

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[**Bookmark It**](#)

Search this book:

**BROWSE**
BY TOPIC

Introduction

Learn the MFC C++ Classes

Acknowledgment

Chapter 1—Windows and MFC

Windows Operating Systems and MFC

C++ Compilers and MFC

Windows

User Inputs to a Window

Messages

MFC and Windows OS Interaction

The Structure of an MFC Application

Creating a Main Window Using MFC

The CFrameWnd::Create() Function

Registering a New Window Class

Resource Files

Customized Icon and Cursor Resources

Summary

Exercise

Chapter 2—Menus and Message Maps

An Example With a Simple Menu

Message Maps

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

Menus

Accelerators

Handler Functions

Setting the Timer

Displaying a Message Box

Adding Message Map Entries With Compiler Tools

An Example That Changes Menus and Submenus

Using CMenu Objects

CWnd Functions for Messages

Summary

Exercise

Chapter 3—Graphics and Text Drawing

The Graphics Device Interface (GDI)

The Device Context

GDI Objects

Device Context Settings

Stock Drawing Objects

The CDC Class

The Device Context Classes

An Example That Draws Text and Shapes

An Example That Sets the Viewport Origin

How a Screen Repaints Itself

Creating a Pen

Creating a Brush

The RGB Macro

The Raster Drawing Mode and SetROP2()

A Graphics Editor Example

C++ Objects for the Rectangle and Ellipse

The Graphics Output Process

Deleting Drawing Objects

Drawing the Rectangles and Ellipses

The OnPaint() Function

Maintaining the Update Region

The Background Color

The Handy Utility Classes: CRect, CPoint, and CSize

[Using a Private Device Context](#)

[An Example With a Private Device Context](#)

[Summary](#)

[Exercise](#)

Chapter 4—Fast Drawing and Bitmap Graphics

[Using Exclusive-or and Exclusive-nor for Fast Redraws](#)

[Details of the Exclusive-or \(Exclusive-nor\) Process](#)

[Limitations of the Exclusive-or \(Exclusive-nor\) Process](#)

[Using Backing Store for Fast Redraws](#)

[Bitmaps](#)

[Using a Memory Device Context](#)

[The CDC::BitBlt\(\) Function](#)

[Using Bitmap Graphics for Animation](#)

[The Message Handler OnCreate\(\)](#)

[The Message Handler OnTimer\(\)](#)

[Device Independent Bitmaps \(DIBs\)](#)

[Palettes](#)

[The System Palette](#)

[Loading and Using the System Palette](#)

[Displaying a DIB Using the System Palette](#)

[Summary](#)

[Exercise](#)

Chapter 5—Child Windows

[A Child Window](#)

[The CWnd::Create\(\)Function](#)

[Message Maps for Child Windows](#)

[User Messages](#)

[A Popup Window](#)

[A Fixed Child Window](#)

[Summary](#)

Chapter 6—Dialogs, Common Dialogs, and Button Controls

Dialogs

Modal vs. Modeless Dialog Boxes

Common Dialogs

Class CFileDialog

Class CPrintDialog

Class CPageSetupDialog

Class CFindReplaceDialog

Class CFontDialog

Class CColorDialog

Using the ChooseColor Common Dialog

Designing Dialog Boxes

Overview of Common Controls

Window Styles for Win3.1 Common Controls

Button Controls

Static Controls

Placing Controls on the Mainframe Window

Messages To and From Button Controls

Messages From the Button Control

Messages To the Button Control

Example Program

Programming the Buttons Example

Generating the Main Window's Code

Generating the Dialog Box Code

The Buttons Program Listing

Discussion of the Buttons Program

Summary

Chapter 7—List Box, Combo Box, and Edit Controls and Data Transfer Functions

Overview of List Box, Combo Box, and Edit Controls

Edit Control Styles

List Box Styles

Combo Box Styles

Operations and Messages for Win3.1 Common Controls

An Example Program

Programming the UsrInput Example

- [Generating the Main Window's Code](#)
- [Generating the Dialog Box Code](#)
- [The UsrInput Program Listing](#)
- [Discussion of the UsrInput Program](#)
- [Data Transfer](#)
 - [Do Data Exchange Functions](#)
 - [Dialog Data Validation \(DDV\) Functions](#)
- [CString Features](#)
- [String Tables and Internationalization](#)
- [Summary](#)

Chapter 8—Communication Between Parent and Child for Modal and Modeless Dialogs

- [The Modal Dialog Example](#)
 - [The MFC Class CCmdUI](#)
 - [The ModalCom Program Listing](#)
 - [Data Transfer](#)
- [Modeless Dialog Example](#)
 - [Modeless Dialog Creation](#)
 - [User Messages](#)
 - [Modeless Program Listing](#)
 - [Data Updating](#)
 - [Special Virtual Functions](#)
- [Summary](#)
- [Exercise](#)

Chapter 9—The Document-View Architecture

- [The Structure of the Four Classes](#)
- [Message Routing](#)
- [An Example Document-View Program](#)
- [The PreCreateWindow\(\) Function](#)
- [Customizing the Mainframe Window](#)
 - [Overriding the CFrameWnd::PreCreateWindow\(\) Function](#)
 - [Mainframe Resources](#)
- [Customizing the View Window](#)

[The OnDraw\(\) Function](#)

[Message Maps](#)

[The Custom Program Listing](#)

[The Document Template](#)

[The RUNTIME_CLASS Macro](#)

[The CView Class](#)

[Views Based on a Dialog Template](#)

[Views Based on a Control](#)

[Summary](#)

[Chapter 10—Document-View Applications With Filing and Printing](#)

[Creating an AppWizard Project](#)

[Designing the Application's Data](#)

[Designing the User Interface](#)

[The Application's Menu](#)

[Printing the View](#)

[The Function OnPrepareDC\(\)](#)

[Mapping Modes](#)

[Functions for Printing](#)

[Print Preview and Print Setup](#)

[Data Persistence—Filing the Document's Data](#)

[Serialization and CArchive](#)

[The OnNewDocument\(\) Function](#)

[Multiple Views of the Document](#)

[ElipsMin Program with Minimum Code](#)

[ElipsMin Program Listing](#)

[Discussion of the “ElipsMin” Program](#)

[Making the Dialog Box Modeless](#)

[Diagnostic Code](#)

[Summary](#)

[Chapter 11—More About Splitter Windows and Filing](#)

[The Starter Application](#)

Multiple View Classes

Static Splitter Windows

Collection Classes

Array Collections

List Collections

Map Collections

Designing the Document's Data

Coding the Document Class

CByteArray Member Functions

Designing the View of the Rules

Using Logical Fonts

Text Metrics

Coding the View Class Containing the Rules

Designing the View of the Game

Drawing The Tic Tac Toe Board

Drawing the Moves

Summary

Chapter 12—An MDI Application

An MDI Application

Class Structure

Characteristics

Creating New Views

Creating New Documents

Multiple Menus

Keyboard Accelerators

The CFormView Class

Creating the “Form” Program

The “Form” Starter Application

The Multiple Document Template

The Trace Macro

The Document Class

Code Additions to the Document Class

The View Class

Code Additions to the View Class

Running the “Form” Program in Debug Mode

The “FormMin” Program With Minimum Code

“FormMin” Program Listing

Discussion of the “FormMin” Program

Summary

Chapter 13—Toolbars and Status Bars

The Bars Example

Creating the Bars Starter Application

Designing the Document Class

Designing the View Class

Drawing

Adding Scrolling

Customizing the Status Bar

Adding the Handler Functions

Customizing the Toolbar

Visual C++ 4 Toolbar Editing

Visual C++ 1.5 Toolbar Editing

Using Two Document Templates

Adding a Dynamic Splitter To An MDI

Adding a Document Template

Resources for Document Templates

Summary

Exercise

Chapter 14—Custom Controls, New Common Controls, and Property Sheets

Custom Controls

The CustCtrl Example

Building the “CustCtrl” Program

The New Common Controls

Creating New Common Controls

The NewCmnCtrls Example

New Common Control Styles

Building the NewCmnCtrls Program

Getting the AppWizard Starter Application

[Add the Menu Item](#)

[Create the Dialog Template](#)

[Creating the Dialog Class](#)

[Notification Messages](#)

[Property Sheets](#)

[The PropertySheet Example](#)

[Creating a Modal Property Sheet](#)

[Building the “PropertySheet” Program](#)

[Using the Apply Button](#)

[Summary](#)

[Appendix A](#)

[Appendix B](#)

[Appendix C](#)

[Appendix D](#)

[Appendix E](#)

[Appendix F](#)

[Appendix G](#)

[Index](#)

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

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

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

Keyword

☒ Brief ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

[Table of Contents](#)

Introduction

It is not necessary to learn the Windows API in order to begin using the Microsoft Foundation Class (MFC) Library of C++ classes; you do not even have to know more than a few basic C++ concepts in order to begin. I have been teaching extension classes through the University of California, Berkeley, since 1993. My students have ranged from expert to beginning Windows programmers. As an MFC teacher, I had to design a system that addressed the “common denominator” in the class, since so many of my students were put onto MFC projects and expected to “ramp up” in a matter of weeks, despite whatever their previous experience may have been. With the help of input from my students, I developed my simple one-idea-at-a-time approach to the MFC Library. I start with the most fundamental concept that a student needs to know and show the student how to master that one concept only. Then, I add the next concept. When exposed to a steady progression of clear ideas and exercises, my students have been able to truly master the fundamental concepts of MFC. This approach has worked for them, and I am sure that it will work for you.

I adapted this book from my lengthy course notes, updating it for the new MFC compilers. But I use the same approach and the same examples. All of the examples have been student-tested for clarity and effectiveness in getting each concept across. All potential misunderstandings and errors have already been caught by my students and corrected. You can type in what you see and follow it line by line, because it will run on either Windows NT, Win3.1x, or Win95. Additionally, I have done my best to ensure that no code was “corrupted” during the production of this book and have included, as a backup (and so that you do not need to retype it), two diskettes with the original, typo-free code. My experience as a teacher has taught me that if you show a programmer effective, error-free code, the code explains itself; therefore, I

have made sure that the original, tested code that was exposed to the scrutiny of my many students is included with this book.

In addition to the examples given in this book, a comprehensive exercise is given—a checkers game—which you code and progressively add more features to as you complete subsequent chapters in this book. This checkers game is discussed in Appendix B as well as in the exercises that are assigned at the end of specific chapters. An example solution for each of the assignments is given on one of the included diskettes.

Who Could Benefit from this Book

This book works well for the beginning or expert programmer who wishes to start programming a Windows application using MFC. A working knowledge of C is required. Because it is helpful to know some C++, Appendix C illustrates all the C++ concepts that you need to know for the examples in this book. I recommend that you review this appendix before delving into this book.

If you are a beginner, you will start at the beginning with a simple program that creates a window. With each additional chapter, you learn how to add new features. Small example programs are used for the first half of the book. For these small example programs, you generate all of the code yourself; you do not need to use compiler tools, such as AppWizard, to generate starter code. In later chapters, you migrate to using a compiler tool that generates starter code and learn how to add your application's code to the starter code.

If you already have some familiarity with MFC, this book will help you to fill in “gaps,” such as learning how to code without using a compiler tool to generate starter code. If you have been coding with the tools, this book will help you to understand the “bones” of the code, without the extraneous lines of tool-generated code that can often obscure the logic of a program. Once you know the bones, you can enjoy a greater understanding and confidence in what you can add and delete in order to make the application your own.

Compiler and Operating System Requirements

The emphasis of this book is on how to use the MFC classes. It does not emphasize the compiler tools, although it covers how to use the tools and then discusses the code that they generate. MFC is hosted by all major C++ compilers, any of which can be used with this book. An appendix is devoted to each C++ compiler that uses MFC and covers the basics of how to use that compiler for this book.

Compiler information is relegated to the appendixes so that it does not interfere with the unfolding of the MFC concepts. I have tunneled down “technical rat-holes” with a single student, trying to troubleshoot specific compiler problems, to the consternation and annoyance of my other students who do not have these problems; there are a number of configurations that host MFC, and they are all idiosyncratic. Because of this, I focus on the fundamentals that apply to the “common denominator” and cover most compiler information in the appendixes.

Your Windows operating system can be Win3.1x, Win95, or NT. The examples given in this book are coded such that they work on any of these operating systems. An exception to this rule is the code for the toolbar in Chapter 13, which differs for MFC 2.5 and for the newer MFC 4.0. Also, the final examples of Chapter 14 illustrate new capabilities added with MFC 4.0 and can be run only on Win95 and NT operating systems. Again, the code for each of the examples in this book is on the included diskettes so you can run it on your system.

Note: The files on the companion diskettes located at the back of this book must be installed on your hard disk. The README file on Disk 1 describes the companion files and their use. For complete installation instructions, see the last page of this book.

How this Book Is Organized

The progression of topics covered in this book starts from the first chapter in which you learn how to write an application that creates a simple window and proceeds to the final chapters in which you write a program that is an MDI application with multiple documents. This MDI application has the following features: a customized toolbar with specialized bitmap buttons, a customized status bar, multiple document templates, windows with or without splitter bars, and the ability to store a document to file and print out a document.

Each new chapter is a lesson that adds new concepts. The first chapter is a lesson on the basics of a window and how to create one. The second chapter is a lesson on menus and message maps, and you learn how to add these new features to the basic window. The third chapter is a lesson on how to draw graphical images; the fourth covers how to move the graphical images using bitmaps and fast drawing techniques. The next chapter covers the basics of child windows so that you have the fundamental understanding to proceed onto dialog boxes and controls which are child windows. Then, you learn how to include dialog boxes and controls in your applications. At this juncture you have learned the fundamentals needed for most applications, and you are ready to proceed on to more complex applications.

Beginning with Chapter 9, you learn how to use the document-view architecture; you can use a compiler tool to generate starter code or you can continue to generate all of your own code. All of the fundamentals that you have mastered in the first eight chapters will be used as you continue to create applications using the document-view architecture. You learn to store documents to files on your hard drive and to retrieve them from the files, how to print a document, and how to create splitter windows either statically or dynamically. You add toolbars and status bars to your application and customize them, and you learn how to customize controls. In the final sections of the last chapter, you learn the new common controls and property sheets introduced with MFC 4.0

When you have completed all of the chapters and the accompanying exercises, you will not only have a greater understanding of MFC, but you will also have created a fully functioning checkers game. I strongly urge you to do the exercises that create this game; my students have told me that the actual

completion of this game, more than anything, helped them to feel that they truly had mastered the MFC Library by understanding the logical “bones.”

Table of Contents

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

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

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

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

[Table of Contents](#)[BROWSE
BY TOPIC](#)

Learn the MFC C++ Classes

Companion Diskette Information

The code for each of the examples given in the book is on the included disks. In addition to the examples given in the book, a checkers game exercise is given; you code the exercise, progressively adding features as you complete chapters in the book. An example solution for each of the checker game assignments is also included. The examples for each chapter are in each chapter's directory, named CHAP1 through CHAPTR14. The example solutions for the checker game exercises are in the directory named EXERCISE. Five megabytes of hard disk space is required for the decompressed files to be stored on your system.

For each example, the executable file as well as the source code files required for you to reconstruct your own project are on the included disks. The executable files have been compiled on a 16-bit system, and thus can be run on any of the operating systems—Win3.1, Win95, or NT. The exceptions are the last two examples of Chapter 14, which have been compiled on a 32-bit system; these two examples can only be compiled and run on Win95 or NT.

Companion Diskette Installation

The installation process creates an /MFC directory structure on your hard drive and then copies all necessary companion files. A README file is available on Disk 1 and is also copied to your hard drive. This file describes the companion files and their use. Install the files as follows:

1. Insert Disk 1 of 2 in your floppy drive and log that drive.
2. Type **INSTALL A C** and press **Enter**.
3. When prompted, remove Disk 1 and insert Disk 2.

4. Type **INSTALL A C** and press **Enter** again to install the remaining files from Disk 2.

When the completion message is displayed, removed Disk 2. All companion files are ready for use.

Table of Contents

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

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

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

Keyword

☒ Brief ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

[Table of Contents](#)

Acknowledgements

To my father, who taught me how to have a sense of humor.

To my children, who taught me patience and endurance.

And to my students, who taught me how to write this book.

[Table of Contents](#)

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Chapter 1

Windows and MFC

The *Microsoft Foundation Class (MFC) Library* is a collection of C++ classes. MFC is used with the *Microsoft Windows operating system* (Windows OS, also Windows). In this chapter, the Windows OS, its fundamental concepts, such as windows, user inputs, and messages, and its interaction with an MFC application are described. The last portion of this chapter presents five MFC programs that demonstrate the creation of windows with varying styles.

The MFC base classes are contained in a C++ class library developed by Microsoft, which is now supplied with many C++ compilers; it is typically stored in the directory C:\MSVC\MFC. It is provided as a *Dynamic Link Library* (DLL) so your application has access to the classes in MFC. A DLL consists of executable functions that are loaded into memory and are independent from any application. Libraries such as MFC are called *application frameworks*, because they give the user a framework for an application. The MFC classes have been built using the operating system's *Application Programming Interface* (API) functions. The API is the original Windows OS library of functions, coded in C, that subsequent libraries were built from, such as MFC. MFC provides easier-to-use functionality that incorporates the API functions. Using the MFC classes means that much of the programming has already been done for you, and you need only add special features to the MFC code to create your application.

This book describes the most frequently used portions of the MFC library. Along the way, it describes API functions that are needed for the example applications. To use the MFC framework, your application must be written in C++. Applications written in MFC can call (evoke) API functions, if needed. Traditional Windows OS applications, which do not use MFC or any other

framework, use API functions; these applications are usually written in C or Pascal. However, MFC provides so much ease of programming that most new applications are being written in C++ so that they can directly access the MFC library and derive functionality from it. Additionally, MFC has gained such notable market acceptance that most C++ compilers now support it.

Windows Operating Systems and MFC

MFC is designed to work with all the available Windows OSs. There are three Windows OSs available in the marketplace today: the relatively new Windows 95 (Win95) and Windows NT (NT), and their predecessor of long standing, Windows 3.1x (Win3.1x). MFC applications can be built and run on any of these operating systems.

An MFC application built on Win3.1x can be run on Win3.1x, Win95, and NT. An MFC application built on Win95 or NT can be run on either Win95 or NT, but not on Win3.1x unless it can be recompiled on Win3.1x. Most example applications in this book can be built on any of the three Windows OSs. The examples in Chapters One through Thirteen and the first section covering custom controls of Chapter Fourteen can be built on Win3.1x, Win95, or NT. The final sections of Chapter Fourteen contain examples that can be built and run on Win95 or NT only; these final examples cannot be built or run on Win3.1x.

Win3.1x has been in the marketplace for a long time and, as of this writing, is the most prevalent operating system. MFC versions 1.0 through version 2.5 were developed and used with Win3.1x. Win3.1x was designed for the then-existing hardware at the time of its initial release, and was built to be compatible with computers that have a 16-bit word; the Win3.1x operating system's design is limited to a 16-bit word. Hence, Win3.1x only runs applications compiled on Win3.1x.

Win95 and NT are the newer versions of the Windows operating system. MFC versions 3.5x and above are used with NT and Win95. They are designed for later processors that have features that fully protect applications and the operating system. Win95 and NT run on computers that have a 32-bit word. They are "forward compatible" and will run 16-bit applications compiled on Win3.1x.

Microsoft uses two terms to describe these 32-bit operating systems. Often you see the programming environment referred to as Win32 rather than NT or Win95, because Win32 is the 32-bit API for the NT and Win95 operating systems; it is the part of NT and Win95 that only programmers see.

Win95, introduced in 1995, is the most recently developed operating system and has a new "look and feel" to its interface elements. The controls have a "3-D look" and the windows have slightly different features, which are discussed later in this chapter. However, the internal aspects of the operating system remain the same as those used in Win3.1x. Programming in Win95 is not affected by the cosmetic differences of the interface elements, because the API functions work identically to their equivalents in Win3.1x.

NT is the superior operating system; it is a high-end operating system, providing networking and *multithreading*, which means that more than one “thread” of processing is occurring at one time. Older versions of NT will have the older Win3.1x “look and feel” for their interface elements. Microsoft has issued a *shell*, an interface that links to the older NT operating system and provides cosmetically different displays. This shell can be used with the older versions of NT to provide the Win95-style interface elements; however, the newer versions of NT incorporate the Win95-style interface elements. Although NT looks and acts just like Win95, the underlying structure of the environment is somewhat different. Despite these underlying differences, Win95 and NT are very similar to use for both users and programmers. Microsoft went to great pains to make NT look and behave as much like Win95 as possible. The operating systems are so similar that the term “Windows OS” refers to both systems.

Previous	Table of Contents	Next

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

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

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

Keyword

☒ Brief
 ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Win95 and Win3.1x use DOS for disk file services. The NT environment differs from Win95 and Win3.1x in that it does not use the MS-DOS file system for file access. NT has its own built-in file functions. This does not affect MFC programming because the API functions work identically to their equivalents in Win95 and Win3.1x. NT is basically a rewritten version of Win95, but NT is not a DOS application like Win95 and Win3.1x. If a computer is running NT, it can boot directly into NT without loading DOS. NT replaces both DOS and Win95 with one streamlined package, while it continues to support the underlying disk file structure used by these systems, so that disk files can be easily exchanged between the systems.

Windows OS allows several programs to run simultaneously. Code for different applications does not execute simultaneously. Instead, the operating system switches execution from one application to another, as needed, so that the user feels that all the running applications are responsive. To run multiple applications simultaneously Windows OS uses *permissive task switching*, which means that the running application (task) must give its permission before the processor is turned over to another waiting application. When an application gains execution in order to perform some action, it must return execution to the operating system as soon as it is done. If an application must do a lot of processing to perform some action, then the system seems to be “sluggish” to the user, since he cannot switch to another application while the processing is being done. Applications that have long computations to perform should therefore be designed to do them in short segments.

Win 95 and NT have several advanced features for controlling program execution that do not exist in Win3.1x. NT will interrupt a running application if it attempts to “hog” the entire system, allowing the user to gracefully terminate the errant program. Win95 and NT have the ability to split the execution of a program into pieces called threads of execution. A computer with more than one CPU chip can route different threads of execution to different CPUs, so that the computer can do more than one thing at one time.

The MFC classes were designed for compatibility between Win95, Win3.1x, and NT; therefore, the classes do not support the unique programming features of NT. However, many advantages, such as the ability to interrupt an errant program, are built into NT. Because they are built in, using them requires no special effort on the part of the programmer, and most MFC applications can be created without modifying the MFC classes or needing multithreaded execution.

C++ Compilers and MFC

C++ compilers that can build MFC applications are: Microsoft, Symantec, and Borland. The MFC examples in this book can be built with any of these compilers. Appendices are included that cover the important compiler aspects to get the reader started. Appendix D covers Microsoft's Visual C++ 4. Appendix E covers Microsoft's Visual C++1.5. Appendix F covers Symantec's C++ 7. Appendix G covers Borland's C++ 5.

The Microsoft compiler, known as Visual C++, is the most prevalent of the compilers that host MFC. Visual C++ 1.5 is for the Win3.1x operating system. Visual C++4 is for Win32 systems. Each of these compilers provides tools to help the programmer through difficult chores. The tools are known as AppStudio, ClassWizard, and AppWizard. These tools are covered, as necessary, in this book. Where it is necessary to mention the tools in this book, the Visual C++ 4 version is covered. There are relatively minor visual differences between the Visual C++ 1.5 and 4 versions of the tools. In one case, in Chapter Thirteen, the differences are sufficiently different that both versions of the tool are covered. Otherwise, the reader is expected to make the translation to his version. The main focus of the examples of this book is the MFC features, not the features of the tools.

The Symantec C++7 compiler also provides tools to assist the programmer. The Symantec tools are known as: ResourceStudio, ClassExpress, and AppExpress. The functionality of each of these tools is equivalent to the corresponding Microsoft tool. The tool functionality is sufficiently similar that the reader is expected to make the translation.

The Borland C++ 5 compiler provides MFC compilation support and the Resource Workshop can be used. Borland users can build all the examples in this book up through Chapter Twelve, since every example given in the first twelve chapters includes a complete listing of all the code that is required for that example.

