a scroll increment value appropriately. This value is used to recalculate the "scroll position" and to scroll the window's contents accordingly. Scrolling also forces an update to repaint the window.

In Phone Book, the scroll bars respond not only to the mouse but also to the RIGHT ARROW, LEFT ARROW, PAGE UP, PAGE DOWN, HOME, and END keys. The arrows scroll horizontally. PAGE UP and PAGE DOWN scroll the height of the window. HOME scrolls to the top of the database list. END scrolls to the bottom of the database list. The `OnKeyDown` member function translates these keys into scrolling actions. These scrolling actions do not change the current selection in the window.

6.8 Add a Keyboard and Mouse Interface

This section explains the tenth step in writing Phone Book: add message-handler functions for the keyboard and the mouse.

▶ To add the keyboard and mouse message-handlers:

1. Add the following `OnUp` message-handler member function to the `CMainWindow` section of your VIEW.CPP file below the `OnHScroll` member function:

```
// CMainWindow::OnUp
// Uses Accelerator tables to link the up arrow key to this
// routine. Decrements the select line with checking for scrolling
// and wrapping off the top of the list.
//
//
void CMainWindow::OnUp()
{
    InvalidateLine();

    if ( m_nSelectLine <= 0 )
    {
        m_nSelectLine = m_people.GetCount() - 1;
        m_nVscrollPos = max( 0, m_nSelectLine + 1 - ( m_cyClient /
                        m_cyChar ) );
        Invalidate( TRUE );
    }
    else
    {
        m_nSelectLine--;
        if ( m_nSelectLine - m_nVscrollPos < 0 )
            OnVScroll( SB_LINEUP, 0, NULL );

        // Selection is off the screen
        if ( m_nSelectLine - m_nVscrollPos > ( m_cyClient / m_cyChar
) )
        {
            m_nVscrollPos = m_nSelectLine + 1 - ( m_cyClient /
                            m_cyChar );
            SetScrollPos( SB_VERT, m_nVscrollPos, TRUE );
            Invalidate( TRUE );
        }
        if ( m_nSelectLine - m_nVscrollPos < 0 )
        {
            m_nVscrollPos = m_nSelectLine;
            SetScrollPos( SB_VERT, m_nVscrollPos, TRUE );
            Invalidate( TRUE );
        }
    }
```

```
    InvalidateLine();
}
```

`OnUp` responds to the UP ARROW key. It moves the selection in the window upward one line. At the top of the database list, the selection wraps around to the bottom of the list. The database record that was formerly selected is unhighlighted and the newly selected line is inverted to highlight it. `OnUp` calls the `InvalidateLine` utility member function to change a line's highlighting. You'll add `InvalidateLine` later.

2. Add the following `OnDown` message-handler member function to the `CMainWindow` section of your VIEW.CPP file below the `OnUp` member function:

```
// CMainWindow::OnDown
// Uses Accelerator tables to link the down arrow key to this
// routine.  Inc the select line with checking for scrolling
// and wrapping off the bottom of the list.
//
void CMainWindow::OnDown()
{
    InvalidateLine();

    if ( m_nSelectLine == (int)( m_people.GetCount() - 1 )
            || m_nSelectLine == -1 )
    {
        m_nSelectLine = 0;
        m_nVscrollPos = 0;
        Invalidate( TRUE );
    }
    else
    {
        m_nSelectLine++;
        if ( ( m_nSelectLine - m_nVscrollPos + 1 ) > ( m_cyClient /
                                m_cyChar ) )
            OnVScroll( SB_LINEDOWN, 0, NULL );

        // Selection is off the screen
        if ( ( m_nSelectLine - m_nVscrollPos ) > ( m_cyClient /
                                m_cyChar ) )
        {
            m_nVscrollPos = m_nSelectLine + 1 - ( m_cyClient /
                            m_cyChar );
            SetScrollPos( SB_VERT, m_nVscrollPos, TRUE );
            Invalidate( TRUE );
        }
```

```
        if ( ( m_nSelectLine - m_nVscrollPos ) < 0 )
        {
            m_nVscrollPos = m_nSelectLine;
            SetScrollPos( SB_VERT, m_nVscrollPos, TRUE );
            Invalidate( TRUE );
        }
    }

    InvalidateLine();
}
```

`OnDown` moves the selection down one line or, if it reaches the last line of the database information, wraps the selection around to the top.

3. Add the following `OnLButtonDown` message-handler member function to the `CMainWindow` section of your VIEW.CPP file below the `OnDown` member function:

```
// CMainWindow::OnLButtonDown
// Turns the location of the mouse pointer into a line number
// and stores that information in m_nSelectLine.  Uses
// InvalidateLine to cause OnPaint to change the screen.
//
void CMainWindow::OnLButtonDown( UINT, CPoint location )
{
    InvalidateLine();

    int pos = m_nVscrollPos + location.y / m_cyChar;

    if ( ( m_nSelectLine != pos )  && ( pos <
       (int)m_people.GetCount() ) )
    {
        m_nSelectLine = pos;
        InvalidateLine();
    }
    else
        m_nSelectLine = -1;
}
```

`OnLButtonDown` responds to a mouse click in the client area of the window. It checks the location of the click and calculates the line number that was clicked. Then it changes the selection to that line. If there is no selection, the member variable `m_nSelectLine` is set to -1 to indicate that status.

4. Add the following `OnLButtonDblClk` message-handler member function to VIEW.CPP below `OnLButtonDown`:

```
// CMainWindow::OnLButtonDblClk
// Translates mouse left button double click into edit person.
//
void CMainWindow::OnLButtonDblClk( UINT wParam, CPoint location )
```

```
{
    if ( m_nSelectLine == -1 )
        OnLButtonDown( wParam, location );
    OnEdit();
}
```

`OnLButtonDblClk` responds to a double-click in the window's client area. A double-click on a line of the display calls the `OnEdit` member function to allow the user to edit data for the selected person.

5. Add the following `OnKeyDown` message-handler member function to VIEW.CPP below `OnLButtonDblClk`:

```
// CMainWindow::OnKeyDown
// Translates keyboard input into scroll messages
//
void CMainWindow::OnKeyDown( UINT wParam, UINT, UINT )
{
    switch ( wParam )
    {
        case VK_HOME:
            OnVScroll( SB_TOP, 0, NULL );
            break;
        case VK_END:
            OnVScroll( SB_BOTTOM, 0, NULL );
            break;
        case VK_PRIOR:
            OnVScroll( SB_PAGEUP, 0, NULL );
            break;
        case VK_NEXT:
            OnVScroll( SB_PAGEDOWN, 0, NULL );
            break;
        case VK_LEFT:
            OnHScroll( SB_PAGEUP, 0, NULL );
            break;
        case VK_RIGHT:
            OnHScroll( SB_PAGEDOWN, 0, NULL );
            break;
    }
}
```

`OnKeyDown` implements keyboard commands for scrolling. Hence it's closely related to the scrolling member functions presented in the previous section. It's presented here as part of the keyboard interface.

To continue the tutorial, see "Add a Member Function to Handle the WM_PAINT Message" on page 235. For more information on the handlers you just added, see the discussion in the next section.

Discussion: Keyboard and Mouse Message Handlers

This section explains the keyboard and mouse message-handler member functions.

OnUp and OnDown

The Phone Book window contains lines of text, each line representing one person's data. For some actions, such as deleting or editing a person, the user must "select" a person—by selecting a line in the window—before giving a menu command. Figure 6.5 shows the display with a person selected.

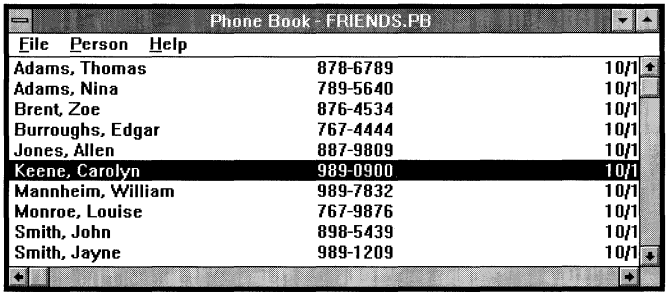


Figure 6.5 A Selection in Phone Book

There are several ways to select a person for an action. One way is to press the UP ARROW or DOWN ARROW keys. The UP ARROW moves the selection up a line. If the selection was already on the first person in the database, UP wraps the selection down to the last person in the database. The DOWN ARROW moves the selection down a line and wraps to the top. Double-clicking a line in the display both selects the person and invokes the main window object's `OnEdit` member function for it.

When the Phone Book window receives an **ON_COMMAND** message containing the ID number of either of these keys (**VK_UP** and **VK_DOWN**), the message map calls the main window object's `OnUp` or `OnDown` message handler.

`OnUp` decrements the line number of the current selection. `OnDown` increments it. Both functions also call a utility member function, `InvalidateLine` to calculate what part of the display must be repainted and to invalidate that area.

OnLButtonDown and OnLButtonDblClk

When the user presses the left mouse button over the display, the message map converts the **WM_LBUTTONDOWN** message into a call to `OnLButtonDown`. When the user double-clicks a line in the display, the message map converts the **WM_LBUTTONDBLCLK** message into a call to `OnLButtonDblClk`.

`OnLButtonDown` works somewhat like the keyboard handlers. It first invalidates the rectangle containing the currently selected line, if any. Then it calculates whether the point at which the mouse was clicked is within the range of the list of persons. If so, it sets the new selection to the line whose rectangle contains that point and invalidates the rectangle. The function causes an update so the window will be repainted with the new selection highlighted.

Similarly, when the user double-clicks the left mouse button, the window object's `OnLButtonDblClk` member function processes a double click in the display. If the mouse click location was in a line of text representing a person's data, `OnLButtonDblClk` invokes the `OnEdit` message-handler function just as if the person had been selected and then that menu command chosen.

Generally a **WM_LBUTTONDOWN** message precedes a **WM_LBUTTONDBLCLK** message, so that `OnLButtonDown` is called to select the database record. The subsequent **WM_LBUTTONDLBLCLK** message causes the `OnLButtonDblClk` member function to be called. It calls `OnEdit`.

Note The default window style is **WS_DBLCLICK**, which ensures that the double-click message is sent by Windows.

6.9 Add a Member Function to Handle the WM_PAINT Message

This section explains the eleventh step in writing Phone Book: add a message-handler function to handle the **WM_PAINT** member function.

▶ **To add the `OnPaint` message-handler:**

- Add the following `OnPaint` message-handler function to the `CMainWindow` section of your VIEW.CPP file below the `OnKeyDown` message handler and above `Save`:

```
// CMainWindow::OnPaint
// This routine does all the painting for the screen.
//
void CMainWindow::OnPaint()
{

    CPaintDC dc( this );

    // Set the Text and background colors for the DC also create a
    // Brush
    CBrush bBack;
    dc.SetTextColor( GetSysColor( COLOR_WINDOWTEXT ) );
    dc.SetBkColor( GetSysColor( COLOR_WINDOW ) );
    bBack.CreateSolidBrush( GetSysColor( COLOR_WINDOW ) );
```

```
    // Compute the lines that need to be redrawn
    int nStart = max( 0, m_nVscrollPos + dc.m_ps.rcPaint.top /
                m_cyChar - 1 );
    int nEnd = min( (int)m_people.GetCount(),
           m_nVscrollPos + ( dc.m_ps.rcPaint.bottom / m_cyChar+1 ) );

    // Create a rect the width of the display.
    CRect area( 0, 0, m_cxClient, 0 );

    CString szDisplay;
    CPerson* pCurrent;
int x,y;
    for ( ;nStart < nEnd; nStart++ )
    {
        // if the current line is the select line then change the
        // colors to the highlight text colors.
        if ( m_nSelectLine == nStart )
        {
            bBack.DeleteObject();
            bBack.CreateSolidBrush( GetSysColor( COLOR_HIGHLIGHT ) );
            dc.SetTextColor( GetSysColor( COLOR_HIGHLIGHTTEXT ) );
            dc.SetBkColor( GetSysColor( COLOR_HIGHLIGHT ) );
        }

        // x is the number of characters horz scrolled * the width of
        // char.  y is the current line no. - number of lines scrolled
        // times the height of a line.
        x = m_cxChar * ( -m_nHscrollPos );
        y = m_cyChar * ( nStart - m_nVscrollPos );

        // Set the rect to y and y + the height of the line.  Fill the
        // rect with the background color.
        area.top = y;
        area.bottom = y+ m_cyChar;
        dc.FillRect( area, &bBack );

        // Get the person and build a string with his name.
        pCurrent = m_people.GetPerson( nStart );
        szDisplay = " " + pCurrent -> GetLastName() + ", " +
                pCurrent -> GetFirstName();

        // Set the dc to write using the point as the left top of the
        // character.  Write the name.
        dc.SetTextAlign( TA_LEFT | TA_TOP );
        dc.TextOut ( x, y,szDisplay, szDisplay.GetLength() );

        // Write the phone number right aligned.
        szDisplay = pCurrent -> GetPhoneNumber();
        dc.SetTextAlign ( TA_RIGHT | TA_TOP );
        dc.TextOut ( x + SIZENAME * m_cxCaps, y, szDisplay,
            szDisplay.GetLength() );
```

```
            // Write the time.
            szDisplay = pCurrent ->
                        GetModTime().Format( "%m/%d/%y %H:%M" );
            dc.TextOut ( x + ( SIZENAME + SIZEPHONE ) * m_cxCaps, y,
                szDisplay, szDisplay.GetLength() );

            // If this is the select line then we need to reset the dc
            // colors back to the original colors.
            if ( m_nSelectLine == nStart )
            {
                bBack.DeleteObject();
                bBack.CreateSolidBrush( GetSysColor( COLOR_WINDOW ) );
                dc.SetTextColor( GetSysColor( COLOR_WINDOWTEXT ) );
                dc.SetBkColor( GetSysColor( COLOR_WINDOW ) );
            }
        }
    }
```

`OnPaint` paints the database's lines of data in the window.

To continue the tutorial, see "Add Utility Member Functions" on page 238. For more information about the `OnPaint` member function that you just added, see the following discussion.

Discussion: OnPaint

Recall from Hello that the main window object's `OnPaint` member does all painting in the window, in response to an update, which generates a **WM_PAINT** message. Of course, Phone Book paints different information in the window, so its `OnPaint` function is different from Hello's.

`OnPaint` creates some utility objects, makes some initial calculations, and then loops through the data, drawing the information for each person in the database on its own line in the window.

The utility objects include a `CPerson` object to store information about the current person being drawn, a **CPaintDC** device-context object to draw into, and a **CBrush** object to represent the Windows brush needed to paint the window background. Recall that a **CPaintDC** object is like a **CDC** device-context object except that it automatically calls **BeginPaint** and **EndPaint** for you.

The function calls the Windows function **GetSysColor** to get the background color for the window. Then it calls the **CreateSolidBrush** member function of a **CBrush** object to create a brush of the right color for the background. This brush is later used to fill areas of the screen.

Initial calculations determine the number of lines from the size of the database and compute the dimensions of the rectangle to paint based on the line height for text in the device context's current font.

The **for** loop paints lines of text for all person objects in the database. It first computes the horizontal and vertical coordinates (relative to the window) of a rectangle in which the next line is to be drawn. These are used to erase the area for the line by painting it with the background brush object. Then the line is set up and displayed in three parts.

First, the current person is retrieved from the database. The `CPerson` object's member functions are used to set up a display string containing the person's last and first names, separated by a comma and a space. Member functions of the device-context object are called to align the text and paint it in the appropriate area.

Second, the display string is reconfigured to contain the person's phone number. This is aligned and painted as a second column of information.

Third, the display string is reconfigured to contain the person's modification date. This is aligned and painted as a third column of information.

If the line currently being displayed is the line containing the "selection," the rectangle containing the line's text is inverted to highlight the selection. Only one person (and one line of the display) can be selected at one time.

When all lines of data have been painted, `OnPaint` returns.

6.10 Add Utility Member Functions

This section explains the twelfth step in writing Phone Book: add utility member functions. These are called by message-handler functions but are not message handlers themselves.

▶ To add the utility member functions:

1. Add the following `FileDlg` utility member function to the `CMainWindow` section of your VIEW.CPP file below the `OnPaint` member function and above `Save`:

```
// CMainWindow::FileDlg
// Call the commdlg routine to display File Open or File Save As
// dialogs.  The setup is the same for either.  If bOpen is TRUE
// then File Open is displayed otherwise File Save As is displayed.
// The File Name and File Title are stored at the string pointer
// passed in.
//
```

```
BOOL CMainWindow::FileDlg( BOOL bOpen, int nMaxFile, LPSTR szFile,
        int nMaxFileTitle, LPSTR szFileTitle )
{

    OPENFILENAME of;

    char szDirName[SIZESTRING];
    char * szFilter[] =
        "Phone Book Files (*.pb)\0"
        "*.pb\0"
        "\0";

    szDirName[0] = '.';

    of.lStructSize = sizeof( OPENFILENAME );
    of.hwndOwner = m_hWnd;
    of.lpstrFilter = (LPSTR)szFilter;
    of.lpstrCustomFilter = (LPSTR)NULL;
    of.nMaxCustFilter = 0L;
    of.nFilterIndex = 1L;
    of.lpstrFile=szFile;
    of.nMaxFile=nMaxFile;
    of.lpszFileTitle = szFileTitle;
    of.nMaxFileTitle = nMaxFileTitle;
    of.lpstrInitialDir = szDirName;
    of.lpstrTitle = (LPSTR)NULL;
    of.nFileOffset = 0;
    of.nFileExtension = 0;
    of.lpstrDefExt = (LPSTR)"pb";
    if ( bOpen )
    {
        of.Flags = OFN_HIDEREADONLY;
        return GetOpenFileName( &of );
    }
    else
    {
        of.Flags = OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT;
        return GetSaveFileName( &of );
    }
}
```

`FileDlg` is a utility function that displays a standard Windows open file dialog box. If the call is to open a file, `FileDlg` calls the **GetOpenFileName** function defined in COMMDLG.DLL. Otherwise, it calls the **GetSaveFileName** function. For more discussion of `FileDlg`, see "OnOpen" on page 214.

2. Add the following `CheckForSave` utility member function to the `CMainWindow` section of your VIEW.CPP file below the `Save` member function:

```
// CMainWindow::CheckForSave
// Whenever a new file is opened this routine will determine if
// there are unsaved changes in the current database. If so it
// will query the user and determine to save or not as appropriate.
//
BOOL CMainWindow::CheckForSave( const char* pszTitle,, const char*
pszMessage )
{
    if( m_people.IsDirty() )
    {
        UINT nButton = MessageBox( pszMessage,, pszTitle,,
                       MB_YESNOCANCEL );
        if( nButton == IDYES )
        {
            if( !Save( m_people.IsNamed() )
                return FALSE;
        }
        else if( nButton == IDCANCEL )
            return FALSE;
    }
    return TRUE;
}
```

`CheckForSave` is called from a number of other member functions that need to check for a current database with unsaved changes. If there are unsaved changes, the function uses the `MessageBox` member function of class **CWnd** to query the user's wishes. If the user clicks the message box's Yes button, the `Save` member function is called to do the work.

3. Add the following `SetMenu` utility member function to the `CMainWindow` section of your VIEW.CPP file below the `CheckForSave` member function:

```
// CMainWindow::SetMenu
// Whenever the Existence of the DataBase is changed this
// routine will reset the menus so only the possible commands
// are accessible.
//
void CMainWindow::SetMenu()
{
    CMenu* pMenu = GetMenu();
    if ( m_people.IsPresent() )
    {
        if ( m_people.IsNamed() )
            pMenu -> EnableMenuItem( IDM_SAVE, MF_ENABLED );
        else
            pMenu -> EnableMenuItem( IDM_SAVE, MF_GRAYED );
        pMenu -> EnableMenuItem( IDM_SAVEAS, MF_ENABLED );
        pMenu -> EnableMenuItem( IDM_CLOSE, MF_ENABLED );
        pMenu -> EnableMenuItem( IDM_PRINT, MF_ENABLED );
```

```
        pMenu -> EnableMenuItem( IDM_ADD, MF_ENABLED );
        pMenu -> EnableMenuItem( IDM_DELETE, MF_ENABLED );
        pMenu -> EnableMenuItem( IDM_FIND, MF_ENABLED );
        pMenu -> EnableMenuItem( IDM_EDIT, MF_ENABLED );
    }
    else
    {
        pMenu -> EnableMenuItem( IDM_SAVE, MF_GRAYED );
        pMenu -> EnableMenuItem( IDM_SAVEAS, MF_GRAYED );
        pMenu -> EnableMenuItem( IDM_CLOSE, MF_GRAYED );
        pMenu -> EnableMenuItem( IDM_PRINT, MF_GRAYED );
        pMenu -> EnableMenuItem( IDM_ADD, MF_GRAYED );
        pMenu -> EnableMenuItem( IDM_DELETE, MF_GRAYED );
        pMenu -> EnableMenuItem( IDM_FIND, MF_GRAYED );
        pMenu -> EnableMenuItem( IDM_FINDALL, MF_GRAYED );
        pMenu -> EnableMenuItem( IDM_EDIT, MF_GRAYED );
    }
}
```

`SetMenu` is a utility function that updates the menus to reflect the current context. It disables (grays or dims) menu commands that are not valid currently and enables menu commands that are valid.

4. Add the following `InvalidateLine` utility member function to the `CMainWindow` section of your VIEW.CPP file below the `SetMenu` member function:

```
// CMainWindow::InvalidateLine
// Marks the screen area of the currently selected person as
// invalid causing windows to call OnPaint to redraw the area.
// This is normally used when the selected line is being changed.
//
void CMainWindow::InvalidateLine()
{
    CRect area( 0, ( m_nSelectLine - m_nVscrollPos ) * m_cyChar,
                   m_cxClient,
                ( m_nSelectLine + 1 - m_nVscrollPos ) * m_cyChar );
    InvalidateRect( area );
}
```

`InvalidateLine` is a utility function that marks the screen area of the currently selected person as invalid. This causes Windows to send a **WM_PAINT** message so that `OnPaint` is called to repaint the area in the window.

This completes the code in file VIEW.CPP. You can compare your file to Listing 2 in Chapter 5. All code for VIEW.H and VIEW.CPP is now complete.

To continue the tutorial, see the next section.

6.11 Prepare Supporting Files

Besides the six source code files, PERSON.H and PERSON.CPP from Chapter 2, DATABASE.H and DATABASE.CPP from Chapter 4, and VIEW.H, and VIEW.CPP from Chapter 5, you will need several additional files to compile Phone Book. The files will look like those shown in Listings 1 through 3. You already started a PHBOOK.RC resource script file in Chapter 5.

► **To prepare these files, do the following:**

1. Create a module-definition file, with the .DEF extension.

   The module-definition file specifies the Windows programming environment. It sets such values as the stack size, the heap size, and the module-loading parameters that Windows uses. All Windows programs require a module-definition file.

   You can use the module-definition file for Phone Book, PHBOOK.DEF, on the distribution disks, or you can copy HELLO.DEF, rename the copy, and modify it slightly. Change references to Hello, and make any other changes you want. The file appears in Listing 1.

2. Complete the resource file, with the .RC extension, that you started earlier in Chapter 5.

   Check your PHBOOK.RC file against Listing 2 and add any missing resource templates. Listing 2 shows the complete resource script file for Phone Book. The items in this file were explained in some detail in Chapter 5.

   To build Phone Book, check that your PHBOOK.RC file has the following resource templates (see Listing 2):

   - An icon resource that specifies the file containing an icon for the program
   - A menu resource that specifies the File, Person, and Help menus
   - An accelerator table resource that specifies five accelerator-key mappings
   - Dialog resources for an About dialog box, two data-entry dialogs, and three Help dialogs

3. Create a resource include file with an .H extension.

   The resource include file defines resource ID numbers that the program uses to associate resources with their templates in the .RC file.

   You can use the resource include file for Phone Book, RESOURCE.H, on the distribution disks, or you can copy RESOURCE.H from Hello and modify it. The file appears in Listing 3.

For more information about module-definition files and resources, see the *Windows SDK Guide to Programming*, the *Windows SDK Reference, Volume 2*, and your Microsoft C/C++ documentation.

At this point, your Phone Book files are complete if you have followed all of the directions in this tutorial. You can check them against the same files on the distribution disks.

6.12 Build the Program

To build your program, follow the instructions given in Chapter 1 of the tutorial. The required files to build Phone Book are PERSON.H, PERSON.CPP, DATABASE.H, DATABASE.CPP, VIEW.H, VIEW.CPP, PHBOOK.DEF, PHBOOK.RC, PHBOOK.ICO, and PHBOOK.DLG. All are available in the MFC\SAMPLE\TUTORIAL directory.

The Programmer's WorkBench (PWB) makefile for Phone Book is called PHBOOK.MAK. The NMAKE makefile is called PHBOOK with no extension.

PHBOOK.EXE builds as a Windows application, so you must run it from Microsoft Windows.

6.13 Summary

This chapter and the two preceding chapters presented a larger, more complete Windows application written with the Microsoft Foundation Class Library.

The principal techniques demonstrated include:

- How to integrate the data model with the Windows interface.
- How to interact with the user through dialog-box objects.
- How to respond to a wide variety of menu commands.
- How to respond to keyboard and mouse commands.

To solidify your Windows programming skills using the Microsoft Foundation Class Library, try modifying the Phone Book program to add address information. You'll need to modify class `CPerson` to hold more data, modify class `CEditPerson` to process additional dialog data-entry fields, modify the message-handler functions of class `CMainWindow` wherever they access the internals of a `CPerson`, and modify the dialog resource template for `CEditDialog` to specify additional data fields. While you're at it, improve the search capabilities as well, so the user can search for other data items besides last name. By the time you've made these modifications, you may be ready to write your own Windows program with the Microsoft Foundation Class Library.

As you move on to write your own Windows programs with the Microsoft Foundation Classes, use your work in this tutorial as a foundation or template on

which to build other applications. Use the cookbook chapters of this manual (chapters 7 through 17) for valuable guidance on specific Microsoft Foundation Class Library programming tasks. And consult the *Class Libraries Reference* for more information on the many capabilities of the Microsoft Foundation Classes.

6.14 File Listings

The code shown in listings 1-3 is available in your distribution disks as PHBOOK.DEF, PHBOOK.RC, and RESOURCE.H.

Listing 1

```
; phbook.def: Defines the module parameters for the application.
;
; This is part of the Microsoft Foundation Classes C++ Library.
; Copyright (C) 1991 Microsoft Corporation
; All rights reserved.
;
; This source code is only intended as a supplement to the
; Microsoft Foundation Class Reference and Microsoft
; QuickHelp documentation provided with the library.
; See these sources for detailed information regarding the
; Microsoft Foundation Classes product.

NAME                    PhBook
DESCRIPTION             'Phone Book Database'

EXETYPE                 WINDOWS
STUB                    'WINSTUB.EXE'

CODE PRELOAD FIXED DISCARDABLE
DATA PRELOAD FIXED MULTIPLE

HEAPSIZE                8096
STACKSIZE               8096
```

Listing 2

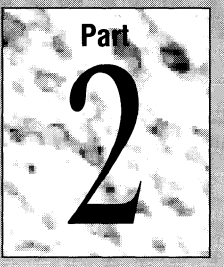
```
/* PhBook.rc : Defines the resources for the application.
//
// This is a part of the Microsoft Foundation Classes C++ Library.
// Copyright (C) 1991 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Microsoft Foundation Classes Reference and Microsoft
// QuickHelp documentation provided with the library.
// See these sources for detailed information regarding the
// Microsoft Foundation Classes product.
*/

#include <windows.h>
#include "resource.h"
#include <afxres.h>

AFX_IDI_STD_FRAME    ICON    phbook.ico

MainMenu MENU
{
    POPUP "&File"
    {
        MENUITEM "&New",              IDC_NEW
        MENUITEM "&Open...",          IDC_OPEN
        MENUITEM "&Save",             IDC_SAVE,    GRAYED
        MENUITEM "Save &As...",       IDC_SAVEAS,  GRAYED
        MENUITEM "&Close",            IDC_CLOSE,   GRAYED
        MENUITEM "&Print...",         IDC_PRINT,   GRAYED
        MENUITEM "E&xit",             IDC_EXIT
    }
    POPUP "&Person"
    {
        MENUITEM "&Add...",           IDC_ADD,     GRAYED
        MENUITEM "&Delete",           IDC_DELETE,  GRAYED
        MENUITEM "&Find...",          IDC_FIND,    GRAYED
        MENUITEM "F&ind All",         IDC_FINDALL, GRAYED
        MENUITEM "&Edit...",          IDC_EDIT,    GRAYED
    }
    POPUP "&Help"
    {
        MENUITEM "&Using Phone Book\tF1",  IDC_HELP
        MENUITEM "&About Phone Book...",   IDC_ABOUT
    }
}
```

```
MainAccelTable ACCELERATORS
{
    VK_F1,              IDC_HELP,       VIRTKEY
    VK_DELETE,    IDC_DELETE,    VIRTKEY
    VK_RETURN,   IDC_EDIT,          VIRTKEY
    VK_UP,              VK_UP,              VIRTKEY
    VK_DOWN,      VK_DOWN,          VIRTKEY
}

rcinclude phbook.dlg
```

Listing 3

```
; resource.h: Defines the resource constants for the application.

#define IDM_NEW             101
#define IDM_OPEN            102
#define IDM_SAVE            103
#define IDM_SAVEAS          104
#define IDM_CLOSE           105
#define IDM_PRINT           106
#define IDM_EXIT            107

#define IDM_ADD             108
#define IDM_DELETE          109
#define IDM_FIND            110
#define IDM_FINDALL         111
#define IDM_EDIT            112

#define IDM_HELP            113
#define IDM_ABOUT           100

#define IDC_DATA            115

#define IDC_STATICLASTNAME  300
#define IDC_LASTNAME        250
#define IDC_STATICFIRSTTNAME 301
#define IDC_FIRSTNAME       251
#define IDC_STATICPHONE     302
#define IDC_PHONE           252
#define IDC_STATICMOD       303
#define IDC_MOD             253
```

The Microsoft Foundation Class Library Cookbook



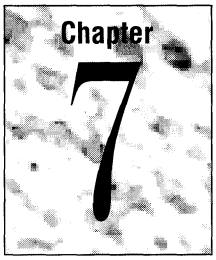
The Microsoft Foundation Class Library Cookbook

Part 2 of the Microsoft Foundation Class Libraries User's Guide contains a cookbook that provides step-by-step for using the Microsoft Foundation Classes to solve common programming tasks. These recipes typically include example code to illustrate the best way to take advantage of the Microsoft Foundation Class Library's features.