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

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

☒ Brief ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Windows

Windows OS provides the necessary support to allow applications to generate a *graphical user interface* (GUI), which allows the user to interact with the application through graphics. In a Windows OS application the GUI consists of *interface elements*, which are the visual displays on the screen. The primary interface elements are *windows*, *dialog boxes*, and *controls*. The functional attributes assigned to these interface elements are predefined by the Windows OS. An application normally displays text and graphics in its windows. The dialog boxes normally contain controls, which display small amounts of information and let the user enter information. Windows OS has a large set of predefined controls, a rich set of API functions, and an extensive message-passing scheme, all of which allow applications to manipulate their interface elements.

In this book, I use the term “window” to refer to a window as described in this section. Technically, dialog boxes and controls are also windows; however, they are specialized windows with specialized behaviors defined in Windows OS. To avoid ambiguity, I will always refer to dialog boxes and controls specifically as dialog boxes and controls. (Dialog boxes and controls are covered in Chapters Six and Seven.)

Windows are the primary interface elements that applications use to present graphic and text output to the user. A window consists of a rectangular *client area*, the screen area on which the application writes or displays graphics and text, surrounded by a rectangular border and various optional non-client visual elements within the border. An application uses the client area to display text and graphics; therefore, the client area is a window’s largest element. For example, the text of a word processor, the spreadsheet of a spreadsheet program, and the graphics and text of a page-layout program are drawn by their applications in the client areas of their windows. A window can also have

child windows, child dialog boxes, and/or child controls in its client area. A child dialog box or control will usually communicate user input directly to its parent window. (Child windows are covered in Chapter Five.)

Every application must have a *main window*. Main windows are also called *mainframe windows*, because they have frames, or borders, that allow the user to control the client area. Figures 1-1a and 1-1b show the elements of the client and non-client areas of a main window for the Win3.1x operating system and Win32 (Win95 and NT operating systems). *Scroll bars*, which are optional elements that help the user control the main window, have been included. A menu bar, which contains menu items (one in this case), has been included. Menu bars appear only on main windows; if a window has a menu bar, it is a main window. Not all main windows have menu bars; the five applications presented later in this chapter are main windows that do not have menu bars.



Figure 1-1a: Win3.1x Window Elements



Figure 1-1b: Win95 and NT (Win32) Window Elements

The client area is the central portion of the window. A window can have a variety of non-client visual elements. The effect these elements have on windows are defined and managed by the operating system. Table 1-1 describes the behavior of these elements.

Table 1-1: Window Non-Client Visual Elements

<i>Element</i>	<i>Description</i>
caption bar	Holds the window’s caption or title and can be dragged to reposition the window.
system menu box	Displays a system menu when clicked; a system menu is defined by the operating system and normally contains such options as “Close,” “Maximize,” “Minimize,” and “Restore.”
minimize box	Minimizes the window when clicked. In Win3.1x the window shrinks to an icon and the text of the window’s caption appears below the icon. In Win95 the minimized application appears in the <i>task bar</i> (also referred to as the “launch bar”), which is the bar at the bottom of the screen; the minimized application’s icon precedes its title in the task bar.
maximize box	Expands the window to full screen when clicked. When a window is maximized, this box becomes a <i>restore box</i> , which restores the window to its previous size.

close box	Closes the window when selected. This feature is included on Win32 applications only.
menu bar	Holds the menu items. When a menu item is selected, pops up a <i>popup menu</i> (a list of menu items that “pops up” or “drops down”) or sends a message to the application.
thick border	Resizes a window when dragged with the mouse.
vertical scroll bar	Scrolls the client area vertically one line when either the Up Arrow or Down Arrow is clicked. Clicking just below the Up Arrow will scroll one page; clicking just above the Down Arrow will scroll one page. The <i>thumb</i> (a square icon in the scroll bar, which you can “grab” with the mouse) can be dragged to scroll rapidly through the client area.
horizontal scroll bar	(Same as vertical scroll bar, but operates the horizontal direction.)
size box	Resizes the window when dragged with the mouse. Appears with scroll bars. This feature is included on Win32 applications only.

The example applications given later in this chapter show how to use the MFC classes to create a main window. These examples demonstrate how your application defines classes that are derived from classes in the MFC library. Applications use objects of these derived classes to represent interface elements. These objects have standard functionality, which is defined by member functions in the MFC base classes. Additional application-specific functionality can be added in member functions which you define.

Previous	Table of Contents	Next

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

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

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

Keyword

☒ Brief
 ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

User Inputs to a Window

The Windows OS has a well-defined set of protocols that let the user perform routine actions, such as moving and resizing windows and switching the *focus* from one window to another window; focus defines the window where user input will be sent. Applications running on Windows OS will automatically conform to these protocols, since Windows OS controls the interface elements. The operating system gives the user the ability to concentrate on a particular application, and within that application on a particular window or dialog box, and then on a particular control in that window or dialog box. In order to make this possible, the concept of focus and *active window* is defined and supported by the operating system.

Note: For key sequences connected by a plus (+) sign (such as Ctrl+C), press and hold the first key while typing the second key. A concatenated keypress such as this is not case sensitive.

When the user clicks on an open window of the same application or a different application from the one he was interacting with, he expects that the window clicked on will gain focus; it will become active and its caption bar will be highlighted. By highlighting the active window, the operating system gives the user further indication of what window is actively accepting user input. Only one window, dialog box, or control can have focus or be active. If the interface element receiving the focus is a control, then the operating system makes its parent window or dialog box active. When a dialog box is given the focus, it immediately gives the focus to one of its controls. When a control has the focus, it indicates this either with a flashing caret (for controls that accept keyboard input) or with a dotted rectangle around the control's text (for all other controls). The caret is a bar or block (applications usually display a flashing caret) that is used to indicate the insertion point for the next character in a text display. Certain key combinations also change the focus. For

example, the user can use the key combination Alt+Esc or Alt+Tab to switch focus between open applications.

Sometimes, the operating system prevents the user from switching the focus to another interface element. When a modal dialog box is displayed, the operating system does not permit the user to change the focus to any other window of that application. This forces the user to complete the dialog before again interacting with the application.

Keyboard input is sent to the active window or control that has focus (for windows and controls that display a flashing caret, indicating that they accept keyboard input). The interface element that has focus receives keyboard messages from the operating system. Mouse messages are sent to the active window or to the control that has focus (for highlighted controls not displaying a flashing caret). The interface element that has focus receives mouse messages from the operating system.

Note: In this book, an API function is always preceded with a double colon (::) for easy identification.

As described above, users expect the focus to change in certain ways according to their actions. It is also possible for an application to give direct focus to a window, dialog box, or control by calling the MFC function `CWnd::SetFocus()` or by calling the API function `::SetFocus()`. `SetFocus()` is a powerful function for an application; if you use it, you should be careful not to violate the user's expectations of where the focus ought to be.

Messages

A central concept of the Windows OS is the *message*, or event. An application responds to messages. All actions result from a window, or application, receiving a message. Every application is event driven. Fundamentally, an MFC application consists of *message handler functions*, functions that respond to specific messages. When a message arrives, its handler function is entered and the code is executed. If no message arrives, no code is executed.

The operating system sends a message to an interface element which can be either a window, a dialog box, or a control. The operating system sends messages to an application's interface elements when an event occurs that may affect the window, dialog box, or control, or when the operating system needs information from the application about the interface element. A message is a structure that contains the data members described in Table 1-2.

Table 1-2: Description of Data Members of the Message Structure

<i>Data Member</i>	<i>Description</i>
hwnd	Identifies the window whose window procedure receives the message (hwnd is a pointer that contains the window's address, or location).

message	Specifies the message type or message number. A C preprocessor constant is defined in the windows.h header file for each message type. Most of the preprocessor constants for messages sent from the operating system start with the characters WM_. The WM stands for Window Message.
wParam	Specifies additional information about the message. The exact meaning of this parameter depends on the message type. For example, a WM_COMMAND message generated when a menu item is selected uses wParam to represent the ID of the menu item. Other messages use wParam differently; for example, the message WM_PAINT, which is generated when the screen needs to be repainted, does not use wParam.
lParam	Specifies additional information about the message. The exact meaning of this parameter depends on the message type. For example, the WM_CREATE message, which is generated when a window is created, uses lParam to represent a pointer to a structure containing information about the window being created. The message WM_PAINT does not use lParam.
time	Specifies the time at which the message was posted.
pt	Specifies the cursor position, in screen coordinates, where the message was posted.

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

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

An application can respond to only the messages to which it is designed to respond; the operating system is designed to handle all the messages that an application declines. In traditional Windows OS programming using only API functions, the wParam and lParam data members of each message have to be interpreted, which is quite a tedious process. In MFC programming, the MFC base classes perform this process for the programmer. MFC reads the messages, interprets the wParam and lParam data members, and provides only the necessary, interpreted data to the function that will respond to that message. Furthermore, MFC provides a *message map* that makes it easy for the application to define functions that only respond to particular messages. Message maps are explained in later chapters, but they basically “map” a message to its handler function.

MFC and Windows OS Interaction

Windows OS has three major components: *User*, *Graphics Device Interface* (GDI), and *Kernel*. User is a module of code that services input devices, such as keyboards. GDI is a module that services output to graphic devices—screens, printers, etc. Kernel services file management and internal memory management. Table 1-3 summarizes the functions of these components. Collectively, these three components are called the API. These components interact with the MFC application. An MFC application calls functions in the API. The base classes in the MFC library incorporate API functions, so the MFC classes also call functions in the API. The three API components are each provided as DLLs. An application can call functions in the DLL as though they were part of the application. The API DLLs are normally found in the Windows OS system directory, where files required by the system are usually stored. This directory is typically C:\WINDOWS\SYSTEM. The filenames for the three API DLLs are USER.EXE, GDI.EXE, and KRNL386.EXE.

Table 1-3: Major Components of the Windows Operating System

Kernel DLL (Kernel)	Provides task switching and message buffering, extensive high-level memory management, the handling of data to and from the input and output ports, and provides file services.
User DLL (User)	Provides services that manage the interface elements and relate to the keyboard and mouse. User sends window-related messages to the application, and contains many functions that the application can call to manipulate its interface elements, including functions to create and destroy them. User uses the keyboard and mouse device drivers to get input from these devices.
GDI DLL	Provides a set of high-level graphics functions that an application can call for graphic services. It manages display and printer drivers, and provides functions that can be called to get information about the video adapter and printers. Also, GDI breaks an application's drawing calls into more primitive drawing operations that can be handled by display and printer device drivers.

The API uses *device drivers* which access *peripheral hardware*, or devices, such as the keyboard, mouse, video display adapter, and printers. Device drivers are stored in the Windows' system directory (typically C:\WINDOWS\SYSTEM) and are loaded by the operating system as they are needed. Peripheral device drivers are generally written by the manufacturers of the devices, because a driver must be hardware specific in order for the video adapter to generate a display, the printer to print, the keyboard to accept input, and the mouse to be functional, etc.

Figure 1-2 graphically depicts the interactions between the Windows OS and MFC. As shown, the Windows OS consists primarily of three DLLs: User, GDI, and Kernel. These DLLs contain API functions that are called either directly from your application or indirectly via the MFC library which incorporates API functions.

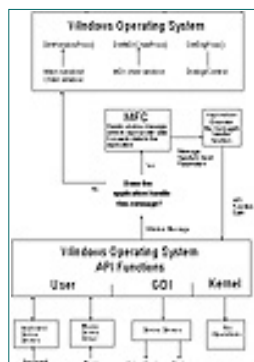


Figure 1-2: Interactions Between an MFC Application and the Windows OS

The window receiving the message can be a main window, a child window of

the main window, a *Multiple Document Interface* (MDI) child window (covered in Chapter Twelve), a dialog box, or a control. If your application is designed to handle the message, MFC intercepts the message, interprets the information in the message, selects the data that is appropriate to forward to your message handler function, and provides that data as input parameters to your message handler.

If your program is not designed to handle the message, the message is forwarded to the operating system's default message processing functions. If a main window or a child of the main window received the message, it is handed off to the DefWindowProc() function. If an MDI child window received the message, it is handed off to the DefMDIChildProc() function. If a dialog or a control received the message, it is handed off to the DefDlgProc() function. These are specific functions that provide default message processing.

Previous	Table of Contents	Next

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

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

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

Keyword

☒ Brief
 ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

The Structure of an MFC Application

An MFC application does two basic things: creates the application's own main window and performs any initial activities, such as allocating memory; and processes messages to that window, or application.

When writing a program, the first step is to define and create the application's main window, which is the "real estate" or piece of the screen that the application will control. Programs only write inside their own windows and not in other program windows. If a program tries to write outside of its area, nothing will show up on the screen. This is how the operating system keeps programs from interfering with each other on the screen. Restricting output to the program's window is the key to having several programs co-exist using the same screen, or display area.

A program's main window is also referred to as the mainframe. Main windows display the menu bar, if there is one. The main window, or mainframe, can be the parent to child windows, which include ordinary windows, dialog boxes, and controls.

A window can have many features. The remainder of this chapter presents five MFC programs that create main windows with varying features. The first example shows you how to create a main window with standard features. The second example shows you how to create a main window with optional non-client visual elements, which are referred to as window *style attributes* (keywords that specify the "style" a window will have). In the third example you are shown how to customize other features of the window, which are: background color, icon, and cursor. These features are customized using a process known as *registering a window class* with the operating system. (The process of registering a class is covered later in this chapter.) In the final two examples you will be introduced to the program's resource file and you will learn how to customize the window's icon and cursor. In these applications, all

messages are processed by the operating system’s default message processing function. (Chapter Two, which covers menus and message maps, discusses applications that handle messages and contains application-specific message handler functions which the programmer writes.)

In your MFC application, you derive your main window or *mainframe class* from the MFC class CFrameWnd, which is derived from CWnd. (Appendix A presents a hierarchy chart containing the MFC classes.) All the MFC classes begin with a “C.” CFrameWnd is the MFC class from which you always derive your mainframe class; it contains the functionality needed for a main window. CWnd is the base class of CFrameWnd. Dialog boxes and controls are also windows, so CWnd is also the base class of dialog boxes and controls. Programs need to create an object of the derived mainframe class which will be the main window object. Thus your main window object inherits standard mainframe behavior from CFrameWnd and it can have additional behavior that corresponds to any member functions that you define in your mainframe class. Additionally, behavior contained in the member functions inherited from CFrameWnd can be modified by overriding these inherited member functions with modified member functions in your mainframe class. Every MFC application will have a mainframe class. In this book, I have chosen to provide the mainframe class in a separate file. This code presentation matches that of the tools that build starter applications for you, because the tools build a separate file of the mainframe class.

Another class that every MFC application must have is an *application class*. An application class is derived from the MFC class CWinApp which contains the functionality needed to set up and run the *message loop*. The message loop is the internal engine that runs the program, processing messages in the message queue. I have chosen to provide the application class in a separate file, which is consistent with how the tools build a starter application. Later in the book, we will be using the tools to build starter applications. For now, the basics of an application will be discussed.

All the MFC classes are defined in the header file <afxwin.h>. The angle brackets around the header file tell the compiler to look in the compiler subdirectories for the file, rather than in the working directory. The prefix “afx” is an acronym for “application framework x.” The <afxwin.h> header file contains the MFC class definitions, which is how the compiler knows to include the CWinApp and CFrameWnd classes. This header file also contains a number of #include directives, which bring in other header files. <afxwin.h> includes the header file afx.h, which in turn includes five other header files: string.h, stdio.h, stdlib.h, time.h, and afx.ini. The <afxwin.h> file results in the compiler reading a number of header files before it starts compiling. Table 1-4 lists header files included in the <afxwin.h> file.

Table 1-4: Contents of the <afxwin.h> Header File

File	Contents
afxwin.h	MFC header file
afx.h	MFC header file for utility functions

	string.h	C run-time library for string functions
	stdio.h	C run-time library for input/output
	stdlib.h	Standard C run-time library
	time.h	C run-time library for time functions
	afx.ini	Inline function declarations
windows.h		Windows OS header file
afxmsg_.h		Message map table for MFC
afxdd_.h		Standard DDX_ and DDV_ routines
afxwin1.in1		Inline function declarations
afxwin2.in1		Inline function declarations

Previous	Table of Contents	Next

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

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

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


```

// MyApp.h:  main header file for the MyApp application
//
#include <afxwin.h>

class C MyApp : public CWinApp // derive the application
{
    public:
        BOOL InitInstance(); // declare funct. to override
}; // inherited InitInstance()
//
// MyApp.cpp: define the class behaviors for the application
//
#include "MyApp.h"
#include "mainfrm.h"

BOOL C MyApp::InitInstance() // define overriding function
{
    m_pMainWnd = new C_MainFrame; // instantiate a main window
    m_pMainWnd->ShowWindow(m_nCmdShow); // make window visible
    return TRUE;
}

C MyApp theApp; // instantiate object which runs program

-----
mainframe class
-----
//
// mainfrm.h:  interface of the C_MainFrame class
//
class C_MainFrame : public CFrameWnd // derive C_MainFrame from
{ // frame class, CFrameWnd
    public:
        C_MainFrame(); // declare constructor
};
//
// mainfrm.cpp:  implementation of the C_MainFrame class
//
#include "MyApp.h"
#include "mainfrm.h"

C_MainFrame::C_MainFrame() // define constructor
{
    Create (NULL, "Main Window A");
}
-----

```

The preceding program makes heavy use of inheritance from the MFC library. An MFC program will always use the CFrameWnd and CWinApp classes, which contain all of the code necessary to create and operate the main window.

To build the internal engine that runs the program we need to derive an application class. The MFC class that defines an application is CWinApp, which is the backbone of any application written in MFC. CWinApp takes care of starting the application, processing messages, and performing all the default activities. (The default activities do not include creating the main window labeled “Main Window A”; this window is created by overriding InitInstance() and calling the CWnd::ShowWindow() function.)

We derive a new class C MyApp from CWinApp and in our derived class define only one overriding function, InitInstance(). You must define your own overriding InitInstance() function; otherwise the application will not create a visible window that the user can see. The CWinApp default constructor is a

very complex function, and one of the many things it does is to always call its virtual function `InitInstance()` when a class object is instantiated. (I use the term instantiated to mean that an instance of an object is created.) `InitInstance()` is the ideal place to put the extra logic, such as allocating memory or initializing variables, that you need your program to perform when the application is started.

The only things that the compiled and executable `MyApp.cpp` needs to do when the application is instantiated is to create a main program window and make it visible on the screen. The `C_MainFrame` class is already defined, so you just instantiate a `C_MainFrame` object with the C++ *new* operator, which is a result of the following line of code:

```
m_pMainWnd = new C_MainFrame;
```

The value returned from the *new* operator is a pointer to the `C_MainFrame` object. This value is saved in the variable `m_pMainWnd`, which is a data member inherited from the `CWinApp` class. The `m_pMainWnd` data member is defined in `CWinApp` since all applications need to know the pointer to the main window of their application. (The prefix “m_” is used to label data members defined in the class definitions.)

After the `C_MainFrame` object is instantiated, the window must be made visible. This is done by calling the `CWnd::ShowWindow()` function. You make use of the pointer to the main window to access the function `ShowWindow()`, which has been inherited by your main window, in the following program statement:

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

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

ShowWindow() is passed an integer value m_nCmdShow. The m_nCmdShow value is passed to the application from the Windows OS environment when the application is started. The value of the m_nCmdShow data member tells the ShowWindow() function whether to start the window minimized, maximized, or default-sized.

Note: The C++ object representing the window and the window itself are two different items. The C++ object was instantiated and existing in memory before we executed the instruction to make it visible on the screen. The C++ object representing a window can be available in memory even though the window is not visible. (This is an important point, and we will make use of it in Chapter Six.)

To start your program running, you simply instantiate a global object of class C MyApp. The only statement in an MFC program that is directly called is the instantiation of the C MyApp object; everything else happens as a consequence of the following one line of code:

```
C MyApp theApp;
```

Instantiating the global object theApp results in the application object's constructor function being called, which calls the InitInstance() function, which creates the main program window and makes it visible. Once the program window is visible, message processing is handled automatically by the default logic in the CWinApp and CFrameWnd base classes. In this example, all messages are declined by our program, since we have no code for handling messages. All of the messages are passed to the default message processing logic. Somewhere deep in the definition of the CFrameWnd class is the call to the API function DefWindowProc(), directly accessing the default logic for message processing. The MFC classes do such a good job of encapsulating the window and application objects that you do not need to be concerned about where this action is taking place.

The MFC base class for a main program window is `CFrameWnd`. In our mainframe class, a new window class `C_MainFrame` is derived from the `CFrameWnd` class. The `CFrameWnd` class is also a derived class, derived from `CWnd`. The `CWnd` class contains most of the logic and functions concerning windows. By deriving a class from `CFrameWnd`, all of the public functions defined in `CFrameWnd` (as well as in the base class of `CFrameWnd`, which is `CWnd`) are available to the new `C_MainFrame` class.

Because constructor functions are not inherited, the only new function defined in this program is in our mainframe class and is the constructor function, `C_MainFrame()`, which will be called when a `C_MainFrame` object is instantiated. The `C_MainFrame()` constructor calls one function, `Create()`, which is a function that is defined as part of the `CFrameWnd` class; the `C_MainFrame` class accesses the `CFrameWnd::Create()` function through inheritance. The `CFrameWnd::Create()` function creates the application's main window.

The next thing you want to do is to create a project, type in the code, compile it, and run it. (General information on how to start up a project is covered in the appendices.)

The `CFrameWnd::Create()` Function

In the preceding program, the `CFrameWnd::Create()` function created the application's main window with standard features as a result of the following line of code:

```
Create (NULL, "Main Window A");
```

The first parameter specifies the default MFC-defined *window registration class*. Windows OS creates the window from a “registered class,” which means it will give the window the style, background color, cursor, and icon that are registered with the operating system. (The use of class here is entirely different than a C++ class. In the next section you will learn how to register window classes.) The first parameter is set to `NULL`, specifying the window will have default features which are: a white client area; a standard arrow cursor; and the Windows OS default icon.

The second parameter is set to the character string that you wish to have appear in the window's caption. The remaining parameters are omitted, so that the defaults are used, which give standard properties to the window. Normally, these defaults are just what you want, but you may use input parameters rather than relying on the defaults.

Actually, the function accepts eight parameters as shown on Table 1-5; all but the first two have default values defined, so these default values are used implicitly by simply not entering any values for the parameters when calling `CFrameWnd::Create()`. In the C++ language, default values can be used for parameters, as long as they are the last parameters passed to the function. You cannot omit parameters in the middle of the list of parameters because the C++ compiler does not have any way of knowing which ones have been omitted.

The next example program shows you how to use the third and fourth parameters. (Chapter Five, which covers child windows, demonstrates the use of the fifth parameter. In Chapter Two, which covers menus and message maps, the sixth parameter is used. The last two parameters are not used in this book. To find all the MFC classes and data types shown in Table 1-5 refer to Appendix A.)

Note: In Table 1-5 the parameters have prefixes that indicate the data type: *lpsz* for “long pointer to a zero-terminated string”; *dw* for “double word”; and *pfor* “pointer.” This notation is known as *Hungarian Notation*. See Appendix A for a list of Hungarian notation prefixes.