Chapters 7 through 12 show how to use the fundamental general-purpose classes for non-Windows programming tasks, including memory management, date and time, strings, collections, files and serialization, diagnostics, and exception handling.

Chapters 13 through 17 show how to use the Microsoft Foundation Classes that relate to Windows programming tasks such as application design, window and dialog management, graphics operations, and user input.

General-Purpose Classes



The Microsoft Foundation Class Library provides services to make programming easier. These services range from general-purpose memory-management services to more advanced Windows facilities. This chapter describes how to take advantage of the general-purpose services related to memory management, time and date management, and string manipulation.

7.1 Memory Management

Memory allocation can be divided into two main categories: frame allocations and heap allocations. One main difference between the two allocation techniques is that with heap allocation you are always given a pointer to the memory block, whereas with frame allocation you typically work with the actual memory block itself. Another major difference between the two schemes is that frame objects are automatically deleted and heap objects must be explicitly deleted by the programmer.

The following sections describe how to use the capabilities of C and C++ to accomplish memory allocations on the frame and on the heap.

Frame Allocation

Allocation on the frame takes its name from the "stack frame" that is set up whenever a function is called. The stack frame is an area of memory that temporarily holds the arguments to the function and any local variables that are defined local to the function. Frame variables are often called "automatic" variables because the compiler automatically allocates the space for them.

There are two key characteristics of frame allocations. First, when you define a local variable enough space is allocated on the stack frame to hold the entire variable, even if it is a large array or data structure. Second, frame variables are automatically deleted when they go out of scope. For local function variables, this scope transition happens when the function exits, but the scope of a frame variable

can be smaller than a function if nested braces are used (or larger, in the case of global variables). This automatic deletion of frame variables is very important. In the case of simple primitive types (such as **int** or **byte**), arrays, or data structures, the automatic deletion simply reclaims the memory used by the variable. Since the variable has gone out of scope, it cannot be accessed anyway. In the case of C++ objects, however, the process of automatic deletion is a bit more complicated.

When an object is defined as a frame variable, its constructor is automatically invoked at the point where the definition is encountered. When the object goes out of scope, its destructor is automatically invoked before the memory for the object is reclaimed. This automatic construction and destruction can be very handy, but you must be aware of the automatic calls, especially to the destructor.

Objects allocated on the frame are automatically deleted.

The key advantage of allocating objects on the frame is that they are automatically deleted. When you allocate your objects on the frame, you don't have to worry about forgotten objects causing memory leaks. (For details on memory leaks, see "Detecting Memory Leaks" on page 290.) A disadvantage of frame allocation is that frame variables cannot be used outside their scope. Another factor in choosing frame vs. heap allocation is that for large structures and objects, it is often better to use the heap instead of the stack for storage, since stack space is often limited.

Heap Allocation

The heap is reserved for the memory allocation needs of the program. It is an area apart from the program code and from the stack. Typical C programs use the functions **malloc** and **free** to allocate and deallocate memory to and from the heap. The Debug version of the Microsoft Foundation Class Library provides modified versions of the C++ built-in operators **new** and **delete** to allocate and deallocate objects in heap memory. When you use **new** and **delete** instead of **malloc** and **free** you are able to take advantage of the Foundation's memory-management debugging enhancements, which can be useful in detecting memory leaks. When you build your program with the Release version of the Microsoft Foundation Class Library, **new** and **delete** still provide an efficient way to allocate and deallocate memory.

Memory Allocation on the Heap and on the Frame

There are three typical kinds of memory allocations:

- An array of bytes
- A data structure
- An object

The following sections describe how the Microsoft Foundation Class Library facilities perform each of these typical tasks for both heap allocation and frame allocation.

Allocation of an Array of Bytes

► **To allocate an array of bytes on the frame:**

- Define the array as shown by the following code. The array is automatically deleted and its memory reclaimed when the array variable exits its scope.

```
{
    const int BUFF_SIZE = 128;

   // allocate on the frame
  char myCharArray[BUFF_SIZE];
  int myIntArray[BUFF_SIZE];
  // reclaimed when exiting scope
}
```

► **To allocate an array of bytes (or any primitive data type) on the heap:**

- Use the **new** operator with the following array syntax:

```
const int BUFF_SIZE = 128;

// allocate on the heap
char* myCharArray = new char[BUFF_SIZE];
int* myIntArray = new int[BUFF_SIZE];
```

► **To deallocate the arrays from the heap:**

- Use the **delete** operator as follows:

```
delete [] myCharArray;
delete [] myIntArray;
```

Allocation of a Data Structure

► **To allocate a data structure on the frame:**

- Define the structure variable as follows:

```
struct MyStructType {...};
void SomeFunc(void)
{
    // frame allocation
    MyStructType myStruct;
```

```
    // use the struct
    myStruct.topScore = 297;

    // reclaimed when exiting scope
}
```

The memory occupied by the structure is reclaimed when it exits its scope.

► **To allocate data structures on the heap:**

- Use **new** to allocate data structures on the heap and deallocate them with **delete** as shown by the following examples:

```
// heap allocation
MyStructType* myStruct = new MyStructType;

// use the struct through the pointer ...
myStruct->topScore = 297;

delete myStruct;
```

Allocation of an Object

► **To allocate an object on the frame:**

- Declare the object as follows:

```
{
    CPerson myPerson;     // automatic constructor call here

    myPerson.SomeMemberFunction();     // use the object

 }
```

The destructor for the object is automatically invoked when the object exits its scope.

► **To allocate an object on the heap:**

- Use the **new** operator, which returns a pointer to the object, to allocate objects on the heap. Use the **delete** operator to delete them.

```
// automatic constructor call here
CPerson* myPerson = new CPerson;

myPerson->SomeMemberFunction();     // use the object

delete myPerson;              // destructor invoked during delete
```

Both the heap and the frame examples assumed that the `CPerson` constructor takes no arguments. Assume that the argument for the `CPerson` constructor is a pointer to **char**.

The statement for frame allocation is:

```
CPerson myPerson( "Joe Smith" );
```

The statement for heap allocation is:

```
CPerson* MyPerson = new CPerson( "Joe Smith" );
```

Resizable Memory Blocks

The **new** and **delete** operators described above are good for allocating and deallocating fixed-size memory blocks and objects. Occasionally, your application may need resizable memory blocks. You must use the standard C run-time library functions **malloc**, **realloc**, and **free** to manage resizable memory blocks on the heap.

Mixing the **new** and **delete** operators and the resizable memory-allocation functions on the same memory block will result in corrupted memory in the **Debug** version of the Foundation Class Library. That is, you should not allocate a memory block with **new** and deallocate it with **free**. Likewise, you should not use the C++ **delete** operator on a memory block allocated with **malloc** and you should not use **realloc** on a memory block allocated with **new**.

7.2 Date and Time

The **CTime** class provides a way to represent date and time information easily. The **CTimeSpan** class represents elapsed time, such as the difference between two **CTime** objects.

Note **CTime** objects cannot be used to represent dates earlier than January 1, 1980. **CTime** objects have a resolution of 1 second.

The first procedure in this section shows how to create a **CTime** object and initialize it with the current time. The next procedure shows how to calculate the difference between two **CTime** objects and get a **CTimeSpan** result.

► To get the current time:

1. Allocate a **CTime** object, as follows:

```
CTime theTime;
```

Note Uninitialized **CTime** objects are automatically set to an invalid time.

2. Call the **CTime::GetCurrentTime** function to get the current time from the operating system. This function returns a **CTime** object that can be used to set the value of **CTime**, as follows:

```
theTime = CTime::GetCurrentTime();
```

Since **GetCurrentTime** is a static member function from the **CTime** class, you must qualify its name with the name of the class and the scope resolution operator (**::**), `CTime::GetCurrentTime()`.

Of course, the two steps outlined above could be combined into a single program statement as follows:

```
CTime theTime = CTime::GetCurrentTime();
```

► **To calculate elapsed time:**

- Use the **CTime** and **CTimeSpan** objects to calculate the elapsed time, as follows:

```
CTime startTime = CTime::GetCurrentTime();

// ... perform time-consuming task ...

CTime endTime = CTime::GetCurrentTime();

CTimeSpan elapsedTime = endTime - startTime;
```

Once you have calculated `elapsedTime`, you can use the member functions of **CTimeSpan** to extract the components of the elapsed-time value.

► **To format a string representation of a time or elapsed time:**

- Use the **Format** member function from either the **CTime** or **CTimeSpan** classes to create a character string representation of the time or elapsed time, as shown by the following example.

```
CTime t( 1991, 3, 19, 22, 15, 0 ); // 10:15PM March 19, 1991
CString s = t.Format( "%A, %B %d, %Y" );
// s == "Tuesday, March 19, 1991"
```

7.3 Strings

The **CString** class provides support for manipulating strings. It is intended to replace and extend the functionality normally provided by the C run-time library string package.

A **CString** object represents a sequence of a variable number of characters. **CString** objects can be thought of as arrays of single-byte characters.

A **CString** object can store up to 32,766 characters. The normal C **char** data type is used to get or set individual characters inside a **CString** object. **CString** objects are automatically growable (that is, you don't have to worry about growing a **CString** object to fit longer strings). A **CString** object also can act like a literal C-style string (a pointer to **const char**).

Basic Operations

The **CString** class provides member functions and overloaded operators that duplicate and in some case surpass the string services of the C run-time libraries (for example, **strcat**). The following sections describe some of the main operations of the **CString** class.

► **To create CString objects from standard C literal strings:**

- Assign the value of a C literal string to a **CString** object:

```
CString myString = "This is a test";
```

- Assign the value of one **CString** to another **CString**:

```
CString oldString = "This is a test";
CString newString = oldString;
```

As explained more completely in the next section on value semantics, the contents of a **CString** are copied when one string is assigned to another **CString**. Thus, the two strings do not share a reference to the actual characters that make up the string.

► **To access individual characters in a CString:**

- You can access individual characters within a **CString** with the **GetAt** and **SetAt** member functions. You can also use the array element operator ([]) instead of **GetAt** to get individual characters (like accessing array elements by index as in standard C-style strings). Index values for **CString** characters are zero-based.

► **To concatenate two CStrings:**

- Use the concatenation operators (+ or +=) as follows:

```
CString s1 = "This ";      //cascading concatenation
s1 += "is a ";
CString s2 = "test";
CString message = s1 + "big " + s2;
//message contains "This is a big test"
```

At least one of the arguments to the concatenation operators (+ or +=) must be a **CString** object, but you can use a constant character string (such as `"big"`) or a **char** (such as `'x'`) for the other argument.

► **To compare two CStrings:**

- While the overloaded equality operator (==) and the **Compare** member functions will determine if two **CString** objects are equivalent character for character, you can also use the **CompareNoCase** and **Collate** member functions to do comparisons that are case insensitive and national- language sensitive. The following table shows the three available **CString** comparison functions and their equivalent C run-time string functions.

CString function	C run-time function
Compare	**strcmp**
CompareNoCase	**stricmp**
Collate	**strcoll**

The **CString** class overrides the relational operators (<, <=, >=, >, ==, and !=) to use the **Compare** function, so you can compare two **CStrings** using these operators, as shown here:

```
CString s1( "Tom" );
CString s2( "Jerry" );
if( s1 < s2 )
    ...
```

CStrings Are Values

Think of CString objects as actual strings.

Even though they are dynamically growable objects, **CString** objects act like built-in primitive types and simple classes. Each **CString** object represents a unique value. **CString** objects should not be thought of as pointers to strings but as the actual strings.

The most obvious consequence of value semantics is that the string contents are copied when you assign one **CString** to another. Thus, even though two **CStrings** may represent the same sequence of characters, they do not share those characters. Each **CString** has its own copy of the character data. When you modify one **CString** object, the copied **CString** object is not modified, as shown by the following example:

```
CString s1, s2;
s1 = s2 = "hi there";

if( s1 == s2 )        // TRUE - they are equal
    ...
```

```
s1.MakeUpper();     // does not modify s2
if( s2[0] == 'h' )     // TRUE - s2 is still "hi there"
```

Notice in the example that the two **CString** objects are considered to be "equal" because they represent the same character string. The **CString** class overloads the equality operator (==) to compare two **CString** objects based on their value (contents) rather than their identity (address).

► **To specify CString formal parameters correctly:**

- For most functions that need a string argument, it is best to specify the formal parameter in the function prototype as a pointer to **const char** (**const char*** or **const char FAR***) instead of a **CString**. With a formal parameter specified as a pointer to **const char**, you can pass either a pointer to a **char** array, a literal string (`"hi there"`), or a **CString** object. The **CString** will be automatically converted to a pointer to **const char**. Any place you can use a pointer to **char**, you can also use a **CString**.
- You can also specify a formal parameter as a constant string reference (that is, **const CString&**) if the argument will not be modified. Drop the **const** modifier if the string will be modified by the function. If a default null value is desired, initialize it to the null string (`""`), as shown below:

```
void AddCustomer( const CString& name,
                  const CString& address,
                  const CString& comment = "" );
```

- For most function results, you can simply return a **CString** object by value.

Operations Related to C-Style Strings

It is often useful to be able to manipulate the contents of a **CString** object as if it were a C-style null-terminated string.

► **To convert to C-style null-terminated strings:**

- In the simplest case, you can cast a **CString** object to be a pointer to **const char**. The **const char*** type conversion operator returns a read-only pointer to a C-style null-terminated string from a **CString** object.

  The pointer to **char** returned by the implicit conversion shown above points into the data area used by the **CString**. If the **CString** goes out of scope and is automatically deleted or something else changes the contents of the **CString**, the **char** pointer will no longer be valid. You should treat this pointer as a temporary read-only pointer. Do not directly modify the characters pointed to.

- You can use **CString** functions such as **SetAt** to modify individual characters in the string object, but if you need a copy of a **CString** object's characters that you can modify directly, use **strcpy** to copy the **CString** object into a separate buffer where the characters can be safely modified, as shown by the following example:

```
CString theString( "This is a test" );
char* psz = new char[theString.GetLength()];
strcpy( psz,theString );
//... modify psz as much as you want
```

Note The second argument to **strcpy** is declared as a constant pointer to **char** (**const char***). The example above passes a **CString** for this argument. The C++ compiler automatically applies the conversion function defined for the **CString** class that converts a **CString** to a **const char***. The ability to define casting operations from one type to another is one of C++'s most useful features.

▶ To work with standard C-library string functions:

- In most situations, you should be able to find **CString** member functions to perform any string operation for which you might consider using the standard C run-time library string functions, such as **strcmp**.
- If you find that you must use the C run-time string functions, you can use the techniques described in the previous procedure to copy the **CString** object to an equivalent C-style string buffer, perform your operations on the buffer, and then assign the resulting C-style string back to a **CString** object.

▶ To modify CString contents directly with GetBuffer and ReleaseBuffer:

- In most situations, you should use **CString** member functions to modify the contents of a **CString** object, or convert the **CString** to a C-style character string as described in the pervious section.
- However, there are certain situations, such as working with operating system functions that require a character buffer, where it is advantageous to directly modify the **CString** contents.

  The **GetBuffer** and **ReleaseBuffer** member functions allow you to gain access to the internal character buffer of a CString and modify it directly. The following steps show how to use these functions for this purpose.

  1. Call **GetBuffer** for a **CString** object, specifying the length of the buffer your require.
  2. Use the pointer returned by **GetBuffer** to write characters directly into the **CString** object.
  3. Call **ReleaseBuffer** for the **CString** object to update all the internal **CString** state information (such as the length of the string). After modifying a

CString object's contents directly, you must call **ReleaseBuffer** before calling any other **CString** member functions.

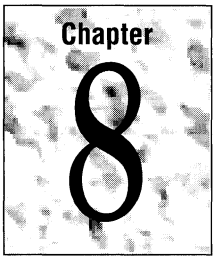
▶ To use CString objects with variable argument functions:

Some C functions take a variable number of arguments. A notable example is **printf**. Because of the way this kind of function is declared, the compiler cannot be sure of the type of the arguments and cannot determine which conversion operation to perform on the argument. Therefore, it is essential that you use an explicit type cast when passing a **CString** object to a function that takes a variable number of arguments.

- Explicitly cast the **CString** to a pointer to a constant **char** string, as shown here:

```
CString kindOfFruit = "bananas";
int   howmany = 25;
printf( "You have %d %s\n", howmany, (const char*)kindOfFruit );
```

The CObject Class



CObject is the root class for most of the Microsoft Foundation Class Library. The Foundation Class **CObject** contains many useful features that you may want to incorporate into your own program objects, including support for serialization, debugging output, and run-time class information. If you derive the class from **CObject**, your class can exploit these **CObject** features.

8.1 How to Derive a Class from CObject

This section describes the minimum steps necessary to derive a class from **CObject**. Other cookbook sections describe the steps needed to take advantage of specific **CObject** features, such as serialization and diagnostic debugging support.

In the following discussions, the terms "interface file" and "implementation file" are used frequently. The interface file (often called the header file, or .H file) contains the class declaration and any other information needed to use the class. The implementation file (or .CPP file) contains the code that implements the class member functions. For example, for a class named `CPerson`, you will typically create an interface file named PERSON.H and an implementation file named PERSON.CPP. For some small classes that will not be shared, it is sometimes easier to combine the interface and implementation into a single .CPP file.

There are three levels of functionality that you can choose from when deriving a class from **CObject**:

- Basic functionality that does not include run-time class information or serialization
- Basic functionality plus run-time class information
- Basic functionality plus run-time class information plus serialization support

Classes designed for reuse should at least include run-time class support and serialization support if any future serialization need is anticipated.

You choose the level of functionality by using specific declaration and implementation macros in the declaration and implementation of the classes you derive from **CObject**. The following sections describe how to specify the level of functionality.

▶ To use basic CObject functionality:

- Use the normal C++ syntax to derive your class from **CObject** (or from a class derived from **CObject**).

  The following example shows the simplest case: the derivation of a class from **CObject:**

```
class CPerson : public CObject
{
    // add CPerson-specific members and functions...
}
```

  Typically, however, you will want to override some of **CObject**'s member functions to handle the specifics of your new class. For example, you will usually want to override the **Dump** function of **CObject** to provide debugging output for the contents of your class. For details on how to override **Dump**, see page 286. You will also want to override the **AssertValid** function of **CObject**. (For a description of how to override **AssertValid**, see the section on **AssertValid** on page 289).

▶ To add run-time class information:

CObject contains support for run-time class information. This means that you can determine the exact class of an object at run-time and also determine the base class from which it was derived. This capability is not supported directly by the C++ language. To take advantage of the run-time class information, you must do the following three steps:

1. Derive your class from **CObject**, as described in the previous section.
2. Use the **DECLARE_DYNAMIC** macro in your class declaration, as shown here:

```
class CPerson : public CObject
{
    DECLARE_DYNAMIC( CPerson )

    // rest of class declaration follows...
};
```

3. Use the **IMPLEMENT_DYNAMIC** macro in the implementation file (.CPP) of your class. This macro takes as arguments the name of the class and the name of its base class, as follows:

```
IMPLEMENT_DYNAMIC( CPerson, CObject )
```

Note The **IMPLEMENT_DYNAMIC** macro should be evaluated only one time during a compilation. Do not use **IMPLEMENT_DYNAMIC** in an interface file (.H) that will be included in more than one file. The best policy is to always put **IMPLEMENT_DYNAMIC** in the implementation file (.CPP) for your class.

If you use these two macros as described, you can then use the **RUNTIME_CLASS** macro and **IsKindOf** member function to determine the class of your objects at run-time. For a description of how to use these features of **CObject**, see "How to Access Run-Time Class Information" on this page.

► **To add serialization support:**

Serialization is the process of writing or reading the contents of an object to and from a file. The following five steps are required to support serialization in your classes.

1. Derive your class from **CObject**.
2. Use the **DECLARE_SERIAL** macro in the class declaration.
3. Define a constructor with no arguments (a default constructor).
4. Use the **IMPLEMENT_SERIAL** macro in the class implementation file.
5. Override the **Serialize** member function.

For more details on how to derive from **CObject** to enable serialization for your class, see "Serialization" on page 279. Each of the steps listed above is described in that section.

Note The serialization features necessarily include the run-time class information features described in the previous section. If you use the **DECLARE_SERIAL** and **IMPLEMENT_SERIAL** macros, you do not have to use the **DECLARE_DYNAMIC** and **IMPLEMENT_DYNAMIC** macros. The run-time class macros are already included in the serialization macros.

8.2 How to Access Run-Time Class Information

If you have derived your class from **CObject** and used the **DECLARE_DYNAMIC** and **IMPLEMENT_DYNAMIC** macros explained previously, the **CObject** class has the ability to determine the exact class of an object at run-time. This ability is an extension of the normal capabilities of C++.

The ability to determine the class of an object at run-time is most useful when extra type checking of function arguments is needed and when you must write special-purpose code based on the class of an object.

The **CObject** member function **IsKindOf** can be used to determine if a particular object belongs to a specified class or if it is derived from a specific class. The argument to **IsKindOf** is a **CRuntimeClass**, which you can get by using the **RUNTIME_CLASS** macro with the name of the class. The use of the **RUNTIME_CLASS** macro is shown below.

▶ To use the RUNTIME_CLASS macro:

- Use **RUNTIME_CLASS** with the name of the class, as shown here for the class **CObject**:

```
CRuntimeClass* pClass = RUNTIME_CLASS( CObject );
```

You will rarely need to access the run-time class object directly. A more common use is to pass the run-time class object to the **IsKindOf** function, as shown in the next section.

▶ To use the IsKindOf function:

IsKindOf will test an object to see if it belongs to a particular class. The following steps show how to use **IsKindOf**.

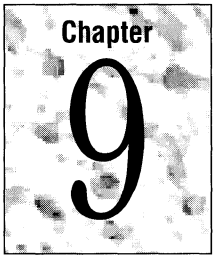
1. Make sure the class has run-time class support. That is, the class must have been derived from **CObject** and used the **DECLARE_DYNAMIC** and **IMPLEMENT_DYNAMIC** macros explained previously on pages 264-265. (Since serialization support requires run-time class support, the class may have been defined with the **DECLARE_SERIAL** and **IMPLEMENT_SERIAL** macros.)
2. Call the **IsKindOf** member function for objects of that class, using the **RUNTIME_CLASS** macro to generate the **CRuntimeClass** argument, as shown here:

```
// in .H file
class CPerson : public CObject
{
    DECLARE_DYNAMIC( CPerson )
public:
    CPerson(){};

    // other declaration stuff left out...
};

// in .CPP file
IMPLEMENT_DYNAMIC( CPerson, CObject )

CObject* myObject = new CPerson;

if( myObject->IsKindOf( RUNTIME_CLASS( CPerson ) ) )
{
   //if IsKindOf is true, then cast is alright
```

```
    CPerson* myPerson = (CPerson*) myObject;
}
```

Note **IsKindOf** returns **TRUE** if the object is a member of the specified class or of a class derived from the specified class. It only works properly in single-inheritance hierarchies.

Collections



The Microsoft Foundation Class Library provides collection classes to manage groups of objects. A collection class is characterized by its "shape" and by the type of its elements. The shape refers to the way the objects are organized and stored by the collection. The Microsoft Foundation Class Library provides three basic collection shapes: arrays, lists, and maps (also known as dictionaries). You can pick the collection shape most suited to your particular programming problem.

Table 9.1 lists the characteristics of the three collection shapes provided with the Microsoft Foundation Class Library. In the table, the term "ordered" means that the order of the items in the collection is determined by the order in which they were inserted and deleted. It does not mean that the items are sorted based on their contents. The term "indexed" means that the items in the collection can be retrieved by an integer index, much like a typical array structure.

Each of the three provided collection shapes is described briefly below, followed by Table 9.1, which compares the features of the shapes to help you decide which is best for your program.

- List

  The list class provides an ordered, nonindexed list of elements, implemented as a doubly linked list. Lists have a "head" and a "tail," and adding or removing elements from the head or tail or inserting or deleting elements in the middle is very fast.

- Array

  The array class provides a dynamically sized, ordered, and integer-indexed array of objects.

- Map (also known as a dictionary)

  A map is a collection that associates a key object with a value object.

Table 9.1 Shape Features

Shape	Ordered?	Indexed?	Insert an Element	Check for Specified Element	Duplicate Elements?
List	Yes	No	Fast	Slow	Yes
Array	Yes	By int	Slow	Slow	Yes
Map	No	By key	Fast	Fast	No (keys) Yes (values)

9.1 How to Make a Type-Safe Collection

The Microsoft Foundation Class Library provides predefined type-safe collections that can be used to contain **CObject, UINT, DWORD** and **CString** elements. You can use these predefined collections (such as **CObList**) to hold collections of any objects derived from **CObject**. The Microsoft Foundation Class Library also provides other predefined collections to hold primitive types such as **UINT** and void pointers (**void***). In general, however, it is often useful to define your own type-safe collections to hold objects of a more specific class and its derivatives.

There are three main ways to use collections with the Microsoft Foundation Class Library, as described by the following sections.

► To use predefined collections:

- The easiest way to use the Microsoft Foundation collections is to use a predefined collection type, such as **CWordArray**. You can create a **CWordArray** and add **UINT** values to it and retrieve them. There is nothing more to do. You just use the predefined functionality.

  You can also use a predefined collection, such as **CObList**, to hold objects that are derived from **CObject**. A **CObList** is defined to hold pointers to **CObject**. You can put any object that is derived from **CObject** into a **CObList**. When you retrieve an object from the list, you may have to cast the result to the proper type, since the **CObList** functions return pointers to **CObject**. For example, if you store `CPerson` objects in a **CObList**, you have to cast a retrieved element to be a pointer to a `CPerson` object. The following example uses a **CObList** to hold `CPerson` objects:

```
class CPerson : public CObject {...};

CPerson* p1 = new CPerson(...);
CObList myList;

myList.AddHead( p1 );   // no cast needed
CPerson* p2 = ( CPerson* )myList.GetHead();
```

This technique of using a predefined collection type and casting as necessary may be adequate for many of your collection needs. If you need further functionality or more type safety, read the next section.

▶ To derive and extend a collection:

- You can also derive your own collection class from one of the predefined collection classes provided with the Microsoft Foundation Class Library. When you derive your class, you can add type-safe wrapper functions to provide a type-safe interface to existing functions.

  For example, if you derived a list from **CObList** to hold `CPerson` objects, you might add the wrapper functions `AddHeadPerson` and `GetHeadPerson`, as shown below.

```
class CPersonList : public CObList
{
public:
    void AddHeadPerson( CPerson* person )
        {AddHead( person );}

    CPerson* GetHeadPerson()
        {return (CPerson*)GetHead();}
};
```

  These wrapper functions provide a type-safe way to add and retrieve `CPerson` objects from the derived list. You can see that for the `GetHeadPerson` function, you are simply encapsulating the casting seen in the previous section.

  You can also add new functionality by defining new functions that extend the capabilities of the collection rather than just wrapping existing functionality in type-safe wrappers. For example, a later section describes a function to delete all the objects contained in a list. This function could be added to the derived class as a member function.

▶ To use templates to create new collection classes:

- The DOC directory (in your distribution disks) contains a technical note (TN004.TXT) that describes the Microsoft Foundation Class Library tool that you can use to create new type-safe collections from template files. These templates and tool allow you to create a version of an existing collection shape that is customized to hold a specified data type or object type.

  The SAMPLE\TEMPLDEF directory (in your distribution disks) contains a sample program that expands templates defined using a subset of the proposed ANSI template syntax. The Foundation collection classes were generated with this program.

9.2 Accessing All Members of a Collection

The Foundation collection classes use a position indicator to describe a given position within the collection. To access one or more members of a collection, first initialize the position indicator and then repeatedly pass that position to the collection and ask it to return the next element. The collection is not responsible for maintaining state information about the progress of the iteration. That information is kept in the position indicator. But, given a particular position, the collection is responsible for returning the next element.

The following examples show how to iterate over the three main types of collections provided with the Microsoft Foundation Class Library.

► **To iterate an array:**

- Use sequential index numbers with the **GetAt** member function:

```
CObArray myArray;

for( int i = 0; i < myArray.GetSize();i++ )
{
    CPerson* thePerson = (CPerson*)myArray.GetAt( i );
    ...
}
```

► **To iterate a list:**

- Use the member functions **GetHeadPosition** and **GetNext** to work your way through the list:

```
CPersonList myList;

POSITION pos = myList.GetHeadPosition();
while( pos != NULL )
{
    CPerson* thePerson = myList.GetNext( pos );
    ...
}
```

► **To iterate a map:**

- Use **GetStartPosition** to get to the beginning of the map and **GetNextAssociation** to repeatedly get the next key and value from the map, as shown by the following example:

```
CMapStringToOb myMap;

POSITION pos = myMap.GetStartPosition();
while( pos != NULL )
```

```
{
    CObject* pObject;
    CPerson* pPerson;
    CString string;
    // gets key ( string ) and value ( pObject )
    myMap.GetNextAssoc( pos, string, pObject );
    if( pObject->IsKindOf( RUNTIME_CLASS(CPerson) ) )
    {
        pPerson = (CPerson*)pObject;
        //...
    }
}
```

How to Delete All Objects in a CObject Collection

To delete all the objects in a collection of **CObjects** (or of objects derived from **CObject**), you use one of the iteration techniques described above to delete each object in turn.

Note Note that objects in collections can be shared. That is, the collection keeps a pointer to the object, but other parts of the program may also have pointers to the same object. You must be careful not to delete an object that is shared until all the parts have finished using the object.

▶ To delete all objects in a CObList:

You can use the following technique to delete all objects in a **CObList** or a list derived from **CObList**.

1. Use **GetHeadPosition** and **GetNext** to iterate through the list.
2. Use the **delete** operator to delete each object as it is encountered in the iteration.
3. Call the **RemoveAll** function to remove all elements from the list after the objects associated with those elements have been deleted.

   The following example shows how to delete all objects from a list of `CPerson` objects. Each object in the list is a pointer to a `CPerson` object that was originally allocated on the heap.

```
class CPersonList : public CObList {...};

CPersonList myList
POSITION pos = myList.GetHeadPosition();

while( pos != NULL )
{
    delete myList.GetNext( pos );
}
myList.RemoveAll();
```

Remove an element after its object is deleted.

The last function call, **RemoveAll**, is a list member function that removes all elements from the list. The member function **RemoveAt** will remove a single element.

Notice the difference between deleting an element's object and removing the element itself. Removing an element from the list merely removes the list's reference to the object. The object still exists in memory. When you delete an object, its memory is reclaimed and it ceases to exist. Thus, it is important to remove an element immediately after the element's object has been deleted so that the list won't try to access objects that no longer exist.

▶ To delete all elements in an array:

1. Use **GetSize** and integer index values to iterate through the array.
2. Use the **delete** operator to delete each element as it is encountered in the iteration.
3. Call the **RemoveAll** function to remove all elements from the array after they have been deleted.

   The code for deleting all elements of an array is as follows:

```
CObArray myArray;

int i = 0;
while (i < myArray.GetSize() )
{
    delete myArray.GetAt( i++ );
}

myArray.RemoveAll();
```

   Like the list example, you can call **RemoveAll** to remove all elements in an array or **RemoveAt** to remove an individual element.

▶ To delete all elements in a map:

1. Use **GetStartPosition** and **GetNextAssociation** to iterate through the array.
2. Use the **delete** operator to delete the key and/or value for each map element as it is encountered in the iteration.
3. Call the **RemoveAll** function to remove all elements from the map after they have been deleted.

   The code for deleting all elements of a **CMapStringToOb** is as follows. Each element in the map has a string as the key and a `CPerson` object (derived from **CObject**) as the value.

```
CMapStringToOb myMap;
// ... add some key-value elements...
// now delete the elements
pos = myMap.GetStartPosition();
```

```
while( pos != NULL )
{
    CObject* pObject;
    CString string;
    // gets key ( string ) and value ( pObject )
    myMap.GetNextAssoc( pos, string, pObject );
    delete pObject;
}
myMap.RemoveAll();
```

You can call **RemoveAll** to remove all elements in a map or **RemoveKey** to remove an individual element with the specified key.

How to Create a Stack Collection

Because the standard list collection has both a head and a tail, it is easy to create a derived list collection that mimics the behavior of a last-in-first-out stack. A stack is like a stack of trays in a cafeteria. As trays are added to the stack, they go on top of the stack. The last tray added is the first to be removed. The list collection member functions **AddHead** and **RemoveHead** can be used to add and remove elements specifically from the head of the list; thus the most recently added element is the first to be removed.

► **To create a stack collection:**

- Derive a new list class from one of the existing list classes provided with the Foundation library and add more member functions to support the functionality of stack operations.

  The following example shows you can add member functions to push elements on to the stack, peek at the top element of the stack, and pop the top element from the stack:

```
class CTray : public CObject { ... };

class CStack : public CObList
{
public:
    // add element to top of stack
    void Push( CTray* newTray )
        { AddHead( newTray ); }

    // peek at top element of stack
    CTray* Peek()
        { return IsEmpty() ? NULL : (CTray*)GetHead(); }

    // pop top element off stack
    CTray* Pop()
        { return (CTray*)RemoveHead(); }
};
```

How to Create a Queue Collection

Because the standard list collection has both a head and a tail, it is also easy to create a derived list collection that mimics the behavior of a first-in-first-out queue. A queue is like a line of people in a cafeteria. The first person in line is the first to be served. As more people come, they go to the end of the line to wait their turn. The list collection member functions **AddTail** and **RemoveHead** can be used to add and remove elements specifically from the head or tail of the list; thus the most recently added element is always the last to be removed.

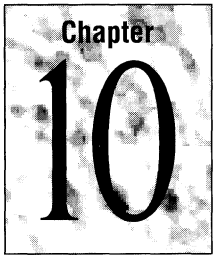
► To create a queue collection:

- Derive a new list class from one of the predefined list classes provided with the Microsoft Foundation Class Library and add more member functions to support the semantics of queue operations.

  The following example shows how you can append member functions to add an element to the end of the queue and get the element from the front of the queue.

```
class CPerson : public CObject { ... };

class CQueue : public CObList
{
public:
    // go to the end of the line
    void AddToEnd( CPerson* newPerson )
        { AddTail( newPerson ); }    // end of the queue

    // get first element in line
    CPerson* GetFromFront()
        { return IsEmpty() ? NULL : (CPerson*)RemoveHead(); }
};
```

Files and Serialization



The Microsoft Foundation Class Library provides the **CFile** class to handle normal file I/O operations. This chapter shows you how to open and close files and read and write arbitrary data to those files. You will also learn about file status operations. You may also be interested in reading section 10.2, "Serialization", of this chapter for a description of how to use the object-based serialization features of the Microsoft Foundation Class Library as an alternative way of reading and writing data in files.

10.1 Files

The Microsoft Foundation Class Library provides the **CFile** class for general-purpose binary file operations. The **CStdioFile** and **CMemFile** classes are derived from **CFile** to provide more specialized file services.

In the Microsoft Foundation Class Library, the most common way to open a file is to go through a two-stage process.

▶ **To open a file:**

1. Create the file object without specifying a path or permission flags.

   You usually create a file object by declaring a **CFile** variable on the frame.

2. Call the **Open** member function for the file object, supplying a path and permission flags.

   Open will return **TRUE** if the file was opened successfully, or **FALSE** if the specified file could not be opened.

   The open flags specify which permissions, such as read-only, you want for the file. The possible flag values are defined as enumerated constants within the **CFile** class, so they are qualified with "**CFile::**", as in **CFile::modeRead**. Use the **CFile::modeCreate** flag if you want to create the file.

The following example shows how to create a new file with read/write permission (replacing any previous file with the same path):

```
char* pszFileName = "\\test\\myfile.dat";
CFile myFile;

if ( ! myFile.Open( pszFileName,
                CFile::modeCreate | CFile::modeReadWrite ) )
{
    TRACE( "Can't open file %s\n",pszFileName );
}
```

▶ To read from and write to the file:

- Use the **Read** and **Write** member functions to read and write data in the file.

  The **Seek** member function is also available for moving to a specific offset within the file.

  Read takes a pointer to a buffer and a **UINT** specifying the number of bytes to read or write and returns a **UINT** with the actual number of bytes that were processed. **Write** takes the same arguments but does not return the number of bytes written. If the requested number of bytes can't be read or written, an exception is thrown. If you have a valid **CFile** object, you can read or write to it with code similar to the following:

```
char     buffer[256];
UINT    actual = 0;

myFile.Write( buffer, sizeof( buffer ) );
myFile.Seek( 0, CFile::begin );
actual = myFile.Read( buffer, sizeof( buffer ) );
```

▶ To close a file:

- Use the **Close** member function. This function closes the file system file and flushes buffers if neccessary.

  If you allocated the **CFile** object on the frame (as in the examples above) it will be automatically destroyed when it goes out of scope. It is important to close the file before the object is deleted. Note that deleting the **CFile** object does not delete the physical file in the file system.

▶ To get file status:

- Use the **CFile** object to get and set information about a file. One useful application is to use the **CFile** static member function **GetStatus** to determine if a file exists. **GetStatus** will return **FALSE** if the specified file does not exist.

  Thus, you could use the result of **GetStatus** to determine whether to use the **CFile::modeCreate** flag when opening a file, as shown by the following example.

```
CFile theFile;
char* szFileName = "c:\\test\\myfile.dat";
BOOL fOpenOK;

CFileStatus status;
// GetStatus will return TRUE if file exists,
// or FALSE if it doesn't exist
if( CFile::GetStatus( szFileName, status ) )
{
    // Open the file without the Create flag
    fOpenOK = theFile.Open( szFileName,
                    CFile::modeWrite );
}
else
{
   // Open the file with the Create flag
   fOpenOK = theFile.Open( szFileName,
                   CFile::modeCreate | CFile::modeWrite );
}
```

10.2 Serialization

Serialization is the process of writing or reading an object to or from a persistent storage medium, such as a disk file. The Microsoft Foundation Class Library provides built-in support for serialization in the class **CObject**. Thus, all classes that are derived from **CObject** can take advantage of **CObject**'s serialization protocol.

The basic idea of serialization is that an object should be able to write its current state, usually indicated by the value of its member variables, to persistent storage. Later, the object can be re-created by reading, or deserializing, the object's state from the storage. A key point here is that the object itself is responsible for reading and writing its own state. Thus, for a class to be serializable, it must implement the basic serialization operations. As you will see in the following sections, this functionality is easy to add to a class.

The Microsoft Foundation Class Library uses the **CArchive** class to perform serialization. The object performs serialization operations on the **CArchive** object without regard to the exact nature of the storage medium. A **CArchive** object is typically associated with a **CFile** object, which is normally a disk file.

A **CArchive** object resembles an I/O stream in that it uses overloaded insertion (<<) and extraction (>>) operators to perform writing and reading operations. But the resemblance to I/O streams is merely cosmetic. Do not confuse the **CArchive** class with general-purpose I/O streams. I/O streams are for formatted text and the **CArchive** class is for binary format serialized objects.

There are two main topics regarding serialization in the Microsoft Foundation Class Library that are covered in the following sections.

1. How to make a serializable class
2. How to serialize an object to and from a file object

How to Make a Serializable Class

There are five main requirements to make a class serializable. They are listed below and explained in the following sections.

1. Derive your class from **CObject** (or from some class derived from **CObject**.)
2. Use the **DECLARE_SERIAL** macro in the class declaration.
3. Define a constructor that takes no arguments.
4. Use the **IMPLEMENT_SERIAL** macro in the implementation file for your class.
5. Override the **Serialize** member function.

Deriving Your Class from CObject and Using the DECLARE_SERIAL Macro

The basic serialization protocol and functionality is defined in the **CObject** class. By deriving your class from **CObject**, (or from some class derived from **CObject**) as shown in the example code below, you gain access to the serialization protocol and functionality of **CObject**.

The **DECLARE_SERIAL** macro is required in the declaration of classes that will support serialization, as shown here:

```
class CPerson : public CObject
{
    DECLARE_SERIAL( CPerson )
    // rest of declaration follows...
};
```

Defining a Constructor with No Arguments

This constructor is used by the Microsoft Foundation Class Library when it re-creates your objects as they are deserialized (loaded from disk.) Typically, this constructor is an empty function, since the deserialization process will fill in all member variables with the values necessary to re-create the object.

This constructor can be declared public, protected, or private. If you make it protected or private, you can be sure that it will only be used by the serialization functions. The constructor must put the object in a state so that it can be safely deleted if necessary.

Note If you forget to define a constructor with no arguments in a class that uses the **DECLARE_SERIAL** and **IMPLEMENT_SERIAL** macros, you will get a "no default constructor available" compiler warning on the line where the **IMPLEMENT_SERIAL** macro is used.

Using the IMPLEMENT_SERIAL Macro in the Implementation File

The **IMPLEMENT_SERIAL** macro is used to define various functions needed when you derive a serializable class from **CObject**. You use this macro in the implementation file (.CPP) for your class. The first two arguments to the macro are the name of the class and the name of its immediate base class.

The third argument to this macro is a schema number. The schema number is essentially a version number for objects of the class. Use an integer greater than or equal to 0 for the schema number.

The Microsoft Foundation Class Library serialization code checks the schema number when reading objects into memory. If the schema number of the object on disk does not match the schema number of the class in memory, then the Foundation will throw an exception, preventing your program from reading an incorrect version of the object.

The following example shows how to use **IMPLEMENT_SERIAL** for a class, `CPerson`, that is derived from **CObject**:

```
IMPLEMENT_SERIAL( CPerson, CObject, 0 )
```

Overriding the Serialize Member Function

The **Serialize** member function, which is defined in the **CObject** class, is responsible for actually serializing the data necessary to capture an object's current state. The **Serialize** function has a **CArchive** argument that it uses to read and write the object data. The **CArchive** object has a member function, **IsStoring**, which indicates whether the serialization is storing (writing data) or loading (reading data). Using this function as a guide, you either insert your object data in the **CArchive** with the insertion operator (<<) or extract data with the extraction operator (>>).

Consider a class that is derived from **CObject** and has two new member variables, a **CString** and a **UINT**. The following class declaration fragment shows the new member variables and the declaration for the overridden **Serialize** member function:

```
class CPerson : public CObject
{
public:
    DECLARE_SERIAL( CPerson, CObject )
    // empty constructor is necessary
    CPerson(){};

    CString m_name;
    UINT    m_number;

    void Serialize( CArchive& archive );

    // rest of class declaration
};
```

► **To override the Serialize member function:**

- First call your base class version of **Serialize** to make sure that the inherited portion of the object is serialized.
- Then insert or extract the member variables that are specific to your class.

  The insertion and extraction operators do all the hard work of interacting with the archive class to read and write the data. The following example shows how to implement `Serialize` for the `CPerson` class declared above:

```
void CPerson::Serialize( CArchive& archive )
{
    // call base class function first
    // base class is CObject in this case
    CObject::Serialize( archive );

    // now do the stuff for our specific class
    if( archive.IsStoring() )
        archive << m_name << m_number;
    else
        archive >> m_name >> m_number;
}
```

How to Serialize an Object

The previous sections showed how to make a class serializable. Once you have a serializable class, you can serialize objects of that class to and from a **CArchive** object. This section shows how to get a **CArchive** object and how to serialize objects to and from the archive.

► To create a CArchive object:

- **CArchive** objects are always associated with files. If you have a **CFile** object or a derived **CFile** object, you can get a **CArchive** object for that file by passing the **CFile** object to the constructor for **CArchive**, as shown in the following example:

```
CFile theFile;
theFile.Open( ..., CFile::modeWrite );

CArchive archive( &theFile, CArchive::store );
```

The second argument to the **CArchive** constructor is an enumerated value that specifies whether the archive will be used for loading (reading) or storing (writing) data. An archive can be for either storing or loading, but not for both. This constructor argument sets the load-store state for the archive. The **Serialize** function of an object checks this state by calling the **IsStoring** function for the archive object.

► To serialize an object:

- Once you have a **CArchive** object, as shown in the previous section, you can serialize objects to and from it (depending on whether it was created for storing or loading) by using the same insertion and extraction operators that you used inside the `Serialize` member function described in an earlier procedure, "Overriding the **Serialize** Member Function," on pages 281-282.

  Assuming that `CPerson` is a class that has overridden the **Serialize** member function, you can serialize a `CPerson` object to an archive as follows:

```
CPerson* aPerson = new CPerson( "Smith" );
archive << aPerson;
```

Close the archive before you close the file.

To flush the buffers and close the connection between the archive and the file when you are done with the serialization operation, call the **Close** member function for the archive. Be sure to close the archive before you close the file, as shown in the following example:

```
archive.Close();

theFile.Close();
```

► To deserialize an object:

- Deserializing an object involves reading the object back in from the disk file to which it was originally serialized. The following example shows how you can deserialize the object:

```
CPerson* aPerson2;
CFile theFile;
theFile.Open( ..., CFile::modeRead );

CArchive archive2( &theFile, CArchive::load );

archive2 >> aPerson2;

archive2.Close();

theFile.Close();
```

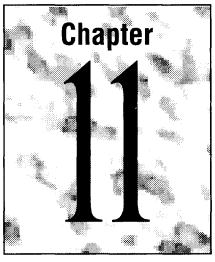
Deserialization will only work if the file position is in the same state as it was when the object was originally serialized. That means your program must keep track of the order in which objects were serialized and deserialize them in the same order.

The suggested way to avoid this bookkeeping problem is to place all your objects in a collection and then serialize the collection as a single object. Your data file can then be thought of as containing a single collection object that can be deserialized. When the collection object is deserialized, it will take care of deserializing all its constituent objects in the proper order.

Note Note that you do not have to do any explicit memory allocation for the objects that you are deserializing. The archive allocates the memory necessary to reconstruct the objects. You are responsible, however, for deleting the deserialized objects when you are done with them.

See the tutorial in this book (Chapters 1-6) for a complete example program that uses serialization to save and restore a name and phone-number data base as a single collection.

Diagnostics



The Microsoft Foundation Class Library contains many diagnostic features to help debug your program during development. These features, especially keeping track of all memory allocations, will slow your program down. Others, such as assertion testing, will cause your program to halt when erroneous conditions are encountered.

In a commercial retail product, slow performance and program interruption are clearly unacceptable. For this reason, the Microsoft Foundation Class Library provides a way to turn the debugging and diagnostic features on or off when building your program. You typically build a debug version of your program and link with the Debug version of the Microsoft Foundation Class Library while developing your program. When it comes time to release your program, you build a release version and link with the Release Foundation libraries.

11.1 Debugging Features

The following list contains the features that are included in the Debug version of the Microsoft Foundation Class Library:

- **Dump** member function to dump object contents to debugging output
- Trace output
- Assertions and **AssertValid** member function
- Memory diagnostics to detect memory leaks
- **DEBUG_NEW** macro to show where objects were allocated

► **To enable the debugging features:**

1. Compile with the symbol **_DEBUG** defined. This is typically done by passing the **/D_DEBUG** flag on the compiler command line. Defining the **_DEBUG** symbol allows sections of code delimited by **#ifdef _DEBUG / #endif** to be compiled.

2. Link with the Debug versions of the Microsoft Foundation Class Library. The Debug versions of the library have a "D" at the end of the library name. For example, the medium-model Debug version of the Microsoft Foundation Class Library for Windows is named MAFXCWD.LIB, and the Release version (non-debug) is named MAFXCW.LIB.

Dumping Object Contents

When deriving a class from **CObject**, you can optionally override the **Dump** member function to write a textual representation of the member variables of the object to a dump context, which is similar to an I/O stream. Like an I/O stream, you can use the insertion (<<) operator to send data to a **CDumpContext**.

Overriding **Dump** is not required when deriving a class from **CObject**, but it is very helpful and highly recommended when using the other diagnostic features for debugging to be able to dump an object and view its contents.

► **To override the Dump member function:**

1. Call the base class version of **Dump** to dump the contents of a base class object.
2. Write a textual description and value for each member variable for your derived class.

The declaration of the `Dump` function in the class declaration looks like the following example:

```
class CPerson : public CObject
{
public:
#ifdef _DEBUG
    virtual void Dump( CDumpContext& dc ) const;
#endif

    CString m_firstName;
    CString m_lastName;
    // etc. ...
};
```

Note Since object dumping only makes sense when debugging your programming, the declaration of the `Dump` function is bracketed with an **#ifdef _DEBUG / #endif** block.

The `Dump` function's first statement should call the **Dump** function for its base class. It should then write a short description of each member variable along with the member's value to the diagnostic stream, as shown by the following example from an implementation file for the class `CPerson`.

```
#ifdef _DEBUG
void CPerson::Dump( CDumpContext& dc ) const
{
    // call base class function first
    CObject::Dump( dc );

    // now do the stuff for our specific class
    dc   << "last name: " << m_lastName << "\n"
         << "first name: " << m_firstName ;
}
#endif
```

Note Again, notice that the definition of the `Dump` function is bracketed by **#ifdef _DEBUG / #endif** directives.

► **To send Dump output to afxDump:**

- You must supply a **CDumpContext** argument to specify where the dump output will go when you call the `Dump` function for an object. The Microsoft Foundation Class Library supplies a predefined **CDumpContext** object named **afxDump** that you will normally use for routine object dumping. The following example shows how to use **afxDump**:

```
CPerson myPerson = new CPerson;
// set some fields of the CPerson object...
//..
// now dump the contents
#ifdef _DEBUG
myPerson->Dump( afxDump );
#endif
```

  In Windows, **afxDump** output is sent to the debugger, if present. In DOS, **afxDump** output is sent to **stderr**.

  Note **afxDump** is defined only in the Debug version of the Microsoft Foundation Class Library.

The TRACE Macro

The **TRACE** macro can be used to print out debugging messages from a program during development. **TRACE** prints a string argument to the current diagnostic output device. For programs with character-based output in DOS, the **TRACE** output will go to **stderr**. For Windows programs, the **TRACE** output will be directed to your debugger.

The **TRACE** macro can handle different numbers of arguments, similar to the way **printf** operates. The following examples show several different ways to use the **TRACE** macros:

```
int x = 1;
int y = 16;
float z = 32.0;
TRACE( "This is a TRACE statement\n" );

TRACE( "The value of x is %d\n", x );

TRACE( "x = %d and y = %d\n", x, y );

TRACE( "x = %d and y = %x and z = %f\n", x, y, z );
```

The **TRACE** macro is active only in the Debug version of the library. After a program has been debugged, you can build a Release version to inactivate all **TRACE** calls in the program.

The ASSERT Macro

The **ASSERT** macro evaluates its argument, prints a diagnostic message, and halts program execution if the argument expression is false (0). The diagnostic message is sent to **afxDump** and has the form

```
assertion failed in file <name> in line <num>
```

where <name> is the name of the source file and <num> is the line number of the assertion that failed in the source file. The **ASSERT** macro takes no action if its argument is true (nonzero).

The **ASSERT** macro is typically used to identify program errors during development. The argument given to **ASSERT** should be chosen so that it holds true only if the program is operating as intended. The following example shows how the **ASSERT** macro could be used to check the validity of a function's return value:

```
int x = SomeFunc(y);
ASSERT(x == 0);   // ASSERT only if x not equal to 0
```

ASSERT can also be used in combination with the **IsKindOf** macro to provide extra checking for function arguments, such as in the following example. (For a discussion of the **IsKindOf** macro, see "How to Access Run-time Class Information" on page 265).

```
ASSERT( object1->IsKindOf( RUNTIME_CLASS( CPerson ) ) );
```

Like the **TRACE** macro described in the previous section, the **ASSERT** macros are only active in the Debug version of your program. Thus, you can inactivate all **ASSERT** statements simply by building the Release version of your program.

Using assertions liberally throughout your programs can catch errors during development. A good rule of thumb is that you should write assertions for any assumptions you make. For example, if you assume that an argument is not **NULL**, then you should use an assertion statement to check for that condition. The good thing about the **ASSERT** macro is that it will catch errors when you are using the Debug version of the Microsoft Foundation Class Library during development but will be turned off (produce no code) when you build your program with the Release version of the library.

Note The expression argument to **ASSERT** will not be executed in the release version of your program. If you want the expression to be executed in both debug and release environments, use the **VERIFY** macro instead of **ASSERT**. In debug versions, **VERIFY** simply passes its expression argument to **ASSERT**. In release environments, **VERIFY** executes the expression argument but does not check the result.

Overriding the AssertValid Function

The **AssertValid** member function is provided in **CObject** to allow run-time checks of an object's internal state. Although it is not required that you override **AssertValid** when you derive your class from **CObject**, you can make using your class safer and more reliable by overriding **AssertValid**.

Typically, **AssertValid** performs assertions on all the object's member variables to see if they contain valid values. For example, **AssertValid** can check that all pointer member variables are not **NULL**. **AssertValid** asserts and halts the program if it finds that the object is invalid. Because it uses the **ASSERT** macro, **AssertValid** will have no effect when used in the Release version of the library.

The declaration of the `AssertValid` function in the class declaration looks like this:

```
class CPerson : public CObject
{
protected:
    CString m_Name;
    float   m_Salary;
public:
    virtual void AssertValid() const;

    // etc. ...
};
```

In your overriding `AssertValid`, first call **AssertValid** for the base class. Then, assert the validity of the members that are unique to your derived class, as shown by the following example:

```
void CPerson::AssertValid()
{
    // call inherited AssertValid first
    CObject::AssertValid()

    // check CPerson members...
    ASSERT( m_Name != NULL )
    ASSERT( m_Salary != 0 )

}
```

Users of an `AssertValid` function of a given class should not rely too heavily on the accuracy of this function. A triggered assertion indicates that the object is definitely bad and execution will be halted. A lack of assertion indicates that no problem was found, but the object isn't guaranteed to be good.

11.2 Detecting Memory Leaks

A memory leak occurs when you allocate memory on the heap and never delete that memory to make it available for reuse. This is a particular problem for programs that are intended to run for extended periods (typically weeks or months). In a long-lived program, even a small incremental memory leak can compound itself, so that eventually all available memory resources are exhausted and the program crashes. Traditionally, memory leaks have been very hard to detect.

The Microsoft Foundation Class Library provides classes and functions that you can use to detect memory leaks during development. Basically, these functions take a snapshot of all memory blocks before and after a particular set of operations. You can use these results to determine if all memory blocks allocated during the operation have been deallocated.

The size of the operation that you choose to bracket with these diagnostic functions is arbitrary. It can be as small as a single program statement, or it can span the entry and exit from your entire program. Either way, these functions will allow you to detect memory leaks and identify the memory blocks that have not been deallocated.

Memory Diagnostics

► **To enable or disable memory diagnostics:**

- Call the **AfxEnableMemoryTracking** to enable or disable the diagnostic memory allocator. Since memory diagnostics are on by default in the Debug library, you will typically use this function to temporarily turn off memory diagnostics to speed program execution and reduce diagnostic output.

► **To select specific memory diagnostic features with afxMemDF:**

- If you want more precise control over the memory diagnostic features, you can selectively turn inidividual memory diagnostic features on and off by setting the value of the Microsoft Foundation Class Library global variable **afxMemDF**. This variable can have the following values as specified by the enumerated type **AfxMemDF**:

Value	Meaning
allocMemDF	Turn on debugging allocator (default).
delayFreeMemDF	Delay freeing memory when calling **delete** or **free**. This will cause maximum memory stress for your program.
checkAlwaysMemDF	Call **AfxCheckMemory** every time memory is allocated or freed.

These possible values can be used in combination by performing a logical-OR operation, as shown here:

```
afxMemDF |= delayFreeMemDF | checkAlwaysMemDF;
```

Detecting a Memory Leak

The following instructions and examples will show you how to detect a memory leak.

► **To detect a memory leak:**

1. Create a **CMemoryState** object and call the **Checkpoint** member function to get the initial snapshot of memory.
2. After you perform the memory allocation and deallocation operations, create another **CMemoryState** object and call **Checkpoint** for that object to get a current snapshot of memory usage.
3. Create a third **CMemoryState** object and call the **Difference** member function, and supply the previous two **CMemoryState** objects as arguments. The **Difference** function will return **TRUE** if there is any difference between the two specified memory states, indicating that some memory blocks that have not been deallocated.

   The following example shows how to check for memory leaks:

```
// Declare the variables needed
#ifdef _DEBUG
    CMemoryState oldMemState, newMemState, diffMemState;
#endif

#ifdef _DEBUG
    oldMemState.Checkpoint();
#endif

    // do your memory allocations and deallocations...
    CString s = "This is a frame variable";
    // the next object is a heap object
    CPerson* p = new CPerson( "Smith", "Alan", "581-0215" );

#ifdef _DEBUG
    newMemState.Checkpoint();
    if( diffMemState.Difference( oldMemState, newMemState ) )
    {
        TRACE( "Memory leaked !\n" );
    }
#endif
```

   Notice that the memory-checking statements are bracketed by **#ifdef _DEBUG / #endif** blocks so that they are only compiled in debug versions of your program.

Dumping Memory Statistics

The **CMemoryState** member function **Difference** will determine the difference between two memory-state objects. It will detect any objects that were not deallocated from the heap between the beginning and end memory-state snapshots.

▶ To dump memory statistics:

- The following example, continuing the example from the previous section, shows how to call **DumpStatistics** to get information about the objects that have not been deallocated.

```
if( diffMemState.Difference( oldMemState, newMemState ) )
{
    TRACE( "Memory leaked !\n" );
    diffMemState.DumpStatistics();
}
```

A sample dump from the example above is shown here:

```
0 bytes in 0 Free Blocks
22 bytes in 1 Object Blocks
45 bytes in 4 Non-Object Blocks
Largest number used: 67 bytes
Total allocations: 67 bytes
```

- The first line describes the number of blocks whose deallocation was delayed if **afxMemDF** was set to **delayFreeMemDF**. For a description of **afxMemDF**, see the section "To select specific memory diagnostic features with afxMemDF" on page 291.
- The second line describes how many objects still remain allocated on the heap.
- The third line describes how many nonobject blocks (arrays or structures allocated with **new**) were allocated on the heap and not deallocated.
- The fourth line gives the maximum memory used by your program at any one time.
- The last line lists the total amount of memory used by your program.

Dumping All Objects

DumpAllObjectsSince dumps out a description of all objects detected on the heap that have not been deallocated. As the name implies, **DumpAllObjectsSince** will dump all objects allocated since the last **Checkpoint**. However, if no **Checkpoint** has taken place, all objects and non-objects currently in memory will be dumped.

► **To dump all objects:**

- Expanding on the code from the previous example, the following code dumps all objects that have not been deallocated when a memory leak is detected:

```
if( diffMemState.Difference( oldMemState, newMemState ) )
{
    TRACE( "Memory leaked !\n" );
    diffMemState.DumpAllObjectsSince();
}
```

A sample dump from the previous code is shown as follows:

```
Dumping objects ->

{5} string.cpp(62) : non-object block at $00A7521A, 9 bytes long
{4} string.cpp(62) : non-object block at $00A751F8, 5 bytes long
{3} string.cpp(62) : non-object block at $00A751D6, 6 bytes long
{2} a CPerson at $51A4

Last Name: Smith
First Name: Alan
Phone #: 581-0215

{1} string.cpp(62) : non-object block at $00A7516E, 25 bytes long
```

The numbers in braces at the beginning of most lines specify the order in which the objects were allocated. The most recently allocated object is displayed first. You can use these ordering numbers to help identify allocated objects.

Interpreting an Object Dump

The preceding dump comes from the original memory checkpoint example in the section "Detecting a Memory Leak" on page 292. Remember that there were only two explicit allocations in that program—one on the frame and one on the heap:

```
// do your memory allocations and deallocations ...
CString s = "This is a frame variable";
// the next object is a heap object
CPerson* p = new CPerson( "Smith", "Alan", "581-0215" );
```

Start with the `CPerson` object; its constructor takes three arguments that are pointers to **char**. The constructor uses those arguments to initialize **CString** member variables for the `CPerson` class. In the memory dump, you can see the `CPerson` object listed along with three nonobject blocks (3, 4, and 5) that hold the characters for the **CString** member variables. These memory blocks will be deleted when the destructor for the `CPerson` object is invoked.

Block number 2 represents the `CPerson` object itself. After the `CPerson` address listing, the contents of the object are displayed. This is a result of **DumpAllObjectsSince** calling the `Dump` member function for the `CPerson` object.

You can guess that block number 1 is associated with the **CString** frame variable because of its sequence number and its size, which matches the number of characters in the frame **CString** variable. The allocations associated with frame variables are automatically deallocated when the frame variable goes out of scope.