Previous	Table of Contents	Next

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

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

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

SEARCH
 ITKNOWLEDGE

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

BROWSE
 BY TOPIC

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes
(Publisher: Wordware Publishing, Inc.)
Author(s): Shirley Wodtke
ISBN: 1556225121
Publication Date: 01/01/97

Bookmark It

Search this book:

[Previous](#)

[Table of Contents](#)

[Next](#)

Table 1-5: CFrameWnd::Create() Parameters

```

BOOL CFrameWnd::Create(LPCSTR lpszClassName, LPCSTR
    lpszWindowName, DWORD dwStyle =
    WS_OVERLAPPEDWINDOW, const RECT& rect = rectDefault,
    CWnd* pParentWnd = NULL, PCSTR lpszMenuName = NULL,
    DWORD dwExStyle = 0, CCreateContext* pContext = NULL);
  
```

Parameter	Description
lpszClassName	A pointer to a null-terminated character string, ending with the null character (\n) and containing the name of the window class upon which the window will be based. NULL specifies a standard window with a white client area, the standard arrow cursor, and the Windows OS default icon.
lpszWindowName	A pointer to a null-terminated character string that is the window name. This string is used as text for the caption bar.
dwExStyle	Specifies the window extended style attributes.
rect	A RECT structure that contains the initial size and position of the window when it is created. Use the predefined rectangle rectDefault to let Windows OS decide the initial size and position of the window.
pParentWnd	A pointer to the parent window of the window being created. This will be NULL for a main program window.
lpszMenuName	A pointer to a null-terminated character string containing the name of the window’s menu in the resource script file. If the menu has an integer ID instead of a string, the MAKEINTRESOURCE() macro can be used. This parameter can be NULL.
dwStyle	Specifies the window style attributes. This value can consist of a series of binary flag values that specify properties of the window, such as if the window should have minimize and maximize boxes, scroll bars, etc. Use the value WS_OVERLAPPEDWINDOW to specify a standard program window with a thick border (for sizing the window), minimize and maximize boxes, system menu, and caption bar. Other dwStyle values are given in Table 1-6.
pContext	Specifies a pointer to a CCreateContext structure. This pointer can be NULL.

The second program given in this chapter shows how to create a window with optional non-client visual features. The executing window shown in Figure 1-4 uses the third and fourth parameters of `CFrameWnd::Create()` to create a main window that initially appears where specified (it can be moved since it has a caption bar), is initially of the specified size (it can be resized because it has a size box), and has vertical and horizontal scroll bars, as well as a system menu. (Notice that it does not have a minimize or maximize button, since these were not specified in the style parameter.)



Figure 1-4: Executing Window with Optional Non-client Features

The program called MyApp2 is given in Listing 1-2. Three of its files—`MyApp.h`, `MyApp.cpp`, and `mainfrm.h`—are identical to the ones used in the previous example (see Listing 1-1), so they are not repeated here. Listing 1-2 shows only the `mainfrm.cpp` file for the program MyApp2.

Listing 1-2: Source Code for MyApp2

```
-----
                                application class
-----
MyApp.h and MyApp.cpp have the same code as in Listing 1-1

-----
                                mainframe class
-----

mainfrm.h has the same code as in Listing 1-1

//
//  mainfrm.cpp:  implementation of the C_MainFrame class
//
#include "MyApp.h"
#include "mainfrm.h"

C_MainFrame::C_MainFrame()
{
    CRect WindowRect(100, 150, 600, 400); //construct CRect object
                                           // for window's left,
                                           // top, right, bottom

    Create (NULL, "Main Window B",
    WS_CAPTION | WS_SYSMENU |               //"Bitwise-OR" all
    WS_VSCROLL | WS_HSCROLL |              //attribute styles
    WS_SIZEBOX, WindowRect);               //CRect variable
                                           //replaces RECT& type
}
-----
```

The third parameter of `CFrameWnd::Create()` is the window style attribute. Table 1-6 lists the window style attributes that are used for main windows. It excludes window styles that apply only to child windows, controls, or dialog boxes. (Child windows, dialog boxes, and controls are discussed in later chapters, and their window style attributes are discussed there.)

The fourth parameter of `CFrameWnd::Create()` is the window size and position and is provided by a

CRect variable, named WindowRect and defined in this program. CRect objects can be used wherever you can use RECT structures. This is because the CRect class is derived from the RECT structure that is defined in the <windows.h> header file, which is:

```
typedef struct {  
    LONGleft;           // x-coord of upper left corner  
    LONGtop;            // y-coord of upper left corner  
    LONGright;          // x-coord of lower right corner  
    LONGbottom;         // y-coord of lower right corner  
} RECT;
```

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes
(Publisher: Wordware Publishing, Inc.)
Author(s): Shirley Wodtke
ISBN: 1556225121
Publication Date: 01/01/97

Bookmark It

Search this book:

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

The class CRect encapsulates the data and provides operations that apply to rectangles. CRect has a large set of functions and operators, which makes it a very useful class. CRect has five constructors; the program uses the constructor that most conveniently initializes the window coordinates.

Table 1-6: Window Styles

WS_BORDER	Creates a window that has a border.
WS_CAPTION	Creates a window that has a title bar (Implies WS_BORDER).
WS_CLIPCHILDREN	Excludes child window areas when drawing in the parent; used when you create the parent window. (Otherwise, Windows OS lets the parent draw over descendants but then repaints the descendants.)
WS_DISABLED	Initially disables the window, so that it will not receive mouse or keyboard messages.
WS_HSCROLL	Creates a window with a horizontal scroll bar.
WS_ICONIC	Creates an <i>iconic</i> , or minimized, window; used with WS_OVERLAPPED only.
WS_MAXIMIZE	Creates a window of maximum size.
WS_MAXIMIZEBOX	Adds a maximize box to window.
WS_MINIMIZE	Creates a window of minimum size.
WS_MINIMIZEBOX	Adds a minimize box to a window.
WS_OVERLAPPED	Create an overlapped window with a caption and a border.

WS_OVERLAPPEDWINDOW	Creates a default window, which is a composite of the following individual styles: WS_OVERLAPPED WS_CAPTION WS_SYSMENU WS_THICKFRAME WS_MINIMIZEBOX WS_MAXIMIZEBOX. This is MFC's default window style.
WS_SIZEBOX	Creates a window with a size box for resizing. Used only with a caption bar and vertical and horizontal scroll bars (Win32 only).
WS_SYSMENU	Adds a system menu box to the window. For child windows, a close box is created instead, which closes the child window only. (For windows with title bars only.)
WS_THICKFRAME	Creates a window with a thick frame. The user can drag the frame to resize the window. Windows OS has a convention that only windows with thick frames or size boxes can be dragged to resize.
WS_VISIBLE	Creates a window that is initially visible.
WS_VSCROLL	Adds vertical scroll bar to the window.

Registering a New Window Class

As you have seen in the previous two programs, MFC provides a standard window class as a default. The first parameter of the `CFrameWnd::Create()` function was `NULL` which specifies that the default window class will be used. The window will be created from the default window class and will have the standard class style, a white client area, a standard arrow cursor, and a default icon. Windows OS keeps track of a set of basic window features which are: class style, background color, cursor, and icon. This set of features is said to define a *window class*. The term window class is a unique usage. In this usage, the word class is specific to the Window OS environment and has nothing to do with the definition of a C++ class. The windows within a class share a few common characteristics, such as having the same class style, using the same brush color to fill the client area of the window (background color), displaying the same cursor shape when the mouse is over the window, and displaying the same icon when minimized.

Any number of windows can be created from a single window class once it is registered with the Windows OS environment. You can register a new window class by calling the `AfxRegisterWndClass()` *global function*. In MFC, there are a number of functions that are preceded by the prefix `Afx`, called global functions, which means they are not member functions of any MFC class and therefore they can be called directly from any place in your program. The function `AfxRegisterWndClass()` has four input parameters as shown in Table 1-7.

Table 1-7: AfxRegisterWndClass() Parameters

LPCSTR AfxRegisterWndClass (UINT nClassStyle, HCURSOR hCursor = 0,
HBRUSH hbrBackground = 0, HICON hIcon = 0;

Parameter	Meaning
nClassStyle	Specifies style values for the window class. This parameter can be any valid window or control style, or a combination of styles created by using the bitwise-OR () operator.
hCursor	Specifies handle of the cursor that will be displayed when the mouse is over the window. This is obtained by using either the CWinApp::LoadStandardCursor() or CWinApp::LoadCursor() function.
hbrBackground	Specifies the handle to the brush to be installed in each window created from the window class. This brush will be used in painting the window's client area.
hIcon	Specifies the handle to the icon resource to be installed in each window created from the window class. This can be either a stock icon loaded with CWinApp::LoadStandardIcon(), or an icon in the program's resource data loaded with CWinApp::LoadIcon(). Set this value to NULL if the windows created from the class will never be minimized.
Return Value	A null-terminated string containing the class name. You can pass this class name to the CFrameWnd::Create() member function to create a window. The name is generated by MFC. The return value is a pointer to a static buffer. To save this string, assign it to a CString variable.

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

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

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

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



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

Go!

[Previous](#)

[Table of Contents](#)

[Next](#)

BROWSE
BY TOPIC

The third program given in this chapter shows how to register a new window class and then how to create a window of the new window class. The executing program shown in Figure 1-5 has an exclamation point icon, a red background, and an UpArrow cursor. Figure 1-5 depicts the executing window of the new window class named MyWindowClass that is registered with the Windows OS environment in MyApp3.



Figure 1-5: Executing Window of MyWindowClass

The program in Listing 1-3, MyApp3, registers a new window class which is given the name MyWindowClass in the program. The program then creates a window based on this registered class. The program MyApp3 has three source code files that are the same as in the previous programs, and one source file, mainfrm.cpp, that has unique code which is the only source code file shown in the following listing.

Listing 1-3: Source Code for MyApp3

```
-----
                                application class
-----
```

MyApp.h and MyApp.cpp have the same code as in Listing 1-1

```
-----
                                mainframe class
-----
```

mainfrm.h has the same code as in Listing 1-1

```
//
```

```
//  mainfrm.cpp:  implementation of the C_MainFrame class
```

```
//
```

```
#include "MyApp.h"
```

```
#include "mainfrm.h"

C_MainFrame::C_MainFrame()
{
    HBRUSH hredBrush = ::CreateSolidBrush(RGB(255,0,0));

    CString MyWindowClass = AfxRegisterWndClass(
        CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW,
        AfxGetApp()->LoadStandardCursor(IDC_UPARROW), hredBrush,
        AfxGetApp()->LoadStandardIcon(IDI_EXCLAMATION));

    Create (MyWindowClass, "Main Window C");
    ::DeleteObject(hredBrush);
}
-----
```

In this program, values are assigned to all four input parameters of the function `AfxRegisterWndClass()`. The class name `MyWindowClass` is assigned to the return value of the function, thereby getting a `CString` variable to pass to the `Create()` function. The operating system then creates a window of the `MyWindowClass` class.

The first input parameter contains the class styles. In this program we use `CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW`, which is the standard set of class styles. Class styles, shown in Table 1-8, are used in more advanced programming. For graphics (covered in Chapters Three and Four) the class styles `CS_CLASSDC`, `CS_OWNDC`, and `CS_PARENTDC` are of interest if you wish to have your own device context rather than using a shared one. Other class styles—`CS_BYTEALIGNCLIENT`, `CS_BYTEALIGNWINDOW`, and `CS_SAVEBITS`—are used to improve drawing speed and efficiency.

Table 1-8: Class Style Flags

Style	Meaning
CS_BYTEALIGNCLIENT	Aligns a window's client area on the byte boundary horizontally. This makes a small savings in memory consumed by windows.
CS_BYTEALIGNWINDOW	Aligns a window on the byte boundary horizontally. This flag should be set by applications that perform bitmap operations in windows by using the <code>BitBlt()</code> function.
CS_CLASSDC	Gives the window class its own device context. Every window (instance) created will share this DC.
CS_DBLCLKS	Sends mouse double-click messages to the window.
CS_GLOBALCLASS	Specifies that the window class is an <i>application global class</i> . An application global class is created by an application or library and is available to all applications. The class is destroyed when the application or library that created the class exits; it is essential, therefore, that all windows created with the application global class be closed before this occurs.
CS_HREDRAW	Redraws the entire window if the horizontal size changes.
CS_NOCLOSE	Stops the close option on the system menu.

CS_OWNDC	Gives each window instance its own device context. (Note that each device context requires 800 bytes of memory.)
CS_PARENTDC	The window class uses its parent window's device context.
CS_VREDRAW	Redraws the entire window when the vertical size changes.
CS_SAVEBITS	Specifies that the system should try to save the screen image behind a window created from this window class as a bitmap. Later, when the window is removed, the system uses the bitmap to quickly restore the screen image. This style is useful for small windows that are displayed briefly and then removed before much other screen activity takes place (for example, menus or dialog boxes). This style increases the time required to display the window since the system must first allocate memory to store the bitmap.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes
(Publisher: Wordware Publishing, Inc.)
Author(s): Shirley Wodtke
ISBN: 1556225121
Publication Date: 01/01/97



Search this book:

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

The second parameter is a handle of a cursor. This example uses a *stock cursor*, which means that the Windows OS environment has the cursor bitmap in “stock,” and we only need to know how to access that bitmap. Pointers to bitmap images are identified by a preprocessor ID number. By protocol an ID number starting with IDC_ is used for cursors. (See Table 1-9 for a listing of all the stock cursor shapes. Note that these stock cursors are used by the Windows OS for resizing a window, etc.) This program uses the stock cursor IDC_UPARROW. This ID is provided as an input to the CWinApp::LoadStandardCursor() function whose return value is the handle to the cursor. The call to the function AfxRegisterWndClass() occurs from within the C_MainFrame class. A pointer to the application object is needed to invoke the LoadStandardCursor() function which is a member function of the application object. The global function AfxGetApp() returns the pointer to the application object. The code that loads the cursor and provides the cursor handle is:

AfxGetApp() ->LoadStandardCursor(IDC_UPARROW)

Table 1-9: Stock Cursor Shapes

Value	Description
IDC_APPSTARTING	Standard arrow and small hourglass (Win32)
IDC_ARROW	Standard arrow
IDC_CROSS	Crosshair
IDC_HELP	Standard arrow and question mark (Win32)
IDC_IBEAM	Text I-Beam
IDC_ICON	A square (used to drag files)
IDC_NO	Slashed circle (Win32)
IDC_SIZE	Four-pointed arrow
IDC_SIZEALL	Same as IDC_SIZE (Win32)

IDC_SIZENESW	The double-headed arrow pointing northeast and southwest
IDC_SIZENS	The double-headed arrow pointing north and south
IDC_SIZENWSE	The double-headed arrow pointing northwest and southeast
IDC_SIZEWE	The double-headed arrow pointing west and east
IDC_UPARROW	Vertical arrow
IDC_WAIT	Hourglass

The third parameter is the handle to a brush. In this program, a brush handle is obtained with the statement:

```
HBRUSH hredBrush = ::CreateSolidBrush( RGB(255, 0, 0) );
```

The API function `::CreateSolidBrush()` is used. The macro `RGB(255, 0, 0)` is a red color. `RGB()` is a macro defined in `<windows.h>` that puts the red, green, and blue intensity values of a color in the correct positions in a block of memory. (Chapter Three covers `RGB()` values. Chapter Three introduces brushes as “GDI objects” which must be deleted. This is accomplished with the API function `::DeleteObject().`)

The fourth input parameter is the handle of an icon. The stock icon which is an exclamation point is used in this example. Table 1-10 lists all the stock icons. By protocol an ID number starting with `IDI_` is used for icons. `LoadStandard Icon()` is also a member function of `CWinApp`. The code that loads the icon and provides the icon handle is:

```
AfxGetApp( )->LoadStandardIcon( IDI_EXCLAMATION)
```

Table 1-10: Stock Icon Shapes

<i>Value</i>	<i>Meaning</i>
IDI_APPLICATION	Default application icon (Win3.1x: “flying windows” and Win95: “miniature window”).
IDI_ASTERISK	“i” shape used in informative messages.
IDI_EXCLAMATION	Exclamation point shape used in warning messages.
IDI_HAND	Stop-sign shape icon used in serious warning messages.
IDI_QUESTION	Question-mark shape used in prompting messages.

The default window class values, which were used in the first two examples of this chapter, `MyApp1` and `MyApp2`, are:

```
class style: CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW;
cursor: IDC_ARROW
background color: COLOR_WINDOW (defined in <windows.h>)
icon: AFX_IDI_STD_FRAME
```

If a program includes an icon given the name `AFX_IDI_STD_FRAME` in its

resource file, the mainframe (main window) object will automatically use this icon image when the main window is minimized. Otherwise, if no icon named AFX_IDI_STD_FRAME is provided in the resource file, then Windows OS uses its default icon image. (Resource files are discussed next in this chapter.)

Previous	Table of Contents	Next

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

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

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

 SEARCH
ITKNOWLEDGE

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

 BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Resource Files

Programs consist of both *dynamic data* and *static data*. Static data is different from dynamic data. Dynamic data changes as the program runs. Dynamic data, which consists of the program code and the data it reads and writes, is stored in files separate from the static data. Static data is any portion of the program that is not executed as a machine instruction and which does not change as the program executes. Examples of static data are resources such as icons, cursors, and menus. Static resource data such as icons and cursors, sometimes called just “resources,” are put into separate source files in your program, because the Windows OS handles resource data separately from the program’s code.

To most efficiently manage available memory, Windows OS usually leaves resources on disk until they are needed, although you can cause individual resources to be loaded at program startup. Once loaded, a resource is usually placed in a “discordable” section of memory so that memory can be freed up for other tasks if needed. When the resource is again required, it is loaded from disk. You will put your resources or resource data such as icons and cursors into a separate source file, referred to as the program’s *resource script file*, or .rc file. Table 1-11 lists the resources covered in this book. (Icons and cursors are covered in the next section of this chapter. Menus and accelerators are covered in Chapter Two. Bitmaps are covered in Chapter Four. Dialog boxes and *string tables* (tables of string characters that are found in the resource script file) are covered in Chapters Six and Seven. Fonts are covered in Chapter Eleven.)

Table 1-11: Resource Data

Resource	Description
Icons	Small bitmap image displayed when the program is minimized.

Cursors	Small (32 x 32 pixel) bitmap image defining the mouse shape.
Menus	Defines the menu structure for the application. Menus include the top menu items located in the menu bar and all the drop-down menus.
Accelerators	Definitions of the keyboard shortcuts (concatenated keypresses) for executing menu items, or other commands using the keyboard instead of the mouse.
Bitmaps	Pictorial data defined as larger images. Can be used to display pictorial information in the client area.
Dialog Boxes	Defines the layout of the dialog box. Dialog boxes typically contain controls that interact with the user.
String Tables	Tables of character strings. They are used as static character data from within the program.
Fonts	Data defining the size and style of characters.

A standard C++ compiler expects to find the resource data separated from the program code. The C++ compiler compiles the code portion of the program and a “resource compiler” compiles the specialized resources. The compilation process for a program with a resource file is shown in Figure 1-6.



Figure 1-6: Compiling a Program with Resources

The compilation of the source code to get an object file is done by the C++ compiler. The object files are combined to make an unfinished .exe file. The resource data is compiled by the resource compiler to make a resource data file with the extension .res. The resource data is added to the unfinished .exe file to create the finished executable program. The resource compiler links the compiled resources with the compiled program code. The resource data is basically stuck onto the end of the program’s code by the resource compiler and becomes part of the program file. Separating the code from the resource data has the advantage of reducing memory demands because Windows OS loads resource data into memory from the disk only as needed. The data separation also speeds the program development process. If the resource source file only is changed, then only the resource data needs to be recompiled and relinked to the already compiled program source code. If only program code is changed, then the code can be recompiled and relinked to the already compiled resources.

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

Customized Icon and Cursor Resources

You can create your own customized, or “tool-generated,” icons and cursors. The next two example programs in this chapter use customized or “tool-generated” icons and cursors. To complete these next two programs, you need to create your own customized icon and cursor. Your C++ compiler has a tool to assist you in generating these resources.

As mentioned in the previous section you can assign your customized icon to the preprocessor constant `AFX_IDI_STD_FRAME` in your .rc file. This provides a unique and simplified way to provide an icon for your main window. The code to implement this method is fairly simple and is given as the next example program, MyApp4, which only addresses using customized icons. The executing code is shown in Figure 1-7. (In Win95, the icon will be displayed in the left portion of the main window’s caption bar when the window is maximized, or default-sized. When the window is minimized (iconized), Win95 will display the icon in the task bar.)



Figure 1-7: Window with a Customized Icon

The program, MyApp4, that displays this customized icon is given in Listing 1-4. The program code has an application class, a mainframe class, and a resource file which contains the file MyApp4.rc. Note that the resource script file includes the library file `afxres.h`. Several resource-related features supported by MFC are automatically available when your resource script file includes this library file. The single line of code in the resource script file that assigns the custom icon to the preprocessor constant `AFX_IDI_STD_FRAME` is:

```
AFX_IDI_STD_FRAME    ICON    "smiley.ico"
```

This line of code uses the key word `ICON` which identifies the resource as an icon and links the file, `smiley.ico`, to the `AFX_IDI_STD_FRAME`. `Smiley.ico`, containing your customized icon, must be available in your working directory. The icon contained in the file `smiley.ico` can be created with your compiler tool.

Listing 1-4: Source Code for MyApp4

```

-----
                                application class
-----

MyApp.h and MyApp.cpp have the same code as in Listing 1-1
-----
                                mainframe class
-----

mainfrm.h has the same code as in Listing 1-1
//
//  mainfrm.cpp:      implementation of the C_MainFrame class
//
#include "MyApp.h"
#include "mainfrm.h"
  
```



```

C_MainFrame::C_MainFrame()
{
Create (NULL, "Main Window D");
}
-----
Resource files
-----
//
// MyApp4.rc
//

#include "afxres.h"

AFX_IDI_STD_FRAME      ICON      "smiley.ico"
-----

```

The final example in this chapter, MyApp5, addresses using both icons and cursors that are customized. The preceding example used the fact that the MFC default class established an ID for the icon, which can be used for customized icons. In this final example, we register our own window class with its customized features, including its customized icon, cursor, and background color. The running program, MyApp5, is shown in Figure 1-8.



Figure 1-8: Window with Customized Icon and Cursor

The program code for this example is given in Listing 1-5. The program registers a window class with a tool-generated hand shaped cursor, a yellow-colored background, and a tool-generated icon (the smiling face) and then creates a window of this window class. The files hand.cur and smiley.ico are generated with the compiler tools.

Listing 1-5: Source Code for MyApp5

```

-----
application class
-----

MyApp.h and MyApp.cpp have the same code as in Listing 1-1

-----
mainframe class
-----

mainfrm.h has the same code as in Listing 1-1
//
// mainfrm.cpp:implementation of the C_MainFrame class
//
#include "MyApp.h"
#include "mainfrm.h"
#include "resource.h"

C_MainFrame::C_MainFrame()
{
    HBRUSH hYellowBrush = CreateSolidBrush(RGB(255,255,0));

    CString MyWindowClass = AfxRegisterWndClass(CS_DBLCLKS |           CS_HREDRAW
| CS_VREDRAW,
        AfxGetApp()->LoadCursor(IDC_HAND),
        hYellowBrush,
        AfxGetApp()->LoadIcon(IDI_SMILEY));
    Create (MyWindowClass, "Main Window E");
    ::DeleteObject(hYellowBrush);
}
-----
resource files
-----

```

```
//
//  MyApp5.rc
//

#include "afxres.h"
#include "resource.h"

IDI_SMILEYICON"smiley.ico"
IDC_HANDCURSOR"hand.cur"
//
//resource.h
//
#define IDI_SMILEY      101
#define IDC_HAND        102
-----
```

The mainfrm.cpp source file looks like MyApp3, except that it uses the functions LoadCursor() instead of LoadStandardCursor() and LoadIcon() instead of LoadStandardIcon().

In this last program, MyApp5, the customized icon and cursor resources are included in the resource files. There are two resource files, MyApp5.rc and resource.h. Resource files are different from other source files; statements in resource files do not end with a semicolon.

The appearance and contents of resources are defined in the resource script file. The source file MyApp5.rc consists of statements linking preprocessor constants to the files that contain the images. Notice that the reserved word CURSOR is used in the resource script file to identify the cursor, just as the reserved word ICON is used to identify the icon. The cursor's name and the icon's name (included to the left of the reserved words CURSOR and ICON) can be anything you choose. In this example, IDI_SMILEY is used for the icon, since by convention icons are named with the prefix IDI_; IDC_HAND is used for the cursor, since cursors are conventionally named with the prefix IDC_.

Previous	Table of Contents	Next

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

This example also uses a resource.h file. The resource.h file contains only statements defining the preprocessor constants. These statements define the constant `IDI_SMILEY` as a unique `UINT` (101), and the constant `IDC_HAND` as another unique `UINT` (102). These numbers must be unique and not used for any other resources.