In general, you shouldn't worry about heap objects associated with frame variables because they are automatically deallocated when the frame variables go out of scope. In fact, you should position your calls to **Checkpoint** so that they are outside the scope of frame variables to avoid clutter in your memory diagnostic dumps. For example, place scope brackets around the previous allocation code, as shown here:

```
oldMemState.Checkpoint();
{
    // do your memory allocations and deallocations ...
    CString s = "This is a frame variable";
    // the next object is a heap object
    CPerson* p = new CPerson( "Smith", "Alan", "581-0215" );
}
newMemState.Checkpoint();
```

With the scope brackets in place, the memory dump for this example is as follows:

```
Dumping objects ->

{5} string.cpp(62) : non-object block at $00A7521A, 9 bytes long
{4} string.cpp(62) : non-object block at $00A751F8, 5 bytes long
{3} string.cpp(62) : non-object block at $00A751D6, 6 bytes long
{2} a CPerson at $51A4

Last Name: Smith
First Name: Alan
Phone #: 581-0215
```

Notice that the memory block associated with the **CString** frame variable has been deallocated automatically and does not show up as a memory leak. The automatic deallocation associated with scoping rules takes care of most memory leaks associated with frame variables.

For objects allocated on the heap, however, you must explicitly delete the object to prevent a memory leak. To clean up the last memory leak in the previous example, you can delete the CPerson object allocated on the heap, as follows:

```
{
    // do your memory allocations and deallocations ...
    CString s = "This is a frame variable";
    // the next object is a heap object
    CPerson* p = new CPerson( "Smith", "Alan", "581-0215" );
    delete p;
}
```

11.3 Using DEBUG_NEW to Aid Debugging

The Microsoft Foundation Class Library defines the macro **DEBUG_NEW** to assist you in tracking down memory leaks. You can use **DEBUG_NEW** everywhere in your program that you would ordinarily use the **new** operator.

When you compile a Debug version of your program, **DEBUG_NEW** will keep track of the filename and line number for each object that it allocates. Then when you call **DumpAllObjectsSince**, as described in the previous section, each object allocated with **DEBUG_NEW** will be shown with the file and line number where it was allocated, thus allowing you to pinpoint the sources of memory leaks.

When you compile a Release version of your program, **DEBUG_NEW** will resolve to a simple **new** operation without the filename and line number information. Thus, you pay no speed penalty in the Release version of your program.

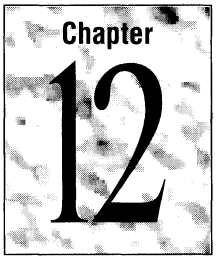
► **To use DEBUG_NEW:**

- Define a macro in your source files that replaces **new** with **DEBUG_NEW**, as shown here:

  ```
  #define new DEBUG_NEW
  ```

  You can then use **new** for all heap allocations. The preprocessor will substitute **DEBUG_NEW** when compiling your code. In the Debug version of the library, **DEBUG_NEW** will create debugging information for each heap block. In the Release version, **DEBUG_NEW** will resolve to a standard memory allocation without the extra debugging information.

Exceptions



There are three categories of outcomes that can occur when a function is called during program execution: normal execution, erroneous execution, or abnormal execution. Each category is described below.

Normal execution
The function may execute normally and return. Some functions return a result code to the caller, which indicates the outcome of the function. The possible result codes are strictly defined for the function and represent the range of possible outcomes of the function. The result code can indicate success or failure or can even indicate a particular type of failure that is within the normal range of expectations. For example, a file-status function can return a code that indicates that the file does not exist. Note that the term "error code" is not used since a result code represents one of many expected outcomes.

Erroneous execution
The caller makes some mistake in passing arguments to the function or calls the function in an inappropriate context. This situation causes an error, and it should be detected by an assertion during program development. (For more information on assertions, see page 288).

Abnormal execution
Abnormal execution includes situations where conditions outside the program's control are influencing the outcome of the function, such as low memory or I/O errors. Abnormal situations should be handled by catching and throwing exceptions.

12.1 Microsoft Foundation Classes Exception Handling

The Microsoft Foundation Class Library uses an exception-handling scheme that is very similar to one proposed by the ANSI standards committee for C++ 2.1. You set up an exception handler before calling functions that you think might encounter abnormal situations. If your program does run into abnormal conditions, then it throws an exception. When an exception is thrown, program execution jumps to the exception handler and execution resumes there.

Exceptions are represented as objects derived from the abstract class **CException**. The Microsoft Foundation Class Library provides several predefined kinds of exceptions as listed below.

Exception handler	Meaning
CMemoryException	Out-of-memory
CFileException	File exception
CArchiveException	Archive/Serialization exception
CNotSupportedException	Response to request for unsupported service
CResourceException	Windows resource allocation exception

Since many parts of the Microsoft Foundation Class Library, especially those dealing with files and serialization, use exceptions to report abnormal conditions, you will find it useful to use the Microsoft Foundation exception-handling mechanism in the parts of your program that call those types of Microsoft Foundation Class Library functions. For a description of each Microsoft Foundation Class Library function and the exceptions that can possibly be thrown by that function, see the *Class Library Reference*. If you see that a function can throw an exception, you should probably surround it with an exception handler.

For a good working example of how to use exceptions with file and serialization functions, see the "Exception Handling" section on page 61 of the tutorial.

12.2 Catching Exceptions

The following instructions and examples will show you how to catch exceptions.

► **To catch exceptions:**

1. Use the **TRY** macro to set up a **TRY** block. Execute any program statements that might throw an exception within a **TRY** block.
2. Use the **CATCH** macro to set up a **CATCH** block. Place exception handling code in a **CATCH** block. The code in the **CATCH** block is executed only if the code within the **TRY** block throws an exception of the type specified in the **CATCH** statement.

   The following code skeleton shows how **TRY** and **CATCH** blocks are normally arranged:

```
// normal program statements
...

TRY
{
    // execute some code that might throw an exception
}
```

```
CATCH( CException, e )
{
    // handle the exception here
    // "e" contains information about the exception
}
END_CATCH

// other normal program statements
...
```

Note Note the **END_CATCH** macro that marks the end of the **CATCH** blocks.

The **CATCH** macro takes an exception type parameter, so you can selectively handle different types of exceptions with sequential **CATCH** and **AND_CATCH** blocks as listed below:

```
TRY
{
    // execute some code that might throw an exception
}
CATCH( CMemoryException, e )
{
    // handle the out-of-memory exception here
}
AND_CATCH( CFileException, e )
{
    // handle the file exceptions here
}
AND_CATCH( CException, e )
{
    // handle all other types of exceptions here
}
END_CATCH
```

12.3 Examining Exception Contents

The **CATCH** macro includes an argument that is used to hold a pointer to a **CException** object (or an object derived from **CException**, such as **CMemoryException**). Depending on the exact type of the exception, you can examine the data members of the exception object to gather information about the specific cause of the exception.

For example, the **CFileException** type has the **m_cause** data member that contains an enumerated type that specifies the cause of the file exception. Some examples of the possible return values are **CFileException::fileNotFound** and **CFileException::readOnly**.

► **To examine exception contents:**

- The following example shows how to examine the contents of a **CFileException**. Other exception types can be examined in a similar way.

```
TRY
{
    // do something to throw a file exception
}
CATCH( CFileException, theException )
{
    if( theException->m_cause == CFileException::fileNotFound )
       TRACE( "File not found\n" );
}
END_CATCH
```

12.4 Freeing Objects in Exceptions

The exception-handling mechanism of the Microsoft Foundation Class Library can interrupt normal program flow. Thus, it is very important to keep close track of objects that have been created on the heap so that you can properly dispose of them in case an exception is thrown.

There are two primary methods to do this.

- Handle exceptions locally using the **TRY** and **CATCH** macros, then destroy all objects with one statement.
- Destroy any object in the **CATCH** block before the exception is thrown outside for further handling.

These two approaches are illustrated below as solutions to the following problematic example code:

```
void SomeFunc()
{
    CPerson* myPerson = new CPerson;

    // do something that might throw an exception
    myPerson->SomeFunc();

    // now destroy the object before exiting
    delete myPerson;
}
```

As written above, `myPerson` will not be deleted if an exception is thrown by `SomeFunc`. Execution jumps directly to the innermost exception handler, bypassing the normal function exit and the code that deletes the object. As written above, the pointer to the object goes out of scope when the exception leaves the function, and

the memory occupied by the object will never be recovered as long as the program is running. This is known as a memory leak and would be detected by using the memory diagnostics.

Handle the Exception Locally

The **TRY/CATCH** paradigm provides a good way to avoid memory leaks by programming defensively to ensure that your objects are destroyed when exceptions occur. For example, the previous example could be rewritten as shown below:

```
void SomeFunc()
{
    CPerson* myPerson = new CPerson;

    TRY
    {
        // do something that might throw an exception
        myPerson->SomeFunc();
    }
    CATCH( CException, e )
    {
        // handle the exception locally
    }
    END_CATCH

    // now destroy the object before exiting
    delete myPerson;
}
```

This new example sets up an exception handler to catch the exception and handle it locally. It then exits the function normally and destroys the object. The important aspect of this example is that a context to catch the exception is established with the **TRY/CATCH** blocks. Without a local exception frame, the function would never know that an exception had been thrown and would not have the chance to exit normally and destroy the object.

Throw Exceptions After Destroying Objects

Another way to handle exceptions is to pass them on to the next outermost exception-handling context. In your **CATCH** block, you can do some cleanup of your locally allocated objects and then throw the exception on for further processing. The following code shows how this can be done:

```
void SomeFunc()
{
    CPerson* myPerson = new CPerson;

    TRY
```

```
    {
        // do something that might throw an exception
        myPerson->SomeFunc();
    }
    CATCH( CException, e )
    {
        // destroy the object before passing exception on
        delete myPerson;
        // throw the exception to the next handler
        THROW_LAST( );
    }
    END_CATCH

    // on normal exits, destroy the object
    delete myPerson;
}
```

If you call functions that can throw exceptions, you can use **TRY/CATCH** blocks to make sure that you catch the exceptions and have a chance to destroy any objects you have created. In particular, be aware that many Microsoft Foundation Class Library functions can throw exceptions.

12.5 Throwing Exceptions from Your Own Functions

It is possible to use the Microsoft Foundation Class Library exception-handling paradigm simply to catch exceptions thrown by functions in the Microsoft Foundation Class Library or other libraries. In addition to simply catching exceptions thrown by library code, you can throw exceptions from your own code if you are writing functions that can encounter exceptional conditions.

► **To throw an exception:**

- Use one of the Foundation Class Library helper functions, such as **AfxThrowMemoryException**, listed in AFX.H. These functions throw a preallocated exception object of the appropriate type.

  When an exception is thrown, execution of the current function is aborted and jumps directly to the **CATCH** block of the innermost exception frame. The exception mechanism bypasses the normal exit path from a function. Therefore, you must be sure to delete those memory blocks that would be deleted in a normal exit. In the following example, a function tries to allocate two memory blocks and throws an exception if either allocation fails:

```
{
    char* p1 = malloc( SIZE_FIRST );
    if( p1 == NULL )
        AfxThrowMemoryException();
    char* p2 = malloc( SIZE_SECOND );
    if( p2 == NULL )
```

```
    {
        free( p1 );
        AfxThrowMemoryException();
    }

    // ... do something with allocated blocks ...

    // in normal exit, both blocks are deleted
    free( p1 );
    free( p2 );
}
```

If the first allocation fails, you can simply throw the memory exception. If the first allocation is successful but the second one fails, you must free the first allocation block before throwing the exception. If both allocations succeed, then you can proceed normally and free the blocks when exiting the function.

12.6 Exceptions in Constructors

When throwing an exception in a constructor, clean up whatever objects and memory allocations you have made prior to throwing the exception, as explained in the previous section.

Throwing an exception in a constructor is tricky, however, because the memory for the object itself has already been allocated by the time the constructor is called. There is no simple way to deallocate the memory occupied by the object from within the constructor for that object. Thus, you will find that throwing an exception in a constructor will result in the object remaining allocated. For a discussion of how to detect objects in your program that have not been deallocated, see "Detecting Memory Leaks" on page 290.

If you are performing operations in your constructor that can fail, it might be a better idea to put those operations into a separate initialization function rather than throwing an exception in the constructor. That way, you can safely construct the object and get a valid pointer to it. Then, you can call the initialization function for the object. If the initialization function fails, you can delete the object directly.

12.7 Frame Variables and Exceptions

Explicitly allocated heap objects must also be deallocated before an exception is thrown. With frame objects, the frame memory will be reclaimed automatically by the exception mechanism. Although the memory occupied by the frame object is reclaimed, the destructor for the frame object is not executed by the exception mechanism.

For most objects the reclamation of frame space is sufficient to clean up the object, but for objects that allocate memory in addition to the frame space that they occupy, such as **CString** objects, the default exception handling is not sufficient to completely deallocate the object. In addition, objects whose destructors are an integral part of their operations need special handling during exceptions.

CString: The Problem of Deallocating Heap Space

When a **CString** object is allocated on the frame, its constructor also allocates memory on the heap to hold the characters of the string. Thus, a **CString** occupies space on the frame and also on the heap. When a **CString** frame variable is destroyed normally, its destructor takes care of deallocating the heap space used by the object. When the normal destruction of the **CString** is bypassed by an exception, this heap space is not deallocated, even though the frame space occupied by the **CString** is reclaimed.

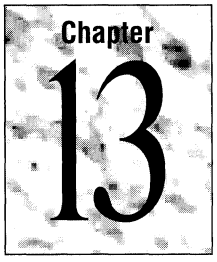
► **To avoid this CString memory leak:**

- Call the **Empty** function for any **CString** frame variables when handling an exception. The following example shows how to do this:

```
{
    CString s1 = "This is a test";

    TRY
    {
        char* p1 = new char[ A_BIG_BLOCK ];
    }
    CATCH( CMemoryException,e )
    {
        // deallocate heap space used by string
        s1.Empty();

        // now you can safely throw the exception
        THROW_LAST( );
    }
    END_CATCH
}
```

The necessity to explicitly deallocate heap resources for a frame-based object is not limited to **CStrings**. Since the destructors for frame objects are not automatically executed when an exception interrupts normal program flow, any frame-based object where the destructor performs significant tasks will need special attention during exception handling.

Application Design



The Microsoft Foundation Classes provide support for writing Windows applications. This chapter describes the specific design principles involved in using the Microsoft Foundation Classes for Windows applications.

13.1 Using Microsoft Foundation Classes to Write Windows Applications

A traditional Windows program that is written without the Microsoft Foundation Class Library typically has at least the following five components:

Component	Purpose
WinMain	Calls initialization function and processes message loop
InitApplication	Initializes window data and registers one or more Windows registration classes
InitInstance	Saves instance handle and creates main window
MainWndProc	Processes messages for main window
About	Processes messages for "About" dialog box

A Windows program that is written with the Microsoft Foundation Class Library has a slightly different structure than a traditional Windows program. Much of the functionality that you have to provide yourself in a traditional Windows program, such as the **WinMain** function and the message loop, is provided automatically by the Microsoft Foundation Class Library. Thus, you have fewer programming tasks when making a Microsoft Foundation-based Windows program.

The following list shows how the parts of a traditional Windows program are replaced by the capabilities of the Microsoft Foundation Class Library.

Windows component	Foundation capability
WinMain	Provided by Microsoft Foundation Class Library. Instead of writing **WinMain** yourself, you derive an application class from **CWinApp** and create an object of that derived class.
InitApplication	Default implementation provided by Microsoft Foundation Class Library. You can override the **InitApplication** function of **CWinApp** to add your own special initialization.
InitInstance	You typically override the **InitInstance** function of **CWinApp** to create the application's main window.
MainWndProc	Instead of writing a window procedure, you derive a window class from one of the existing Foundation window classes.
About	Instead of writing a dialog window procedure, you create a **CModalDialog** object and let it process the user interaction.

The following list shows the typical steps you will complete to take advantage of the Microsoft Foundation Class Library to write a Windows application:

▶ To use Microsoft Foundation classes to write a Windows program:

1. Derive your own window class to serve as the main window. You typically derive from the Microsoft Foundation class **CFrameWnd** or **CMDIFrameWnd**. (For more information on this topic, see the cookbook section "Deriving Frame Windows.")
2. Define a message map to associate specified window messages with member functions of your derived window class. For a complete description of how to define message maps, see the cookbook section "Handling Window Messages."
3. Derive your own application class from **CWinApp**.
4. Override the **InitInstance** member function of **CWinApp** to create a main window.
5. Define an object of your application class. You define the application object as a global variable in the main .CPP file for your program, as shown in the example below. This object will be automatically initialized by the Microsoft Foundation-provided **WinMain** at program startup.

```
// in .H file
class CTheApp : public CWinApp
{
    // ... class declaration ...
}

// in .CPP file
CTheApp  myApp;    // define global application object
```

For an example of how to derive the minimum necessary application and window objects for a Microsoft Foundation-based Windows application, see the sample program files HELLO.H and HELLO.CPP. The following section describes some of the specific tasks that must be done by overridden member functions of the application class. Later sections describe the functions of the window class that can be overridden.

13.2 Deriving Classes from CWinApp

The Microsoft Foundation class **CWinApp** provides much of the framework of a basic Windows application. You will typically not need to change very much of this functionality except for the initialization parts. The following six member functions of **CWinApp** are designed to be overridden in derived classes. You will almost always override **InitInstance**. The other functions can be overridden as needed to provide special processing beyond the default behavior.

CWinApp function	Purpose	When to override
InitApplication	One-time application initialization	Seldom
InitInstance	Creates main window	Almost always
Run	Message loop	Seldom
PreTranslateMessage	Extra message processing	Seldom
OnIdle	Idle loop processing	If needed to perform background tasks when no messages are pending
ExitInstance	Called when program terminates, returns exit code to Windows	Sometimes, to perform application cleanup

Initializing Your Application

Windows allows several copies of the same program to be running at the same time. Thus, application initialization is conceptually divided into two sections: one-time initialization that is done the first time the program runs, and instance initialization that runs each time a copy of the program runs, including the first time.

The one-time application initialization of a typical Windows program calls **RegisterClass** for each of the unique Windows registration classes used by the program. Windows programs that use the Microsoft Foundation Class Library do not normally need to call **RegisterClass**, so there is typically no need to do one-time initialization in a Microsoft Foundation-based Windows program.

The Microsoft Foundation class **CWinApp** provides the member function **InitApplication**, which you can override to perform any one-time application initialization tasks for your application. The Microsoft Foundation classes already provide much of the default initialization (such as creating Windows registration classes) that is normally found in traditional Windows initialization functions. If you do not need to do any other special one-time initialization, you do not have to override **InitApplication**.

How to Initialize Each Application Instance

Each instance (copy) of a Windows program is given a chance to perform instance-specific application initialization. Typically, each instance must at least create a main window. The Microsoft Foundation class **CWinApp** provides the member function **InitInstance** that you will normally override to perform instance-specific application initialization.

For example, the sample program in HELLO.CPP overrides **InitInstance** to create a main window from the derived window class `CMainWindow`, and assigns the window pointer to the **CWinApp** member variable **m_pMainWnd**. The overriding function `InitInstance` then shows the Window and updates it, as shown here:

```
//in .H file
class CTheApp : public CWinApp
{
public:
    BOOL InitInstance();
};

// in .CPP file
BOOL CTheApp::InitInstance()
{
    m_pMainWnd = new CMainWindow();
    m_pMainWnd->ShowWindow( m_nCmdShow );
    m_pMainWnd->UpdateWindow();

    return TRUE;
}
```

The member variable **m_nCmdShow** is passed to the **ShowWindow** function to specify whether the window is visible or not. **m_nCmdShow** is the value passed to **WinMain** in the **nCmdShow** argument.

In your derived application class, you will typically override **InitInstance** in much the same way as shown, substituting your own window class for `CMainWindow`. You may also use `InitInstance` to perform additional initialization tasks (buffer allocation, etc.) as well.

Idle Loop Processing

Idle loop processing allows your program to perform specified tasks when there is not pending user input or system messages. The default message loop provided by the Microsoft Foundation Class Library calls the virtual function **OnIdle** in class **CWinApp** whenever there are no pending messages in the message queue for the application.

If you want to do idle processing, you can override **OnIdle** in your derived application class. This function has a **LONG** argument and returns a **BOOL** result. By default, it returns **FALSE** to indicate that no idle processing time is required. The **LONG** argument indicates how many times **OnIdle** has been called since the last message was taken from the event queue.

Idle loop processing proceeds according to the following list of actions:

1. If the message loop in the Microsoft Foundation Class Library checks the message queue and finds no pending messages, it calls `OnIdle` for the current application object, supplying 0 as the *lCount* argument.
2. `OnIdle` performs some processing, and returns **TRUE** if it wants to be called again to do further processing.
3. The message loop checks the message queue again. If no messages are pending, it calls `OnIdle` again, incrementing the *lCount* argument.
4. Eventually, `OnIdle` finishes processing all its idle tasks and returns **FALSE**. This tells the message loop to stop calling `OnIdle` until the next message is retrieved from the message queue, at which point the idle cycle restarts with the argument reset to 0.

An example may make this process clearer. Since the argument to `OnIdle` is incremented each time it is called, you can essentially prioritize the idle tasks that you want to perform. By checking the value of the argument to `OnIdle`, you can determine how long (relatively) the message queue has been empty.

For instance, assume that you have two idle tasks you want to do. The first is high priority, and you want to do it whenever there is an idle moment. The second task is less important, and should be done only when there is a long pause in user input. To handle these two cases, you could implement your `OnIdle` function as shown here. Note that you must call **OnIdle** for the base class (**CWinApp**) to ensure that the default idle processing gets done too.

```
BOOL CTheApp::OnIdle( LONG lCount )
{
   CWinApp::OnIdle( lCount );
```

```
    switch( lCount )
    {
    case 0:
        TRACE( "frequent\n" );
        return TRUE;
    case 10000:
        TRACE( "infrequent\n" );
        return FALSE;
    default:
        return TRUE;
    }
}
```

Since message processing does not occur until `OnIdle` has returned control to the message loop, you should never retain control for too long. Trying to do too much in `OnIdle` can hamper your program's responsiveness to user input. If you have lengthy idle processing, break it up into smaller pieces and do one piece each time `OnIdle` is called.

13.3 The Resource File

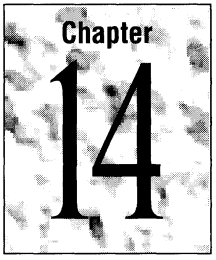
The resource file for Windows applications based on the Microsoft Foundation Class Library is typically the same as the resource file for an equivalent program written without the Microsoft Foundation classes. You normally define your program resources, such as menus and dialog boxes, in a text file (*.RC) that is processed by the Windows resource compiler (RC.EXE) into a binary resource file (*.RES). The binary resource file is combined with your compiled and linked program code to form the final executable file. You can also use the dialog editor to create dialog resources for inclusion in your program.

For example, the About dialog box for the sample program in HELLO.RC is defined by the following text:

```
AboutBox DIALOG 22, 17, 144, 75
    STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
    CAPTION "About Hello"
{
    CTEXT "Microsoft Windows"               -1,  0,  5, 144,  8
    CTEXT "Microsoft Foundation Classes"    -1,  0, 14, 144,  8
    CTEXT "Hello, Windows!"                 -1,  0, 23, 144,  8
    CTEXT "Version 1.0"                     -1,  0, 36, 144,  8
    DEFPUSHBUTTON "OK"          IDOK, 56, 56,  32, 14, WS_GROUP
}
```

See the example programs provided with the Microsoft Foundation Class Library for sample resource definition files. Also see the *Windows Software Development Kit* documentation of the resource compiler for more specific information about resource definition formats.

Window Management



The Microsoft Foundation Classes provide support for creating and managing Windows-based windows for displaying data and handling user input. This support is built on top of the existing Windows window types and messaging mechanism. This chapter describes how to use the Microsoft Foundation Classes to create windows and handle window messages.

14.1 Creating a Frame Window

In a traditional Windows program, you create your own Windows registration class by filling in a **WNDCLASS** structure and passing it to the **RegisterClass** function. If you are using the Microsoft Foundation Class Library, you typically derive a window class from an existing Foundation class using the normal C++ syntax for class inheritance. For frame windows, the Microsoft Foundation Class Library provides three base classes from which you can derive your frame window classes.

Base class	Frame window class
CFrameWnd	Standard frame window
CMDIFrameWnd	Multiple Document Interface frame window
CMDIChildWnd	Multiple Document Interface child window

The Microsoft Foundation Class Library also provides other C++ classes that support other types of windows, such as dialog boxes and controls. These other classes are covered in later sections of the cookbook. Each of these base classes provides default functionality that is appropriate to the function of the window.

When you derive your frame window from an existing Microsoft Foundation frame window class, you will complete two main tasks:

1. Define a constructor for the derived class.
2. Define message-handling functions and a message map for the derived class so that window messages coming to the derived window can be correctly routed to the proper handler functions.

These two tasks are described in the following two sections.

14.2 Constructors for Derived Window Classes

When you derive your window class from a Microsoft Foundation base window class, you must at least implement a constructor for your window class. To initialize the class, the constructor can call the **Create** member function, which calls the Windows API function that actually create the window that is displayed on screen by Windows.

For example, the example program file HELLO.H contains a class declaration, a fragment of which is shown here, for a derived window class and its constructor. The example program file HELLO.CPP contains the definition for the constructor, also shown here.

```
// in HELLO.H
class CMainWindow : public CFrameWnd
{
public:
    CMainWindow();

    // other parts of declaration left out ...
};

// in HELLO.CPP
CMainWindow::CMainWindow()
{
    LoadAccelTable( "MainAccelTable" );
    Create( NULL, "Hello Foundation Application",
        WS_OVERLAPPEDWINDOW, rectDefault, NULL, "MainMenu" );
}
```

The constructor in the example above loads an accelerator table resource for the window and then calls the **Create** member function, passing in arguments that specify various characteristics of the window.

If you want to add other child windows to your main frame window, you can create those windows in the frame window constructor.

Note If you are defining a window class from which you plan to derive other window classes, you should not call **Create** in the constructor of the base class, because in derived classes more than one window will be created by successive calls to base class constructors. Instead, you should implement `Create` in your base window class.

14.3 Handling Window Messages

When you use the C++ derivation mechanism of the Microsoft Foundation Class Library, the derived class inherits all the functionality of the base window class, including its window procedure. The window procedure is where incoming window messages are processed. A traditional window procedure is made up of a large **switch** statement that examines the window message and its arguments to determine what action to take.

When you use the Microsoft Foundation window classes, you are still free to override the window procedure for the base window class and use **switch** statements to decode incoming window messages. However, to take full advantage of the Microsoft Foundation Class Library functionality and to make your code more compatible with future enhancements to the library and its associated programming tools, you should use the message-map mechanism to associate specific window messages to message-handler functions that you write.

► **To use the message-map mechanism in your derived window classes:**

1. Define message-handler member functions in your derived window class.
2. Use the **DECLARE_MESSAGE_MAP** macro in your derived window class declaration.
3. Use the **BEGIN_MESSAGE_MAP**, **END_MESSAGE_MAP**, and message-specific macros in the implementation file (.CPP) for your derived window class.

You can define message-handler functions as member functions for your derived window class. Typically, you define one message-handler function for each window message that you handle. The base window class from which you derive your window will handle all the other window messages that you don't explicitly handle.

There are three main categories of messages that a window receives, as listed here:

- **WM_COMMAND** messages generated by user menu selection or accelerator invocation.
- Notification messages from child windows. These are also **WM_COMMAND** messages, but the window procedure arguments contain the control ID of the

child window and an event code, such as **BN_CLICKED**, to identify why the child window is sending the notification.

- Other **WM_XXX** messages, such as **WM_PAINT**, generated by the system or user input.

Menu-Command Messages

When a user chooses a menu item in a Windows program, the system sends a **WM_COMMAND** message to the frame window that contains the menu bar. The arguments that are sent with the **WM_COMMAND** message contain a menu-item ID number. The ID number is the same as defined in the resource definition file (.RC). In the Microsoft Foundation Class Library, you typically define one messagehandler function for each menu item that your window supports. You associate these message-handler functions with specific menu-item ID numbers by defining a message map, as explained below.

► To handle menu-command messages:

1. Define one message-handler function for each menu item. For example, if your menu includes two items, Open and Save, with the ID numbers **IDM_OPEN** and **IDM_SAVE**, you would declare your window class as follows:

```
class CMyWnd : public CFrameWnd
{
    // constructor not shown...

public:
    afx_msg void OnOpen();
    afx_msg void OnSave();

    DECLARE_MESSAGE_MAP()
};
```

Note The declaration of all message-handler functions should use the **afx_msg** prefix to show that they will be called through the message-map mechanism, although the prefix is not required.

Notice that message-handler functions for menu commands have no arguments and return no value. Since they are member functions of the window class, they have access to the other member functions and member variables of the window object to get the information they need to perform their task.

Notice also the **DECLARE_MESSAGE_MAP** macro in the class declaration. This macro is required to enable the message-map mechanism for the class.

2. Once the message-handler functions are defined and the message map enabled by the **DECLARE_MESSAGE_MAP** macro, define the message map itself to indicate which message-handler functions are to be associated with which messages, as follows:

```
BEGIN_MESSAGE_MAP( CMyWnd, CFrameWnd )
    ON_COMMAND( IDM_OPEN, OnOpen )
    ON_COMMAND( IDM_SAVE, OnSave )
END_MESSAGE_MAP()
```

   The **BEGIN_MESSAGE_MAP** macro has two arguments: the name of the derived class and the name of the base class. The **ON_COMMAND** message-map entry macro takes the ID number of the menu item and a pointer to the member function (the function pointer is produced by simply using the name of the function). Finally, the **END_MESSAGE_MAP** macro has no arguments.

The above message map directs the window procedure for the base window class to call the derived window class member function **OnOpen** (which you must define) when the window receives a **WM_COMMAND** menu message where *wParam* is equal to **IDM_OPEN**. Likewise, the **IDM_SAVE** menu command is associated with the **OnSave** function. Other menu item commands can be associated with other message-handler member functions in a similar manner. The message map shown above is equivalent to the following **switch** statement in a traditional window procedure:

```
switch( msg )
{
    case WM_COMMAND:
    {
        if( LOWORD( lParam ) == 0 )
        {
            switch( wParam )
            {
               case IDM_OPEN:
                    DoOpen();
                    break;
                case IDM_SAVE:
                    DoSave();
                    break;
                default:
                    return FALSE;
            }
        }
        break;
    }
    default:
        return FALSE;
}
```

The message-map definition macros (eg. **BEGIN_MESSAGE_MAP**) for a particular class should be evaluated only one time during a compilation; thus they typically appear in the implementation file (.CPP) for your window class rather than in the interface file (.H).

The next section shows how to use the message-map mechanism to respond to notification messages from child windows.

Notification Messages from Child Windows

Windows programs often use a main frame window containing one or more child windows. These child windows are often predefined control windows such as buttons or edit text fields. These controls communicate with their parent window by sending **WM_COMMAND** notification messages. For example, a child button control responds to a user mouse click by sending a **WM_COMMAND** message to its parent window. The arguments to the window message procedure contain the control ID of the button and the the constant **BN_CLICKED**. Thus, the parent window gets a notification message that tells it the ID of the control and what happened to that control.

The Microsoft Foundation Class Library provides support through message maps for handling notification messages from child windows. A set of macros is provided to support notification messages that are generated by standard control windows. For example, the message-map entry macro for a **BN_CLICKED** notification message is **ON_BN_CLICKED**. Macros for other notification messages are formatted in a similar fashion. For a list of available notification message-map entry macros, see AFXMSG.H or the reference manual.

► To handle notification messages from child windows:

- Provide for each possible message a control ID number and a function pointer to the member function to be called when that message is received. Add entries to the message map and member functions for your frame window class to handle the possible notification messages for your window.

  For example, assume that you have a child button window with an ID of **ID_MY_FIRST_BUTTON**. The following class declaration and message-map definition calls the member function **OnMyFirstButtonClick** when the button sends a **BN_CLICKED** notification message to the frame window:

```
class CMyWnd : public CFrameWnd
{
public:
    afx_msg void OnMyFirstButtonClick();
```

```
    // other class declaration stuff ...

    DECLARE_MESSAGE_MAP();
};

// in .CPP file
BEGIN_MESSAGE_MAP( CMyWnd, CFrameWnd )
    ON_BN_CLICKED( ID_MY_FIRST_BUTTON, OnMyFirstButtonClick )
END_MESSAGE_MAP()
```

► **To differentiate between messages sent by several child windows:**

- Use different control ID arguments to the message-map macro. For example, you can have two different child button windows that send the same **BN_CLICKED** message. Then the message map described above would have a second entry for another button child window, as follows:

```
BEGIN_MESSAGE_MAP( CMyWnd,CFrameWnd )
    ON_BN_CLICKED( ID_MY_FIRST_BUTTON, OnMyFirstButtonClick )
    ON_BN_CLICKED( ID_MY_SECOND_BUTTON, OnMySecondButtonClick )
END_MESSAGE_MAP()
```

  Notice that the second message-map entry refers to a different member function, which you would have to define for your frame window class.

Like the message-handler functions for menu-command messages, all message-handler functions for notification messages are declared with the **afx_msg** prefix and take no arguments and return no value. They are similar to the **ON_COMMAND** macro used for message-map entries for menu commands. The message-map macros for the notification messages have two arguments: the ID of the child window and a function pointer to the user-defined message-handler function for that message.

Dialog boxes, which are described in the "Dialogs and Controls Windows" chapter of the cookbook, use this same notification mechanism: within the message-map entries, each child window notification message is matched to a user-defined message-handler function.

The next section shows how to use the message-map mechanism to respond to messages that are not from menu commands or notifications from child windows.

Other Window Messages

Unlike the message-handler functions for menu commands and child notification messages, the message-handler functions for other types of **WM_XXX** messages, such as **WM_PAINT** or **WM_RBUTTONDOWN**, variously take one or more arguments and may have a return a value. The function name and the argument signature required for each of these message-handler functions is predefined by the message-map macro for each particular message.

For example, the message-handler function for the **WM_RBUTTONDOWN** message must be declared as shown here:

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

The function prototypes for all the message handler functions are declared in the **CWnd** declaration in AFXWIN.H. Note the difference here from the previous discussions of message-handler functions for menu commands and notification messages. For menu commands and notification messages, you are free to make up your own name for your message-handler function. You identify the message-handler function by passing the function name to the message-map entry macro for that message.

For example, if you called the message-handler function for the **IDM_ABOUT** menu command `MyWonderfulMenuCommandHandler`, your message map would look like this:

```
BEGIN_MESSAGE_MAP( CMyWnd, CFrameWnd )
    ON_COMMAND( IDM_ABOUT, MyWonderfulMenuCommandHandler )
END_MESSAGE_MAP()
```

In contrast, the names for the message-handler functions for the other **WM_XXX** messages are predefined by the Microsoft Foundation Class Library. Thus, you must use the Foundation-defined name and the required argument signature for each window message that you handle through the message map. The file AFXWIN.H contains function prototypes for all the message-handler functions for the **WM_XXX** messages. You can find these prototypes by searching AFXWIN.H for the word **afx_msg**. This word, when prefixed to a function prototype, identifies the function as a message-handler function.

Note Message-handler functions prefixed with **afx_msg** can be thought of just like virtual functions in that they are meant to be overridden in derived classes. They differ from virtual functions in the way they are actually implemented and dispatched, which is more efficient than standard virtual functions.

The macro that defines the message-map entry for the **WM_RBUTTONDOWN** message is named **ON_WM_RBUTTONDOWN**. This macro expects to find a message-handler member function named **OnRButtonDown** defined for the window class. Since the names of the message-handler functions for **WM_XXX** window messages are predefined, the macros for the message-map entries for these messages do not need any arguments, as shown in the following procedure.

► **To create a window class that responds to a right mouse button click:**

- Declare a class and define a message map as follows:

```
class CMyWnd : public CFrameWnd
{
    // constructor not shown...

public:
    afx_msg void OnRButtonDown( UINT nFlags, CPoint point );

    DECLARE_MESSAGE_MAP()
};

// in .CPP file
BEGIN_MESSAGE_MAP( CMyWnd, CFrameWnd )
    ON_WM_RBUTTONDOWN()
END_MESSAGE_MAP()
```

The prototype function declaration for **OnRButtonDown** is contained in the **CWnd** class declaration in AFXWIN.H. Consult this file to determine the proper function name and argument/return value signature for the message-handler function for the message that you want to handle.

14.4 Calling the Default Window Procedure from a Message-Handler Function

A common practice in Windows programming is to handle a particular window message in a window procedure and then call the default window procedure, **DefWindowProc**, to finish the processing for the message. If you are using the Microsoft Foundation Class Library's message-map mechanism to handle window messages, you call your base class's message-handler function if you want to finish processing the message in the default manner. You should not change the arguments to the message handler before passing them on to the base class message handler.

The following example shows how a derived class might handle a **WM_CHAR** message under certain circumstances and pass the message on to its base class for further processing as necessary.

► **To filter keystrokes to a CEdit control, allowing only digits:**

1. Derive a class from **CEdit** and define the `OnChar` message-handler function to handle **WM_CHAR** messages.
2. For each **WM_CHAR** message, add code to examine the incoming character to see if it is a digit.
   - If the character is a digit, call the base class version of **OnChar** to allow the character to be inserted into the control in the default way. You can also allow the BACKSPACE key and TAB keys to be processed in the default manner.
   - If the character isn't a digit, simply return without calling the base class **OnChar**. This will cause the character to be ignored by the control.

The following class declaration shows how this might be done:

```
class CNumEdit : public CEdit
{
public:
    afx_msg void OnChar( UINT nChar, UINT nRepCnt, UINT nFlags )
    {
        if( (( nChar <= '9' ) && ( nChar >= '0' ))
            || ( nChar == VK_TAB )
            || ( nChar == VK_BACK ))
        {
            CEdit::OnChar( nChar, nRepCnt, nFlags );
        }
    }

    DECLARE_MESSAGE_MAP()
};


BEGIN_MESSAGE_MAP( CNumEdit, CEdit )
    ON_WM_CHAR()
END_MESSAGE_MAP()
```

For a more complete discussion of the **CNumEdit** example shown above, see "Deriving Controls from a Standard Control" on page 337.

14.5 Overriding Window Procedure for a Window Class

The preferred way to handle window messages with the Microsoft Foundation Class Library is to use message maps and message-handler functions. You can, however, override the window procedure for a Foundation window class to use the more traditional Windows-style **switch** statement to respond to window messages.

► **To use a traditional window procedure in your derived window class:**

1. Override base window class **WindowProc**.
2. Use a traditional **switch** statement that uses **WindowProc** parameters.
3. Call base class **WindowProc** to get message-map processing for those messages that are not handled directly.

The following example code shows how to override **WindowProc** for a class derived from **CEdit**. Like the previous section, this derived class, named `CNumEdit`, accepts only digits. You can compare the message-map version in the previous section with this more traditional version. The first code fragment is a partial class declaration for `CNumEdit`, showing the declaration for overriding **WindowProc**. After that is the implementation of the overriding `WindowProc`.

```
class CNumEdit : public CEdit
{
    // other declaration stuff left out...

    virtual LONG WindowProc( UINT message, UINT wParam, LONG lParam );

    // other declaration details left out...

};

LONG CNumEdit::WindowProc( UINT message, UINT wParam, LONG lParam )
{
    switch( message )
    {
        case WM_CHAR:
            if( ( wParam <= '9' ) && ( wParam >= '0' ) )
            {
                return CEdit::WindowProc( message, wParam, lParam );
            }
            else
            {
                return FALSE;
            }
         default:
            return CEdit::WindowProc( message, wParam, lParam );

    }
}
```

14.6 Scrolling

Scroll bars are typically created as special child windows in the parent frame window. They react to user mouse and keyboard events by sending notification messages to the parent window. This is similar to the mechanism used for other control windows, which is described in the preceding section, "Notification Messages from Child Windows," on page 316.

One difference between scroll bars and other types of control windows is that scroll bars do not use **WM_COMMAND** messages for notification. Instead, scroll bars have their own messages, **WM_VSCROLL** and **WM_HSCROLL**, which are used to notify the parent window of user interaction with the scroll bar.

► **To use the Foundation's message maps to handle scrolling:**

- Define functions to handle messages from the vertical and/or horizontal scroll bars and define entries in the frame window's message map to point to these functions. The functions must be named **OnVScroll** and **OnHScroll**.

  The macros that define the message-map entries are named **ON_WM_VSCROLL** and **ON_WM_HSCROLL**. Consider the following class declaration and message map:

```
class CMyWnd : public CFrameWnd
{
public:
 afx_msg void OnVScroll( UINT nSBCode, UINT nPos, CWnd* pScrollBar );
 afx_msg void OnHScroll( UINT nSBCode, UINT nPos, CWnd* pScrollBar );

 // other class declaration stuff ...

 DECLARE_MESSAGE_MAP();
};

BEGIN_MESSAGE_MAP( CMyWnd, CFrameWnd )
    ON_WM_VSCROLL()
    ON_WM_HSCROLL()
END_MESSAGE_MAP()
```

  Notice that the functions **OnVScroll** and **OnHScroll** have three arguments.

  - The first **UINT** argument contains a code that specifies which scrolling action has been requested, such as **SB_LINEDOWN**, **SB_PAGEUP**, or **SB_THUMBPOSITION**. The possible values are the same as those for the *wParam* argument to a **WM_VSCROLL** or **WM_HSCROLL** message.
  - The second **UINT** argument contains an absolute position indicator that applies only to **SB_THUMBPOSITION** and **SB_THUMBTRACK** messages.

- The final argument is a **CWnd** pointer that points to the scroll-bar object that sent the message. If the frame window was created with the **WS_HSCROLL** or **WS_VSCROLL**, then the scroll bars are not real child windows and this argument will be **NULL**. If it is not **NULL**, you can cast this pointer to a **CScrollBar** pointer and call scroll-bar member functions to get or set information about the scroll bar.

14.7 Using MDI Window Classes

The Microsoft Foundation Class Library provides support for Multiple Document Interface (MDI) windows programs with the base window classes **CMDIFrameWnd** and **CMDIChildWnd.** To create an MDI program, you can derive your main frame window class from **CMDIFrameWnd**. You derive child windows from **CMDIChildWnd**. The derivation of these window classes is the same as described for normal frame windows earlier in this chapter. The rest of this section describes some of the special situations that you must allow for in MDI programs.

Deallocating Memory Used by MDI Child Windows

Because many MDI child windows can come and go during a program's execution, it is important to define a function for your MDI child window class that cleans up any memory used by the window when it is closed. For main frame windows, this is less important since there is typically only one frame window per application and the memory allocated by that window is automatically freed by Windows when the application terminates.

You can respond to the **WM_NCDESTROY** message to free resources and memory used by your MDI child window since this message is the last message that the window receives before it is destroyed. To handle the **WM_NCDESTROY** message, you can define the **OnNcDestroy** message-handler function for your MDI child window class and include the **ON_WM_NCDESTROY** macro in your message map.

Accessing the MDI Parent Window

The function **CMDIChildWnd::GetParentFrame()** returns a pointer to the MDI parent frame window for the child. Use this function rather than the normal Windows API function **GetParent**. Since **GetParentFrame** is declared to return a pointer to a **CFrameWnd** object, you will have to typecast the result to a **CMDIFrameWnd** object. For an example of how to use **GetParentFrame**, see the next section, "Changing Frame Window Menus to Match MDI Child Windows."

Changing Frame Window Menus to Match MDI Child Windows

You will often want to change the menus of the MDI parent window to reflect the state of the currently active MDI child window. This is especially true when there is more than one type of child window. However, it is true even when there is only one type of child window because menus often must change to correspond to the state of a particular window.

▶ **To change an MDI window menu:**

1. Use the **GetParentFrame** function, as explained previously in "Accessing the MDI Parent Window."
2. Once you have the frame window, you can call the **MDISetMenu** member function to change the menu state for the frame window.

The following code shows how to respond to the **WM_MDIACTIVATE** message with the **OnMDIActivate** message-map message-handler function to change the frame window's menus. The flag argument to the handler is **TRUE** if the window is being activated, **FALSE** if it is being deactivated.

```
void
CMyChildWnd::OnMDIActivate( BOOL flag, CWnd* pActive, CWnd* pDeActive )
{
    CMDIFrameWnd* pFrame = (CMDIFrameWnd*)GetParentFrame();
    m_pMenuChartWindow = m_pMenuChart->GetSubMenu( CHART_MENU_POS );
    m_pMenuInitWindow = m_pMenuInit->GetSubMenu( INIT_MENU_POS );

    if( flag == TRUE )
    {
        pFrame->MDISetMenu( m_pMenuChart, m_pMenuChartWindow );
    }
    else
    if( flag == FALSE )
    {
        pFrame->MDISetMenu( m_pMenuInit, m_pMenuInitWindow );
    }

    pFrame->DrawMenuBar();
}
```

14.8 Using the AfxRegisterWndClass Function

The attributes of a **CWnd** object are stored in two places, the window object and the Windows registration class. A Windows registration class is different from a C++ class. When Windows creates a window, it requires the name of a registration class. All windows created with a particular registration class share the attributes contained in that registration class.

The window classes of the Microsoft Foundation Class Library also require that the name of a Windows registration class be passed to the **Create** member function. Most of the time you will pass **NULL** for the registration class name to get the default registration class for the particular Foundation window class that you are using. You can use the name of a different Windows registration class if you want to change the attributes of the window.

In traditional Windows programs, you use the Windows function **RegisterClass** to create a registration class. In a Microsoft Foundation Class Library windows program it is often easier to use the Microsoft Foundation function **AfxRegisterWndClass** to create a registration class.

Although a Windows registration class has numerous fields, there are four key attributes of a registration class that you will typically want to change for a window class:

- The class style
- The default mouse cursor
- The brush used to paint the background
- The icon used to represent the minimized window

The Microsoft Foundation Class Library automatically creates Windows registration classes for its predefined window classes (**CFrameWnd**, **CMDIFrameWnd**, and **CMDIChildWnd**). When you derive your own classes from one of these base classes, you inherit the attributes defined for the base class.

For those situations where you want to change one of the four window class attributes listed above, the Foundation function **AfxRegisterWndClass** allows you to create a Windows registration class by specifying as arguments the class style, cursor, background brush, and icon. When you supply **NULL** as the value for any of the four arguments except the class style, you specify that you want the default value for that attribute. Thus, it is easy to change any combination of the four attributes.

► To register a Windows registration class and change its four class attributes:

- Specify the arguments for class style, cursor, background brush, and icon. **AfxRegisterWndClass** generates a synthetic name (it can be different each time your program runs) and creates a Windows registration class that contains your attribute arguments. You can use the name that is returned as an argument to the **Create** member function in your derived window classes.

```
CString className = AfxRegisterWndClass( CS_HREDRAW | CS_VREDRAW,
                                ::LoadCursor( NULL,IDC_UPARROW ),
                                ( COLOR_WINDOW+1 ),
                                NULL);
```

Note The scope resolution operator "::" is used without a class name in front of the **LoadCursor** function name in the example above to show that the name refers to the Windows function rather than any class member function with the same name.

► **To pass on the attributes of a Windows registration class to CWnd::Create:**

- You can assign the name of the newly registered class to a **CString** and then pass the string to **CWnd::Create** to create a window that has the attributes defined for that Windows registration class.

```
// assign class name to intermediate CString
CString myClassName = AfxRegisterWndClass( CS_HREDRAW | CS_VREDRAW,
                                ::LoadCursor( NULL,IDC_UPARROW ),
                                ( COLOR_WINDOW+1 ),
                                NULL);
CMyWnd* myWnd = new CMyWnd;
myWnd->Create( myClassName, ... );
```

- Since **AfxRegisterWndClass** returns a pointer to the registration class name and checks for redundant registration, it can also be used inline as the first argument to **CWnd::Create**, as shown by the following example from BOUNCE.CPP in the MDI sample program:

```
// use class name directly
CMDIChildWnd::Create( AfxRegisterWndClass( CS_HREDRAW | CS_VREDRAW,
                                ::LoadCursor( NULL,IDC_UPARROW ),
                                ( COLOR_WINDOW+1 ),
                                NULL),
                            szTitle,
                            style,
                            rect,
                            parent );
}
```

14.9 Simple Way to Change a Window Icon

The previous section described how to use **AfxRegisterWndClass** to change the icon that Windows uses to display a minimized window. If all you want to do is change the icon and leave the class style, cursor, and background brush unchanged, then you can define an icon resource with the proper ID in your application's resource file. The Microsoft Foundation window classes will automatically use that icon. This technique is simpler than using **AfxRegisterWndClass** because you do not have to write any code; defining the icon resource is sufficient.

By default, the Foundation window classes use an empty rectangle icon to represent a minimized window. The classes **CFrameWnd** and **CMDIChildWnd** look in the application's resource file for an icon with the ID number **AFX_IDI_STD_FRAME**. These classes will use that icon instead of the default if it exists.

▶ **To change a window icon:**

- Design an icon with the Windows Icon Editor and include it in your application resource file with the following command taken from the sample program file HELLO.RC:

```
AFX_IDI_STD_FRAME    ICON    hello.ico
```

Within one application:

- All window classes derived from **CFrameWnd** and **CMDIChildWnd** will use the **AFX_IDI_STD_FRAME** icon, or the default application icon if it is not defined.
- All windows derived from **CMDIFrameWnd** use the icon with the ID number **AFX_IDI_STD_MDIFRAME**, or the default application icon if it is not defined.

If you have a program that uses more than one type of child window derived from **CFrameWnd** or **CMDIChildWnd**, such as an MDI program with more than one type of child window, the Microsoft Foundation Class Library's default mechanism for choosing custom minimized icons does not support different minimized icons for those child windows. To have a custom icon for different types of child windows in the same application, use **AfxRegisterWndClass** to create a Windows registration class for each child window class. For a description of how to use **AfxRegisterWndClass**, see the section "Using the AFXRegisterWnd Class Function" on page 324. Also, see the MDI sample program.

14.10 Using Member Variables Instead of cbWndExtra Bytes

Windows provides a mechanism by which an arbitrary number of extra bytes can be attached to each window of a particular class. Programmers typically use these extra bytes to store extra information about the window or the data displayed by the window.

In a traditional Windows program, you specify the number of extra bytes for a particular window class in the **cbWndExtra** field of the **WNDCLASS** structure, which is used when you call **RegisterClass** to create the Windows registration class. For each window of that registration class that you create, Windows allocates the specified number of extra bytes for the window structure. The bytes are

accessed as an untyped array of bytes with the **GetWindowLong** and **SetWindowLong** functions. These bytes are allocated in the USER's heap, which is a limited Windows resource used by all running Windows applications.

In a Windows program based on the Microsoft Foundation Class Library, you seldom call **RegisterClass**. Instead, you derive a new window class from an existing Foundation window class using the C++ derivation mechanism.

If you need extra data attached to each window that you have created from a derived class, you can simply add member variables to the class declaration when you derive the class. You can then use normal C++ syntax to access these member variables for each window object that you create.

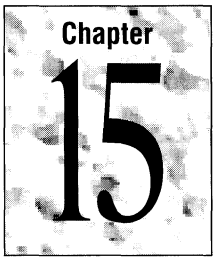
Access to these member variables has the advantage of being type-safe, as opposed to the untyped array-of-bytes method used in traditional Windows programs. In addition, the member variables are allocated in the program's data segment, which is not as precious as the USER heap.

▶ To use member variables to add extra bytes:

- Declare the member variables to hold window-specific data and then access those members:

```
class CMyWnd : public CFrameWnd
{
public:
    // window-specific extra data
    char* m_FileName;
    long m_cbChars;
    BOOL m_isDirty;
    int m_topLineDisplayed;

    // other class declaration stuff ...
};

CMyWnd* myWindow = new CMyWnd( ... );

if( myWindow->m_cbChars > MAX_SIZE )
    //...
```

Dialogs and Control Windows



The Microsoft Foundation classes provide support for creating and managing Windows dialog boxes. The dialog classes are derived from the **CWnd** class described in the "Window Management" chapter, so many of the principles described in that chapter apply to the dialog classes as well.

15.1 Dialog Boxes

The Microsoft Foundation Class Library provides support for dialog box windows with the window classes **CDialog** and **CModalDialog**. For simple modal dialogs, you can sometimes use the **CModalDialog** class directly, such as when creating an "About" dialog box, but most often you will have to derive your own dialog classes in much the same way as described in "Creating a Frame Window" on page 311.

Modal Dialog Boxes

Modal dialog boxes require that the user dismiss the dialog box before going to another application window. The simplest example of a modal dialog box is an About dialog box.

► **To create an About dialog box:**

1. Define a dialog resource template for your dialog box and then create a **CModalDialog** window to use that template. You do not need to derive your own dialog window class box.
2. Call the **DoModal** member function to process user input until the user dismisses the dialog.

The following dialog resource template is taken from HELLO.RC:

```
AboutBox DIALOG 22, 17, 144, 75
    STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
    CAPTION "About Hello"
{
    CTEXT "Microsoft Windows"               -1,  0,  5, 144,  8
    CTEXT "Microsoft Foundation Classes"    -1,  0, 14, 144,  8
    CTEXT "Hello, Windows!"                 -1,  0, 23, 144,  8
    CTEXT "Version 1.0"                     -1,  0, 36, 144,  8
    DEFPUSHBUTTON "OK"          IDOK, 56, 56,  32, 14, WS_GROUP
}
```

The following code is taken from HELLO.CPP. It shows how the main window message-handler function for the About menu command creates a **CModalDialog** window based on the dialog template shown above and then calls **DoModal** for the dialog.

The arguments to the constructor for a **CModalDialog** window are the name of the dialog template and a pointer to a **CWnd** object which is the parent window of the dialog box window. In the following example, the dialog can pass **this** as the parent window pointer since the dialog is being created in a member function of **CMainWindow** and the main window is the parent window of the dialog box:

```
void CMainWindow::OnAbout()
{
    CModalDialog about( "AboutBox", this );
    about.DoModal();
}
```

► **To initialize the modal dialog box before it is displayed:**

1. Derive a dialog class from **CModalDialog**.
2. Override the virtual **OnInitDialog** function. **OnInitDialog** is called when the dialog receives the **WM_INITDIALOG** message.

   An example of a situation in which you would need to do this would be calculating available memory for an About dialog box.

   Note Because **OnInitDialog** is so commonly overridden by derived dialog classes, it is a virtual function that is automatically called by the base dialog class when the dialog receives a **WM_INITDIALOG** message. **OnInitDialog** is not called through the normal Microsoft Foundation message map mechanism, so you are not required to have to create a message-map entry for the **WM_INITDIALOG** message.

The following class declaration shows how you might derive from **CModalDialog** to handle the **WM_INITDIALOG** message:

```
class CMyModalDlg : public CModalDialog
{
    // override OnInitDialog
    virtual BOOL OnInitDialog();
};

BOOL CMyModalDialog::OnInitDialog()
{
 //...
}
```

▶ To customize the response of the buttons in a CModalDialog object:

1. Override the virtual **CModalDialog** functions **OnOK** and **OnCancel** to handle mouse clicks in the OK and Cancel buttons of your dialog box.
2. Use the predefined control IDs **IDOK** and **IDCANCEL** for these buttons. These two control IDs are special in **CModalDialog** objects in that they do not require message-map entries. **BN_CLICKED** events in buttons with these IDs are automatically sent to the **OnOK** and **OnCancel** virtual functions.

▶ To customize the response to other events in your modal dialog box so that it is the same as choosing the OK or Cancel buttons:

- Define a message-map entry for that notification event and hook it to the **OnOK** or **OnCancel** function, as shown here for a double-click in a list box:

```
class CMyModalDlg : public CModalDialog
{
    // override OnInitDialog
    virtual    BOOL OnInitDialog();

public:
    virtual void OnOK()
    {
        // do something with list box selection...

        // and dismiss dialog
        EndDialog( IDOK );
    }

    DECLARE_MESSAGE_MAP()
};

BEGIN_MESSAGE_MAP( CMyModalDlg,CModalDialog )
    // double-click list box aliases for OK
    ON_LBN_DBLCLK( ID_TYPEFACE, OnOK )
END_MESSAGE_MAP()
```

The above declaration includes the **DECLARE_MESSAGE_MAP** macro which is required to enable message map mechanism for the derived class.

The default implementations of **OnOK** and **OnCancel** call **EndDialog** to dismiss the modal dialog. If you override either of these functions, you should call **EndDialog** in your overridden versions.

Deriving from CDialog

Unlike **CModalDialog**, where you have the choice of whether or not to derive your own dialog window class box, you must derive your dialog from the **CDialog** class.

▶ **To derive a dialog class from CDialog:**

1. Define a contructor for the derived class and optionally define message-handler functions and a message map. This is essentially the same technique as that described for the standard frame window.

   A typical constructor simply calls the **Create** function and supplies the name of the dialog template and the parent window that owns the dialog.

2. Optionally, override the virtual function **OnInitDialog** to perform any necessary dialog initialization before the dialog is displayed. You do not have to define a message-map entry for the **WM_INITDIALOG** message for classes derived from **CDialog**. The **CDialog** class automatically calls the **OnInitDialog** member function when the dialog receives a **WM_INITDIALOG** message.

The advantage of a dialog derived from **CDialog**, as opposed to **CModalDialog**, is that it can be modeless. That is, the user can switch back and forth from the modeless dialog to other application windows.

More on Dialog Boxes

If you want an even more elaborate modal dialog box where the user can interact with several controls, you can add message-handler functions and message-map entries to handle notification events from the dialog box controls, as described in "Notification Messages from Child Windows" on page 316. If any of the controls other than the OK and Cancel buttons allow the user to close the dialog, you can call **EndDialog** in the message-handler functions for those control notifications. See the example program code in SHOWFONT.CPP for examples of derived modal dialog boxes that handle notification messages from child controls.

► **To derive a simple class from CDialog:**

- Use the following class declaration as a model:

```
class CMyDialog : public CDialog
{
    public:
    // constructor calls Create
    CMyDialog( CWnd* parentWnd )
    {
        Create( "MyTemplateName", parentWnd );
    }

    virtual BOOL OnInitDialog();

};
```

Beyond these two basic tasks, you will typically define message-handler functions and message-map entries to handle notification messages from other controls in your dialog. This process is described in "Notification Messages from Child Windows" on page 316.

Type-Safe Dialog Item Access

One optional technique that you might find useful in dialog boxes is to define type-safe member functions to access individual controls in the dialog box. For example, if your dialog contains a button control with the ID number **ID_MYBUTTON**, you can define the following function in your derived dialog class:

```
CButton* CMyDialog::GetMyButton()
{
    return (CButton*)GetDlgItem( ID_MYBUTTON );
}
```

Once this function is defined, you can access the button with code similar to the following:

```
myDlg.GetMyButton()->SetState( TRUE );
```

See the sample program RESTOOL.

Using a Dialog Box as a Main Window

Although it is possible to fill a standard frame window with one or more child control windows to imitate the look of a dialog box, using a real dialog window gives you access to extra functionality, such as allowing the user to move between controls with the TAB key, that is not easily available in a standard frame window. In addition, when you use a dialog window, many operations on child control windows, such as getting the text from an edit text control, are simplified. Thus, it is sometimes desirable to use a dialog window as the main window of your application.

As described in the "Cookbook" section on instance initialization, you typically create your main application window in the **InitInstance** function of **CWinApp**. When you are using a standard frame window, you create the window and assign it to the **m_pMainWnd** member variable of the application object. When you are using a dialog window as the main window, you still create the dialog window in **InitInstance** just as you would for a normal frame window.

► **To use a dialog box as the main window:**

1. Derive your main window from **CDialog**.

   At least one of its message-handler functions should call the **PostQuitMessage** function to signal that the user wishes to quit the program. A typical way to do this is to use the **WS_SYSMENU** style in the dialog template and handle the **WM_CLOSE** message from the Control menu to call **PostQuitMessage**.

   The following resource definition, class declaration, and message-map definition show how this strategy could be implemented.

```
main DIALOG 22, 17, 250, 120
STYLE WS_DLGFRAME | WS_CAPTION | WS_SYSMENU
CAPTION "Dialog Title"
BEGIN
    /* control definitions go here ...*/
END

class CMainDlgWindow : public CDialog
{
public:
    CMainDlgWindow()
    {
        Create( "main" );
    }
```

```
    afx_msg void OnClose()
    {
        PostQuitMessage( 0 );
    }

    DECLARE_MESSAGE_MAP()
};

// in .CPP file
BEGIN_MESSAGE_MAP( CMainDlgWindow, CDialog )
    ON_WM_CLOSE()
END_MESSAGE_MAP()
```

2. Create the dialog box in your application's overriding `InitInstance` function:

```
BOOL CMyApp::InitInstance()
{
    m_pMainWnd = new CMainDlgWindow();
    m_pMainWnd->ShowWindow( m_nCmdShow );
    m_pMainWnd->UpdateWindow();
    return TRUE;
}
```

Your dialog might also have a Quit button, in which case the message-handler function for the **BN_CLICKED** message from that button would also call **PostQuitMessage**.