The `LoadCursor()` and the `LoadIcon()` functions have *overloaded constructors* (constructors that have more than one definition where a definition is a set of input parameters). One of the two constructors for each of these functions is overloaded to accept as its input parameter a `UINT`, which is the ID number of the resource. This constructor form is convenient to use if you are generating your cursors and icons with compiler tools. The compiler tools will always generate two resource files, the `.rc` file and the `resource.h` file. Your compiler tool will generate resource files in which the preprocessor constants of your resources are defined as unique numerical values (`UINTs`) in the `resource.h` file, which the tool also generates. If you use this constructor as used in the preceding program, `MyApp5`, you will not have to alter the resource files generated by your compiler tool, but you will have to incorporate the ID names chosen by your compiler tool into your program.

The preceding example, `MyApp5`, presents a `.rc` and `resource.h` file that give the code needed for the source code program. Note that when a compiler tool is used, it gives the code needed for the source code program, but it also adds lines of code to be used by the tool itself when it is re-entered to add another resource. In all the examples shown in this book the resource files will contain only that code necessary for the source code program. The resource files shown in the examples will not contain any of the code added by the tool to be used only by the tool.

The other constructor for each of these functions accepts a null-terminated string as its input parameter. This string contains the name of the resource defined in the `.rc` file. To use this constructor, you use

LoadCursor("IDC_HAND") and LoadIcon("IDI_SMILEY"). You would not include #define statements in the resource.h file that defined these preprocessor constants to be UINTs. If you made this mistake, the constructor would be looking for a string and would find a UINT, and it would not load anything.

Summary

MFC provides an efficient way to build programs for Windows OSs. MFC is designed to work with all the available Window OSs. Most C++ compilers on the market today can build MFC applications. The central concept of a Windows OS application is its main window which displays the output of the application. The main window has a menu and can be designed with varying features. The user interacts with the application through keyboard presses and mouse clicks. The Windows OS sends messages to the application's windows as a result of user inputs or other events. The MFC framework intercepts these messages, reads and interprets them, and forwards the relevant data as inputs to the application's handler functions.

Using the MFC library allows you to create powerful programs with minimal coding. Your classes are derived from the MFC classes and inherit most of the functionality that you need. An application that creates a fully functional main window with standard features contains less than 30 lines of code. It contains an application class derived from CWinApp, from which it inherits all the functionality required to set up and run the message loop. It also contains a mainframe class derived from CFrameWnd, from which it inherits standard mainframe behavior.

Customized main windows can be created with specialized non-client features. The inherited CFrameWnd::Create() function is used to provide specialized window style attributes, initial size and position of the window when it opens up, the menu (if there is one), and the window class which tells Windows OS which cursor, icon, and background color to provide for that window. The default window class which provides a standard icon, the standard arrow cursor, and a white background is used unless the user specifies otherwise. The user can specify a unique window class by first registering a new window class using the global function AfxRegisterWndClass() and then using that registered window class in the call to the CFrameWnd::Create() function. A registered window class has a specific class style, cursor, background color, and icon. All windows created of that class will have the same class style, cursor, background color, and icon.

Resource files contain your program's static data. Resources such as icons, cursors, bitmaps, menus, etc. are placed in the resource file. Resource files are compiled by the resource compiler and attached to the end of the executable program. During run time resources are loaded as needed to save memory. Your customized icons and cursors are placed into your resource files.

Exercise

In this book, the exercises all involve the building of a checkers game

program. The program will be built incrementally as you go along. As you learn each new subject, that portion of the checkers game program will be built. In this chapter, you learned about creating the main window and adding a customized icon. Your exercise is to create a main window with the title “Checkers Game—Player A vs. Player B” and with a customized icon of your choosing. A complete description of the checkers game along with helpful hints is presented in Appendix B. Appendix B has a section titled “Chapter 1 Exercise”; refer to that section for more information.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

☒ Advanced

☒ Search

☒ Search Tips

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Chapter 2

Menus and Message Maps

In the previous chapter, you learned how to create a main window. In this chapter you learn how to add menus and message response functions to your main window. It is through menus and response functions to messages that the user interacts with an application. An example with a simple menu that incorporates an accelerator table is first explored. This example is then used to demonstrate how to add message map entries with the compiler tools. The second example in this chapter presents a more complex menu, one where the main menu items and the submenus change as the program progresses. In this second example, you declare your menus to be objects of class CMenu and use inherited functions to modify submenus and to change the main menu. Finally, we look at the CWnd response functions for messages.

Menus are used to let the user select one of a set of items. A menu is always associated with a main window. The main window “owns” the menu. A menu always has *top-level menu items*, which are in the menu bar that appears just below the caption bar of the main window. A menu item may have a submenu that drops down (called a popup menu) when the user selects that menu item. Submenus (popups) may have further submenus (popups). The term *end menu item* indicates a menu item that does not cause a submenu to appear. Ordinarily, when the user selects an end menu item, the application takes some action in response. The action in response to a menu-item selection occurs because the MFC message map causes the handler function (handler) for that message to be activated. Within the body of your program, you write the code for the handler function.

In this chapter, the first example presented is one with a simple menu. The simple menu has two items in the menu bar. One menu bar item has a submenu associated with it. The menu choices can be selected with a mouse click; however, with most menus, you associate a keyboard input with each menu selection. In this first example, you create a menu and attach it to your main window. You also learn how to associate keyboard input (accelerators) with each menu item.

An Example With a Simple Menu

Figure 2-1 shows a simple menu example which has two main menu items. When the user selects the menu item “One&Four Beeps,” a submenu appears with two end menu items: “OneBeep” and “FourBeeps F4.” In this example, selecting the menu item “Greeting” causes a message box (also shown) to appear.



Figure 2-1: Executing Code for MenuA Program

Listing 2-1 presents the source code for the MenuA program. It has a message map which maps the menu items to their handlers. When the selection “FourBeeps” is made, a timer is set. The timer is needed to space the beeps to make them audible. Since the window will be receiving the WM_TIMER messages during execution, you need to create a message map entry and a handler function for the WM_TIMER message. The message map, the menu, the accelerators, and the handler functions are discussed following the listing.

Listing 2-1: Source Code for MenuA Program

```
-----
                                application class
-----

MyApp.h and MyAppcpp have the same code as in Listing 1-1

-----
                                mainframe class
-----
//
//  mainfrm.h          interface of the C_MainFrame class
//

class C_MainFrame : public CFrameWnd
{
    public:
        C_MainFrame();
    private:
        int m_nBeeps;
        void OnOneBeep();           // declare fxns that respond
        void OnFourBeeps();         // to menu items
        void OnGreeting();
        void OnTimer(UINT);         // declare fxn that responds
                                    // to WM_TIMER messages

    DECLARE_MESSAGE_MAP()
}

//
//  mainfrm.cpp        implementation of the C_MainFrame class
//
#include "MyApp.h"
#include "mainfrm.h"
#include "resource.h"

BEGIN_MESSAGE_MAP (C_MainFrame, CFrameWnd)    // define message map
    ON_COMMAND(ID_ONE_BEEP, OnOneBeep)         // WM_COMMAND messages
    ON_COMMAND(ID_FOUR_BEEPS, OnFourBeeps)     // are from menu items
    ON_COMMAND(ID_GREETING, OnGreeting)
    ON_WM_TIMER()                             // for WM_TIMER message
END_MESSAGE_MAP()

C_MainFrame::C_MainFrame()
{

```

```

        Create (NULL, "Menus - Example A", WS_OVERLAPPEDWINDOW,
                rectDefault, NULL, MAKEINTRESOURCE(IDR_MENU1));
        LoadAccelerTable(MAKEINTRESOURCE(IDR_ACCELERATOR1));
        m_nBeeps = 0;
    }

//-----Response functions to menu items-----

void C_MainFrame::OnOneBeep()
{
    ::MessageBeep(0);
}

void C_MainFrame::OnFourBeeps()
{
    SetTimer(1, 200, NULL);
}

void C_MainFrame::OnGreeting()
{
    int MyChoice = MessageBox("Hello there!\nDo you want four beeps?",
                              "Greeting", MB_ICONEXCLAMATION | MB_YESNOCANCEL);
    if (MyChoice == IDYES)
        OnFourBeeps();
}

//-----Response function to WM_TIMERmessages-----

void C_MainFrame::OnTimer(UINT nID)
{
    if(m_nBeeps == 4)
    {
        KillTimer(1);
        m_nBeeps = 0;
        return;
    }
    else
    {
        ::MessageBeep(0);
        m_nBeeps++;
    }
}

-----
resource files
-----
//
//      Script1.rc
//
#include "afxres.h"
#include "resource.h"
////////////////////////////////////
//      Menu
IDR_MENU1      MENU            DISCARDABLE
BEGIN
    POPUP      "One&&Four&Beeps"
    BEGIN

```



```

        MENUITEM "&OneBeep",          ID_ONE_BEEP
        MENUITEM "&FourBeeps      F4",      ID_FOUR_BEEPS
    END
        MENUITEM "&Greeting",          ID_GREETING
END
////////////////////////////////////
//      Accelerator

IDR_ACCELERATOR1      ACCELERATORS      DISCARDABLE
BEGIN
    VK_F4,          ID_FOUR_BEEPS,      VIRTKEY, NOINVERT
    "G",          ID_GREETING,          ASCII, NOINVERT
    "g",          ID_GREETING,          ASCII, NOINVERT
END
//
//  resource.h      --      this program's resource IDs
//
#define IDR_MENU1          101
#define IDR_ACCELERATOR1  102
#define ID_ONE_BEEP        40001
#define ID_FOUR_BEEPS      40002
#define ID_GREETING        40003
-----

```

Previous	Table of Contents	Next

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

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

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

Keyword

☒ Brief
 ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Message Maps

The MFC approach to processing messages is to “jump,” or map, to the message’s handler function when a message that the program will process is received. The handler function does whatever action is appropriate in the program for that one message, then returns control to the operating system and waits for the arrival of the next message. The “jumping” to individual handler functions is done with a message map. The message map specifies which function is called for each message and then lets messages not in the message map pass on to the Windows OS default message processing logic.

The creation of a message map is done using several macros that are defined in the <afxwin.h> header file. The declaration of any class using message maps must contain the macro DECLARE_MESSAGE_MAP(). By convention, it is placed at the end of the class declaration. The message map’s table is defined with a set of macros that expand to message map entries. The table begins with a BEGIN_MESSAGE_MAP() macro call, which defines the class that is handled by this message map and the parent class to which unhandled messages are passed. The table ends with the END_MESSAGE_MAP() macro call. Between these two macros can be any number of references to messages that will be processed by the program. Listing 2-1 shows a program where the WM_COMMAND and WM_TIMER messages are processed. All other messages are not diverted by the message map and are therefore sent to the Window OS default message processing logic.

The operating system sends a WM_COMMAND message when a menu item is selected. A WM_COMMAND message is sent regardless of which menu item was selected, so there needs to be a way to decide which menu item was chosen. This is done using the menu item’s *ID identifier* that was specified in the menu definition in the .rc file. This ID value is passed along with the WM_COMMAND message. The MFC message map specifies which function

gets executed depending on the ID value passed with the WM_COMMAND message. There will be one function in the window object's class definition for each "end item" the menu contains. The following message map table in Excerpt 2-1 is from Listing 2-1:

Excerpt 2-1: Message Map Table from Listing 2-1

```
BEGIN_MESSAGE_MAP (C_MainFrame, CFrameWnd)
    ON_COMMAND(ID_ONE_BEEP, OnOneBeep)
    ON_COMMAND(ID_FOUR_BEEPS, OnFourBeeps)
    ON_COMMAND(ID_GREETING, OnGreeting)
    ON_WM_TIMER( )
END_MESSAGE_MAP( )
```

Note the following features from the preceding message map table:

- The message map table must be defined outside the scope of any function or class definition.
- The table begins with a BEGIN_MESSAGE_MAP() macro call. The first parameter is the name of the window class (in this example, C_MainFrame) receiving the messages. The second parameter is the MFC base class from which the window class was derived (in this example, CFrameWnd) and to which unhandled messages are passed. These parameters allow the MFC logic to segregate the message maps for different windows. The table ends with the END_MESSAGE_MAP() call.
- Between these two macro calls is an entry for each message to be handled by this message map. The WM_COMMAND messages will have the message map entry ON_COMMAND(). The ON_COMMAND() entry has two parameters; the first parameter is the ID identifier of the end menu item, and the second parameter is the name (chosen by the programmer) of the response function for that end menu item. The response function for a WM_COMMAND message will always have the return value void. There is also an entry ON_WM_TIMER() which maps the *ordinary message*, WM_TIMER, which is always processed by a function named OnTimer(). (In this book, I use the term *ordinary message* to mean any WM_ message other than the WM_COMMAND message. Ordinary messages are discussed later in this chapter.) The message map entry for an ordinary message has the form ON_WM_XXXX() for the message WM_XXXX.

Previous	Table of Contents	Next

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

[Previous](#)[Table of Contents](#)[Next](#)[BROWSE
BY TOPIC](#)

Menus

The menu, which is named IDR_MENU1 and is present in the script1.rc file, is attached to the main window with the CFrameWnd::Create() function that was discussed in Chapter One. This is accomplished with the following line of code found in the C_MainFrame() constructor:

```
Create (NULL, "Menus - Example A", WS_OVERLAPPEDWINDOW,
        rectDefault, NULL, MAKEINTRESOURCE(IDR_MENU1));
```

All API and MFC functions that have a *resource identifier argument* can accept either a string or an integer. Since function overloading is not done in C, the argument is always declared to be of type LPCSTR (pointer to a character string). If you are supplying an integer, you use the macro MAKEINTRESOURCE to convert it to type LPCSTR. The macro MAKEINTRESOURCE casts an integer argument to a far pointer to a character.

You can create your menu resource script manually or you can use your compiler tool. The compiler tools for creating menus always generate two resource files: the .rc file and the resource.h file. Your compiler tool generates a resource file in which the preprocessor constant of your menu resource is defined as a unique numerical value in the resource.h file, which the tool also generates. If you use MAKEINTRESOURCE (IDR_MENU1) as the input argument, you do not have to alter the files generated by your compiler tool.

If you are generating your menus by hand, or choose to alter the tool-generated menus, then you can use the following line of code, which accepts the name of the menu as a string that has not been changed (by a #define in the resource.h file) to be a UINT.

```
Create (NULL, "Menus - Example A", WS_OVERLAPPEDWINDOW,
        rectDefault, NULL, "IDR_MENU1");
```

Note: Bear in mind, that if you use the preceding line of code, then the following line of code must not be present in the resource.h file:

```
#define IDR_MENU1
```

101

The menu script, shown in Listing 2-1, is present in the .rc file and is shown in the following Excerpt 2-2. Menu resource scripts generated by compiler tools will have lines of comment code which the tool uses to re-enter the file and make alterations or additions, as required. Such lines are not operative code. The lines of code shown in Excerpt 2-2 are the operative code from the menu resource script.

Excerpt 2-2: Menu Script From the .rc File

```
IDR_MENU1      MENU      DISCARDABLE
BEGIN
    POPUP      "One&&Four&Beeps"
    BEGIN
        MENUITEM "&OneBeep",      ID_ONE_BEEP
        MENUITEM "&FourBeeps      F4",      ID_FOUR_BEEPS
    END
    MENUITEM "&Greeting",      ID_GREETING
END
```

The first line of the script contains first the string that identifies the menu, in this case IDR_MENU1. Then the next item (separated from the first item by one or more blanks) is the keyword MENU, which identifies that the following lines of script represent a menu. The last item, DISCARDABLE, tells the compiler that this resource can be moved from *accessible memory* to disk.

The menu items for the menu IDR_MENU1 are contained between the BEGIN and END statements, or block, which immediately follows the initial identifying line. Each line in the menu script must begin with either POPUP or MENUITEM. A POPUP line item will contain the string that will appear on the menu, and the POPUP line must then be followed by another BEGIN and END pair that encloses the menu entries for the submenu. In a .rc file, you could use braces instead of BEGIN and END. By convention, BEGIN and END is almost always used instead of braces ({}). Menu end items are identified on the lines beginning with MENUITEM. The keyword MENUITEM is then followed by the string that will appear in the menu item and is separated by a comma from its ID. It is in these lines of code that you assign IDs to your end menu items.

Tip: It is important to assign a descriptive ID that is easy to remember, as this ID will be used on the message maps, as you have seen.

Keyboard alternatives to mouse actions are always a good idea, giving the user a choice as to the most convenient way to select an item in a menu. The simplest keyboard alternatives are provided by preceding a selected character in the menu item's title string with an ampersand character. (The character following the ampersand ends up underlined in the menu when it appears.) The ampersand character itself is not visible when the menu is displayed. For example, the string "&Greeting" produces a menu item that responds to Alt+G (no case sensitivity) to open the message box. The string "One&&Four&Beeps" will produce a menu item that responds to Alt+B to bring up the submenu. The Alt key must be used in conjunction with the underlined character for menu items on the main menu bar. Once the submenu has appeared, a concatenated keypress is not necessary, because the Alt key is not used once the submenu has appeared; the user may select "O" for one beep or "F" for four beeps.

Tip: If you want "&" to appear in the string, you must use "&&" in the code.

Using the ampersand approach is fine for a single menu bar, but becomes cumbersome when there are popup menus. A better keyboard shortcut would require only one concatenated

keypress to execute the command, no matter where it was in the menu structure. The operating system provides this much simpler alternative in the form of *keyboard accelerators*.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

☒ [Advanced Search](#)

☒ [Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Accelerators

In this section, the basic ideas that you need to know to use accelerators are presented. The example given is limited and is not meant to be a comprehensive discussion of all the accelerator options that are available. However, it does give you an understanding of what accelerators are and a “roadmap” of how to incorporate them into your program.

The basic idea of keyboard accelerators is that you define the key combinations in the resource script file. Each key combination that is defined ends up sending a WM_COMMAND message to the program; the WM_COMMAND messages imitate menu selections.

In the MenuA example, an accelerator table is used to obtain the following keyboard functionality:

1. Pressing either the “G” or “g” character without using the Alt key to open the message box.
2. Pressing the F4 key (before the submenu items appear) to initiate four beeps.

The following excerpt shows and discusses the operative code from the accelerator resource file from Listing 2-1:

Excerpt 2-3: Accelerator Resource Files from Listing 2-1

```

IDR_ACCELERATOR1      ACCELERATORS      DISCARDABLE
BEGIN
    VK_F4,              ID_FOUR_BEEPS,      VIRTKEY, NOINVERT
    "G",                ID_GREETING,      ASCII, NOINVERT
    "g",                ID_GREETING,      ASCII, NOINVERT
END
  
```

The accelerator table can be generated manually, or you can use your compiler tool. The accelerator table is given a name, followed by the keyword ACCELERATORS, which can then be followed (if you choose) by the keyword DISCARDABLE to help the operating system with memory management. Adding an accelerator table to the program’s resource

script file does not immediately make the data available to the program when the program runs. To be used, the accelerator table must be loaded into memory and attached to the CFrameWnd class object for the program's main window. The CFrameWnd::LoadAccelTable() function loads the accelerator table into memory, ready for use. The accelerator table name is used to load the accelerator data with the following line of code from the C_MainFrame() constructor:

```
LoadAccelTable(MAKEINTRESOURCE(IDR_ACCELERATOR1));
```

As with menus, if you have generated the accelerator table with a compiler tool, you will want to use this form to load the accelerator table. Otherwise you must alter the resource.h file and eliminate the line:

```
#define IDR_ACCELERATOR1 102
```

If you do not have this line in your resource.h file, you may use the following line of code to load the accelerator table:

```
LoadAccelTable("IDR_ACCELERATOR1");
```

The accelerator table syntax is as follows:

```
TABLENAME          ACCELERATORS          DISCARDABLE
BEGIN
    Key, IDvalue, [VIRTKEY/ASCII][NOINVERT][ALT][SHIFT][CONTROL]
END
```

The keystroke for each key combination is called the Key. There are two basic types of Keys, those specified as virtual keys (VIRTKEY), and those specified as letters (ASCII). After the Key is a comma, and then the ID value that the accelerator will imitate. This is the integer value that will be sent as the ID value with a WM_COMMAND message when the key combination is pressed. This should be the ID value of the corresponding menu item that you want the keystroke to activate. Following the ID value is another comma, and then zero or more flags specifying the type of accelerator. If the Key type is ASCII, then the Key entry must be a letter in double quotes. In the ASCII type of accelerator definition, the keyboard accelerator is a case-sensitive match of the character in double quotes. If the Key type is VIRTKEY, the Key is interpreted as a virtual key code, such as function keys, which are defined in <windows.h>. For example, the Key identifier VK_F4 used in this program is defined in <windows.h> as the virtual key code for the F4 key. You can also specify the ALT, SHIFT, and CONTROL keywords, separated by commas, in any combination.

Tip: Avoid the commonly used accelerators, such as the F1 function key that is often used to invoke help.

One additional keyword that can be used with an accelerator is NOINVERT. This stops the corresponding menu item (if any) from flashing when the accelerator is activated. Normally, the top menu bar selection corresponding to the accelerator is switched quickly to reverse and back again to simulate a mouse click. NOINVERT stops this default action.

Tip: You should attempt to give the user notice, wherever practical, of the existence of the accelerator keys. For example, the string "&FourBeeps F4" tells the user that the F4 key is the accelerator for that menu item. One drawback of using accelerator keys is that, unless it can be put onto a submenu, it is difficult to give the user information on the accelerator keys that are

available.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Handler Functions

Handler functions are those functions which are activated, via the message map, when a message arrives. There are two types of handler functions. The first are those handler functions that respond to a WM_COMMAND message from a menu item. WM_COMMAND messages have a message map entry that takes two entries, one of which is your handler function's name (you may choose any name). The second are those that respond to all other WM_ messages. For this second type, the message map simply contains the ID of the message, e.g., the ON_WM_TIMER() message map entry found in the preceding example. The handler functions for WM_ messages other than the WM_COMMAND message are CWnd member functions and must have the same name and prototype as found in the CWnd function. This second type of handler function is discussed later in this chapter.

In this example there are three handler functions that respond to WM_COMMAND messages, which are: OnOneBeep(), OnFourBeeps(), and OnGreeting(). There is one function that responds to an ordinary message, WM_TIMER, which is OnTimer(). For this function you can only use the name OnTimer(). In the following paragraphs, code found in the handler functions is discussed.

In the handler functions OnOneBeep() and OnTimer(), the API function ::MessageBeep(0) is used. Whenever an API function is used, it will be preceded by the scope resolution operator (::). This is done for two reasons. First, it makes it easy for the reader to know which function is being used, and, secondly, the compiler knows to look for the API function rather than first searching all base classes and then finally looking for the API function.

You can deliberately choose to call an API function in place of an MFC member function with the same name. For example, you may wish to call the API MessageBox function. In this case, you must use the :: operator to ensure that

you call the API function. This call would also need to give the window's handle as the first parameter, which is the `m_hWnd` data member of `CWnd`. This API call is shown in the following line of code:

```
int MyChoice = ::MessageBox(m_hWnd, "Hello there!\n\nDo you want four beeps?", "Greeting", MB_ICONEXCLAMATION | MB_YESNOCANCEL);
```

Functions inherited from `CWnd` are used in this example. They are: `CWnd::SetTimer()`, `CWnd::KillTimer()`, and `CWnd::MessageBox()`. Any of these `CWnd` functions can be called three ways, as illustrated by the `KillTimer()` calls:

```
(1)  CWnd::KillTimer(1);  
(2)  this->KillTimer(1);  
(3)  KillTimer(1);
```

Setting the Timer

The `CWnd::SetTimer()` function has three input arguments. The first argument is an ID number which allows you to keep track of the timers that you have set. The ID number allows you to set more than one timer in your application. Win3.1x allows up to 32 timers to be set at one time. Win95 has essentially unlimited timers. In Win3.1x, the limit of 32 applies to the sum of all the timers set by every application running on the system, not just to one program. `CWnd::SetTimer()` will return zero if all 32 timers have been used. So you can check whether the timer was set and alert the user if the system has used all 32 timers.

Once the timer has been set, your program begins receiving `WM_TIMER` messages at a fixed time interval specified by the second input argument. This argument specifies the number of milliseconds that the operating system should wait between sending `WM_TIMER` messages. The operating system will not put more than one `WM_TIMER` message in a program's message queue, so there is no danger of having `WM_TIMER` messages "pile up" waiting to be processed. The third argument of `CWnd::SetTimer()` is `NULL` if you want the `CWnd` object to handle the `WM_TIMER` messages.