15.2 Using Microsoft Foundation Control Classes

The Microsoft Foundation Class Library provides a set of classes that correspond to the standard Windows control windows, such as buttons, edit text controls, and scroll bars. You can use these control classes in any situation where you would normally use a Windows control.

The following list shows the Microsoft Foundation control classes and the corresponding Windows controls:

Foundation class	Windows control
CStatic	Static control
CButton	Button control
CListBox	List box control
CComboBox	Combo box control
CEdit	Edit control
CScrollBar	Scroll bar control

The Microsoft Foundation control classes require two-stage contruction. That is, you first allocate the object and then call a separate initialization function (named **Create**) to create the Windows control and attach it to the Foundation control object. You must call **Create** on a Foundation control object before calling any other member functions for the object.

► To create a standard Microsoft Foundation control object:

1. Allocate a control object, either by
   - Defining it on the frame, or by
   - Calling the **new** operator.
2. Call the **Create** function for the object, supplying the necessary arguments to create the corresponding Windows control window and attach it to the Microsoft Foundation control object.

   The following code shows how you might declare a **CEdit** object in the class declaration of your derived dialog class, and then call the **Create** member function in **OnInitDialog** function. Because the **CEdit** object is declared as an embedded object (not as a pointer) it is automatically allocated when the dialog object is contructed. Although the **CEdit** object is constructed at the same time as the dialog, it must still be initialized with the **Create** member function.

```
class CMyDialog : public CDialog { protected:    CEdit m_edit;
public:     virtual BOOL OnInitDialog(); };
```

The **OnInitDialog** function sets up a rectangle, and then calls **Create** to create the Windows edit control and attach it to the uninitialized **CEdit** object. After creating the edit control, you can also set the input focus to the control by calling the **SetFocus** member function. Finally, you return **FALSE** from **OnInitDialog** to show that you set the focus. If you return **TRUE**, the dialog manager will set the focus to the first control item in the dialog item list.

```
BOOL CMyDialog::OnInitDialog()
{
    RECT r;
    r.top = 85;
    r.bottom = 110;
    r.left = 180;
    r.right = 210;

    m_edit.Create( WS_CHILD | WS_VISIBLE | WS_TABSTOP |
                        ES_AUTOHSCROLL | WS_BORDER,
                              r,
                              this,
                              ID_EXTRA_EDIT);
```

```
    // set input focus to new edit control
    m_edit.SetFocus();

    // return FALSE only if you have set the focus
    return FALSE;
}
```

15.3 Deriving Controls from a Standard Control

As described in the previous section, the Microsoft Foundation control classes provides classes that correspond to the predefined control windows supported by Windows, such as buttons, edit controls, and scroll bars. For the most part, you will be able to use these control classes just as they are defined since their default behavior conforms to the normal user-interface guidelines for Windows programs.

There are some situations, however, where you may want to modify the behavior of a control. You can often modify behavior of a control by deriving a new class from an existing control class, as shown by the following example.

Suppose you wanted to modify the standard edit control, **CEdit**, so that it only accepted digits. You could call this derived control `CNumEdit`, since it would be good for user-entered numeric values. In addition to modifying its behavior so that it only accepted digits, you could add member functions to get and set its contents by numeric value, thus encapsulating the conversion between text and numeric values.

There are three steps to creating a derived control class, as listed here:

1. Derive your class from an existing control class and override the **Create** function so that it provides the necessary arguments to the base class **Create** function.
2. Provide message-handler functions and message-map entries to modify the control's behavior in response to specific window messages.
3. Provide new member functions to extend the functionality of the control (optional).

These three steps are described in the following three sections.

▶ To override the Create function class:

1. Override the **Create** function for your derived class that takes the same arguments as the base class **Create** and pass the arguments on to the base class and let it do most of the default initialization of the control.
2. Perform other initialization tasks specific to your derived class. For the `CNumEdit` example, call the **LimitText** member function to limit the number of characters that the derived control will accept.

```
class CNumEdit : public CEdit
{
public:

    BOOL Create( DWORD dwStyle, const RECT& rect,
            CWnd* pParentWnd, UINT nID )
    {           if ( ! CEdit::Create( dwStyle, rect, pParentWnd,
nID ) )
            return FALSE;

        LimitText( 16 );
        return TRUE;
    }
    // other class declaration stuff removed...
};
```

▶ To use message-handler functions to modify behavior:

1. Define the message-handler function **OnChar** to examine each incoming character. If the character is a digit, the BACKSPACE key, or the TAB key, call the **OnChar** function of **CEdit** to pass it to the base class for normal processing. If the character is not a digit, then don't call the **CEdit** class's **OnChar** function and the base class will never see the character. The implementation of the `OnChar` member function of **CNumEdit** is as follows.

```
class CNumEdit : public CEdit
{
public:

    BOOL Create(DWORD dwStyle, const RECT& rect,
                CWnd* pParentWnd, UINT nID);

    // filter incoming characters
    afx_msg void OnChar( UINT nChar, UINT nRepCnt, UINT nFlags );

    DECLARE_MESSAGE_MAP()

};
```

```
void CNumEdit::OnChar( UINT nChar, UINT nRepCnt, UINT nFlags )
{
    if( (( nChar <= '9' ) && ( nChar >= '0' ))
        || ( nChar == VK_TAB )
        || ( nChar == VK_BACK ))
    {
        CEdit::OnChar( nChar, nRepCnt, nFlags );
    }
}
```

2. Define a message-map entry for the **WM_CHAR** message:

```
BEGIN_MESSAGE_MAP( CNumEdit, CEdit )
    ON_WM_CHAR()
END_MESSAGE_MAP()
```

3. Filtering **WM_CHAR** messages works fine for removing nondigits from keyboard input, but to be complete you would also have to handle the **WM_PASTE** message to filter nondigits from Clipboard data. A discussion of how to filter Clipboard data is beyond the scope of this section.

► To extend functionality with new member functions:

- To be truly useful, the `CNumEdit` class can provide new functions to get and set the value of the control with numeric values. That way, the code that converts the text to a number can be encapsulated in the derived control. Programmers who use the control can simply get and set its value without worrying about conversion.

```
class CNumEdit : public CEdit
{
public:

    BOOL Create( DWORD dwStyle, const RECT& rect,
                 CWnd* pParentWnd, UINT nID );

    // filter incoming characters
    afx_msg void OnChar( UINT nChar, UINT nRepCnt, UINT nFlags );


    // provide Get and Set functions to deal with numeric value
    LONG GetValue();
    void SetValue( LONG v );

    DECLARE_MESSAGE_MAP()

};
```

► **To implement GetValue and SetValue:**

- Use the standard C run-time functions **sscanf** and **sprintf** to convert between number and text. Use the **CWnd** member functions **GetWindowText** and **SetWindowText** to access the text shown in the edit control. Possible implementations of these functions are shown below:

```
LONG CNumEdit::GetValue()
{
    char buff[64];
    LONG v;

    GetWindowText( buff, sizeof( buff ) );

    if( sscanf( buff, "%d", &v ) )
    {
        return v;
    }
    else
    {
        return 0;
    }
}

void CNumEdit::SetValue( LONG v )
{

    char buff[64];

    sprintf( buff, "%d", v );

    SetWindowText( buff );
}
```

Using a Derived Control in a Dialog

Dialogs normally contain one or more control windows such as buttons, list boxes, and edit controls. The type and position of those controls is typically specified in a dialog template resource. If you create a derived control class as described in the previous sections, you cannot specify the derived control in a dialog template since the resource compiler does not know anything about your derived class.

► To place a derived control in a dialog:

1. Embed an object of the derived control class in the declaration of your derived dialog class, as shown in the following class declaration.

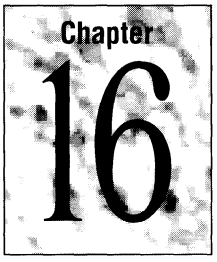
```
class CMyDialog : public CDialog { protected:    CNumEdit m_numEdit;
public:     virtual BOOL OnInitDialog(); };
```

2. Override the **OnInitDialog** message-handler function in your dialog class to call **Create** for the derived control that you embedded in your dialog box. A sample `OnInitDialog` is shown here:

```
BOOL CMyDialog::OnInitDialog()
{
    // create the derived control
    RECT r;
    r.top = 50;
    r.bottom = 75;
    r.left = 20;
    r.right = 120;

    m_numEdit.Create( WS_CHILD | WS_VISIBLE | WS_TABSTOP |
                              ES_AUTOHSCROLL | WS_BORDER,
                              r,
                              this,
                              ID_MYNUMEDIT );

    // other initialization tasks...

    return TRUE;
}
```

Because the derived controls are embedded in the dialog, they will be automatically destroyed when the dialog is destroyed.

Graphics



The Microsoft Foundation classes provide support for Windows graphics functions, including device contexts, standard drawing objects such as pens, brushes, and bitmaps, and a mechanism for handling drawing related Window messages.

16.1 Handling the Paint Message

Windows programs do most of their drawing in response to **WM_PAINT** messages. Using the Microsoft Foundation classes, your derived window class can handle the **WM_PAINT** message by implementing the **OnPaint** message handler function and defining a message-map entry with the **ON_WM_PAINT** macro. The following list describes the steps necessary to handle **WM_PAINT** messages using the Microsoft Foundation classes:

1. Derive your window class from a Foundation window class.
2. Define the **OnPaint** message-handler member function for your derived window class.
3. Define a message-map entry with the **ON_WM_PAINT** macro.

The following class declaration shows how to derive a window class from a Microsoft Foundation window class and declare the **OnPaint** message-handler member function necessary to handle the **WM_PAINT** member function. It also shows the required inclusion of the **DECLARE_MESSAGE_MAP** macro to enable the Foundation message-map mechanism for the derived window class.

```
class CMainWnd : public CFrameWnd
{
public:
    // constructor is required for derived window classes
    CMainWnd();

    // message-handler function for WM_PAINT message
    afx_msg void OnPaint();

    // enable message-map mechanism
    DECLARE_MESSAGE_MAP()
};
```

The `OnPaint` message-handler function for a derived window class is called when the window receives a **WM_PAINT** message. In a traditional Windows program, you call **BeginPaint** to get a device context for the window, draw your window contents, and finally call **EndPaint** to release the device context when responding to a **WM_PAINT** message.

When using the Microsoft Foundation classes, you can create a **CPaintDC** object instead of calling **BeginPaint** and **EndPaint**. The constructor for this object automatically calls **BeginPaint** and its destructor calls **EndPaint**. You must specify a pointer to a **CWnd** object when creating the **CPaintDC** object. Since you typically create a **CPaintDC** object in the `OnPaint` member function of your derived window class, you can pass **this** as the **CWnd** pointer argument, as shown in the following example implementation of `OnPaint`:

```
void CMainWnd::OnPaint()
{
    CPaintDC dc( this );

    // do painting here with dc...

}
```

When the **CPaintDC** object is created, its constructor calls **BeginPaint**. Notice that since the **CPaintDC** is created on the stack, it is automatically deallocated when you exit `OnPaint`. This automatic deallocation implies that the destructor for the object, which calls **EndPaint**, will be invoked as the function terminates.

The **CPaintDC** device context can be used to perform any drawing operation that you could normally do in a normal Windows device context since all the device context GDI drawing functions are available as member function in the device context class. For example, to draw text in the window, you can use the **TextOut** member function, which is equivalent to the Windows Graphical Device Interface function **TextOut**. The following code fragment shows how to use a **CPaintDC** device context to draw in a window during the **WM_PAINT** message.

```
void CMainWnd::OnPaint()
{
    CString s = "Hello, Windows!";
    CPaintDC dc( this );
    CRect rect;

    GetClientRect( &rect );
    dc.SetTextAlign( TA_BASELINE | TA_CENTER );
    dc.TextOut( ( rect.right / 2 ), ( rect.bottom / 2 ),
                s, s.GetLength() );
}
```

To associate the `OnPaint` message-handler function with the **WM_PAINT** message, you must define a message-map entry for the derived window class. The following message-map definition shows how this is done:

```
BEGIN_MESSAGE_MAP( CMainWnd, CFrameWnd )
    ON_WM_PAINT()
END_MESSAGE_MAP()
```

16.2 Getting the Device Context from a CWnd Window

Although most drawing in a Windows program occurs during **WM_PAINT** message processing, there are times, such as when you are providing tracking feedback to mouse movement, that you want to draw directly into a window without generating a **WM_PAINT** message. At these times, you can create a **CClientDC** device context from a pointer to a **CWnd** window. You can use the **CClientDC** device context to perform any graphics operation that you would typically do with a normal Windows device context on the client area of a window. The client area of a window is all the area not including the title bar, size border, menu bar, and scroll bars.

Use a **CClientDC** device context at those times when you would use the Windows API functions **GetDC** and **ReleaseDC** in a traditional Windows program. The constructor for the **CClientDC** device context calls **GetDC** and the destructor calls **ReleaseDC**. If you create the **CClientDC** object on the stack, its destructor will automatically ensure that the device context is properly released when you are done with it. The following example shows how to create a **CClientDC** device context in a member function of a derived window class:

```
CMyWnd::FeedBack()
{
    CClientDC dc( this );

    // do drawing here ...
}
```

Use **CClientDC** member functions for drawing in the same way that **CPaintDC** member functions are used, as described in the previous section, "Handling the Paint Message."

16.3 Graphic Objects

Windows provides a variety of drawing tools to draw within device contexts. It provides pens to draw lines, brushes to fill interiors, and fonts to draw text. The Microsoft Foundation classes provide graphic object classes that are equivalent to the drawing tools in Windows. The following list shows the available Foundation graphic object classes and the equivalent Windows GDI handle types:

Foundation Classes	Windows Tools
CPen	**HPEN**
CBrush	**HBRUSH**
CFont	**HFONT**
CBitmap	**HBITMAP**
CPalette	**HPALLETE**
CRgn	**HRGN**

Each of the Microsoft Foundation graphic object classes has a constructor that allows you to create graphic objects of that class.

The following four steps are typically used when a graphic object is needed:

1. Define a graphic object on the frame, athough you can also use the **new** operator. Perform initialization of the object as necessary.
2. Select the object into the current device context, saving the old graphic object that was selected before.
3. When done with the current graphic object, select the old graphic object back into the device context to restore state.
4. Allow the frame allocated graphic object to be automatically deleted when exiting scope (or use the **delete** operator it if allocated with **new**.)

Note If you have graphic objects in your program that will be used repeatedly, it is often a good idea to allocate the objects just one time, in the **CWinApp::InitInstance** function for example, and select them into a device context whenever you need to use them. Delete these objects when you no longer need them, which is typically just before the program terminates.

Creating and Deleting Graphic Objects

There are two techniques for creating the graphic objects of the Microsoft Foundation Class Library. The first technique uses a single-stage constructor that also initializes the object and performs any necessary memory allocation for the object. The other technique uses a two-stage contruction strategy in which the object is first constructed and then initialized with a separate function.

Two-stage construction is generally safer than single-stage construction.

The advantage of single-stage construction is that it involves only a single programming statement, but the disadvantage is that the constructor may throw an exception if incorrect arguments are provided or memory allocations fail. On the other hand, two-stage construction will not throw an exception, but it is slightly more bothersome in that you must write two program statements to create the object. In either case, the method for destroying the object is the same.

The following two sections show examples of single-stage and two-stage construction of graphic objects, as well as examples of destruction.

Single-stage construction of a graphic object

▶ **To create a pen object**

- Specify the pen style, the pen width, and the pen color to the **CPen** constructor.

  Possible pen styles are the same as those used for traditional Windows programs and listed in the file WINDOWS.H. The pen color is specified as an **RGB** color whose three components represent the relative saturation of red, green, and blue in the color.

  The following code example shows how to create a black **CPen** object with a single-stage constructor.

```
CPen myPen( PS_DOT, 5, RGB( 0, 0, 0 ) );
```

  If the object is created on the frame, as in the above example, its destructor will automatically be invoked when the object goes out of scope. If you allocate the object on the heap with the **new** operator, use the **delete** operator to deallocate it.

Other graphic object classes have similar constructors with different arguments to specify the attributes necessary to construct them. Before you can use the graphic object to draw, it must be selected into a device context. For more information on how to do this, see "Selecting a Drawing Object in a Device Context" later in this chapter.

Two-stage construction of a graphic object

▶ **To construct a CPen object:**

- Call the constructor with no arguments.

  This constructs the object and initializes it just enough so that it can be deleted if necessary. You cannot use a pen constructed in this way until it has been initialized.

▶ **To initialize this CPen object:**

- Use the **CreatePen** and **CreatePenIndirect** functions.

  The following example shows how to construct the empty **CPen** object and then initialize it with **CreatePen**. The Boolean return value from **CreatePen** allows you to detect if the pen object was not successfully initialized.

```
CPen myPen;
if( pPen.CreatePen( PS_DOT, 5, RGB( 0, 0, 0 ));
    //... use the pen
```

Other graphic object classes have similar two-stage constructors and initialization functions, with different arguments to specify the attributes necessary to construct and initialize each type of object. The next section describes how to select a graphic object into a device context, which is necessary before you can use the object to do any drawing.

Selecting a Drawing Object into a Device Context

The previous section described how to create Microsoft Foundation graphic objects. However, before you can use these objects to draw, you must select them into a device context. By selecting a graphic object into a device context, you are modifying the drawing environment for the device context. This is the standard programming model for traditional Windows programming as well as for the Microsoft Foundation classes.

The Microsoft Foundation device context classes (**CPaintDC**, **CClientDC**, and **CWindowDC**) provide the **SelectObject** member function. You supply a pointer to a graphic object and **SelectObject** selects the object into the device context, returning a pointer to the previous graphic object for the device context.

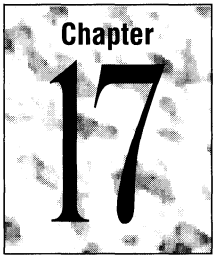
For instance, if you call **SelectObject** with a **CPen** pointer, **SelectObject** will return a **CPen** pointer for the old device context pen. You can later select the old pen back into the device context to restore the original drawing state. Likewise, **SelectObject** will return a **CFont** pointer when you select a new **CFont**, and a **CBrush** pointer when you select a new **CBrush**.

The following example shows how to create a **CPen** object, select it into the device context, do some drawing, and then restore the original pen; all in the context of responding to a **WM_PAINT** message.

```
void CMyWnd::OnPaint()
{
    CPaintDC myDC( this );

    // create the pen
    CPen newPen( PS_SOLID, 2, RGB( 0, 0, 255 ) );

    // select it into the device context
    // save old pen at the same time
    CPen* pOldPen = myDC.SelectObject( &newPen );

    // draw some lines with the pen
    myDC.MoveTo(...);
    myDC.LineTo(...);
    myDC.LineTo(...);
    myDC.LineTo(...);

    // now restore old pen
    myDC.SelectObject( pOldPen );

}
```

Note The graphic object returned by **SelectObject** is a "temporary" object. That is, it will be deleted by **CWinApp::OnIdle** the next time the program gets idle time. As long as you use the object returned by **SelectObject** in a single function without returning control to the main event loop, you will have no problem. But if you try to use the object later, after you have exited the function and the program has executed its event loop again, you risk having the object deleted by the idle time handler. See "Tracking the Mouse in a Window" on page 353 for an example of how to avoid problems when using a temporary graphic object.

User Input



Windows handles user input by sending messages to the active window. The Microsoft Foundation classes support user input with the message map mechanism described in the "Window Management" chapter. The current chapter describes how to use the message map mechanism to handle specific types of user input.

17.1 Handling a Mouse Click in a Window

To handle a mouse click in your window, you need to handle the **WM_LBUTTONDOWN** or **WM_RBUTTONDOWN** message. Using the Microsoft Foundation classes mechanism for handling messages, you will find the following three steps necessary to handle mouse-click messages:

1. Derive your window class from an existing Foundation window class (such as **CFrameWnd** or **CMDIChildWnd**).
2. Define the **OnLButtonDown** or **OnRButtonDown** message-handler member functions for your window class, depending on whether you want to handle left or right mouse button events.
3. Define-message map entries using the **ON_WM_LBUTTONDOWN** or **ON_WM_RBUTTONDOWN** macros, depending on whether you want to handle left or right mouse button events.

The following class declaration fragment shows how to declare the **OnLButtonDown** function in a derived window class. It also shows the necessary inclusion of the **DECLARE_MESSAGE_MAP** macro to enable the message-map mechanism for the derived class.

```
class CMainWnd : public CFrameWnd
{
    // other declaration stuff left out...

    // declare message-handler function
    afx_msg void OnLButtonDown( UINT nFlags, CPoint point );

    DECLARE_MESSAGE_MAP()
};
```

The **OnLButtonDown** message-handler function has two arguments; a **UINT** which indicates whether the CONTROL and/or SHIFT keys were pressed when the mouse was clicked, and a **CPoint** argument containing the coordinates of the mouse at the time of the click. The following function definition shows how to look at the information in these arguments.

```
void CMainWnd::OnLButtonDown(UINT nFlags, CPoint point)
{
    TRACE("LBUTTON down message, point = %d , %d\n",point.x,point.y);
    if(nFlags & MK_CONTROL)
        TRACE("Control key was down\n");
    if(nFlags & MK_SHIFT)
        TRACE("Shift key was down\n");
}
```

The definition of the message-map entries for the mouse down messages is the same as described for other messages elsewhere in this documentation. The following example shows how to define a message-map entry for the **WM_LBUTTONDOWN** message for the derived window class described in this section.

```
// in .CPP file
BEGIN_MESSAGE_MAP( CMainWnd, CFrameWnd )
    ON_WM_LBUTTONDOWN()
END_MESSAGE_MAP()
```

Note You can handle the **WM_RBUTTONDBLCLK** or **WM_LBUTTONDBLCLK** messages for double clicks in your window.

17.2 Tracking the Mouse in a Window

Tracking mouse movements in a window while the mouse button is pressed involves three different window messages. You must first handle the message that signals that the mouse button has been pressed, **WM_LBUTTONDOWN** or **WM_RBUTTONDOWN**, as described in the previous section. You can then handle **WM_MOUSEMOVE** messages until you get a **WM_LBUTTONUP** or **WM_RBUTTONUP** message.

The following list describes the steps necessary to track the mouse in a window. The list assumes that you have derived a window class from an existing Foundation window class, as described in the previous section on handling mouse clicks in a window.

1. Call **SetCapture** in **OnLButtonDown** (or **OnRButtonDown**)
2. Handle mouse movements in **OnMouseMove**
3. Call **ReleaseCapture** in **OnLButtonUp** (or **OnRButtonDown**)

The following example shows the code necessary for a simple doodling window. This window allows the user to draw freehand lines in the window by pressing the left mouse button and holding it down. The code shows how to capture the mouse and how to create a device context for drawing. It also shows how to create a pen drawing tool as described in the section "Graphic Objects" on page 346.

The class declaration for the doodling window is shown as follows. Notice that the class is derived from **CFrameWnd** and that it declares message-handler functions to handle the three mouse messages listed in the first part of this section. Notice also that the class declares several member variables to help with the drawing in the window.

```
class CMainWnd : public CFrameWnd
{
protected:
    BOOL m_bTracking;
    CClientDC* m_pDCDoodle;
    CPen* m_pPenDoodle;
    HPEN m_hPenSaved;
    int  m_nPenWidth;

public:

    CMainWindow();
```

```
// WM_ message handlers
    afx_msg void OnLButtonDown( UINT nFlags, CPoint p );

    afx_msg void OnLButtonUp( UINT nFlags, CPoint p );

    afx_msg void OnMouseMove( UINT nFlags, CPoint p );

    DECLARE_MESSAGE_MAP()
};
```

As described elsewhere, you must define a message map for the window class. The following definition is from the .CPP file for the doodle window class.

```
BEGIN_MESSAGE_MAP( CMainWnd, CFrameWnd )
    ON_WM_LBUTTONDOWN()
    ON_WM_MOUSEMOVE()
    ON_WM_LBUTTONUP()
END_MESSAGE_MAP()
```

In the **OnLButtonDown** message-handler function, you call **SetCapture** to tell Windows that you want to retain exclusive control of the mouse while tracking its movement. This prevents another window from getting mouse messages while you are trying to track the mouse. You will call **ReleaseCapture** when the user releases the mouse button and you stop tracking.

You also create a device context and a drawing pen for the window in response to the **WM_LBUTTONDOWN** message. You then call **SelectObject** to install your pen in the device context. **SelectObject** returns a pointer to a **CPen** object that represents the previous pen settings for the device context.

As mentioned in the section "Selecting a Drawing Object into a Device Context," on page 348, the object returned by **SelectObject** is a temporary object that is liable to be deleted the next time the application is idle. Since idle time can occur while tracking the mouse (when the user stops moving the mouse), you cannot depend on the object returned from **SelectObject** in the mouse down message handler to still be around when the mouse up message is handled.

To avoid relying on the temporary Foundation **CPen** object returned by **SelectObject**, you call **GetSafeHandle** to extract the **HPEN** handle from the **CPen** returned by **SelectObject**. The **HPEN** returned by **GetSafeHandle** is a permanent Windows graphical device interface object that will not be automatically deleted.

Finally, after selecting the newly created pen into the device context, you call **MoveTo** to establish the anchor point for the tracking.

The code for **OnLButtonDown** is shown as follows:

```
void CMainWnd::OnLButtonDown( UINT nFlags, CPoint p )
{
    m_bTracking = TRUE;
    SetCapture();

    // Create the Device Context for this window
    m_pDCDoodle = new CClientDC( this );

    // Create a new pen to draw in the window
    m_pPenDoodle = new CPen( PS_SOLID, m_penWidth, RGB( 0, 0, 0) );

    // Select the pen into the Device Context
    CPen* pPenTemp = m_pDCDoodle->SelectObject( m_pPenDoodle );

    // extract an HPEN from CPen to save for later restoration
    m_hPenSaved = (HPEN)pPenTemp->GetSafeHandle();

    // establish anchor point for doodling
    m_pDCDoodle->MoveTo( p );
}
```

The message-handler function **OnMouseMove** is called when your window receives a **WM_MOUSEMOVE** message. If you are tracking, as indicated by the Boolean member variable **m_bTracking**, you call the **LineTo** function to draw a line from the last anchor point to the current mouse location. Drawing the line also establishes a new anchor point. The code for **OnMouseMove** is shown here:

```
void CMainWnd::OnMouseMove( UINT nFlags, CPoint p )
{
    if( m_bTracking )
    {
        m_pDCDoodle->LineTo( p );
    }
}
```

Finally, the `OnLButtonUp` message-handler function is called when the window receives a **WM_LBUTTONUP** message. In this function, you call **ReleaseCapture** to give other windows access to mouse messages. You also use the **HPEN** saved in the `OnLButtonDown` function to create a **CPen** object that you then use to select the original pen characteristics back into the device context of the window, thus restoring the original drawing environment.

Finally, you delete the doodling pen and device context that you created in `OnLButtonDown`, as follows:

```
void CMainWindow::OnLButtonUp( UINT nFlags, CPoint p )
{
    if( m_bTracking )
    {
        m_bTracking = FALSE;
        ReleaseCapture();

        // Create a CPen from an HPEN
        m_pDCDoodle->SelectObject( CPen::FromHandle( m_hPenSaved ));

        delete m_pPenDoodle;
        delete m_pDCDoodle;
    }
}
```

17.3 Keyboard Events

When a user presses a key on the keyboard, Windows sends a series of messages to the current active window. These messages include **WM_KEYDOWN**, **WM_KEYUP**, and **WM_CHAR**. The Foundation window classes support these messages with message map entries and message-handler functions, just as it does for other window messages. The keyboard messages and their associated message-map entries and message-handler functions are shown in the following list:

Message	Message-map macro	Message-handler function
WM_KEYDOWN	**ON_WM_KEYDOWN**	**OnKeyDown**
WM_KEYUP	**ON_WM_KEYUP**	**OnKeyUp**
WM_CHAR	**ON_WM_CHAR**	**OnChar**

► **To handle any of these key messages:**

1. Derive a window class from an existing Foundation window class
2. Implement the appropriate message-handler functions
3. Define message-map entries for the messages that you want to handle

The following example shows a window class that handles the **WM_CHAR** message.

First, the class is derived from an existing Foundation window class and the `OnChar` message-handler function is declared:

```
class CMainWnd : public CFrameWnd
{
public:
    // constructor
    CMyWnd()

    void OnChar( UINT nChar, UINT nRepCnt, UINT nFlags );

    DECLARE_MESSAGE_MAP()
};
```

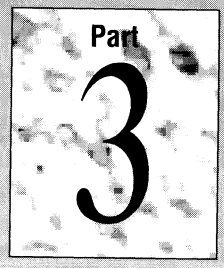
Next, the message map for the window class is defined in the .CPP file for the class:

```
BEGIN_MESSAGE_MAP( CMainWnd, CFrameWnd )
    ON_WM_CHAR()
END_MESSAGE_MAP()
```

Finally, you define the `OnChar` message-handler function. It has three arguments. The first contains the ASCII code for the character typed on the keyboard, the second contains the repeat count if the message is the result of keyboard autorepeat, and the third contains flags that indicate various other conditions of the keyboard, such as whether or not the ALT key was down when the key was pressed. The following code shows a skeleton for the `OnChar` function:

```
void CMainWnd::OnChar( UINT nChar, UINT nRepCnt, UINT nFlags )
{
    TRACE( "The %c character was pressed\n", wChar );
}
```

The Microsoft iostream Class Library Tutorial



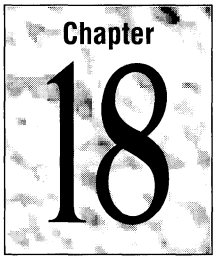
The Microsoft iostream Class Library Tutorial

Microsoft C/C++ Version 7.0 contains a C++ class library that supports object-oriented input/output. This library follows the syntax originally established by the authors of the C++ language and thus represents a de facto standars for C++ input/output.

Chapter 18 provides the training necessary to get you started with the iostream classes. After you work through the examples, you will be able to write programs that process formatted text character streams and binary disk files, and you will be able to customize the library in limited ways. As you work with the iostream classes, you will learn about the C++ "multiple inheritance" feature.