`WM_TIMER` messages are mapped to the `CWnd::OnTimer()` function, which has the following function prototype:

```
void OnTimer(UINT nID)
```

Where `nID` is the ID number for the timer (passed by the operating system with the `WM_TIMER` message). MFC makes the value of `nID` available to your response function. Check this value to find out which timer generated the `WM_TIMER` message if your program uses more than one timer. If your program only uses one timer, the message must be from that timer, so there is no need to check the `nID` value. Timers are specific to one window object, so there is no way your program will get a stray `WM_TIMER` message from another application.

When the timer is no longer needed, remove it from memory with the `CWnd::KillTimer()` function. This must be done explicitly, as terminating the program will not automatically remove the timer. The `CWnd::KillTimer()` function has only one argument, which is the identifier of the timer to kill.

Displaying a Message Box

Message boxes are *modal dialogs*. Modal dialogs are the subject of Chapters Six and Seven where their full capabilities are explored. When a modal dialog is activated, you can't do anything else in your application, which "owns" (parents) the modal dialog. You can switch to another application, which does not own the modal dialog box, and work in that application. In the application that owns the dialog box, you must close the modal dialog box before you can access any code in that application.

Message boxes use the modal technique to force the user's attention on the information at hand. They are modal dialogs that are packaged in a programmer-friendly way. The programmer can use message boxes to display, during the running of the program, small amounts of information important for the user to see. They also have a return value which gives the user's choice when the message box is closed. Message boxes are very easy to implement. For these reasons, they are introduced here; you will find them very useful.

The call to the `MessageBox()` function causes a message box to be displayed. The value returned by the `MessageBox()` function depends on the button selected. The first argument is the string (or a pointer to a buffer containing the string) that you wish to be displayed within the message box. You embed "\n" characters in the string to produce multiline text (the text will not automatically break into multiple lines). The second argument is the string (or a pointer to the string) that you want displayed in the caption bar of the message box. The third argument of `MessageBox()` should consist of one of the style choices in the following Button Style list, optionally "or"-ed with one of the icon specifiers listed under Icon Specifiers, and optionally "or"-ed with one or two of the modifiers in the Modifiers list.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Return Value

Zero is returned if there is not enough memory to display the message box; otherwise one of the following values is returned:

IDABORT	The Abort button was selected.
IDCANCEL	The Cancel button was selected.
IDIGNORE	The Ignore button was selected.
IDNO	The No button was selected.
IDOK	The OK button was selected.
IDRETRY	The Retry button was selected.
IDYES	The Yes button was selected.

Button Style List

The third argument of MessageBox() should include one of these preprocessor constants. The indicated buttons will be displayed in the message box. The buttons are displayed left to right, in the order listed.

MB_OK	“OK” button
MB_YESNO	“Yes” and “No” buttons
MB_YESNOCANCEL	“Yes,” “No,” and “Cancel” buttons
MB_RETRYCANCEL	“Retry” and “Cancel” buttons
MB_OKCANCEL	“OK” and “Cancel” buttons
MB_ABORTRETRYIGNORE	“Abort,” “Retry,” and “Ignore” buttons

Icon Specifier List

If one of these specifiers is *or*-ed into MessageBox()’s third argument, then the corresponding icon will be displayed in the upper left part of the message box.

MB_ICONINFORMATION	Lowercase “i” in a circle
MB_ICONEXCLAMATION	Exclamation point in a circle
MB_ICONSTOP	Stop sign
MB_ICONQUESTION	Question mark

Modifier List

One choice for the modality type and one choice for the default button may be “or”-ed into `MessageBox()`’s third argument. It is not meaningful to include more than one modality or more than one default button. The default button is visually indicated by a thick border and will be invoked if the user hits the Enter key.

MB_SYSTEMMODAL All applications are suspended until the user responds to the message box. System-modal message boxes are used to notify the user of serious, potentially damaging errors that require immediate attention. They should be used sparingly.

Note: System modal dialogs are supported in 32-bit Windows operating systems. However, in Win32, the system modal dialog behaves the same as an application modal dialog box. Using Win32 system modal dialogs is not recommended.

MB_APPLMODAL The user must respond to the message box before continuing work in the current window. However, the user can move to the windows of other applications and work in those windows. This is the default if no modality style is specified.

MB_DEFBUTTON1 The first button is the default. (Note that the first button is always the default unless specified otherwise.)

MB_DEFBUTTON2 The second button is the default.

MB_DEFBUTTON3 The third button is the default.

Adding Message Map Entries With Compiler Tools

A compiler tool (ClassWizard for Visual C++ or ClassExpress for Symantec’s C++) can be used to add the message map entries and handler functions to your program, saving you a lot of time typing in code. Using the MenuA example, the steps on how to design your program such that your compiler tool will participate are shown. In the following example, the code is shown before the compiler tool participates and the code is then shown after the compiler tool has made its entries, so that you can observe what code the compiler tool has inserted for you.

For the compiler tool to participate, the code must be separated into a .h and a .cpp file (which has been done). The compiler tool wants to start with the message map macros already inserted into the code, and it looks for certain text (comment) lines within the code. In the file `mainfrm.h`, the lines that the compiler tool needs to see (located within the class declaration block) are:

```
//{{AFX_MSG(C_MainFrame)
//}}AFX_MSG
DECLARE_MESSAGE_MAP( )
```

In the file mainfrm.cpp, the lines that the compiler tool needs to see (located outside of any block of code) are:

```
BEGIN_MESSAGE_MAP (C_MainFrame, CFrameWnd)
    //{{AFX_MSG_MAP(C_MainFrame)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP( )
```

Note: You must be very careful to type these comment lines in exactly as they appear in the preceding examples. You cannot leave any extra blanks in these comment lines. The compiler will not catch typing errors because these are comment lines. If there is any error whatsoever in these comment lines, your compiler tool will not be able to perform and will give unusual messages or simply come up with a dialog box with empty entries. If there is any problem getting your compiler tool to work, be sure to very carefully re-examine these comment lines for correctness. I cannot emphasize this point strongly enough!

You must have the preceding lines available in the code. To add message map entries for your menu, the resource files must be added to your project. Then you are ready to enlist the services of the compiler tool to write code for you. Listing 2-1a (following) gives the code for this program as it should look when it is ready for the compiler tool to participate in writing code for you. Only the mainframe class is shown in Listing 2-1a, since it is the file that has been modified to work with the compiler tool. The application class and the resource files are the same as those shown in Listing 2-1. (Note that the code includes the message map macros and the specific comment lines discussed above.)

Previous	Table of Contents	Next

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

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

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

Keyword

☒ Brief
 ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

You can compile the code as given in Listing 2-1a (you must include the resource file, of course). The code will compile with no compiler errors before you add the message maps. The lack of message maps and handler functions does not inhibit the compiler from building your program. However, you will notice in your executing program that the popup menu items that are not completed with message maps and handler functions are grayed. The application framework automatically disables (grays) the popup menu items for which it does not find a handler function. This automatic behavior occurs when the data member CMainFrame::m_bAutoMenuEnable is set to TRUE. This data member is initialized to TRUE by the framework, but it can be directly accessed since it is a public data member. If you access this data member and set it to FALSE, the automatic graying of popup menu items without handler functions does not occur and you can enable/disable the menu item yourself. Enabling/disabling a menu item is discussed later in this chapter.

Listing 2-1a: Modified MenuA Code Before Using Compiler Tool to Add Message Map Entries

```

-----
                                     mainframe class
-----
//
//  mainfrm.h          interface of the C_MainFrame class
//
class C_MainFrame : public CFrameWnd
{
    public:
        C_MainFrame();
    private:
        int m_nBeeps;
        //{AFX_MSG(C_MainFrame)
        //}AFX_MSG
        DECLARE_MESSAGE_MAP()
};

//
//  mainfrm.cpp        implementation of the C_MainFrame class
//

#include "MyApp.h"
  
```



```
#include "mainfrm.h"
#include "resource.h"

BEGIN_MESSAGE_MAP (C_MainFrame, CFrameWnd)
   //{{AFX_MSG_MAP(C_MainFrame)
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()

C_MainFrame::C_MainFrame()
{
    Create (NULL, "Menus - Example A", WS_OVERLAPPEDWINDOW,
            rectDefault, NULL, MAKEINTRESOURCE(IDR_MENU1));
    LoadAccelerTable(MAKEINTRESOURCE(IDR_ACCELERATOR1));
    m_nBeeps = 0;
}
```

The compiler tool will add code in both the mainfrm.h and mainfrm.cpp files. In fact, your code must be separated into a .h and a .cpp file as we have done in this example. The compiler tool looks for both of these files.

Figure 2-2 shows the ClassWizard compiler tool adding message maps.

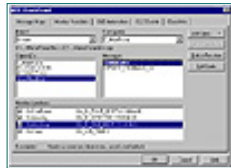


Figure 2-2: Adding Message Maps with ClassWizard

The compiler tool inserts the handler function prototype into the mainfrm.h file between the comment lines that have been provided. The prototype functions added will be preceded by the no-op prefix *afx_msg*. All function prototypes added between the comment lines must be preceded by *afx_msg* if you wish to later invoke the services of the compiler tool. If you add function prototypes manually, you must still use the *afx_msg* prefix if you wish to retain your file to be useable again by the compiler tool. If you are adding them manually and do not wish to use the prefix *afx_msg*, then locate that function prototype outside the comment lines.

The compiler tool provides the message map entries in the mainfrm.cpp file between the comment lines that have been provided. It also provides the “stubbed-out” handler function for each message. You, of course, must fill in the body of each of the “stubbed-out” handler functions, which will be done manually. The filled-out handler functions would be the same as those in Listing 2-1.

When your compiler tool is finished, you will have the code in your mainframe class that is given in Listing 2-1b. You will be able to compile the code as given in Listing 2-1b. The code will compile with “stubbed-out” handler functions; there will be no compiler errors, but of course your program will not function as you desire it to until these handler functions have been finished. However, you will note that the popup menu items will no longer be grayed, since a handler function exists for those menu items.

Listing 2-1b: MenuA code After Adding Message Map Entries with the Compiler Tool

```
-----
                                mainframe class
-----
//
//  mainfrm.h          interface of the C_MainFrame class
//
class C_MainFrame : public CFrameWnd
```

```

{
    public:
        C_MainFrame();
    private:
        int m_nBeeps;
        //{AFX_MSG(C_MainFrame)
        afx_msg void OnTimer(UINT nIDEvent);
        afx_msg void OnFourBeeps();
        afx_msg void OnGreeting();
        afx_msg void OnOneBeep();
        //}}AFX_MSG

        DECLARE_MESSAGE_MAP()
};

//
//  mainfrm.cpp      implementation of the C_MainFrame class
//

#include "MyApp.h"
#include "mainfrm.h"
#include "resource.h"

BEGIN_MESSAGE_MAP (C_MainFrame, CFrameWnd)
    //{AFX_MSG_MAP(C_MainFrame)
    ON_WM_TIMER()
    ON_COMMAND(ID_FOUR_BEEPS, OnFourBeeps)
    ON_COMMAND(ID_GREETING, OnGreeting)
    ON_COMMAND(ID_ONE_BEEP, OnOneBeep)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

C_MainFrame::C_MainFrame()
{
    Create (NULL, "Menus - Example A", WS_OVERLAPPEDWINDOW,
            rectDefault, NULL, MAKEINTRESOURCE(IDR_MENU1));
    LoadAccelerTable(MAKEINTRESOURCE(IDR_ACCELERATOR1));
    m_nBeeps = 0;
}

void C_MainFrame::OnTimer(UINT nIDEvent)
{
    // TODO: Add your message handler code here and/or call default

    CFrameWnd::OnTimer(nIDEvent);
}

void C_MainFrame::OnFourBeeps()
{
    // TODO: Add your command handler code here
}

void C_MainFrame::OnGreeting()
{
    // TODO: Add your command handler code here
}

```

```
}  
  
void C_MainFrame::OnOneBeep()  
{  
    // TODO: Add your command handler code here  
}  
-----
```

Previous	Table of Contents	Next

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

An Example That Changes Menus and Submenus

The next example demonstrates a number of menu features. Changing the entire main menu while the program is running is demonstrated in Figure 2-3a and Figure 2-3b. Also explained is the changing of the submenu based on events occurring during the running of the program, and placing and removing a checkmark on menu items that have been selected or deselected is demonstrated.

When this program, MenuB, is first started a menu appears with the two items: “CursorShape” and “Install PositionMenu.” This program has two menus: a position menu (for the position of the cursor) and a cursor shape menu (for the shape of the cursor). When the program first opens, the cursor menu is the one showing, and it has the default cursor, the standard arrow, as shown in Figure 2-3a.



Figure 2-3a: Initial Executing Window of MenuB

If the position menu is installed prior to a cursor change, the menu changes in the specific way shown in Figure 2-3b. Choose the menu item “InstallPositionMenu” and the position menu will be installed. The position menu appears, with the position submenu items as shown. At this point in the program, the cursor has not been changed and is still the initial, standard cursor.



Figure 2-3b: Executing Window of MenuB with a Modified Main Menu

To return to the original menu (the cursor menu), choose the menu item “InstallCursorMenu.” When you select the main menu item “CursorShape,” a dropdown, or popup, menu appears with the selections: “Hourglass” and “UpArrow.” These items will be checked or unchecked (toggled “on” and “off”) as they are selected or deselected. (See Figure 2-3c.)



Figure 2-3c: Executing Window of MenuB with a Toggled Dropdown Menu

Once a cursor (other than the standard arrow) is chosen, the menus are altered when they are installed. Figure 2-3d shows the installed position menu after the hourglass cursor has been selected. Notice that when the position menu is installed, the menu’s title string is different. Also, the contents of the dropdown menu are different. When the hourglass cursor has been selected and the “Install PositionMenu” item is selected, the entire main menu is changed to contain the following two items: “InstallCursorMenu” and a second menu item, which has a different title depending on which cursor has been selected. If you have selected the hourglass cursor, it is titled “HourglassPosition,” and the following two items appear on the dropdown menu: “UpperLeft” and “LowerLeft.” (See Figure 2-3d.)



Figure 2-3d: Executing Window of MenuB with an Hourglass Cursor

If you have selected the uparrow cursor and have subsequently installed the position menu, which is the case shown in Figure 2-3e, then the second menu item will be titled “UpArrowPosition” and the two items on the dropdown menu are: “UpperRight” and “LowerRight.” As you can see, the menu is being dynamically altered as the program progresses and its alteration depends on the selected cursor shape.



Figure 2-3e: Executing Window of MenuB with an UpArrow Cursor

The MFC CMenu class is used because it provides member functions that allow you to load, create, alter, and destroy menu items. Four CMenu objects

are instantiated; the following CMenu functions are used: LoadMenu(), CreatePopupMenu(), AppendMenu(), DestroyMenu(), RemoveMenu(), and CheckMenuItem(). The use of the CMenu functions is discussed later. In addition, two CWnd class member functions that deal with a window's menu are used. These functions are SetMenu() and DrawMenuBar(). The use of these functions is discussed following the listing. Listing 2-2 (following) gives the source code for this example. Note that in this program CursorMenu and PositionMenu must be the same string that identified these menus in the resource file.

Note: Hungarian Notation is used in the following listing. Class data members have the prefix m_. BOOLEAN variables have a prefix of b; HANDLE variables have a prefix of h, etc. (Appendix A describes the Hungarian notation in more detail.)

Previous	Table of Contents	Next

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

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

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

Keyword

☒ Brief ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

Listing 2-2: Source Code for MenuB Program

```

-----
                                application class
-----

MyApp.h and MyApp.cpp have the same code as in Listing 1-1

-----
                                mainframe class
-----
//
//  mainfrm.h          interface of the C_MainFrame class
//
class C_MainFrame : public CFrameWnd
{
public:
    C_MainFrame();
    ~C_MainFrame();
private:
    CMenu CursorMenu;
    CMenu PositionMenu;
    CMenu m_AddHourglassMenu;
    CMenu m_AddUpArrowMenu;
    BOOL m_bHourglass;
    BOOL m_bUpArrow;
    HCURSOR m_hCurrentCursor;

    //{AFX_MSG(C_MainFrame)
    afx_msg void OnHourglass();
    afx_msg void OnUpArrow();
    afx_msg void OnInstallPositionMenu();
    afx_msg void OnInstallCursorMenu();
    afx_msg void OnUpperRight();
    afx_msg void OnLowerRight();
    afx_msg void OnUpperLeft();
    afx_msg void OnLowerLeft();
    afx_msg BOOL OnSetCursor(CWnd* pWnd, UINT nHitTest, UINT nMessage);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
}
  
```

```

//
//  mainfrm.cpp
//
#include "MyApp.h"
#include "mainfrm.h"
#include "resource.h"

BEGIN_MESSAGE_MAP (C_MainFrame, CFrameWnd)
   //{{AFX_MSG_MAP(C_MainFrame)
    ON_COMMAND(ID_HOURLASS, OnHourglass)
    ON_COMMAND(ID_UPARROW, OnUpArrow)
    ON_COMMAND(ID_INSTALL_POS_MENU, OnInstallPositionMenu)
    ON_COMMAND(ID_INSTALL_CURSOR_MENU, OnInstallCursorMenu)
    ON_COMMAND(ID_UPPER_RIGHT, OnUpperRight)
    ON_COMMAND(ID_LOWER_RIGHT, OnLowerRight)
    ON_COMMAND(ID_UPPER_LEFT, OnUpperLeft)
    ON_COMMAND(ID_LOWER_LEFT, OnLowerLeft)
    ON_WM_SETCURSOR()
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()

C_MainFrame::C_MainFrame()
{
    //do NOT attach menu in Create()
    Create(NULL, "Menus - Example B");
    CursorMenu.LoadMenu("CursorMenu");
    PositionMenu.LoadMenu("PositionMenu");
    SetMenu(&CursorMenu); // attach CursorMenu to window
    m_hCurrentCursor = AfxGetApp()->LoadStandardCursor(IDC_ARROW);
    m_bHourglass = FALSE;
    m_bUpArrow = FALSE;

    // create added popup menu for hourglass positions
    m_AddHourglassMenu.CreatePopupMenu();
    m_AddHourglassMenu.AppendMenu(MF_ENABLED, ID_UPPER_LEFT, "U&pperLeft");
    m_AddHourglassMenu.AppendMenu(MF_ENABLED, ID_LOWER_LEFT, "L&owerLeft");
    // create added popup menu for uparrow positions
    m_AddUpArrowMenu.CreatePopupMenu();
    m_AddUpArrowMenu.AppendMenu(MF_ENABLED, ID_UPPER_RIGHT, "&UpperRight");
    m_AddUpArrowMenu.AppendMenu(MF_ENABLED, ID_LOWER_RIGHT, "&LowerRight");
}

C_MainFrame::~C_MainFrame()
{
    CursorMenu.DestroyMenu(); // destroy menus
    PositionMenu.DestroyMenu();
    m_AddHourglassMenu.DestroyMenu();
    m_AddUpArrowMenu.DestroyMenu();
}

//-----Response functions to menu items-----

void C_MainFrame::OnHourglass()
{
    m_bHourglass = TRUE;
    m_bUpArrow = FALSE;
    CursorMenu.CheckMenuItem(ID_UPARROW, MF_BYCOMMAND | MF_UNCHECKED);
    CursorMenu.CheckMenuItem(ID_HOURLASS, MF_BYCOMMAND | MF_CHECKED);
    m_hCurrentCursor = AfxGetApp()->LoadStandardCursor(IDC_WAIT);
}

void C_MainFrame::OnUpArrow()
{

```



```

        m_bUpArrow = TRUE;
        m_bHourglass = FALSE;
        CursorMenu.CheckMenuItem(ID_HOURLASS, MF_BYCOMMAND | MF_UNCHECKED);
        CursorMenu.CheckMenuItem(ID_UPARROW, MF_BYCOMMAND | MF_CHECKED);
        m_hCurrentCursor = AfxGetApp()->LoadStandardCursor(IDC_UPARROW);
    }

void C_MainFrame::OnInstallPositionMenu()
{
    SetMenu(&PositionMenu);

    if (m_bHourglass)
    {
        PositionMenu.RemoveMenu(1, MF_BYPOSITION);
        PositionMenu.AppendMenu(MF_POPUP,
                                (UINT)m_AddHourglassMenu.m_hMenu,
                                "&HourglassPosition");
        DrawMenuBar();
    }
    if (m_bUpArrow)
    {
        PositionMenu.RemoveMenu(1, MF_BYPOSITION);
        PositionMenu.AppendMenu(MF_POPUP,
                                (UINT)m_AddUpArrowMenu.m_hMenu,
                                "&UpArrowPosition");
        DrawMenuBar();
    }
}

void C_MainFrame::OnInstallCursorMenu()
{
    SetMenu(&CursorMenu);
}

void C_MainFrame::OnUpperRight()
{
    RECT rect;
    GetClientRect(&rect);
    ClientToScreen(&rect);
    ::SetCursorPos( (rect.right - 60), (rect.top + 60) );
}

void C_MainFrame::OnLowerRight()
{
    RECT rect;
    GetClientRect(&rect);
    ClientToScreen(&rect);
    ::SetCursorPos( (rect.right - 60), (rect.bottom - 60) );
}

void C_MainFrame::OnUpperLeft()
{
    RECT rect;
    GetClientRect(&rect);
    ClientToScreen(&rect);
    ::SetCursorPos( (rect.left + 60), (rect.top + 60) );
}

void C_MainFrame::OnLowerLeft()
{
    RECT rect;
    GetClientRect(&rect);

```

```

        ClientToScreen(&rect);
        ::SetCursorPos( (rect.left + 60), (rect.bottom - 60) );
    }

//-----Response function for WM_SETCURSOR messages-----

BOOL C_MainFrame::OnSetCursor(CWnd* pWnd, UINT nHitTest, UINT nMessage)
{
    ::SetCursor(m_hCurrentCursor);
    return TRUE;
}

-----
                                resource files
-----
//
//  menub.rc      ---      resource script file for menub
//
#include "afxres.h"
#include "resource.h"

CursorMenu    MENU    DISCARDABLE
BEGIN
    POPUP "&CursorShape"
    BEGIN
        MENUITEM "&Hourglass",          ID_HOURLASS
        MENUITEM "&UpArrow",            ID_UPARROW
    END
    MENUITEM "Install&PositionMenu",      ID_INSTALL_POS_MENU
END
PositionMenu    MENU DISCARDABLE
BEGIN
    MENUITEM "&InstallCursorMenu",        ID_INSTALL_CURSOR_MENU
    POPUP "&CursorPosition"
    BEGIN
        MENUITEM "&UpperRight",          ID_UPPER_RIGHT
        MENUITEM "&LowerRight",          ID_LOWER_RIGHT
        MENUITEM "U&pperLeft",           ID_UPPER_LEFT
        MENUITEM "L&owerLeft",           ID_LOWER_LEFT
    END
END
//
//  resource.h    --    this program's resource IDs
//
#define ID_HOURLASS                40001
#define ID_UPARROW                 40002
#define ID_INSTALL_POS_MENU        40003
#define ID_INSTALL_CURSOR_MENU     40004
#define ID_UPPER_RIGHT             40005
#define ID_LOWER_RIGHT             40006
#define ID_UPPER_LEFT              40007
#define ID_LOWER_LEFT              40008
-----

```

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Using CMenu Objects

In the previous listing, the key item to note is the declaration of four CMenu objects: CursorMenu, PositionMenu, m_AddHourglassMenu, and m_AddUpArrowMenu. These objects are the foundation for the manipulation of these menus as the program operates.

The menu is not attached in the C_MainFrame() constructor call to CFrameWnd::Create(). Instead the menus are explicitly loaded by calling the CMenu::LoadMenu() function. CMenu::LoadMenu() loads the menu data out of the program's resources into memory. Then the desired menu can be attached to the program's main window by the CWnd::SetMenu() function. In the C_MainFrame() constructor, the CursorMenu is initially attached to the main window. In the response function to the menu item "InstallPositionMenu," which is OnInstallPositionMenu(), the PositionMenu will be attached to the main window using the CWnd function SetMenu(). Likewise, in the response function to the menu item "InstallCursorMenu," which is OnInstallCursorMenu(), the CursorMenu will be attached to the main window. Once the menu has been loaded and attached to the main window, it behaves just like any other menu.