Chapter 19 presents some advanced topics that include iostream class derivation and creation of custom multiparameter "manipulators." The derivation topic gets you started on extending the library and the manipulators are useful for specialized formatting. You will also begin to understand the relationship between the "stream" classes and their subsidiary "buffer" classes. You can then apply some of the iostream library design principles to you own class libraries.

The Fundamentals of iostream Programming



The *C++ Tutorial* introduced the **cin** and **cout** objects together with the insetion (<<) and extraction (>>) operators. Many of the *C++ Tutorial* programming examples use these iostream objects and operators to read input from the keyboard and to display results on the screen. This chapter presents the theory behind input/output (I/O) streams and gives step-by-step instructions for using these useful tools.

18.1 Introduction

This chapter begins with a general description of the iostream classes and then describes output streams, input streams, and input/output streams. Part 3 of the *Class Libraries Reference* contains an alphabetical reference that documents the technical details of the Microsoft iostream Class Library. The *Class Libraries Reference* also contains a detailed class hierarchy diagram.

What Is a Stream?

If you are a C programmer, you know that the C language has no built-in input/output capability. You must use C run-time library functions such as **printf**, **_open**, **_read**, **fwrite**, and **getchar** for disk, display, and keyboard I/O. Variations such as **sprintf** support in-memory formatting.

Like C, C++ does not have built-in I/O capability. All C++ compilers, however, come bundled with a systematic, object-oriented I/O package, known as the "iostream classes." These stream classes are the most standard of all C++ classes because they were developed by the authors of the C++ language.

The "stream" is the central concept in the iostream classes. You can think of a stream object as a kind of "smart file" that acts as a source and/or destination for bytes. A "stream" is a C++ object with characteristics determined by its specific class and by your customized insertion and extraction operators.

The disk operating system, through its device drivers, extends the concept of a file to include the keyboard, display, printer, and communication ports. The iostream classes, like the C run-time library I/O routines, interact with these extended files. The Microsoft iostream Class Library, however, goes further. Built-in classes support reading and writing to and from memory with syntax identical to the syntax for disk I/O. As you gain familiarity with the library, you can derive your own stream classes to support, for example, interprocess communication or a graphical user interface.

Microsoft C/C++ Input/Output Alternatives

When you use Microsoft C/C++ version 7.0, you have several options for I/O programming:

- C run-time library direct, unbuffered I/O

  The functions **_open**, **_read**, **_write**, and **_close**, among others, interact directly with the operating system—there is no buffering or formatting. Many existing data management programs use these functions because they are efficient and allow custom buffering schemes. These functions are oriented to the C programmer, but they can be called from C++ programs.

- ANSI C run-time library stream I/O

  Here the word "stream" refers to the C "stdio" run-time library and does not directly relate to the C++ I/O stream classes. Functions such as **fopen, fread, fwrite,** and **printf** do their own buffering before they call the direct functions. These functions are oriented to the C programmer, but they can be called from C++ programs.

- Console and port direct I/O

  C run-time library functions such as **_kbhit** are useful in non-Windows applications because they give the programmer direct access to the hardware. There are no C++ class equivalents.

- The Microsoft Foundation Class Library

  If you are using the Microsoft Foundation Class Library, then you should use the **CFile** classes for disk I/O, particularly in the Windows environment. This ensures that your applications are portable and extensible.

- The Microsoft iostream Class Library

  The iostream classes are particularly useful for buffered, formatted text input and output. Use them for unbuffered or binary I/O if you have decided not to use the Microsoft Foundation Classes and if you need a C++ programming interface. Remember that the iostream classes are an object-oriented I/O *alternative* to C run-time functions such as **printf** and **_read**. Their use is not mandatory in C++ programs.

It is possible to use the iostream classes with Microsoft Windows. The string and file streams work without restrictions, but the character-mode stream objects **cin**, **cout**, and **cerr** are inconsistent with the Windows graphical user interface. If you link with the QuickWin Library (see Chapter 8 in *Programming Techniques*), however, the **cin**, **cout**, **cerr**, and **clog** objects are assigned to special windows because they are connected to the predefined files **stdin**, **stdout**, and **stderr**. If you are an advanced C++ programmer, you can derive custom stream classes that interact directly with the Windows environment.

The iostream Class Hierarchy

The class hierarchy diagram shown in the *Class Libraries Reference* exposes the some of the interrelationships between the iostream classes. There are further "membership" relationships between the **ios** family of classes and the **streambuf** family.

The hierarchy diagram is useful mainly in conjunction with Part 3, "iostream Class Reference." Use it for locating the base classes that provide inherited member functions for their derived classes. You don't need to understand all the interclass relationships in order to work through the examples in this tutorial.

18.2 Output Streams

An output stream object is a destination for bytes. The three most important output stream classes are **ostream**, **ofstream**, and **ostrstream**, as described below.

The ostream Class

The **ostream** class, through the derived class **ostream_withassign**, supports the predefined stream objects:

- **cout,** standard output
- **cerr**, standard error with limited buffering
- **clog**, similar to **cerr** but with full buffering

You will rarely construct an object from class **ostream** (or from class **ostream_withassign**); you will generally use the predefined objects. Under certain circumstances, you may assign these standard names to other stream objects after program startup.

The **ostream** class is best suited to sequential text-mode output. All this chapter's formatting options apply to **ostream** objects, and these objects can be configured for buffered or unbuffered operation. All the functionality of the base class, **ios**, is included in **ostream**. See the *Class Libraries Reference* for details.

If you do construct an object of class **ostream**, then you must specify a **streambuf** object to the constructor. This is an advanced use of streams that is covered in "Deriving Your Own Stream Classes" in Chapter 19.

The ofstream Class

The **ofstream** class supports disk file output. Construct an object of class **ofstream** if you need an output-only disk file. You can specify whether **ofstream** objects accept binary or text-mode data. Indeed, you can change this mode after you have opened the file.

If you specify a filename in the constructor, that file is automatically opened when the object is constructed. Otherwise you can use the **open** member function after invoking the **void**-argument constructor, or you can construct an **ofstream** object based on an open file that is identified by a file descriptor.

Many formatting options and member functions apply to **ofstream** objects. All the functionality of the base classes **ios** and **ostream** is included in **ofstream**, and that includes the ability to open files in either buffered or unbufferd mode. See the iostream section of the *Class Libraries Reference* for details.

The ostrstream Class

The **ostrstream** class supports output to in-memory strings in a manner similar to the C run-time library function **sprintf**. When you need to create a string in memory using the I/O stream formatting facilities, construct an object of class **ostrstream**. Because **ostrstream** objects are write-only, your program must access the resulting string directly through a pointer to **char**.

Constructing Output Stream Objects

If you use only **cout**, **cerr**, or **clog**, you don't have to worry about output stream construction. If you use file streams or string streams, then you must use constructors.

Output File Stream Constructors

If you need an output file stream, you have three choices:

- You can use the **void**-argument constructor, then call the **open** member function.

```
ofstream myFile; // Static or on the stack
myFile.open( "filename", iosmode );

ofstream* pmyFile = new ofstream; // On the heap
pmyFile->open( "filename", iosmode );
```

- You can specify a filename and mode flags in the constructor call, thereby opening the file during the construction process.

```
ofstream myFile( "filename", iosmode );
```

- You can specify an integer file descriptor for a file that is already open for output. In this case you have the option to specify unbuffered output or a pointer to your own buffer.

```
int fd = _open( "filename", dosmode );
ofstream myFile1( fd );  // Buffered mode (default)
ofstream myFile2( fd, NULL, 0 );  // Unbuffered mode ofstream
myFile3( fd, pch, buflen);  // User-supplied buffer
```

For a description of how to use the mode flags of the **open** member function, see the *Class Libraries Reference*.

Output String Stream Constructors

There are two **ostrstream** constructors: one that dynamically allocates its own storage and one that requires the address and size of a preallocated buffer.

- The dynamic constructor is used like this:

```
char* sp;
ostrstream myString << "this is a test" << ends;
sp = myString.str();  // Get a pointer to the string
```

The **ends** "manipulator" in this example adds a null terminator to the string. This terminator is necessary if the string is to be used by C run-time library functions that expect standard strings. For a description of manipulators, see "Format Control" on page 368.

- The use of the "preallocation" constructor is demonstrated by the following example:

```
char s[32];
ostrstream myString( s, sizeof( s ) );
myString << "this is a test" << ends; // Text stored in s
```

Using Insertion Operators

The insertion (<<) operator is the most familiar means of sending bytes to an output stream object. This operator is preprogrammed for all the standard C++ data types, including integers, floating-point values, null-terminated strings, addresses, and so forth. This chapter presents examples that illustrate output format control and the technique for creating insertion operators for your own classes.

Format Control

You can change the width, the justification (right or left), the precision of floating-point values, and the radix, among others. This section explains some of the available output stream formatting options.

This section introduces the term "manipulator." If you are interested in how manipulators fit into the C++ language, see "Writing Your Own Manipulators without Parameters" on page 381. Otherwise, consider the predefined manipulators to be elements of the iostream syntax expressions used in conjunction with the insertion (and extraction) operators.

Output Width

In this book, many examples show data values separated by spaces. If you need output that is lined up in columns, you must specify the output width.

You can specify a width for each displayed item by placing the **setw** manipulator in the stream or by calling the **width** member function. Both the **setw** manipulator and the **width** member function take a *width* parameter.

Example 1

Example 1 prints a column of numbers without specifying an output width:

```
// exios101.cpp
// Displaying columns of numbers
#include <iostream.h>

void main()
{
   double values[] = { 1.23, 35.36, 653.7, 4358.24 };
   for( int i = 0; i < 4; i++ )
      cout << values[i] << '\n';
}
```

The output looks like this:

```
1.23
35.36
653.7
4358.24
```

Example 2

Example 2 shows how the **width** member function manages output width. By calling the **width** function with an argument of 10, the program specifies that the displayed values are to appear right-aligned in a column at least 10 characters wide.

```
// exios102.cpp
// The width member function
```

```
#include <iostream.h>

void main()
{
   double values[] = { 1.23, 35.36, 653.7, 4358.24 };
   for( int i = 0; i < 4; i++ )
   {
      cout.width(10);
      cout << values[i] << '\n';
   }
}
```

The output from this example looks like this:

```
      1.23
     35.36
     653.7
   4358.24
```

In this example, leading blanks are added to any value that is less than 10 characters wide. Later, you will learn how to replace the leading blanks with specified padding characters.

Example 3

Sometimes you need different widths for different data elements in the same line. The **width** member function provides this capability, but the **setw** manipulator is more convenient, as Example 3 illustrates:

```
// exios103.cpp
// The width member function
#include <iostream.h>
#include <iomanip.h>

void main()
{
   double values[] = { 1.23, 35.36, 653.7, 4358.24 };
   char *names[] = { "Zoot", "Jimmy", "Al", "Stan" };
   for( int i = 0; i < 4; i++ )
      cout << setw( 6 )  << names[i]
           << setw( 10 ) << values[i] << endl;
}
```

Note that the **width** member function is declared in IOSTREAM.H, but if you use the **setw** manipulator (or any other manipulator with parameters), you must include both IOSTREAM.H and IOMANIP.H. Also note that the newline character ('\n') has been replaced by the **endl** manipulator.

In the output from this example, the strings are printed with a width of six and the integers with a width of ten.

```
 Zoot       1.23
Jimmy      35.36
   Al      653.7
 Stan    4358.24
```

Neither **setw** nor **width** truncate values. If the formatted output exceeds the current width, the entire value prints, subject to the stream's current precision setting. You should be aware of this behavior when you design formatted displays that use **setw** or **width**.

Note Both **setw** and **width** affect the following field only. The field width reverts to its default behavior (the necessary width) after one field has been printed. The other stream format options remain in effect until changed.

Padding

You can use the **fill** member function to set the value of the padding character for fields that have a specified width. The default padding character is a blank.

Example 4

In Example 4 a column of numbers is padded with asterisks:

```
// exios104.cpp
// The fill member function
#include <iostream.h>

void main()
{
   double values[] = { 1.23, 35.36, 653.7, 4358.24 };
   for( int i = 0; i < 4; i++ )
   {
      cout.width( 10 );
      cout.fill( '*' );
      cout << values[i] << endl;
   }
}
```

The values print as follows:

```
******1.23
*****35.36
*****653.7
***4358.24
```

Alignment

So far, all output from the example programs has printed flush right within the width field. Output streams default to right-aligned text.

Example 5

Suppose that you want the names in Example 3 to be left-aligned and the number to remain right-aligned. You can use the **setiosflags** manipulator to specify that the output is to be left- or right-aligned. Example 5 shows how to left-align the names:

```
// exios105.cpp
// The setiosflags and resetiosflags manipulators
#include <iostream.h>
#include <iomanip.h>

void main()
{
   double values[] = { 1.23, 35.36, 653.7, 4358.24 };
   char *names[] = { "Zoot", "Jimmy", "Al", "Stan "};
   for ( int i = 0; i < 4; i++ )
      cout << setiosflags( ios::left )
           << setw( 6 )  << names[i]
           << resetiosflags( ios::left )
           << setw( 10 ) << values[i] << endl;
}
```

The output looks like this:

```
Zoot        1.23
Jimmy      35.36
Al         653.7
Stan     4358.24
```

The example sets the left-align flag by using the **setiosflags** manipulator with an argument of **ios::left**. This argument is an enumerator that is defined in the **ios** class, so its reference must include the **ios::** prefix. The **resetiosflags** manipulator turns off the left-align flag to return to the default right-align mode.

Note Unlike **width** and **setw**, the effect of **setiosflags** and **resetiosflags** is permanent. If you turn on left-alignment, for example, it stays on until you turn it off.

Precision

Suppose you want the floating-point numbers in Example 5 to display with only one significant digit. The **setprecision** manipulator tells the object to use a specified number of digits of precision.

Example 6

Example 6 limits the displayed precision to one significant digit.

```
// exios106.cpp
// The setprecision manipulator
#include <iostream.h>
#include <iomanip.h>
```

```
void main()
{
   double values[] = { 1.23, 35.36, 653.7, 4358.24 };
   char *names[] = { "Zoot", "Jimmy", "Al", "Stan" };
   for ( int i = 0; i < 4; i++ )
      cout << setiosflags( ios::left )
           << setw( 6 )
           << names[i]
           << resetiosflags( ios::left )
           << setw( 10 )
           << setprecision( 1 )
           << values[i]
           << endl;
}
```

The program prints this list:

```
Zoot          1
Jimmy      4e+001
Al         7e+002
Stan       4e+003
```

Example 7

You might want to eliminate the scientific notation in Example 6. Two flags, **ios::fixed** and **ios::scientific**, control how a floating-point number prints. You can set and clear these flags with the **setiosflags** and **resetiosflags** manipulators:

```
// exios107.cpp
// The fixed flag
#include <iostream.h>
#include <iomanip.h>

void main()
{
   double values[] = { 1.23, 35.36, 653.7, 4358.24 };
   char *names[] = { "Zoot", "Jimmy", "Al", "Stan" };
   cout << setiosflags( ios::fixed );
   for ( int i = 0; i < 4; i++ )
      cout << setiosflags( ios::left )
           << setw( 6  )
           << names[i]
           << resetiosflags( ios::left )
           << setw( 10 )
           << setprecision( 1 )
           << values[i]
           << endl;
}
```

With fixed notation, the program prints with one digit *after the decimal point.*

```
Zoot          1.2
Jimmy        35.4
Al          653.7
Stan       4358.2
```

If, in Example 7, the **ios::fixed** flag is changed to **ios::scientific**, the program's behavior changes, and it prints this:

```
Zoot     1.2e+000
Jimmy    3.5e+001
Al       6.5e+002
Stan     4.4e+003
```

Again, the program prints one digit after the decimal point. Example 7 shows that if *either* the **ios::fixed** flag or the **ios::scientific** flag is set, then the precision value determines the number of digits after the decimal point. If neither flag is set, then the precision value determines the total number of significant digits.

Note The default value of precision is six. This means that a number such as 3466.9768 prints as 3466.98 unless **ios::fixed** or **ios::scientific** is set, in which case it prints as 3466.976800 or 3.466977e+003, respectively.

Radix

The **dec**, **oct**, and **hex** manipulators set the default radix for both input and output. If you insert the **hex** manipulator into the output stream, for example, the object correctly translates the internal data representation of integers into a hexadecimal (base 16) output format. The default radix is **dec** (decimal).

On insertion, the **hex** manipulator causes integers and long integers to be converted to hexadecimal format. The numbers are displayed with digits a through f in lower case if the flag **ios::uppercase** is clear (the default); otherwise they are displayed in upper case. Thus, for example, the decimal integer 28 is displayed as 1c. The **ios::uppercase** flag is described in Part 3 of the *Class Libraries Reference* (iostream section) under the **flags** member function of class **ios**.

Output File Stream Member Functions

You have already seen the formatting member function **width**. The **setw** manipulator served as a substitute and was more convenient for formatted I/O that used the insertion operator. There are three classes of output stream member functions:

- Member functions that are equivalent to manipulators
- Member functions that perform unformatted write operations
- Member functions that otherwise modify the stream state and have no equivalent manipulator or insertion operator

If you need sequential, formatted output, you might use only manipulators and insertion operators, but if you need random-access binary disk output, you would use several other member functions. In this second case, you may choose to use no insertion operators at all.

The following member functions are particularly useful for disk output. For a complete list of stream member functions, see Part 3 of the *Class Libraries Reference*.

The open Function

If you are using an output file stream (**ofstream**), then you must associate that stream with a specific disk file. You can make this association in the constructor, or you can use the **open** function. The second method allows you to reuse the same stream object with a series of separate files. In either case, the parameters describing the file are the same.

You generally specify an **open_mode** flag when you open the file associated with an output stream (there is a default mode parameter). Here is a list of the **open_mode** flags defined as enumerators in the **ios** class. Note that the list includes flags for both input and output streams. The flags can be combined as appropriate using the bitwise OR (|) operator.

Flag	Function
ios::app	Opens an output file for appending. Every output operation occurs at the physical end of file.
ios::ate	Opens an existing file and seeks to the end. This mode works with both input and output files.
ios::in	Opens an input file. If you use ios::in as an **open_mode** for an **ofstream** file, it prevents the truncation of an existing file.
ios::out	Opens an output file. When you use ios::out for an **ofstream** object without **ios::app**, **ios::ate**, or **ios::in**, then **ios::trunc** is implied.
ios::nocreate	Opens a file only if it already exists; otherwise the open fails.
ios::noreplace	Opens a file only if it does not exist; otherwise the open fails.

Flag	Function
ios::trunc	Opens a file and deletes the old file (if it already exists).
ios::binary	Opens a file in binary mode (default is text mode). See "Binary Output Files" on page 378 for an explanation of binary mode.

Here are three common output stream situations that involve the mode options:

- You want to create a file. If it already exists, delete the old version.

```
ostream ofile( "FILENAME" );  // Default is ios::out
ofstream ofile( "FILENAME", ios::out ); // Equivalent to above
```

- You want to append records to an existing file or create one it if it does not exist.

```
ofstream ofile( "FILENAME", ios::app );
```

- You want to open two files, one at a time, on the same stream.

```
ofstream ofile();
ofile.open( "FILE1", ios::in );
// Do some output
ofile.close(); // FILE1 closed
ofile.open( "FILE2", ios::in );
// Do some more output
ofile.close(); // FILE2 closed
// when ofile goes out of scope it is destroyed
```

The put Function

The **put** function writes a single character to the output stream. The following two statements are the same by default, but the second is affected by the stream's format parameters:

```
cout.put( 'A' ); // Exactly one character written
cout << 'A'; // Format parameters 'width' and 'fill' apply
```

The write Function

The **write** member function writes a block of memory to an output file stream. The number of bytes written is determined by the specified length parameter. The function accommodates the complex binary data structures found in many business and scientific applications.

Example 8

Example 8 shows the **write** member function being used to write binary data to an output file stream.

```
// exios108.cpp
// The write member function
#include <fstream.h>

struct Date
{
   int mo, da, yr;
};

void main()
{
   Date dt = { 6, 10, 91 };
   ofstream tfile( "date.dat" , ios::binary );
   tfile.write( (char *) &dt, sizeof dt );
}
```

The program creates an output file stream and writes the binary value of the `Date` structure to it. The **write** function does not stop writing when it reaches a null character, so the complete class structure is written regardless of its content. For an explanation of **ios::binary**, see "Binary Output Files" on page 378.

The **write** function takes two arguments: a **char** pointer and a count of characters to write. Note the cast to **char*** before the address of the structure object. Without this cast, the program would not compile.

The seekp and tellp Functions

An output file stream keeps an internal pointer corresponding to the position in the file where data will be written next. The **seekp** member function sets that pointer and thus allows random-access disk file output. The **seekp** function is often used in conjunction with the **seekg** function in input/output streams. For a programming example that uses the input stream equivalent to the **seekp** function, see the description for **seekg** in "Input Streams" on page 382.

The **tellp** member function returns the current file position. For a programming example that uses the input stream equivalant to the **tellp** function, see the description for **tellg** in "Input Streams" on page 382.

The close Function

Use the **close** member function to close the disk file associated with an output file stream. The file must be closed in order to properly update the disk. If necessary, the **ofstream** destructor closes the file for you, but you can use the **close** function if you need to open another file for the same stream object.

Note The output stream destructor automatically closes the stream's file only if the file was opened by the constructor or by the **open** member function. If you have passed the constructor a file descriptor for an already-open file or if you have used the **attach** member function, then it is your responsibility to close the file.

Error Processing Functions

These member functions enable you to test for errors while writing to a stream:

Function	Return Value
bad	Returns **TRUE** if there is an unrecoverable error.
fail	Returns **TRUE** if there is an unrecoverable error *or* an "expected" condition, such as a conversion error or the file is not found. Processing can often resume after a call to **clear** with a zero parameter.
good	Returns **TRUE** if there is no error condition (unrecoverable or otherwise) and the end-of-file flag is not set.
eof	Returns **TRUE** on the end-of-file condition.
clear	Sets the internal error state. If called with the default parameters, clears all error bits.
rdstate	Returns the current error state. See the *Class Libraries Reference* for a complete description of error bits.

The **!** operator is overloaded to perform the same function as the **fail** function. Thus the expression `if( !cout)...` is equivalent to `if( cout.fail() )...`

The **void*()** operator is overloaded to be the opposite of the **!** operator. Thus the expression `if( cout)...` is equivalent to `if( !cout.fail() )...`

Note that the **void*()** operator is not the same as **good** because it doesn't test for the end-of-file condition.

The Effects of Buffering

What are the effects of buffering? Consider Example 9. You might expect the program to print `please wait`, wait five seconds, and then proceed. It won't necessarily work as you anticipated, however, because the output is buffered.

Example 9

Example 9 clearly demonstrates the effects of buffering.

```
// exios109.cpp
// A buffered stream object
#include <iostream.h>
#include <time.h>

void main()
{
   time_t tm = time( NULL ) + 5;
   cout << "Please wait...";
   while ( time( NULL ) < tm )
      ;
   cout << "\nAll done" << endl;
}
```

In order to make the program work sensibly, you must somehow tell **cout** to empty itself as soon as you want the message to appear. You can tell an **ostream** object to flush itself by sending it the **flush** manipulator. To fix the program above, change one line:

```
cout << "Please wait..." << flush;
```

This extra step flushes the buffer, which ensures that the message prints before the wait instead of after. You may choose to use **endl** instead of **flush**. The **endl** manipulator flushes the buffer and also outputs a carriage return–linefeed combination.

Note The **cin** object (along with **cerr** and **clog**) is normally tied to **cout**. Thus any use of **cin** (or of **cerr** and **clog**) causes **cout** to be flushed.

Binary Output Files

Streams were originally designed for text, and thus text mode is the default output mode. In text mode, a newline character (hexadecimal 10) always expands to a carriage return–linefeed pair. This expansion could cause problems as shown in Example 10:

Example 10

Example 10 demonstrates one of the chief characteristics of text file output.

```
// exios110.cpp
#include <fstream.h>
int iarray[2] = { 99, 10 };
void main()
{
    ofstream os( "test.dat" );
    os.write( (char *) iarray, sizeof( iarray ) );
}
```

You might expect this program to output the byte sequence { 99, 0, 10, 0 }; actually it outputs the sequence { 99, 0, 13, 10, 0 }. This would clearly cause problems for another program that expected binary input.

If you need true binary output, in which characters are written untranslated, then you have several choices, as shown in Examples 11, 12 and 13.

Example 11

You can specify binary output by using the **ofstream** constuctor mode parameter as shown:

```
// exios111.cpp
// Binary example 1
#include <fstream.h>
#include <fcntl.h>
#include <io.h>
int iarray[2] = { 99, 10 };
void main()
{
    ofstream os( "test.dat", ios::binary );
    ofs.write( iarray, 4 ); // Exactly 4 bytes written
}
```

Example 12

You can construct a stream in the usual way, and then use the **setmode** member function. This function allows you to change the mode after you have opened the file.

```
// exios112.cpp
// Binary example 2
#include <fstream.h>
int iarray[2] = { 99, 10 };
void main()
{
    ofstream ofs ( "test.dat" );
    ofs.setmode( filebuf::binary );
    ofs.write( char*iarray, 4 );
// Exactly 4 bytes written
}
```

As an alternative to the **setmode** member function, you can use the **binary** manipulator:

```
ofs << binary;
```

There is a companion **text** manipulator that switches the stream back to text translation mode.

Example 13

Finally, you can first open the file using the C run-time library **_open** function with a binary mode flag:

```
// exios113.cpp
// Binary example 3
#include <fstream.h>
#include <fcntl.h>
#include <io.h>
int iarray[2] = { 99, 10 };
void main()
{
    filedesc fd = _open( "test.dat",
             _O_BINARY | _O_CREAT | _O_WRONLY );
    ofstream ofs( fd );
    ofs.write( char*iarray, 4 ); // Exactly 4 bytes written
}
```

Overloading the << Operator for Your Own Classes

Output streams use the insertion (<<) operator for the standard types. Now you will learn how to overload the << operator for your own classes.

Example 8, on page 376, illustrated the use of a `Date` structure. A date is an ideal candidate for a C++ class in which the data members (month, day and year) are hidden from view. A `Date` object should know how to display itself, and an output stream is the logical destination.

The following code displays a date on **cout** in a manner consistent with the preceding examples.

```
Date dt( 1, 2, 90 );
cout << dt;
```

To get the **cout** object to accept a `Date` object after the insertion operator, you must overload the insertion operator to recognize an **ostream** object on the left and a `Date` on the right. The overloaded << operator function must then be declared as a **friend** of class `Date` so that it can access the private data within a `Date` object.

Example 14

Example 14 overloads the << operator to accept an **ostream** object on the left and a `Date` object on the right:

```
// exios114.cpp
// Overloading the << operator
#include <iostream.h>
```

```
class Date
{
   int mo, da, yr;
public:
   Date( int m, int d, int y )
   {
      mo = m; da = d; yr = y;
   }
   friend ostream& operator<< ( ostream& os, Date& dt );
};

ostream& operator<< ( ostream& os, Date& dt )
{
   os << dt.mo << '/' << dt.da << '/' << dt.yr;
   return os;
}

void main()
{
   Date dt( 5, 6, 77 );
   cout << dt;
}
```

When you run this program, it prints the date:

```
5/6/77
```

Note that the overloaded operator returns a reference to the original **ostream** object, which means you can combine various insertions:

```
cout << "The date is" << dt << flush;
```

Writing Your Own Manipulators Without Parameters

You have already seen some of the built-in output stream manipulators, including **flush**, **hex**, and **setw**. Now you will learn how to write your own custom manipulators that don't use parameters. This task requires neither class derivation nor use of complex macros.

Suppose you have a printer that requires the sequence `<ESC> [` to enter boldface mode. You could, of course, insert these characters directly into the stream like this:

```
cout << "regular " << '\033' << '[' << "boldface" << endl;
```

A less tedious strategy is to define a manipulator `bold` that inserts the desired sequence. The output statement now becomes:

```
cout << "regular " << bold << "boldface" << endl;
```

All that's necessary to define the `bold` manipulator is the following function:

```
ostream& bold( ostream& os ) {
    return os << '\033' << '[';
}
```

The `bold` function is simply a globally defined function that takes an **ostream** reference argument and returns the **ostream** reference. It is not a member function or a friend because it doesn't need access to any private class elements. The `bold` function connects to the stream because the stream's << operator is overloaded to accept that type of function by a declaration that looks something like this:

```
ostream& ostream::operator<< ( ostream& (*_f)( ostream& ) ); {
    (*_f)( *this );
    return *this;
}
```

Indeed, you could use this C++ feature to extend other overloaded operators.

It is incidental that, in this case, `bold` inserts characters into the stream. The function could do anything at all, but you must remember that it is called when it is inserted *into* the stream, not necessarily when the adjacent characters are printed. Thus printing could be delayed because of the stream's buffering.

More Complex Manipulators

Chapter 19 describes the process of writing manipulators that take one or more arguments. This is a more difficult process that requires use of the macros contained in the IOMANIP.H header file.

18.3 Input Streams

An input stream object is a source of bytes. Refer to the class hierarchy diagram in the *Class Libraries Reference*. The three most important input stream classes are **istream**, **ifstream**, and **istrstream** as described below.

The istream Class

The **istream** class is best suited to sequential text-mode input. Objects of class **istream** can be configured for buffered or unbuffered operation. All the functionality of the base class, **ios**, is included in **istream**. See the *Class Libraries Reference* for details.

Rarely will you construct an object from class **istream**. You will generally use the predefined **cin** object (actually an object of class **istream_withassign**). Under certain circumstances, you may assign **cin** to other stream objects after program startup.

All the functionality of the base class, **ios**, is included in **istream**. For details, see Part 3 of the *Class Libraries Reference.*

The ifstream Class

The **ifstream** class supports disk file input. Construct an object of class **ifstream** if you need an input-only disk file. You can specify whether **ifstream** objects process binary or text-mode data.

If you specify a filename in the constructor, that file is automatically opened when the object is constructed. Otherwise you can use the open member function after invoking the **void**-argument constructor.

Many formatting options and member functions apply to **ifstream** objects. All the functionality of the base classes **ios** and **istream** is included in **ifstream**, and that includes the ability to open files in either buffered or unbufferd mode. See Part 3 of the *Class Libraries Reference* for details.