In the C_MainFrame() constructor, the additional popup menus m_AddHourglassMenu and m_AddUpArrowMenu were created. The functions CMenu::CreatePopupMenu() and CMenu::AppendMenu() are used to create these menus. Once these submenus have been created, they are available to install onto the position menu. The appropriate submenu (depending on whether the hourglass cursor or the uparrow cursor has been selected) will be installed in the response function OnInstallPositionMenu(). This is achieved by removing the second menu item (in position 1) on the PositionMenu using the function CMenu::RemoveMenu(). The appropriate submenu is then appended to the main menu using the function

CMenu::AppendMenu(). Since these submenus change the appearance of the main menu bar, a call is made to CWnd::DrawMenuBar() to redraw the menu bar to show the new wording.

In this example a destructor ~C_MainFrame() must be used. In the destructor, the menus are destroyed using the function CMenu::DestroyMenu(). If the menu, including all popups, is attached to the window when the window is destroyed, the menu will automatically be removed from memory. If the menu is not attached to a window, it must be explicitly destroyed. Otherwise, the menu data will remain in memory for the duration of the Windows OS session.

The CMenu class also contains functions for menu-item options such as whether the menu item is checked, grayed, or inactive. In this program, the function CMenu::CheckMenuItem() is used to check and uncheck menu items. The use of this function can be seen in the handler functions OnHourglass() and OnUpArrow().

The handler functions OnUpperRight(), OnLowerRight(), OnUpperLeft(), and OnLowerLeft() sets the cursor to the respective position in the client area. In each of these handler functions, the client area rectangle is obtained through the call to GetClientRect(). The client area coordinates are then converted to screen coordinates by calling the ClientToScreen() function. The cursor is always in screen coordinates. The API function ::SetCursorPos() is then called, which sets the cursor to the desired position.

The payoff for using CMenu objects is the access to a variety of CMenu member functions. The CMenu member functions for creating menus and for menu-item operations are listed in the following tables, Tables 2-1 and 2-2.

Table 2-1: CMenu Member Functions for Creating Menus

<i>Function</i>	<i>Purpose</i>
AppendMenu()	Adds a new menu item or a popup menu to the end of a menu.
CreateMenu()	Creates a new menu, ready to add items.
CreatePopupMenu()	Creates a new popup menu, ready to add items.
DeleteMenu()	Deletes an item from the menu. If the menu item has an associated popup menu, it destroys the handle to the popup menu and frees the memory used by the popup menu.
DestroyMenu()	Deletes an entire menu, removing it from memory. This is necessary only if the menu has been loaded and is not attached to a window. Menus attached to windows are automatically removed from memory when the window is destroyed.
InsertMenu()	Inserts a new menu item or popup into a menu or popup menu.

LoadMenu()	Loads a menu from the program's resource data, ready to be attached to a window with CWnd::SetMenu(). You can define more than one menu in the resource data, and switch between menus as the program operates. LoadMenu() will load only one copy of each menu into memory no matter how often the function is called.
ModifyMenu()	Changes a menu item in place. This function is handy for changing the wording of a menu item.
RemoveMenu()	Removes a menu item including any associated popup menus from the menu. It does not destroy the handle for a popup menu, so the menu can be reused.

Table 2-2: CMenu Member Functions for Menu-Item Operations

Function	Purpose
CheckMenuItem()	Places a check mark next to, or removes a check mark from, a menu item in a popup menu.
EnableMenuItem()	Enables, disables, or dims (grays) a menu item.
GetMenuItemCount()	Determines the number of items in a popup or top-level menu.
GetMenuItemID()	Obtains the menu-item identifier for a menu item located at the specified position.
GetMenuState()	Returns the status of the specified menu item or the number of items in a popup menu.
GetMenuString()	Retrieves the label of the specified menu item.
GetSubMenu()	Retrieves a pointer to a popup menu.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

CWnd Functions for Messages

In the earlier sections of this chapter, we discussed the special Windows OS message WM_COMMAND and its message map. You saw that the WM_COMMAND message map entries were unique for that special message and that you could choose your own name for your handler functions. All other WM_ messages will be called by the term “regular Windows OS messages” or simply “Windows OS messages,” which is meant to exclude the special WM_COMMAND message.

Regular Windows OS messages have message handlers that are member functions of CWnd. Since the handlers are member functions of CWnd, each message has a name and prototype found in CWnd. When you use one of the regular Windows OS messages, you must use the name and prototype exactly as found in CWnd. You cannot choose your own message handler name (as you can with WM_COMMAND messages). You must use the exact prototype as found in CWnd. As you recall, MFC intercepts all messages destined for your window, interprets the message’s parameters, and then forwards the interpreted information as inputs to your message handler. The input parameters of your message handler are provided by MFC. You will receive these input parameters, whether you choose to use them or not in your handler function. Within the body of your message handler, you write the code to be executed when this message is sent by the Windows OS.

There are many regular Windows OS messages. The most frequently used messages from two message categories are given in Tables 2-3 and 2-4. In these tables, the WM_ message that “jumps,” or maps, to the CWnd handler function is given, the CWnd function prototype is given, and a brief description of the handler function is given. Table 2-3 gives messages that are categorized as *input messages*, that is, they are generated by keystrokes, mouse actions, the timer, etc. Table 2-4 gives messages that are categorized as

general messages. All regular Windows OS messages used in this book are found on these tables. In this chapter we used WM_TIMER and WM_SETCURSOR, which are listed on Table 2-3. In Chapters Three, Four, and Five, we will use WM_PAINT, WM_LBUTTONDOWN, WM_MOUSEMOVE, WM_LBUTTONUP, WM_CREATE, and WM_DESTROY. Only those messages that are supported by compiler tools are found on these tables. The compiler tool supports only the most frequently used messages. CWnd supports numerous messages not supported by the compiler tool. For example, CWnd supports the middle button clicks, WM_MBUTTONDOWN, WM_MBUTTONDOWN, and WM_MBUTTONUP, which are not supported by the compiler tool. If you wish to use one of the less frequently used messages, such as the middle button messages, you will need to add the code manually. Also, there are categories of messages that are not as frequently used which are not given here. These less frequently used categories are: *system messages*, *non-client area messages*, and *clipboard messages*. (Selected messages from these less frequently used categories can be found in the compiler tool.)

Table 2-3: Input Messages

<i>Message Description</i>	<i>CWnd Function Prototype and Message Handler</i>
WM_CHAR	void OnChar(UINT, UINT, UINT); called when a keystroke translates to a nonsystem character.
WM_HSCROLL	void OnHScroll(UINT, UINT,UINT); called when the user clicks the horizontal scroll bar of CWnd.
WM_KEYDOWN	void OnKeyDown(UINT,UINT,UINT); called when a nonsystem key is pressed.
WM_KEYUP	void OnKeyUp(UINT,UINT,UINT); called when a nonsystem key is released.
WM_LBUTTONDOWNDBLCLK	void OnLButtonDownDblClk(UINT, Cpoint); called when the user double-clicks the left mouse button.
WM_LBUTTONDOWN	void OnLButtonDown(UINT,CPoint); called when the user presses the left mouse button.
WM_LBUTTONUP	void OnLButtonUp(UINT, Cpoint); called when the user releases the left mouse button.
WM_MOUSEMOVE	void OnMouseMove(UINT,CPoint); called when the mouse cursor moves.
WM_RBUTTONDOWNDBLCLK	void OnRButtonDownDblClk(UINT, Cpoint); called when the user double-clicks the right mouse button.
WM_RBUTTONDOWN	void OnRButtonDown(UINT,CPoint); called when the user presses the right mouse button.
WM_RBUTTONUP	void OnRButtonUp(UINT,CPoint); called when the user releases the right mouse button.

WM_SETCURSOR	BOOL OnSetCursor(CWnd*, UINT, UINT); called if mouse input is not captured and the mouse causes cursor movement within a window.
WM_TIMER	void OnTimer(UINT); called after each interval specified in SetTimer.
WM_VSCROLL	void OnVScroll(UINT,UINT,CWnd*); called when the user clicks the window's vertical scroll bar.

Previous	Table of Contents	Next

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

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

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

Keyword

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes
(Publisher: Wordware Publishing, Inc.)
Author(s): Shirley Wodtke
ISBN: 1556225121
Publication Date: 01/01/97

Bookmark It

Search this book:

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

Table 2-4: General Messages

<i>Message Description</i>	<i>CWnd Function Prototype and Message Handler</i>
WM_ACTIVATE	void OnActivate(UINT, CWnd*, BOOL); called when CWnd is being activated or deactivated.
WM_ACTIVATEAPP	void OnActivateApp(BOOL, HANDLE); called when the application is about to be activated or deactivated.
WM_CANCELMODE	void OnCancelMode(); called to allow CWnd to cancel any internal modes, such as mouse capture.
WM_CLOSE	void OnClose(); called as a signal that CWnd should be closed.
WM_CREATE	int OnCreate(LPCREATESTRUCT); called as a part of window creation.
WM_DESTROY	void OnDestroy(); called when CWnd is being destroyed.
WM_ENDSESSION	void OnEndSession(BOOL); called when the session is enabled or disabled.
WM_ENTERIDLE	void OnEnterIdle(UINT, CWnd*); called to inform an application's main window procedure that a modal dialog box or a menu is entering an idle state.
WM_ERASEBKGND	BOOL OnEraseBkgnd(CDC*); called when the window background needs erasing.

WM_GETMINMAXINFO	void OnGetMinMaxInfo(LPPOINT); called whenever Windows OS needs to know the maximized position or dimensions, or the minimum or maximum tracking size.
WM_ICONERASEBKGND	void OnIconEraseBkgnd(CDC*); called when CWnd is minimized and the background of the icon must be filled before painting the icon.
WM_KILLFOCUS	void OnKillFocus(CWnd*); called immediately before CWnd loses the input focus.
WM_MOVE	void OnMove(int, int); called after the position of the CWnd has been changed.
WM_PAINT	void OnPaint(); called to repaint a portion of the window.
WM_QUERYENDSESSION	BOOL OnQueryEndSession(); called when the user chooses to end the Windows OS session.
WM_QUERYNEWPALETTE	BOOL OnQueryNewPalette(); informs CWnd that it is about to receive input focus, so it can realize its palette.
WM_SETFOCUS	void OnSetFocus(CWnd*); called after CWnd gains the input focus.
WM_SHOWWINDOW	void OnShowWindow(BOOL, UINT); called when CWnd is to be hidden or shown.
WM_SIZE	void OnSize(UINT, int, int); called after the size of CWnd has changed.

Summary

Menus and mouse clicks provide user input to an application. Menu scripts are created and stored in the resource files, then attached to the main window using the `CFrameWnd::Create()` function. Menu items are activated by a mouse click or by using a keyboard accelerator. Keyboard accelerator tables are created and stored in the resource files, then loaded into memory, ready for use, with the `CFrameWnd::LoadAccelTable()` function. The functions which attach or load the resources can accommodate resources identified either as a string or as an int; this is done either through overloading the function or by using the macro `MAKEINTRESOURCE`, the method depends on the function. Message boxes (used in the first example in this chapter) are very useful modal dialog boxes; they are simple to program and can display small amounts of information important for the user to see.

Message maps contain “entries” or macros that map incoming messages to their handler functions. If the message was generated by a click on a menu item or by using a keyboard accelerator, the incoming message will be a `WM_COMMAND` message; its message map entry will be the macro `ON_COMMAND()`, which maps an incoming message identified by its menu

item ID to its handler function. Messages other than the WM_COMMAND message, e.g., WM_TIMER, are sent via message map entries, e.g., ON_WM_TIMER(), to their handler function which is a CWnd function inherited by your window class. The ClassWizard tool can be used to speed the process of adding message map entries and handler functions to your program; however, to do this, your files need have a .h file and a .cpp file and specific lines of code need to be present in your code to enlist the services of the compiler tool.

The CMenu class is used to provide menus that can be altered based on internal events occurring during run time. The entire main menu can be changed, submenus and submenu items can be changed, the text in any menu item can be changed, menu items can be checked and unchecked, etc.

Handler functions for “ordinary” messages (all messages other than the WM_COMMAND message) are handled by CWnd functions that are inherited by your window class. To use these messages, you must use the exact CWnd prototype in your .h file; the MFC framework will provide inputs to your handler function whether you use them or not. There are CWnd functions not supported by ClassWizard, which, if needed, can be used by manually inserting them in your code.

Exercise

In this chapter, you learned about menus, accelerators, and message handler functions. The exercise is to continue building your checkers game. Your checkers game will have menu selections for the colors of the board and pieces as well as selections for the shape of the pieces. Now you can create and add the menu to your program. You can provide accelerators if you wish. You can check and uncheck the menu item selections. You can add entries to the message maps and provide “stubbed-out” handler functions either coding the message map entries manually or using the services of ClassWizard. Refer to the section titled “Chapter 2 Exercise” of Appendix B for more information.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Chapter 3

Graphics and Text Drawing

For many applications, the main way the application communicates with the user is through the client area of its window. This chapter presents a set of functions for drawing shapes and characters in the client area of a window; these functions are provided by the Windows OS. Furthermore, Windows OS also lets an application draw bitmaps by creating a block of memory, called a *memory device context*, that holds a rectangular image. The drawing functions can draw on the memory device context. (Bitmaps and memory device contexts are covered in Chapter Four.) Also, Windows OS generalizes this process so that if the hardware supports it, these same things can be drawn on printers or other output devices using the same function calls. (Printing is covered in Chapter Ten.)

The Graphics Device Interface (GDI)

The set of functions for doing drawing is called the GDI (Graphics Device Interface), which is a DLL supplied with the operating system. Anytime your program draws directly on the display or printer, it must use the GDI functions. There are two parts of Windows OS that do the conversion from the graphics function calls to the actual commands sent to the hardware. One part is the GDI, or Graphics Device Interface. Most of the GDI resides in a program file called GDI.EXE that is stored in the Windows OS System directory. The Windows OS environment will load GDI.EXE into memory when it is needed for graphical output. The operating system will also load a device driver program if the hardware conversions are not part of GDI.EXE. Drivers are just programs that assist the GDI in converting graphics commands to hardware commands. Common examples of drivers are VGA.DRV for a

The Device Context

The Windows operating system provides *device independence* through the *device context* concept. This means that your program should be able to work using different keyboards, screens, and printers without modification to the program. The Windows operating system takes care of the interface with the hardware, allowing the programmer to concentrate on the program itself.

Windows OS programs do not send data directly to the screen or printer. Instead, the program obtains a handle for the screen or printer's device context. The output data is sent to the device context, and then the operating system takes care of sending it to the hardware. The advantage of using the device context is that the graphics and text commands you send to the device context are always the same, regardless of whether the physical output is showing up on a VGA screen, printer, or some new device that was invented after you completed your program.

The device context is a fundamental concept of the Windows operating system. A device context represents one of the following: the video display, the printer or plotter, or a memory device. In addition to representing a specific output device, the device context holds information that specifies certain aspects of the appearance of drawn objects.

The device context is a set of data (a struct, to be exact) maintained by Windows OS and identified by a device context handle. A device context not only represents the destination entity for the drawing operations, but also holds information about the drawing objects and parameters that specify the modes that the GDI drawing functions use when drawing to the device context. For example, the text color drawing parameter specifies the color with which characters are drawn. Functions are provided for setting the drawing parameters for a device context. These functions are in the CDC class and are discussed later in this chapter.

The device context holds information about drawing objects and drawing parameters that will be used when drawing into the device context. These are collectively called the *drawing attributes* of the device context. The drawing attributes and their default values are listed in Table 3-1. In the examples given in this book, most of these device context drawing attributes will be used.

Table 3-1: Device Context Drawing Attributes

<i>Attribute</i>	<i>Default Value</i>
Color palette handle	Default palette
Brush handle	Solid white
Brush origin	0,0 (for aligning brush patterns)
Pen handle	Solid black, one pixel wide
Current pen position	0,0
Font handle	System font
Bitmap handle	None

Clipping region handle	The whole region
Background color	White (for the interstices when drawing hatch brushes and characters in OPAQUE mode)
Background mode	OPAQUE (for hatch brushes and characters)
Drawing mode	R2_COPYPEN (new graphics replace previous graphics)
Text alignment	Top left
Text color	Black
Text justification	0 break, 0 char
Spacing between characters	0
Polygon fill mode	Alternate
Stretching mode	Black on white
Mapping mode	MM_TEXT (1:1 conversion between coordinates passed to drawing functions and device pixels; origin is upper left and coordinates increase down and to the right.)
†Window extent	1,1
†Window origin	0,0
†Viewport extent	1,1
†Viewport origin	0,0

†This attribute may be redefined if you wish to have other than a direct mapping between the logical window coordinates and the physical viewport coordinates.

The device context obtained for a video display ordinarily comes from a global pool that holds a limited number of device contexts for use by all the programs that are running. To avoid exhausting this pool, applications should obtain a display device context immediately before drawing and relinquish it immediately afterward. If the pool of global device contexts is exhausted, MFC will inform the user.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

The device context obtained is a data structure that is initialized with default attributes. If the application changes any device context attribute values, these changes are lost when the application releases or deletes the device context. There are two companion CDC functions, `SaveDC()` and `RestoreDC()`, which can be used to save the device context's current state by copying the values of the drawing attributes (such as selected objects, mapping mode, etc.) to a special stack maintained by Windows OS. The saved device context states can later be restored by using the companion function `RestoreDC()`. With these functions there will still be the time consumption of saving and restoring device context settings, but some efficiency can be realized with these functions. (These functions are not used in this chapter, since you first must master the fundamental steps of using the device context.)

There is a way to totally avoid the time consumption involved in obtaining a device context and customizing attribute values each time the application needs to draw. This is done by using a private device context. This saves time for programs that frequently perform small drawing updates. The use of private device contexts does run the risk of exhausting the GDI's internal stack, especially if you use several of them. (Using a private device context is covered in the last section of this chapter.)

The device context contains descriptive information about its output device (video display, printer, etc.) This information can be obtained by a call to the CDC function `GetDeviceCaps(nindex)`, where `nindex` indicates the kind of information desired. For example, the function call `GetDeviceCaps(PLANES)` returns the number of color planes per pixel. You can find out if a printer can print graphics, if the video display is monochrome or has 16, 256, or 6 million colors, etc.

GDI Objects

The device context contains handles to *drawing objects*. (The term object is

used to describe a drawing object, but a drawing object is not the same as a C++ object.) The drawing objects are: a brush, a pen, a font, a bitmap (used with memory device contexts only), a palette, and a region (used for drawing and clipping).

The MFC class `CGdiObject` provides a base class for the drawing objects. You never create a `CGdiObject` object directly. Rather, you create an object from one of its derived classes, which are: `CPen`, `CBrush`, `CFont`, `CBitmap`, `CPalette`, and `CRegion`. These are described in Table 3-2. (The classes `CPen`, `CBrush`, and `CFont` are discussed in this chapter. The classes `CBitmap` and `CPalette` are discussed in Chapter Four.)

Table 3-2: Classes Derived from CGdiObject

<code>CPen</code>	A pen is a tool for drawing lines and shape borders. You can specify a pen's color and thickness and whether it draws solid, dotted, etc. lines.
<code>CBrush</code>	A brush is used to fill areas with color. You can specify the brush's color, if it is solid, or you can specify a bitmap pattern for your brush. Brushes can also have hatched styles.
<code>CFont</code>	A font is a collection of characters of a particular typeface and size. Fonts are generally stored on disk as resources.
<code>CBitmap</code>	A bitmap is an array of bits in which one or more bits correspond to each pixel. You can use bitmaps to create brushes.
<code>CPalette</code>	A palette is a color mapping interface that allows an application to take full advantage of the color capability of an output device without interfering with other applications.
<code>CRegion</code>	A region is an area that is a combination of polygons and ellipses. You can use regions for filling and clipping.

Never construct an object of class `CGdiObject`; rather construct objects of the derived classes. Constructors for some GDI derived classes, such as `CPen` and `CBrush`, allow you to specify enough information to create the object in one step. Others, such as `CFont`, can only be constructed in two steps. For these classes, you first construct a C++ object with the default constructor, and then you call a create function such as `CreateFont()`. The `CGdiObject` class has a virtual destructor. The derived destructor deletes the GDI object that is attached to the C++ object. The rules governing the use of GDI objects are: 1) If you construct a GDI object, you must delete it prior to exiting the program; 2) Don't delete GDI objects while they are selected into a device context. (You must first separate the GDI object from the device context before you delete the GDI object.)

Device Context Settings

Table 3-1 presents default settings for device contexts when they are first

obtained. You can customize, or change, the device context settings to fit your needs. The device context holds and remembers the current settings for the character font, character color, background color, background mode, etc. In the case of a drawing object, the device context is changed by selecting the drawing object into the device context using the function CDC::SelectObject(). This just means that the handle to the drawing object is made available to the device context for immediate use. The following table (Table 3-3) shows some CDC functions that are used to change the device context settings; most of these functions are presented in the examples that are given in this chapter and in Chapter Four. (There are many more CDC functions not used in these two chapters.)

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes
(Publisher: Wordware Publishing, Inc.)
Author(s): Shirley Wodtke
ISBN: 1556225121
Publication Date: 01/01/97

Bookmark It

Search this book:

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

Table 3-3: Changing Device Context Settings	
<i>Setting</i>	<i>Selected By and Function Description</i>
Text color	SetTextColor() sets the color of the text.
Background color	SetBkColor() sets the background color that surrounds each character.
Background mode	SetBkMode() sets the painting of text and graphics if they cover everything underneath (OPAQUE) or if the white areas are transparent (TRANSPARENT).
Stock Font	SelectStockObject() sets the font used by text output functions.
Pen	SelectObject() sets the pen used to draw lines and outlines of objects.
Stock Pen	SelectStockObject() selects predefined stock pens.
Brush	SelectObject() selects the brush pattern used to fill the interior of objects.
Stock Brush	SelectStockObject() selects predefined stock brushes.
Raster Drawing Mode	SetROP2() determines how the existing pixels on the screen are combined with the new pixels when drawing with pens and brushes.
Polygon Fill Mode	SetPolyFillMode() determines how polygons are filled. (This only changes the image if the polygon lines cross.)
Clipping Region	SelectClipRgn() limits the area within which output will be displayed.

As demonstrated in Table 3-3, the function SelectObject() has been overloaded for every type of GDI object. SelectObject returns a pointer to the type of GDI object that is currently selected into the device context, i.e., it will return a

CPen* for a CPen, a CBrush* for a CBrush, etc. This pointer is temporary and may only be stored as an automatic variable. SelectObject() has all of the following prototypes:

```
CPen* CDC::SelectObject(CPen* p)
CBrush* CDC::SelectObject(CBrush* b)
CFont* CDC::SelectObject(CFont* f)
CBitmap* CDC::SelectObject(CBitmap* bm)
CRgn* CDC::SelectObject(CRgn* r)
```

Tip: If you need to store the pointer to the GdiObject as a persistent variable, then you must use the function CGdiObject::GetSafeHandle().

Tip: When a new GDI object is selected into the device context, it is common practice to save, as an automatic variable, the pointer to the old GDI object that was displaced from the device context. This old GDI object will then be selected back into the device context in order to displace the new GDI object for deleting. An alternative to this procedure is to select a stock object into the device context in order to displace the new GDI object from the device context for deleting.

Stock Drawing Objects

The operating system maintains a number of useful stock drawing objects. If a stock drawing object is used, it does not have to be deleted. In fact, if you try to delete it, the operating system ignores your request. If there is an available stock drawing object that meets your requirements, then the function SelectStockObject(index) can be used to select the stock object into the device context.

Using a stock object has the advantage of convenience; it is already defined and does not have to be deleted when it is no longer required. To use a stock object, supply the identifying number of the stock object to the SelectStockObject(index) function as its index argument. The available stock objects and their identifiers are listed in Table 3-4. (The identifying number is a preprocessor constant supplied by Windows OS.)