The istrstreamClass

The **istrstream** class supports input from in-memory strings in a manner similar to the C run-time library function **sscanf**. Construct an object of class **istrstream** when you need to extract data from an existing null-terminated character array (C string). You must have allocated and initialized your string before calling the constructor.

Constructing Input Stream Objects

If you use only **cin**, then you don't have to worry about input stream construction. If you use file streams or string streams, then constructors are important.

Input File Stream Constructors

If you need an input file stream, you have three choices:

- You can use the **void**-argument constructor, then call the **open** member function.

```
ifstream myFile; // On the stack
myFile.open( "filename", iosmode );

ifstream* pmyFile = new ifstream; // On the heap
pmyFile->open( "filename", iosmode );
```

- You can specify a filename and mode flags in the constructor invokation, thereby opening the file during the construction process.

```
ifstream myFile( "filename", iosmode );
```

- You can specify an integer file descriptor for a file that is already open for input. In this case you have the option to specify unbuffered input or a pointer to your own buffer.

```
int fd = _open( "filename", dosmode );
ifstream myFile1( fd );  // Buffered mode (default)
ifstream myFile2( fd, NULL, 0 );  // Unbuffered mode
ifstream myFile3( fd, pch, buflen );  // User-supplied buffer
```

For a description of mode flag usage, see the description of the **open** member function in "Output Streams" on page 365.

Input String Stream Constructors

The input string stream constructor requires the address of preallocated, preinitialized storage:

```
char s[] = "123.45";
double amt;
istrstream myString( s );
myString >> amt; // Amt should contain 123.45
```

Using Extraction Operators

The extraction operator (>>) is the most familiar means of getting bytes from an input stream object. This operator is preprogrammed for all the standard C++ data types, including integers, floating-point values, null-terminated strings, and so forth.

Formatted text input extraction operators depend on white space to separate incoming data values. This behavior is inconvenient when a text field contains multiple words or when numbers are separated by commas. One alternative is the use the unformatted input member function **getline** to read a block of text with white space included. The block can then be parsed with special-purpose functions. Another alternative is derivation of an input stream class with a member function such as `GetNextToken`. This special-purpose function could call **istream** member functions to extract and format character data.

Testing for Extraction Errors

It is very important to test for errors during the extraction process. Consider, for example, the statement:

```
cin >> n;
```

If `n` is a signed integer, then an input value of 33,000 (32,767 is the maximum allowed value) will set the stream's **fail** bit. If you don't deal with the error immediately, then **cin** will be unusable. All subsequent extractions result in an immedate return with no value stored.

Example 15

The error processing functions discussed under "Output Streams" apply also to input streams. Example 15 demonstrates how the **fail** and **clear** functions can be used with **cin**.

```
// exios115.cpp
#include <iostream.h>
int n[5], i;
void main()
{
    cout << "Enter 5 values, separated by spaces" << endl;
    for( i = 0; i < 5; i++ ) {
        cin >> n[i];
        if( cin.eof() || cin.bad() ) break; // Tests for end-of-file
                                            // or unrecoverable error
        if( cin.fail() ) { // Tests for format conversion error
            cin.clear(); // Clear stream's fail bit
            n[i] = 0;    // and continue processing
        }
    }
    for( i = 0; i < 5; i++ ) { // Prints the values just read
        cout << n[i];
    }
}
```

Input Stream Manipulators

Many manipulators, such as **setprecision**, are defined for the **ios** class and thus technically apply to input streams. Few manipulators, however, actually affect input stream objects. The most important are the manipulators **dec**, **oct**, and **hex**. These radix manipulators determine the conversion base used with numbers from the input stream.

On extraction, the **hex** manipulator enables the processing of a variety of input formats. For example, the sequences c, C, 0xc, 0xC, 0Xc, and 0XC are all interpreted as the decimal integer 12. Any character other than 0 through 9, A through F, a through f, x, and X terminates the numeric conversion. Thus the sequence "124n5" is converted to the number 124 with the **ios::fail** bit set in the process.

Input Stream Member Functions

The following member functions are particularly useful for disk input. See the *Class Libraries Reference* for a complete list of stream member functions.

The open Function

If you are using an input *file* stream (**ifstream**), then you must associate that stream with a specific disk file. You can make this association in the constructor, or you can use the **open** function. In either case, the parameters are the same.

You generally specify an **open_mode** when you open the file associated with an input stream (the default mode is **ios::in**). For a list of the **open_mode** flags, see the description of the **open** function in "Output Streams" on page 365. The flags can be combined as appropriate using the bitwise OR (|) operator.

If you want to read an existing file, you must test for the possibility that the file does not exist. For a description of the **fail** member function, see "Error Processing Functions" on page 377.

```
istream ifile( "FILENAME", ios::nocreate );
if ( ifile.fail() )
// The file does not exist ...
```

The get Function

The unformatted **get** member function works like the >> operator with two exceptions. First, the **get** function always includes white-space characters, whereas the extractor excludes white space when the **ios::skipws** flag is set (the default). Secondly, the **get** function is less likely to cause a tied output stream (**cout**, for example) to be flushed.

Example 16

Example 16 illustrates how the behavior of the extraction operator is different than that of the **get** member function.

```
// exios116.cpp
// The istream get member function
#include <iostream.h>

void main()
{
   char line[100], ch = 0, *cp;
```

```
    cout << " Type a line terminated by 'x'\n>";
    cp = line;
    while ( ch != 'x' )
    {
       cin >> ch;
       if( !cin.good() ) break; // Exits on EOF or failure
       *cp++ = ch;
    }
    *cp = '\0';
    cout << ' ' << line;
    cin.seekg( 0L, ios::end ); // Empties the input stream
    cout << "\n Type another one\n>";
    cp = line;
    ch = 0;
    while ( ch != 'x' )
    {
       cin.get( ch );
       if( !cin.good() ) break; // Exits on EOF or failure
       *cp++ = ch;
    }
    *cp = '\0';
    cout << ' ' << line;
}
```

The program reads two strings from the keyboard one character at a time. The first input operation uses the extraction operator, and the second one uses the **get** member function. If you type `Time to exit` for both entries, the screen would look like this:

```
Type a line terminated by 'x'
Time to exit <-------------------- Entered by you
Timetoex <---------------------- Echoed by the program
Type another one
Time to exit <-------------------- Entered by you
Time to ex <---------------------- Echoed by the program
```

You can see that the extraction operator skips over the white-space characters (because the **ios::skipws** flag is set by default) and the **get** function does not. The program needs the `x` terminator because it needs to know when to stop reading. Because **cin** is attached to the C run-time library **stdin** file, the program does not see any characters until you type the carriage return.

Example 17

A variation of the **get** function allows you to specify a buffer address and the maximum number of characters to read. This is useful for limiting the number of characters sent to a specific variable, as the following example code shows.

```
// exios117.cpp
// Using get with a buffer and length
#include <iostream.h>

void main()
{
   char line[25];
   cout << " Type a line terminated by carriage return\n>";
   cin.get( line, 25 );
   cout << ' ' << line;
}
```

In this particular example, you can type up to 24 characters (because strings must end with a null). If you type more than 24, the additional characters remain in the stream and are available for later extraction.

The getline Function

The **getline** function works in almost the same way as the **get** function. Both functions allow a third argument that specifies the terminating character for input. If you do not include that argument, its default value is the newline character.

The difference between the two is that **get** leaves the terminating character in the stream and **getline** removes the terminating character and throws it away. Both functions reserve one character for a terminating null (in C and C++, all strings end with a null).

Example 18

Example 18 uses the **getline** function with a third argument to specify a terminating character for the input stream:

```
// exios118.cpp
// The istream getline member function
#include <iostream.h>

void main()
{
   char line[100];
   cout << " Type a line terminated by 't'" << endl;
   cin.getline( line, 100, 't' );
   cout << line;
}
```

If you type `I like elephants.`, the program screen contains these lines:

```
Type a line terminated by 't'
I like elephants. <-------------- Entered by you
I like elephan <----------------- Echoed by the program
```

The program stops after reading the letter `t` and leaves the remaining characters, starting with `s`, in the stream. The next extraction operation or call to **get** will read the letter `s`.

If you had used the **get** member function instead of the getline function, the screen display would be the same, but the next character retrieved would be the letter `t`.

The read Function

The **read** member function reads a sequence of bytes from a file to a specified area of memory. The number of bytes read is determined by the specified length parameter. Reading will otherwise stop if the physical end of file is reached or, in the case of a text-mode file, if an embedded **EOF** character is read.

Example 19

Example 19 shows how to use the **read** function to read a binary record from a payroll file into a structure:

```
// exios119.cpp
// The istream read function
#include <fstream.h>
#include <fcntl.h>
#include <io.h>

void main()
{
   struct
   {
      double salary;
      char name[23];
   } employee;

   ifstream is( "payroll", ios::binary | ios::nocreate );
   if( is ) {  // ios::operator void*()
      is.read( (char *) &employee, sizeof( employee ) );
      cout << employee.name << ' ' << employee.salary << endl;
   }
   else {
      cout << "ERROR: Cannot open file 'payroll'." << endl;
   }
}
```

This program assumes that the data records are formatted *exactly* as specified by the structure with no terminating carriage return or linefeed characters. For a description of binary file processing, see "Binary Output Files" on page 378.

The seekg and tellg Functions

An input file stream keeps an internal pointer that corresponds to the position in the file where data will be read next. The **seekg** member function sets that pointer.

Example 20

Example 20 opens a file, changes the input position, and then reads to end of the file:

```
// exios120.cpp
// The seekg member function
#include <fstream.h>

void main()
{
   char ch;

   ifstream tfile( "payroll", ios::binary | ios::nocreate );
   if( tfile ) {
      tfile.seekg( 8 );      // Seek eight bytes in (past salary)
      while ( tfile.good() ) { // EOF or failure stops the reading
         tfile.get( ch );
         if( !ch ) break; // quit on null
         cout << ch;
      }
   }
   else {
      cout << "ERROR: Cannot open file 'payroll'." << endl;
   }
}
```

You can use the **seekg** function to implement record-oriented data management systems. Multiply the fixed-length record size by the record number to obtain the byte position relative to the end of the file, then use the two-argument **get** function to read the record.

Example 21

The **tellg** member function returns the current file position for reading, as illustrated by Example 21:

```
// exios121.cxx
// The tellg function
#include <fstream.h>

void main()
{
   char ch;
```

```
    ifstream tfile( "payroll", ios::binary | ios::nocreate );
    if( tfile ) {
        while ( tfile.good() ) {
           streampos here = tfile.tellg();
           tfile.get( ch );
           if ( ch == ' ' )
              cout << "\nPosition " << here << " is a space";
        }
    }
    else {
       cout << "ERROR: Cannot open file 'payroll'." << endl;
    }
}
```

The program reads the file built by an earlier example and displays messages that show the character positions of any spaces it finds. The **tellg** function returns a value of type **streampos**, which is a **typedef** defined in IOSTREAM.H.

The close Function

Use the **close** member function to close the disk file associated with an input file stream. The file should be closed to free the operating system file handle. The **ifstream** destructor closes the file for you (unless you called **attach** or passed your own file descriptor to the constructor), but you can use the **close** member function if you need to open another file for the same stream object.

Overloading the >> Operator for Your Own Classes

Input streams use the extraction (>>) operator for the standard types. You can write similar extraction operators for your own types, but, as with all extraction operations, success depends on the precise use of white space.

Below is an example of an extraction operator for the `Date` class that was presented in Example 14 on pages 380-381:

```
istream& operator>> ( istream& is, Date& dt )
{
   is >> dt.mo >> dt.da >> dt.yr;
   return is;
}
```

A more complete extraction operator might check the validity of the date that was entered.

18.4 Input/Output Streams

An **iostream** object is both a source and a destination for bytes. The two most important I/O stream classes, both derived from **iostream**, are **fstream** and **strstream**, as described below.

Note The **fstream** and **strstream** classes inherit the functionality of the **istream** and **ostream** classes described earlier.

The fstream Class

The **fstream** class supports disk file input/output. Construct an **fstream** object if you need an disk file that is to be both read from and written to in the same program.

The strstream Class

The **strstream** class supports input and output of in-memory strings. Construct an **strstream** object when you need to manipulate a string in memory using the iostream formatting facilities.

An Input/Output Stream Example

An object of type **fstream** is a single stream with two logical substreams, one for input and one for output. Although there are separately designated put and get positions in the underlying buffer, those positions are tied together.

If a file is not opened in append mode, changing the get position (with a **seekg** call) changes the put position (returned by a **tellp** call).

If a file is opened in append mode, write operations always occur at the end of the file. Each such write operation also changes the get position pointer to just past the last character in the file. A call to **seekp** also changes the get position.

Example 22

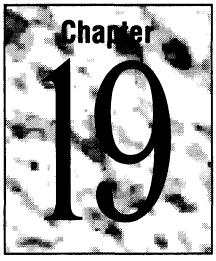
Example 22 reads the text file into a character array and writes an uppercase-only copy of the bytes at the end of the file:

```
// exios122.cpp
// An fstream file
#include <fstream.h>
#include <ctype.h>

void main()
{
   fstream tfile( "test.dat", ios::in | ios::app );
   char tdata[100];
   int i = 0;
   char ch;
```

```
    while ( i < 100 ) {
        tfile.get( ch );
        if ( tfile.istream::eof() ) break;
        tdata[i++] = ch;
    }
    tfile.istream::clear();
    for ( int j = 0; j < i; j++ ) {
        tfile.put( toupper( tdata[j] ) );
    }
}
```

The program must call the **clear** member function before starting the write operations because the stream is at the end of the file. The **clear** function resets the end-of-file flag and all other flags, which allows the program to proceed with the output.

Advanced iostream Programming



The fundamentals of iostream programming were presented Chapter 18. This chapter offers several advanced topics; in particular, parameterized manipulators and stream class derivation.

19.1 Custom Manipulators with Parameters

Chapter 18 introduced manipulators and described the process of creating your own manipulators without parameters on page 381. This section describes the steps necessary to create single-parameter and multi-parameter output stream manipulators. It also describes use of manipulators for streams other than output streams.

Output Stream Manipulators with One Parameter (int or long)

The Microsoft iostream Class Library provides a set of macros for creating parameterized manipulators. Manipulators with a single **int** or **long** parameter are a special case and will be discussed first.

Example 1

To create an output stream manipulator that accepts a single **int** or **long** parameter (like **setw**), you must use the **OMANIP** macro, defined in IOMANIP.H. Example 1 defines a `fillblank` manipulator that inserts a specified number of blanks into the stream:

```
// exios201.cpp
// A custom manipulator with an integer parameter
#include <iostream.h>
#include <iomanip.h>
```

```
ostream& fb( ostream& os, int l )
{
   for( int i=0; i < l; i++ )
        os << ' ';
   return os;
}

OMANIP(int) fillblank( int l )
{
   return OMANIP(int) ( fb, l );
}

void main()
{
    cout << "10 blanks follow" << fillblank( 10 ) << ".\n";
}
```

The IOMANIP.H header file contains a macro that expands **OMANIP(int)** into a class, **__OMANIP_int**. This class definition includes a constructor and an overloaded **ostream** insertion operator for an object of the class. The `fillblank` function, inserted in the stream, calls the **__OMANIP_int** constructor in order to return an object of class **__OMANIP_int**. Thus `fillblank` can be used with an **ostream** insertion operator. The constructor, in turn, calls your `fb` function.

Note The **OMANIP** macro represents an advanced use of C++. It will ultimately be superseded once C++ accommodates parameterized types. In the meantime, it is easier to adapt the code above than to analyze the syntax. The expression **OMANIP(long)** expands to another built-in class, **__OMANIP_long**, which accommodates functions with a long integer parameter.

Other One-Parameter Output Stream Manipulators

You can create manipulators that take arguments other than **int** and **long**. In addition to the **OMANIP** macro, you must use the **IOMANIPdeclare** macro, which declares the classes for your new type.

Example 2

Example 2 uses a class `money`, which is a thinly disguised **long** type. A new manipulator, `setpic`, attaches a formatting "picture" string to the class that can be used by the overloaded stream insertion operator of the class `money`. Note that the picture string is stored as a static variable in the `money` class rather than as data member of a stream class. This means you do not have to derive a new output stream class.

```
// exios202.cpp
// A custom manipulator with a char* parameter
#include <iostream.h>
```

```
#include <iomanip.h>
#include <string.h>

typedef char* charp;
IOMANIPdeclare( charp );

class money {
private:
    long value;
    static char *szCurrentPic;
public:
    money( long val ) { value = val; }
    friend ostream& operator << ( ostream& os, money m ) {
        // A more complete function would merge the picture
        // with the value rather than simply appending it
        os << m.value << '[' << money::szCurrentPic << ']';
        return os;
    }
    friend ostream& setpic( ostream& os, char* szPic ) {
        money::szCurrentPic = new char[strlen( szPic ) + 1];
        strcpy( money::szCurrentPic, szPic );
        return os;
    }
};
char *money::szCurrentPic;  // Static pointer to picture

OMANIP(charp) setpic(charp c)
{
    return OMANIP(charp) (setpic, c);
}

void main()
{
    money amt = 35235.22;
    cout << setiosflags( ios::fixed );
    cout << setpic( "###,###,###.##" ) << "amount = " << amt << endl;
}
```

Output Stream Manipulators with More Than One Parameter

Example 3 shows you how to write a manipulator that takes two arguments. As you can see, it is similar to the previous example except that the character pointer type declaration is replaced by a structure declaration.

Example 3

The following program illustrates the definition of the `fill` manipulator, which inserts a specified quantity of a particular character:

```
// exios203.cpp
// 2-argument manipulator example
```

```
#include <iostream.h>
#include <iomanip.h>

struct fillpair {
       char ch;
       int  cch;
};

IOMANIPdeclare( fillpair );

ostream& fp( ostream& os, fillpair pair )
{
    for ( int c = 0; c < pair.cch; c++ ) {
        os << pair.ch;
    }
    return os;
}

OMANIP(fillpair) fill( char ch, int cch )
{
    fillpair pair;

    pair.cch = cch;
    pair.ch  = ch;
    return OMANIP (fillpair)( fp, pair );
}


void main()
{
    cout << "10 dots coming" << fill( '.', 10 ) << "done" << endl;
}
```

Example 3 could be easily rewritten with the manipulator definition in a separate program file. The header file must contain the neccesary declarations as follows:

```
struct fillpair {
       char ch;
       int  cch;
};
IOMANIPdeclare( fillpair );
ostream& fp( ostream& o, fillpair pair );
OMANIP(fillpair) fill( char ch, int cch );
```

Custom Manipulators for Input Streams and I/O Streams

The **OMANIP** macro works with **ostream** and its derived classes. The **SMANIP**, **IMANIP**, and **IOMANIP** macros work with the classes **ios**, **istream**, and **iostream**, respectively.

Using Manipulators with Derived Stream Classes

If you define a custom manipulator that works with, say, the **ostream** class, it will work with all classes derived from **ostream**. If, however, you need manipulators that work *only* with a derived class `xstream`, then you must add the overloaded insertion operator, shown below, which is *not* a member of the class.

```
xstream& operator<< ( xstream& xs, xstream& (*_f)( xstream& ) ) {
    (*_f)( xs );
    return xs;
}
```

Now the manipulator code looks like this:

```
xstream& bold( xstream& xs ) {
    return xs << '\033' << '[';
}
```

If the manipulator needs to access `xstream` protected data members functions functions, you can declare the `bold` function as a **friend** of the `xstream` class.

19.2 Deriving Your Own Stream Classes

This section is intended for C++ programmers who are experienced in deriving classes.

A Straightforward Stream Class Derivation

Like any C++ class, a stream class can be derived for the simple purpose of adding new member functions, data members, or manipulators. If you need an input file stream that tokenizes its input data, for example, you can derive from the **ifstream** class. This new derived class can include a member function that returns the next token by calling its base class's public member functions or extractors. You might need some new data members to hold the stream object's state between operations, but you probably will not need to use the base class's protected member functions or data members.

For the straightforward stream class deriviation, you need only write the necessary constructors and the new member functions.

The class derivation example that follows is more complex because it exploits the relationship between the stream class and the associated stream buffer class.

The streambuf Class

You have probably noticed some references to a **streambuf** class. This class is unimportant unless you plan major customizations of the iostream library. As you explore the iostream classes, you will learn that **streambuf** really does most of the work for the other stream classes. You will create a modified output stream by deriving *only* a new **streambuf** class and connecting it to the standard **ostream** class.

Why Derive a Custom streambuf Class?

The existing output streams communicate to the file system and to in-memory strings. You can invent your own streams that address a memory-mapped video display, a window as defined by Microsoft Windows, some new physical device, and so forth. A simpler goal is the alteration of the byte stream as it goes to a file system device. This customization technique, as you will see, is sufficient to produce an object-oriented interface to laser printers such as the Hewlett-Packard LaserJet series.

A streambuf Derivation Example

Example 4 shows you how to modify the **cout** object such that prints in two-column landscape (horizontal) mode on a Hewlett-Packard LaserJet printer or other printer that uses the PCL control language. As the test driver program shows, all the member functions and manipulators that work with the original **cout** also work with the special version. The application programming interface is the same!

Example 4

The example is divided into three source files.

- HSTREAM.H—the LaserJet class declaration that must be included in both the implementation file and your application file
- HSTREAM.CPP—the LaserJet class implementation that must be linked with your application
- EXIOS204.CPP—the test driver program that sends output to a LaserJet printer

The HSTREAM.H file contains only the class declaration for `hstreambuf`. This class is derived from **filebuf** and overrides the appropriate **filebuf** virtual functions.

```
// htream.h - HP LaserJet output stream header
#include <fstream.h> // Accesses 'filebuf' class
#include <string.h>
#include <stdio.h> // for sprintf
```

```
class hstreambuf : public filebuf
{
public:
    hstreambuf( int filed );
    virtual int sync();
    virtual int overflow( int ch );
    ~hstreambuf();
private:
    int column, line, page;
    char* buffer;
    void convert( long cnt );
    void newline( char*& pd, int& jj );
    void heading( char*& pd, int& jj );
    void pstring( char* ph, char*& pd, int& jj );
};
ostream& und( ostream& os );
ostream& reg( ostream& os );
```

HSTREAM.CPP contains the the `hstreambuf` class implementation.

```
// hstream.cpp  - HP LaserJet output stream
#include "hstream.h"

#define REG 0x01    // Regular font code
#define UND 0x02    // Underline font code
#define CR 0x0d     // Carriage return character
#define NL 0x0a     // Newline character
#define FF 0x0c     // Formfeed character
#define TAB 0x09    // Tab character

#define LPP 57      // Lines per Page
#define TABW 5      // Tab width

// Prolog defines printer initialization (font, orientation, etc.
char prolog[] =
{ 0x1B, 0x45,                                   // Reset printer
  0x1B, 0x28, 0x31, 0x30, 0x55,          // IBM PC char set
  0x1B, 0x26, 0x6C, 0x31, 0x4F,          // Landscape
  0x1B, 0x26, 0x6C, 0x38, 0x44,               // 8 lines-per-inch
  0x1B, 0x26, 0x6B, 0x32, 0x53};               // Lineprinter font

// Epilog prints the final page and terminates the output
char epilog[] = { 0x0C, 0x1B, 0x45 };   // Formfeed, reset

char uon[] = { 0x1B, 0x26, 0x64, 0x44, 0 }; // Underline on
char uoff[] = { 0x1B, 0x26, 0x64, 0x40, 0 };//  Underline off

hstreambuf::hstreambuf( int filed ) : filebuf( filed )
{
    column = line = page = 0;
    int size = sizeof( prolog );
    setp( prolog, prolog + size );
```

```
    pbump( size );   // Puts the prolog in the put area
    filebuf::sync(); // Sends the prolog to the output file
    buffer = new char[1024]; // Allocates destination buffer
}

hstreambuf::~hstreambuf()
{
    sync(); // Makes sure the current buffer is empty
    delete buffer; // Free the memory
    int size = sizeof( epilog );
    setp( epilog, epilog + size );
    pbump( size );   // Puts the epilog in the put area
    filebuf::sync(); // Sends the epilog to the output file
}

virtual int hstreambuf::sync()
{
    long count = out_waiting();
    if ( count ) {
        convert( count );
    }
    return filebuf::sync();
}

virtual int hstreambuf::overflow( int ch )
{
    long count = out_waiting();
    if ( count ) {
        convert( count );
    }
    return filebuf::overflow( ch );
}
//*** The following code is specific to the HP LaserJet printer ***

// Converts a buffer to HP, then writes it
void hstreambuf::convert( long cnt )
{
    char *bufs, *bufd; // Source, destination pointers
    int j = 0;

    bufs = pbase();
    bufd = buffer;
    if( page == 0 ) {
        newline( bufd, j );
    }
    for( int i = 0; i < cnt; i++ ) {
        char c = *( bufs++ );  // Gets character from source buffer
        if( c >= ' ' ) {     // Character is printable
            * ( bufd++ ) = c;
            j++;
            column++;
        }
```

```
        else if( c == NL ) { // Moves down one line
            *( bufd++ ) = c;   // Passes character through
            j++;
            line++;
            newline( bufd, j ); // Checks for page break, etc.
        }
        else if( c == FF ) {    // Ejects paper on formfeed
            line = line - line % LPP + LPP;
            newline( bufd, j ); // Checks for page break, etc.
        }
        else if( c == TAB ) {   // Expands tabs
            do {
                *( bufd++ ) = ' ';
                j++;
                column++;
            } while ( column % TABW );
        }
        else if( c == UND ) { // Responds to 'und' manipulator
            pstring( uon, bufd, j );
        }
        else if( c == REG ) { // Responds to 'reg' manipulator
            pstring( uoff, bufd, j );  //
        }
    }
    setp( buffer, buffer + 1024 ); // Sets new put area
    pbump( j ); // Indicates the number of characters in the dest buffer
}

// simple manipulators - apply to all ostream classes
ostream& und( ostream& os ) // Turns on underscore mode
{
    os << (char) UND; return os;
}

ostream& reg( ostream& os ) // Turns off underscore mode
{
    os << (char) REG; return os;
}

void hstreambuf::newline( char*& pd, int& jj )
// Called for each newline character
    column = 0;
    if ( ( line % ( LPP*2 ) ) == 0 ) // Even page
    {
    page++;
    pstring( "\033&a+0L", pd, jj );  //  Set left margin to zero
    heading( pd, jj );      /*  print heading  */
    pstring( "\033*p0x77Y", pd, jj ); // Cursor to (0,77) dots
    }
```

```
    if ( ( ( line % LPP ) == 0 ) && ( line % ( LPP*2 ) ) != 0 ) // Odd page
    { //  prepare to move to right column
    page++;
    pstring( "\033*p0x77Y", pd, jj ); // Cursor to (0,77) dots
    pstring( "\033&a+88L", pd, jj ); // Left margin to 88th column
    }
    }
void hstreambuf::heading( char*& pd, int& jj ) // Prints page heading
{
        char hdg[20];
        int i;

        if( page > 1 ) {
            *( pd++ ) = FF;
            jj++;
        }
        pstring( "\033*p0x0Y", pd, jj ); // Top of page
        pstring( uon, pd, jj ); // Underline on
        sprintf( hdg, "Page %-3d", page );
        pstring( hdg, pd, jj );
        for( i=0; i < 80; i++ ) { // Pads with blanks
            *( pd++ ) = ' ';
            jj++;
        }
        sprintf( hdg, "Page %-3d", page+1 ) ;
        pstring( hdg, pd, jj );
        for( i=0; i < 80; i++ ) { // Pads with blanks
            *( pd++ ) = ' ';
            jj++;
        }
        pstring( uoff, pd, jj ); // Underline off
}
// Outputs a string to the buffer
void hstreambuf::pstring( char* ph, char*& pd, int& jj )
{
    int len = strlen( ph );
    strncpy( pd, ph, len );
    pd += len;
    jj += len;
}
```

EXIOS204.CPP is the test driver program. It reads text lines from **cin** and writes them to the modified **cout**.

```
// exios204.cpp
// hstream Driver program copies 'cin' to 'cout' until end-of-file
#include "hstream.h"

hstreambuf hsb( 1 ); // 1=stdout

void main()
{
```

```
    char line[200];
    cout = &hsb; // Associates the HP LaserJet streambuf to cout
    while( 1 ) {
       cin.getline( line, 200 );
       if( !cin.good() ) break;
       cout << line << endl;
    }
}
```

Here are the main points in the code above:

- The new class `hstreambuf` is derived from **filebuf**, the buffer class for disk file I/O. The **filebuf** class does the actual writing to disk in response to commands from its associated **ostream** class. The **hstreambuf** constructor takes an argument that corresponds to the operating system file number, in this case 1 for **sdtout**. This constructor is invoked by the following line:

  ```
  hstreambuf hsb( 1 );
  ```

- The `hstreambuf` object is associated with **cout** by the **ostream_withassign** assignment operator:

  ```
  ostream& operator =( streambuf* sbp );
  ```

  The following statement in EXIOS204.CPP executes the assignment.

  ```
  cout = &hsb;
  ```

- The `hstreambuf` constructor prints the prolog that sets up the laser printer, and it allocates a temporary print buffer.
- The destructor outputs the epilog text and frees the print buffer when the object goes out of scope. In this example, the object goes out of scope *after* the exit from **main**.
- The **streambuf** virtual **overflow** and **sync** functions do the low-level output. The `hstreambuf` class overrides these functions in order to gain control of the byte stream. The functions call the private `convert` member function.
- The `convert` function processes the characters inside the **hstreambuf** buffer and stores them in the object's temporary buffer. The tempory buffer is then processed by the **filebuf** functions.
- The details of `convert` relate more to the PCL language than to the iostream library. Note that there are private data members that keep track of column number, line number and page number.
- The `und` and `reg` manipulators control the underscore print attribute. They simply insert codes 0x02 and 0x03 into the stream. These codes are later translated into the printer-specific sequences by `convert`.

- Example 4 is a useful program now, but you can easily extend it to embellish the heading, add more formatting features, and so forth.
- In a more general example, the `hstreambuf` class would have been derived from **streambuf** rather than from **filebuf**. The **filebuf** derivation gets the most leverage from existing iostream library code, but it makes certain assumptions about the implementation of **filebuf**, particularly the **overflow** and **sync** functions. Thus you could not necessarily expect the example to work with other derived **streambuf** classes or with the **filebuf** classes provided by other software publishers.

Index

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399

Microsoft®

1191 Part No. 28113