Table 3-4: Stock Drawing Objects

<i>Identifier</i>	<i>Meaning</i>
BLACK_BRUSH	Solid black brush
DKGRAY_BRUSH	Dark gray brush
GRAY_BRUSH	Gray brush
NULL_BRUSH	Null brush (filling not performed)
HOLLOW_BRUSH	Same as NULL_BRUSH
LTGRAY_BRUSH	Light gray brush
WHITE_BRUSH	White brush (a default)
BLACK_PEN	One-pixel-wide, solid, black line (a default)
NULL_PEN	No line is drawn

WHITE_PEN	One-pixel-wide, solid, white line
†ANSI_FIXED_FONT	ANSI fixed-pitch font
†ANSI_VAR_FONT	ANSI variable-pitch font
DEVICE_DEFAULT_FONT	The default font for a device. (Tip: The built-in font for a printer can be used for faster output to a printer, if this is a hardware printer font.)
OEM_FIXED_FONT	Built-in fixed-pitch font on a printer
SYSTEM_FONT	Default font for captions and menus
DEFAULT_PALETTE	Default color palette

†ANSI is a type of ASCII that has accented letters for many of the codes above 0x80. Windows OS uses ANSI internally.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

The CDC Class

All drawing is performed through the member functions of the CDC object. CDC provides the drawing functions: Rectangle(), Ellipse(), Polygon(), PolyPolygon(), Chord(), RoundRect(), Pie(), LineTo(), Polyline(), and Arc(). The CDC class provides member functions for device-context operations, working with drawing tools, GDI object selection, and colors and palettes. It also provides member functions for getting and setting drawing attributes and mapping, working with the viewport, the window extent, and regions, converting coordinates, and clipping. Member functions are also provided for drawing text, working with fonts, using printer escapes, scrolling, and playing metafiles (metafiles are discussed in the following paragraphs).

The CDC class also provides for device-context objects. The CDC class provides member functions, as well as data members for working with a display context associated with a window. CDC contains data members for the two device context handles used in MFC. These are: m_hDC, which is the handle to the output-device context, and m_hAttribDC, which is the handle to the attribute-device context. On instantiation of a CDC object, these two handles refer to the same device. CDC directs all output GDI calls to m_hDC and most attribute GDI calls to m_hAttribDC. An example of an attribute call is GetTextColor(); SetTextColor() is an example of an output call.

The Device Context Classes

MFC provides several device context classes derived from CDC, which are: CClientDC, CPaintDC, CWindowDC, and CMetaFileDC. All of the output functions, like TextOut(), are inherited from the CDC base class, so the device context classes use the same set of inherited functions.

CClientDC is designed to initialize a device context to the settings for a computer screen. If a CClientDC object is created, a device context that is mapped only to the window's client area is created—drawing cannot occur outside of it. The point (0,0) usually refers to the upper left corner of the client area. The CClientDC class works everywhere except in OnPaint() functions.

When processing the WM_PAINT message in an OnPaint() function, the specialized CPaintDC class must be used. CPaintDC encapsulates calls to BeginPaint() and EndPaint().

When an object of class CWindowDC is constructed, point (0,0) is at the upper left corner of the non-client area of the window. With this whole window device context, drawing can occur in the window's border, caption area, etc.

CMetafileDC is a specialized device context class for use with *metafiles*. Metafiles consist of files of code for drawing to the screen. It frequently requires less memory to store the code, rather than storing the bitmaps of the results produced on the screen. When this is done, the files are called metafiles and producing the screen results is called “playing the metafiles.”

In MFC, your device context object will take care of obtaining the device context. The device context object will also take care of releasing the device context. Most frequently, you'll construct a device context object

inside a message handler function. When your C++ device context object goes out of scope, it takes care of releasing the device context.

An Example That Draws Text and Shapes

The first example of this chapter illustrates the use of the device context and drawing functions. Figure 3-1, following, shows the output of this example program, which is named Hello. When the “DrawText” menu item is selected, a rectangle is drawn, and the text “Hello, World!” is drawn in red onto the rectangle. Also, an ellipse is drawn, the font is changed, and white text is drawn on a black background inside the ellipse. This program demonstrates the use of the class CClientDC and some of its inherited member functions. The drawing functions CDC::Rectangle() and CDC::Ellipse() are also demonstrated. Both of these drawing functions take as input: *intxl*, *intyt*, *intxr*, *intyb*, where (*xl*,*yt*) is the left-top pixel and (*xr*,*yb*) is the right-bottom pixel of the rectangle in client-area coordinates. In the case of the ellipse, these inputs define the bounding rectangle inside which the ellipse is drawn.



Figure 3-1: Execution of Program “Hello”

The program listing for this example, which is named Hello, is given in Listing 3-1.

Listing 3-1: Source Code for the “Hello” Program

```
-----
                                application class
-----

    MyApp.h and MyApp.cpp have the same code as in Listing 1-1

-----
                                mainframe class
-----
//
//  mainfrm.h
//
class C_MainFrame : public CFrameWnd
{
    public:
        C_MainFrame();
    private:
        void OnDrawText();

    DECLARE_MESSAGE_MAP()
};

//
//  mainfrm.cpp
//
#include "MyApp.h"
#include "mainfrm.h"
#include "resource.h"

BEGIN_MESSAGE_MAP(C_MainFrame, CFrameWnd)
    ON_COMMAND(ID_DRAWTEXT, OnDrawText)
END_MESSAGE_MAP()

C_MainFrame::C_MainFrame()
{
    Create( NULL, "Hello World Example", WS_OVERLAPPEDWINDOW,
        rectDefault, NULL, MAKEINTRESOURCE(IDR_MENU1));
}
```

```

}

void C_MainFrame::OnDrawText()
{
    CClientDC MyDc (this);

    MyDc.Rectangle(25, 25, 200, 100);           // draws rectangle
    MyDc.SetTextColor(RGB(255, 0, 0));          // sets text color to red
    MyDc.TextOut( 50, 50, "Hello, World!");      // draws text on rectangle
    MyDc.Ellipse(250, 25, 450, 100);           // draws ellipse
    MyDc.SetTextColor(RGB(255, 255, 255));       // sets white text color
    MyDc.SetBkColor(RGB( 0, 0, 0));             // sets black bkg color
    MyDc.SelectStockObject(ANSI_FIXED_FONT);     // select new stock font
    MyDc.TextOut( 300, 50, "Hello, World!");     // draws text on ellipse
}

-----
                                resource files
-----
//
//  hello.rc
//
#include "resource.h"

IDR_MENU1      MENU
BEGIN
    MENUITEM "&DrawText",  ID_DRAWTEXT
END

//
//  resource.h
//
#define IDR_MENU1          101
#define ID_DRAWTEXT       40001
-----

```

The preceding program, Hello, draws text and shapes onto the screen. A CClientDC device context object was used to draw in the client area of the window. The CClientDC constructor function is passed a pointer to the window object which will receive the graphics output. The line

```
CClientDC MyDc(this);
```

Previous	Table of Contents	Next

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

passes the pointer of the window to the constructor function for the CClientDC class. The *this* pointer points to the C_MainFrame class object, since *this* is used within the body of the C_MainFrame class function. The CClientDC object is given the name MyDc. The device context is obtained by the MyDc object when it is constructed. The device context when newly obtained will have the default values given in Table 3-1. After the CClientDC object is created, all sorts of output to the screen can be performed, using the functions inherited by object MyDc. In the preceding “Hello” program, a rectangle using the default values for the pen and brush are first drawn. Next, the text color setting of the device context is changed to red, and a line of text is drawn on the rectangle. Then, using the default values which had not been changed for the pen and brush, an ellipse is drawn. The text color setting is changed to white and the text background color is changed to black. Subsequently, a line of text is drawn on the ellipse. At the end of the handler function, the C++ object MyDc goes out of scope; its destructor is automatically called, and the MyDc destructor will take care of releasing the device context.

An Example That Sets the Viewport Origin

In the MM_TEXT mapping mode, which is the default mapping mode, coordinates map to pixels. Values of x increase as you move right, and values of y increase as you move down, but you are allowed to change the origin through calls to the CDC function SetViewportOrg(). The next code example, named HelloB, sets the viewport origin to (25,25) and draws the same rectangle as was drawn in the first program Hello. This has the effect of shifting the origin of the client area to (25,25). The executing program is shown in Figure 3-2.

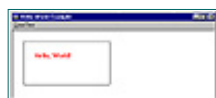


Figure 3-2: Execution of Program “HelloB”

The program listing for HelloB is given in Listing 3-2, which shows only the functions that have been changed from Listing 3-1. (The only part of the code that changes is the handler function C_MainFrame::OnDrawText(), which is shown in the following listing.)

Listing 3-2: Changes to Hello in Source Code for “HelloB”

```
//
//  mainfrm.cpp
//

. . . . .

void C_MainFrame::OnDrawText()
{
    CClientDC MyDc (this);

    MyDc.SetViewportOrg(CPoint(25,25));
```

```

MyDc.Rectangle(0, 0, 200, 100);           // draws rectangle
MyDc.SetTextColor( RGB(255, 0, 0));       // sets text color to red
MyDc.TextOut( 25, 25, "Hello, World!");   // draws text on rectangle
}

```

As can be seen from the preceding example, when the viewport origin is set to the point (25,25), drawing begins at point (25,25). To draw the rectangle in the same place (25,25) as it was drawn in the previous Hello example, the left-top pixel of the rectangle's inputs must be set to (0,0). As you can see from this example, setting the viewport origin to some value, (25,25) in our example, essentially resets the origin of the client area to that value. Thus, in our example, the client area origin has been reset to (25,25).

Tip: Changing the viewport origin can be very useful when drawing to and from a memory device context (covered in Chapter Four) or when printing (covered in Chapter Ten).

How a Screen Repaints Itself

In the preceding two examples, a CClientDC object was used to draw in the client area. This approach has the shortcoming that when the window is resized, the graphics are not repainted; in the preceding examples, they must be restored by again choosing the menu item. Using a CPaintDC object and putting the graphics calls into the OnPaint() function changes this behavior, so that the graphics are redrawn each time a WM_PAINT message is sent. This is done in all subsequent examples.

Windows OS sets the paint flag whenever a window gains one or more pixels. Windows OS sends a WM_PAINT message to the window whenever the client area needs to be repainted. When there are no higher-priority messages to be processed, the WM_PAINT message is processed and the window is repainted. The WM_PAINT message will be sent whenever any of the following events occur:

- The window has one or more invalid regions. (This is caused by one or more previously hidden areas of the window being brought back into view. Your window is brought back into view when a window that was above your window is closed or moved. Windows OS keeps track of all the invalid regions since the last repainting occurred and will repaint the invalid rectangle that encloses all invalid regions.)
- The user resizes the window.
- Part of the client area is scrolled.
- The application invalidates the client area, using one of the function calls: CWnd::Invalidate(), CWnd::InvalidateRect(), or CWnd::InvalidateRgn().

Creating a Pen

Pen drawing objects are used for drawing the outlines of the shapes drawn by the functions Rectangle(), Ellipse(), Polygon(), PolyPolygon(), Chord(), RoundRect(), Pie(), LineTo(), Polyline(), and Arc(). Pens can specify lines of any pixel width.

The pen's line will be centered on the outline boundary. Brush colors will normally be dithered to match the specified RGB color, but pens use the closest palette color. There is one exception to this rule; if the PS_INSIDEFRAME pen style is used, then dithered colors can be used for the outline, and the outline is drawn just up to the outline boundary from the inside.

Custom pens can be solid, dashed, or dotted in a variety of combinations. The functions CDC::CreatePen() and CDC::CreatePenIndirect() are used to create custom pens.

When a CPen object is constructed, the pen style, the pen width in pixels, and the color must be specified. The parameter pen style can be one of the styles shown in Table 3-5.

Table 3-5: Pen Styles

<i>Pen Style</i>	<i>Description</i>
PS_SOLID	Creates a solid pen.
PS_DASH	Creates a dashed pen. (Valid only when the pen width is 1.)
PS_DOT	Creates a dotted pen. (Valid only when the pen width is 1.)

PS_DASHDOT	Creates a pen with alternating dashes and dots. (Valid only when the pen width is 1.)
PS_DASHDOTDOT	Creates a pen with alternating dashes and double dots. (Valid only when the pen width is 1.)
PS_NULL	Creates a null pen.
PS_INSIDEFRAME	Creates a pen that draws a line inside the frame of closed shapes or inside lines.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

Pens can be created in a one-step construction, or they can be created with a two-step construction. A logpen can also be constructed, which is a structure that has members that contain the pen style, color, and width. A pen can be constructed in any one of the following ways:

1. Provide the necessary three arguments to the constructor:

```
CPen RPen(PS_SOLID, 3, RGB(255, 0, 0)); //solid red pen-3 pixels
```

2. Use the constructor with no arguments; then initialize the resulting CPen object using its member function CreatePen():

```
CPen RPen;
RPen.CreatePen(PS_SOLID, 3, RGB(255,0,0)); //solid red pen-3 wide
```

3. Use the constructor with no arguments; then initialize the resulting CPen object using its member function CreatePenIndirect(), which takes as its argument a pointer to the LOGPEN structure that contains information about the pen:

```
LOGPEN Pen_kind;
Pen_kind.lopnStyle = PS_SOLID;
Pen_kind.lopnWidth.x = 3;
Pen_kind.lopnColor = RGB(255, 0, 0);
CPen RPen;
RPen.CreatePenIndirect(&Pen_kind); // solid red pen of 3 pixels
```

A LOGPEN structure has the following form:

```
typedef struct tagLOGPEN{
    UINT        lopnStyle;
    POINT        lopnWidth;
    COLORREF     lopnColor;
} LOGPEN;
```

Creating a Brush

Brushes can be solid, hatched, or patterned. Only solid and hatched brushes are used in this chapter.

A solid brush is a solid color in that it has no crosshatches or bitmap pattern. When the RGB macro is used to specify the color, then the GDI will produce a dithered color if the specified color does not exactly match a color in the default palette. (The default palette is described in Chapter Four.) Dithering consists of drawing adjacent eight-pixel by eight-pixel squares to fill the drawing area, where each square has a pattern of colors

such that the average color in the square is the color specified by the RGB macro.

A hatch brush can be created to produce a pattern of: closely spaced horizontal, vertical, or diagonal lines; a horizontal-vertical crosshatch; or a diagonal crosshatch. A bitmap-pattern brush draws adjacent eight-pixel by eight-pixel squares to fill the drawing area. The pattern is taken from a bitmap, the handle of which is supplied to the brush creation function. (Bitmaps are discussed in Chapter Four.)

Brush drawing objects specify the way that the filled areas of filled figures are drawn when the following functions are used: Polygon(), PolyPolygon(), Ellipse(), Chord(), Pie(), Rectangle(), RoundRect(), FillRect(), FillRgn(), and FloodFill(). The brush applies when these functions are called. Also, Windows OS uses a brush to fill the client area of a window just before it sends a WM_PAINT message to the window. This brush is called the background brush. This filling process is called erasing the client area. As discussed in Chapter One, the input variable hbrBackground of the AfxRegisterWndClass() function specifies the background brush when you register your window class. The window background color is usually white, since Windows OS sets it to white as the default value. However, as mentioned, the user can change it to any other color. Your application should respecify the background brush if it requires a particular background color or requires a hatch or bitmap pattern in the background of a window's client area.

The hatched parameter style for a custom brush can be any one of those given in Table 3-6.

Table 3-6: Hatched Brush Styles

<i>Hatch Style</i>	<i>Description</i>
HS_BDIAGONAL	Downward hatch (left to right) at 45 degrees
HS_CROSS	Horizontal and vertical crosshatch
HS_DIAGONAL	Crosshatch at 45-degrees.
HS_FDIAGONAL	Upward hatch (left to right) at 45 degrees
HS_HORIZONTAL	Horizontal hatch
HS_VERTICAL	Vertical hatch

Brushes can be created in many ways. The first set of ways to create a brush is based on using the CBrush overloaded constructors. The constructor with a single COLORREF argument constructs a solid brush with the specified color. The constructor with two arguments constructs a hatch brush. The first argument specifies the hatched style; the second argument specifies the color.

Creating a solid brush

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

Creating a hatched brush

```
CBrush MyBrush( HS_CROSS, RGB(255, 0, 0) );
```

The second set of ways to create a brush is based on instantiating the CBrush object using the constructor with no arguments, then initializing the CBrush object using one of its member functions: CreateSolidBrush(), which initializes a brush with the specified solid color; CreateHatchBrush(), which initializes a brush with the specified hatched pattern and color; or CreateBrushIndirect(), which initializes a brush with the style, color, and pattern specified in a LOGBRUSH structure.

Creating a solid brush:

```
CBrush MyBrush;  
MyBrush.CreateSolidBrush( RGB(255, 0, 0) );
```

Creating a hatched brush:

```
CBrush MyBrush;  
MyBrush.CreateHatchBrush( HS_CROSS, RGB(255, 0, 0) );
```

Using the LOGBRUSH structure:

```
LOGBRUSH Brush_kind;  
Brush_kind.lbStyle = BS_HATCHED;  
Brush_kind.lbColor = RGB(255, 0, 0);  
Brush_kind.lbHatch = HS_CROSS;  
CBrush MyBrush;  
MyBrush.CreateBrushIndirect(&Brush_kind);
```

A LOGBRUSH structure has the following form:

```
typedef struct tagLOGBRUSH{  
    UINT          lbStyle;  
    COLORREF      lbColor;  
    int           lbHatch;  
} LOGBRUSH;
```

where lbStyle can be BS_SOLID, BS_PATTERN (based on a bitmap), BS_NULL, BS_HATCHED, or BS_DIBPATTERN (based on a DIB). (If the lbStyle is BS_SOLID or BS_NULL, lbHatch is ignored.)

Previous	Table of Contents	Next

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

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

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

Keyword

☒ Brief
 ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

The RGB Macro

The operating system uses a COLORREF, which is a 32-bit (4-byte) number to represent colors. The data type COLORREF is a DWORD value declared in <windows.h>. Only the least significant three bytes are used to store color data. The most significant byte is used to distinguish between the RGB color model and color palettes for devices that support more than 16 colors.

The basic color data consists of three values that represent the intensity of the red, green, and blue contributions to that color. Each intensity value can vary from 0 to 255. This provides a possible 16,777,216 colors. RGB is a macro defined in <windows.h> that puts the red, green, and blue intensity values in the correct byte positions. RGB(0,0,0) encodes black, as black is created by using zero intensity for all three color elements. RGB(255,255,255) encodes white, which is the combination of maximum amounts of red, green, and blue colors. To create a red color use the following code:

```
COLORREF red;
red = RGB(255,0,0);
```

In many cases, you can skip declaring a COLORREF value and use the RGB macro right inside a function call. For example, to set the text color red, as was done in the previous programs in this chapter, use the following code:

```
CClientDC MyDc(this);
MyDc.SetTextColor(RGB(255,0,0));
```

With the default color model, Windows OS assumes that the device is limited to 20 system colors. If fewer than 20 are available, some of the 20 will be identical. This is the case for most VGA systems, which have only 16 colors. If a color value is specified with RGB() that does not match one of the system

colors, Windows OS will approximate the requested color by *dithering*. Dithering puts pixels of available colors in a pattern to create the illusion of a color between the two colors being mixed. The dithering logic is very fast, and it does not slow down painting appreciably.

Windows OS supports devices with more than 20 colors, via color palettes. The same RGB color encoding is used, but the high-order byte contains a code for either an index into a palette, or a color value.

The Raster Drawing Mode and SetROP2()

The *raster drawing mode* is a device context attribute. It is known as ROP2 (Raster Operation) and applies to drawing to the screen. It defines how colors from the brush, or pen, and the colors in the existing image on the screen, are to be combined. Its default value is R2_COPYPEN but may be set to another value by the function CDC::SetROP2(). If the raster drawing mode is not R2_COPYPEN, then the resulting color when drawing a shape (rectangle, ellipse, etc), a line, or a pixel, is modified. The modification occurs on a pixel-by-pixel basis, after each pixel value is emitted from the drawing function. The pixel value being drawn by these functions is logically combined with the value currently in that pixel in the display adapter (or in memory if drawing to a memory device context) to yield a new pixel value that is then placed in that pixel. This results in a color different from the one being drawn with.

The fill pixels that are actually drawn depend on the brush style and the background mode. If the brush has style BS_HOLLOW (the null brush), then there are no drawn fill pixels. Otherwise, if SetBkMode() has not been called or has most recently been called with the OPAQUE mode, or if the brush style is BS_SOLID, then the drawn fill pixels include all the pixels within the outline. If the brush style is BS_HATCHED and SetBkMode() has been called with the TRANSPARENT mode, then the drawn pixels are just those of the hatch lines (the interstices are not drawn).

The term shape area means the set of pixels defined, as follows: for a call to SetPixel(), shape area is that pixel; for a call to a line-drawing function, shape area is the set of pixels that form the line; for a filled shape, such as a rectangle, shape area includes the pixels of the outline (which is drawn according to the selected pen) and the fill pixels that are actually drawn. For each pixel in the shape area, the pixel is set to a new value that depends on the drawing mode. In general this new value depends on the pixel's present value and on the value being drawn into the pixel by the drawing function. If *D* represents the value in a given pixel and *P* represents the value being drawn into that pixel, then the new value of the pixel will be as given in Table 3-7 for the three ROP2 codes explored in this chapter and in Chapter Four. There are additional drawing modes, which are less commonly used and listed under the function CDC::SetROP2() in the reference books.

Table 3-7: ROP2 Codes of Special Interest

Drawing mode	New value that pixel is set to
R2_COPYPEN	<i>P</i> (The value being drawn.)

R2_XORPEN	$P \wedge D$ (The value being drawn <i>exclusive or</i> -ed to the previous pixel value; where two similar bits result in a zero, two dissimilar bits result in a one.)
R2_NOTXORPEN	$\sim(P \wedge D)$ (The value being drawn <i>exclusive nor</i> -ed to the previous pixel value; where two similar bits result in a one and two dissimilar bits result in a zero.)

The R2_XORPEN and R2_NOTXORPEN modes are commonly used for rubberbanding and crude animation, since any shape drawn twice in these modes disappears. Therefore, to make a shape disappear, draw the shape once to make it appear, then when you need to move it, draw it again in its old position. Draw the same shape in its new position to make it reappear as if moved. Of course, when the shape appears, its colors are modified by what was there. In particular, when exclusive or-ing onto a white background, the shape's colors are usually inverted. (In Chapter Four, we further explore fast drawing using these techniques.)

The raster drawing modes act on pixel values as sent to and retrieved from the display adapter. They do not act on the logical RGB values that Windows OS applications usually deal with. Thus the exact action of R2_XORPEN and R2_NOTXORPEN, for instance, will depend on the physical pixel values that Windows OS generates from the RGB logical color. For the 16-color mode, assuming the palette is not changed from the default, the pixel values and their corresponding RGB values are listed in Table 3-8.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

☒ Advanced

[Search](#)

☒ Search Tips

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Table 3-8: Pixel and RGB Values for 16 Basic Colors

Color	Pixel Value (Dec.)	(Binary)	Red	Green	Blue
Black	0	0 0 0 0	0	0	0
Dark Red	1	0 0 0 1	128	0	0
Dark Green	2	0 0 1 0	0	128	0
Olive	3	0 0 1 1	128	128	0
Dark Blue	4	0 1 0 0	0	0	128
Purple	5	0 1 0 1	128	0	128
Teal	6	0 1 1 0	0	128	128
Dark Gray	7	0 1 1 1	128	128	128
Gray	8	1 0 0 0	192	192	192
Red	9	1 0 0 1	255	0	0
Green	10	1 0 1 0	0	255	0
Yellow	11	1 0 1 1	255	255	0
Blue	12	1 1 0 0	0	0	255
Fuschia	13	1 1 0 1	255	0	255
Aqua	14	1 1 1 0	0	255	255
White	15	1 1 1 1	255	255	255

When drawing in the R2_XORPEN mode:

Red ^ White is Teal since $9 \wedge 15$ is 6.

Teal ^ White is Red since $6 \wedge 15$ is 9.

Olive ^ White is Blue since $3 \wedge 15$ is 12.

Blue ^ White is Olive since $12 \wedge 15$ is 3.

When drawing in the R2_NOTXORPEN mode:

$\sim(\text{Red} \wedge \text{White})$ is Red since ~ 6 is 9.

$\sim(\text{Teal} \wedge \text{White})$ is Teal since ~ 9 is 6.

$\sim(\text{Olive} \wedge \text{White})$ is Olive since ~ 12 is 3.

$\sim(\text{Blue} \wedge \text{White})$ is Blue since ~ 3 is 12.

The next example program allows you to explore the effects of the ROP2 drawing mode.

A Graphics Editor Example

In this section a graphics editor program is presented. With this graphics editor the graphics features that have been discussed can be explored; either a rectangle or ellipse shape can be chosen and drawn with a choice of pen colors and styles and also a choice of brush colors and styles. Either an OPAQUE or TRANSPARENT background mode can be chosen. The background mode choices have an instantaneous effect—the screen is immediately redrawn using the new setting. With a hatched brush the background below the hatch design is either obscured (OPAQUE), or the underlying colors show through the interstices (TRANSPARENT). Different ROP2 modes may also be chosen, and the screen will be immediately redrawn using the newly selected ROP2 mode. An example of the graphics editor output is shown in Figure 3-3.

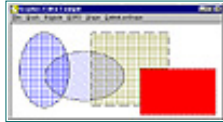


Figure 3-3: Output of Graphics Editor Program

The source code for the graphics editor program is given in Listing 3-3. Following the listing, the code is discussed.

Listing 3-3: Source Code for the Graphics Program

```
-----
                                application class
-----

    MyApp.h and MyApp.cpp have the same code as in Listing 1-1

-----
                                mainframe class
-----

//
//  mainfrm.h    interface of the C_MainFrame class
//

class C_Shape
{
    protected:
        CRect m_BoundingRect;
        LOGPEN  m_MyLogPen;
        LOGBRUSH m_MyLogBrush;
    public:
        void SetBoundingRect(int left, int top, int right, int bottom);
        CRect GetBoundingRect();
        void SetBottomRight(int right, int bottom);
        void SetParams(LOGPEN*, LOGBRUSH*);
        virtual void Draw(CPaintDC*) = 0;
};

class C_Rectangle : public C_Shape
{
    public:
        virtual void Draw(CPaintDC*);
};

class C_Ellipse : public C_Shape
```

```

{
    public:
        virtual void Draw(CPaintDC*);
};

class C_MainFrame : public CFrameWnd
{
    public:
        C_MainFrame();
        ~C_MainFrame();
    private:
        int m_nSape;
        BOOL m_bButtonDown;
        int m_BkMode;
        int m_ROP2;
        LOGPEN m_LogPen;
        LOGBRUSH m_LogBrush;
        enum EnumShape{eRectangle, eEllipse};
        EnumShape m_eShapeKind;
        #define MAX_SHAPES    100
        C_Shape* m_apShape[MAX_SHAPES];
        CRect m_RepaintArea;

        //{AFX_MSG(C_MainFrame)
        afx_msg void OnPaint();
        afx_msg void OnLButtonDown(UINT nFlags, CPoint Mouse);
        afx_msg void OnMouseMove(UINT nFlags, CPoint Mouse);
        afx_msg void OnLButtonUp(UINT nFlags, CPoint Mouse);
        afx_msg void OnPenGreen();
        afx_msg void OnPenFuschia();
        afx_msg void OnPenDash();
        afx_msg void OnPenDot();
        afx_msg void OnBrushRed();
        afx_msg void OnBrushTeal();
        afx_msg void OnBrushOlive();
        afx_msg void OnBrushBlue();
        afx_msg void OnBrushSolid();
        afx_msg void OnBrushCross();
        afx_msg void OnBrushHatchDiag();
        afx_msg void OnOpaque();
        afx_msg void OnTransparent();
        afx_msg void OnCopyPen();
        afx_msg void OnXORPen();
        afx_msg void OnNOTXORPen();
        afx_msg void OnRectangle();
        afx_msg void OnEllipse();
        afx_msg void OnDelete();
        //}}AFX_MSG

        DECLARE_MESSAGE_MAP()
};
//
//  mainfrm.cpp:      implementation of the C_MainFrame class
//
#include "MyApp.h"
#include "mainfrm.h"
#include "resource.h"

```

```

BEGIN_MESSAGE_MAP(C_MainFrame, CFrameWnd)
   //{{AFX_MSG_MAP(C_MainFrame)
    ON_WM_PAINT()
    ON_WM_LBUTTONDOWN()
    ON_WM_MOUSEMOVE()
    ON_WM_LBUTTONUP()
    ON_COMMAND(ID_PC_GREEN, OnPenGreen)
    ON_COMMAND(ID_PC_FUSCHIA, OnPenFuschia)
    ON_COMMAND(ID_PS_DASH, OnPenDash)
    ON_COMMAND(ID_PS_DOT, OnPenDot)
    ON_COMMAND(ID_BC_RED, OnBrushRed)
    ON_COMMAND(ID_BC_TEAL, OnBrushTeal)
    ON_COMMAND(ID_BC_OLIVE, OnBrushOlive)
    ON_COMMAND(ID_BC_BLUE, OnBrushBlue)
    ON_COMMAND(ID_BS_SOLID, OnBrushSolid)
    ON_COMMAND(ID_HS_CROSS, OnBrushCross)
    ON_COMMAND(ID_HS_FDIAGONAL, OnBrushHatchDiag)
    ON_COMMAND(ID_OPAQUE, OnOpaque)
    ON_COMMAND(ID_TRANSPARENT, OnTransparent)
    ON_COMMAND(ID_COPYPEN, OnCopyPen)
    ON_COMMAND(ID_XORPEN, OnXORPen)
    ON_COMMAND(ID_NOTXORPEN, OnNOTXORPen)
    ON_COMMAND(ID_RECTANGLE, OnRectangle)
    ON_COMMAND(ID_ELLIPSE, OnEllipse)
    ON_COMMAND(ID_DELETE, OnDelete)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////
///      Definitions of C_Shape, C_Rectangle, and C_Ellipse functions
///

void C_Shape::SetBoundingRect(int left, int top, int right, int bottom)
{
    CRect NewRect(left, top, right, bottom);
    m_BoundingRect = NewRect;
}

CRect C_Shape::GetBoundingRect()
{
    return m_BoundingRect;
}

void C_Shape::SetRightBottom(int right, int bottom)
{
    m_BoundingRect.right = right;
    m_BoundingRect.bottom = bottom;
}

void C_Shape::SetParams(LOGPEN* pPen, LOGBRUSH* pBrush)
{
    m_MyLogPen = *pPen;
    m_MyLogBrush = *pBrush;
}

```

```

void C_Rectangle::Draw(CPaintDC* pMyDc)
{
    CPen NewPen;
    CBrush NewBrush;
    NewPen.CreatePenIndirect(&m_MyLogPen);
    NewBrush.CreateBrushIndirect(&m_MyLogBrush);
    CPen* pOldPen = pMyDc->SelectObject(&NewPen);
    CBrush* pOldBrush = pMyDc->SelectObject(&NewBrush);
    pMyDc->Rectangle(&m_BoundingRect);
    pMyDc->SelectObject(pOldPen);        // displace drawing objects
    pMyDc->SelectObject(pOldBrush);      // from device context
}

void C_Ellipse::Draw(CPaintDC* pMyDc)
{
    CPen NewPen;
    CBrush NewBrush;
    NewPen.CreatePenIndirect(&m_MyLogPen);
    NewBrush.CreateBrushIndirect(&m_MyLogBrush);
    CPen* pOldPen = pMyDc->SelectObject(&NewPen);
    CBrush* pOldBrush = pMyDc->SelectObject(&NewBrush);
    pMyDc->Ellipse(&m_BoundingRect);
    pMyDc->SelectObject(pOldPen);        // displace drawing objects
    pMyDc->SelectObject(pOldBrush);      // from device context
}

////////////////////////////////////
// Defintion of C_MainFrame functions
//
C_MainFrame::C_MainFrame()
{
    Create(NULL, "Graphics Editor Example", WS_OVERLAPPEDWINDOW,
    rectDefault, NULL, MAKEINTRESOURCE(IDR_MENU1));

    m_bButtonDown = FALSE;
    m_BkMode = OPAQUE;
    m_ROP2 = R2_COPYPEN;
    m_LogBrush.lbColor = RGB(255, 0, 0);
    m_eShapeKind = eRectangle;
    m_nShape = 0;
}

C_MainFrame::~C_MainFrame()                // destructs array of shapes
{
    if ( m_nShape > 0)
    {
        for (int shape = 0; shape < m_nShape; shape++)
            delete m_apShape[shape];
    }
}

//----- handler functions for WM messages -----

void C_MainFrame::OnPaint()
{
    CPaintDC MyDc(this);
    MyDc.SetBkMode(m_BkMode);
    MyDc.SetROP2(m_ROP2);
}

```

```

        for (int shape = 0; shape < m_nShape; shape++)
            m_apShape[shape]->Draw(&MyDc);
    }

void C_MainFrame::OnLButtonDown(UINT nFlags, CPoint Mouse)
{
    if (m_nShape >= MAX_SHAPES || m_bButtonDown) return;
    switch(m_eShapeKind)
    {
        case eEllipse:
            m_apShape[m_nShape] = new C_Ellipse;
            break;
        default:
            m_apShape[m_nShape] = new C_Rectangle;
    }

    m_apShape[m_nShape]->SetParams(&m_LogPen, &m_LogBrush);
    m_apShape[m_nShape]->SetBoundingRect(Mouse.x, Mouse.y,
                                           Mouse.x, Mouse.y);

    m_nShape++;
    m_bButtonDown = TRUE;
    SetCapture(); // window gets mouse input if mouse outside window
}

void C_MainFrame::OnMouseMove(UINT nFlags, CPoint Mouse)
{
    if (!m_bButtonDown) return;

    m_apShape[m_nShape - 1]->SetRightBottom(Mouse.x, Mouse.y);

    InvalidateRect(&m_RepaintArea);

    CRect r = m_apShape[m_nShape - 1]->GetBoundingRect();
    int left = min(r.left, r.right) - 1;
    int top = min(r.top, r.bottom) - 1;
    POINT LeftTop;
    LeftTop.x = left;
    LeftTop.y = top;
    int width = abs(r.right - r.left) + 2;
    int height = abs(r.bottom - r.top) + 2;
    SIZE RectSize;
    RectSize.cx = width;
    RectSize.cy = height;
    CRect NewArea(LeftTop, RectSize);
    m_RepaintArea = NewArea;
}

void C_MainFrame::OnLButtonUp(UINT nFlags, CPoint Mouse)
{
    m_bButtonDown = FALSE;
    ::ReleaseCapture();
}

// ----- handler functions for menu items -----
void C_MainFrame::OnPenGreen()
{
    m_LogPen.lopnColor = RGB ( 0, 255, 0);
}

```

```

}

void C_MainFrame::OnPenFuschia()
{
    m_LogPen.lpnColor = RGB ( 255, 0, 255);
}

void C_MainFrame::OnPenDash()
{
    m_LogPen.lpnStyle = PS_DASH;
}

void C_MainFrame::OnPenDot()
{
    m_LogPen.lpnStyle = PS_DOT;
}

void C_MainFrame::OnBrushRed()
{
    m_LogBrush.lbColor = RGB(255, 0, 0 );
}

void C_MainFrame::OnBrushTeal()
{
    m_LogBrush.lbColor = RGB( 0, 128, 128);
}

void C_MainFrame::OnBrushOlive()
{
    m_LogBrush.lbColor = RGB(128, 128, 0);
}

void C_MainFrame::OnBrushBlue()
{
    m_LogBrush.lbColor = RGB( 0, 0, 255);
}

void C_MainFrame::OnBrushSolid()
{
    m_LogBrush.lbStyle = BS_SOLID;
}

void C_MainFrame::OnBrushCross()
{
    m_LogBrush.lbStyle = BS_HATCHED;
    m_LogBrush.lbHatch = HS_CROSS;
}

void C_MainFrame::OnBrushHatchDiag()
{
    m_LogBrush.lbStyle = BS_HATCHED;
    m_LogBrush.lbHatch = HS_FDIAGONAL;
}

void C_MainFrame::OnOpaque()
{
    m_BkMode = OPAQUE;
}

```



```

        Invalidate();
    }

void C_MainFrame::OnTransparent()
{
    m_BkMode = TRANSPARENT;
    Invalidate();
}

void C_MainFrame::OnCopyPen()
{
    m_ROP2 = R2_COPYPEN;
    Invalidate();
}

void C_MainFrame::OnXORPen()
{
    m_ROP2 = R2_XORPEN;
    Invalidate();
}

void C_MainFrame::OnNOTXORPen()
{
    m_ROP2 = R2_NOTXORPEN;
    Invalidate();
}

void C_MainFrame::OnRectangle()
{
    m_eShapeKind = eRectangle;
}

void C_MainFrame::OnEllipse()
{
    m_eShapeKind = eEllipse;
}

void C_MainFrame::OnDelete()
{
    if(m_nShape > 0)
    {
        delete m_apShape[m_nShape - 1];
        m_nShape--;
        Invalidate();
    }
}

-----
                                resource files
-----
//
//  graphics.rc  -  icon and menu for graphics editor
//
#include "afxres.h"
#include "resource.h"

AFX_IDI_STD_FRAME      ICON      "graphics.ico"

```

```

IDR_MENU1    MENU
BEGIN
    POPUP "&Pen"
    BEGIN
        POPUP "&Color"
        BEGIN
            MENUITEM "&Green",          ID_PC_GREEN
            MENUITEM "&Fuschia",        ID_PC_FUSCHIA
        END
        POPUP "&Style"
        BEGIN
            MENUITEM "&Dash",          ID_PS_DASH
            MENUITEM "&Do&t",          ID_PS_DOT
        END
    END
    POPUP "&Brush"
    BEGIN
        POPUP "&Color"
        BEGIN
            MENUITEM "&Red",            ID_BC_RED
            MENUITEM "&Teal",          ID_BC_TEAL
            MENUITEM "&Olive",         ID_BC_OLIVE
            MENUITEM "&Blue",          ID_BC_BLUE
        END
        POPUP "&Style"
        BEGIN
            MENUITEM "&Solid",          ID_BS_SOLID
            MENUITEM "&CrossHatch",     ID_HS_CROSS
            MENUITEM "&DownwardHatch", ID_HS_FDIAGONAL
        END
    END
    POPUP "Bk&Mode"
    BEGIN
        MENUITEM "&Opaque",            ID_OPAQUE
        MENUITEM "&Transparent",       ID_TRANSPARENT
    END
    POPUP "&ROP2"
    BEGIN
        MENUITEM "&CopyPen",          ID_COPYPEN
        MENUITEM "&XORPen",           ID_XORPEN
        MENUITEM "&NotXORPen",        ID_NOTXORPEN
    END
    POPUP "&Shape"
    BEGIN
        MENUITEM "&Rectangle",        ID_RECTANGLE
        MENUITEM "&Ellipse",          ID_ELLIPSE
    END
    MENUITEM "&DeleteLastShape",      ID_DELETE
END

//
// resource.h
//
#define IDR_MENU1                101
#define ID_PC_GREEN              40001
#define ID_PC_FUSCHIA           40002

```

```
#define ID_PS_DASH 40003
#define ID_PS_DOT 40004
#define ID_BC_RED 40005
#define ID_BC_TEAL 40006
#define ID_BC_OLIVE 40007
#define ID_BC_BLUE 40008
#define ID_BS_SOLID 40009
#define ID_HS_CROSS 40010
#define ID_HS_FDIAGONAL 40011
#define ID_OPAQUE 40012
#define ID_TRANSPARENT 40013
#define ID_COPYPEN 40014
#define ID_XORPEN 40015
#define ID_NOTXORPEN 40016
#define ID_RECTANGLE 40017
#define ID_ELLIPSE 40018
#define ID_DELETE 40019
-----
```

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

C++ Objects for the Rectangle and Ellipse

In the graphics program, classes C_Rectangle and C_Ellipse represent rectangles and ellipses respectively. These classes are both derived from class C_Shape. In fact, all of the data and most of the functionality that ellipses and rectangles have in this graphics program are common to the two types of shapes; therefore, this data and functionality are encapsulated in the base class C_Shape. The class C_Shape holds the data required to describe a shape. This includes data members for: a CRect named m_BoundingRect, which holds the coordinates of the shape's bounding rectangle—left, top, right, and bottom; the LOGPEN variable m_MyLogPen that describes the pen that will be selected into the device context when the shape is drawn; and the LOGBRUSH variable m_MyLogBrush that describes the brush that will be selected into the device context when the shape is drawn. Class C_Shape contains member functions: SetBoundingRect() which sets the bounding rectangle's coordinates; GetBoundingRect() which gets the bounding rectangle's coordinates; SetRightBottom() which sets the right bottom coordinates of the shape's bounding rectangle; SetParams() which sets the data members m_MyLogPen and m_MyLogBrush according to supplied arguments; and the Draw() function since each shape must draw itself.

The function Draw() is declared in C_Shape with the syntax:

```
virtual void Draw(CPaintDC*) = 0;
```

This syntax declares the member function Draw() to be a *pure virtual function*. This means that the member function's code is undefined. In any class that is derived from a class that has a pure virtual function, the pure virtual functions have to be overridden with member functions that have defined code in order for the compiler to permit objects of the class to be instantiated. The pure virtual function declaration is therefore a statement that this function must be

defined in any derived class that will have instantiated objects. In this example application, the objects of classes derived from `C_Shape` must be able to draw themselves, but `C_Shape` itself does not know how to draw a shape. (After all, how do you draw a generic shape?)

The derived classes `C_Rectangle` and `C_Ellipse` just define one member function, the `Draw()` function. These `Draw()` functions simply activate the object's custom brush and pen and draw the shape using functions `CDC::Rectangle()` and `CDC::Ellipse()`, respectively. Then they displace the drawing objects from the device context, and the custom drawing objects are deleted automatically when they go out of scope.

The Graphics Output Process

The preceding code for the graphics program illustrates the generalized process for outputting the graphics to the screen, which is as follows:

1. Create a device context with either the `CClientDC` or `CPaintDC` class. (In rare cases you may be using `CWindowDC`.) `CPaintDC` is used only when processing `WM_PAINT` messages in the `CWnd::OnPaint()` function.
2. Create your specialized pens and brushes first and then select them into the device context.
3. Draw the graphics.
4. Displace your pens and brushes from the device context. Drawing objects must not be deleted if they are currently selected into the device context.
5. Then delete your pens and brushes. `CPen` and `CBrush` automatic objects will, like any other C++ automatic objects, have their destructor called and be automatically deleted when they go out of scope.

Deleting Drawing Objects

Before it ends execution, your application should delete all drawing objects that it creates. Use the function `CGdiObject::DeleteObject()` to delete your drawing objects. Created drawing objects that are not deleted before the termination of execution remain in the Windows OS system, using up memory. Windows OS may behave in unexpected ways if you destroy drawing objects while they are selected into the device context. Your program may behave unexpectedly when Windows OS attempts to use the device context which contains pointers to deleted pens and brushes. To avoid this problem, you must make sure that all drawing objects are displaced out of the device context before they are deleted.

An important property of the `CDC::SelectObject()` function is that it returns a pointer to the previously selected object in the device context of the same type. For example, if you are selecting a `CPen` object into a device context, `CDC::SelectObject()` returns a pointer to the `CPen` object that was selected previously. An elegant way to displace your drawing objects from the device context is to save the pointers to the old pen and brush objects as the `pOldPen` and `pOldBrush` variables when a new one is selected. The old objects can then

be selected back into the device context to displace your new ones, making the new ones safe to delete. Another way to make sure that a pen or brush is not selected into the device context is to select a stock pen or brush. Selecting a stock pen displaces the current pen from the device context.

When you create the object CPaintDC within the body of the OnPaint() function it is stored on the program's stack. When the OnPaint() function finishes, the device context object is destroyed when the stack is destroyed, such that the stack can be reused by the next function. Thus, the destructor function for CPaintDC is automatically called at the end of the OnPaint() function. This is convenient, and it means the memory consumed by this object is released automatically. Similarly, the CPen and CBrush objects were created on the stack of the Draw() function when it was called from within the OnPaint() function. When the Draw() function closes, its stack is destroyed and the destructor function is called for the CPen and CBrush objects at the end of the Draw() function.

Selecting a Menu Item

The menu item handler functions set the parameter values for the pen and the brush and the variable m_eShapeKind to reflect the user's choices. Logpens and logbrushes are very conveniently used in this program, because you can set one pen or brush variable at a time. That is, you can change the brush color regardless of whether the brush is solid or hatched. The variables for the background mode and the ROP2 code are also changed in the corresponding handler functions; in these handler functions the entire screen is immediately repainted reflecting the new choices. This is done by a call to the CWnd::Invalidate() function which causes a WM_PAINT message to be generated.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Drawing the Rectangles and Ellipses

Left Mouse Button Down

There is an array `m_apShape` of pointers to `C_Shape` objects to be used by some of the message handlers to operate with the list of shapes. The integer `m_nShape` holds the number of shapes that are currently in the array. Whenever the user presses the left mouse button in the client area, the handler function `OnLButtonDown()` is called. This message handler instantiates an ellipse or rectangle object from class `C_Ellipse` or `C_Rectangle`, respectively, adds a pointer to it to the `m_apShape` array, calls its `SetParams()` member function to set its drawing attributes (as given in `m_MyLogPen` and `m_MyLogBrush`) and also calls its `SetBoundingRect()`. It also records that the left mouse button is down (the `m_bButtonDown` flag is set to `TRUE`) and captures the mouse input with the function `CWnd::SetCapture()`. If the mouse input is not captured, another window may appear on top of your window when the mouse goes outside your window. The necessary effect of capturing the mouse input is that all subsequent mouse messages are sent to the window, including the `WM_LBUTTONDOWNUP` message, which would otherwise be lost.

Moving the Mouse with the Left Button Down

The handler function `OnMouseMove()` responds to `WM_MOUSEMOVE` messages. This function is invoked whenever the mouse moves. The new shape, which has the right bottom corner of the newest mouse position, should be drawn while the user holds the left mouse button down and moves the mouse.

If the left mouse button is down (indicated by the flag `m_bButtonDown` being `TRUE`), then `OnMouseMove()` changes the right bottom corner of the shape's bounding rectangle to correspond to the current mouse position. To draw only

that portion of the screen in which the new shape is being drawn (which is recommended), call the function `CWnd::InvalidateRect(&m_RepaintArea)`. The `CRect m_RepaintArea` holds the bounding rectangle of the area that we want to be repainted. The second input parameter to the call `CWnd::InvalidateRect()` is not specified, so its default value of `NULL` is used, which means that the window's background needs to be repainted for the area being redrawn.

Releasing the Left Mouse Button

When the left mouse button is released, the `OnLButtonUp()` function has only two things to do: it must note that the left mouse button is no longer down (the `m_bButtonDown` flag is set to `FALSE`), and it must release the mouse capture. This is done with the API function call `::ReleaseCapture()`. (The API function `ReleaseCapture()` is not wrapped by an MFC function.)

Note: For the drawing functions `CDC::Rectangle()` and `CDC::Ellipse()`, the left and right coordinates may be swapped and the top and bottom coordinates may be swapped without changing the shapes drawn. However, the input parameter, `CRect m_RepaintArea`, must always be such that the left-top point is above and to the left of the right-bottom point (i.e., they cannot be swapped). If they are swapped, simply reswap them. Do this by selecting the minimum of the top or bottom as the top and by selecting the minimum of the left or right as the left. Also, increase the area to be sure to completely erase the previous graphics.

The call to `CWnd::InvalidateRect()` causes Windows OS to send a `WM_PAINT` message to the main window, and this message in turn causes MFC to invoke the `OnPaint()` member function. `OnPaint()` is a member function of `C_MainFrame`.

The OnPaint() Function

The easiest way for an application to handle the drawing of graphics and text in its client area is to call `CWnd::Invalidate()` (or `CWnd::InvalidateRect()` when the client area is to be repainted) and put all the drawing code in the member function `OnPaint()`. The reason this way is the easiest is that `OnPaint()` needs to contain the screen-drawing code anyway, in order to respond to resizing and uncovering of the window. The graphics program uses this approach.

As discussed earlier, the `OnPaint()` function will redraw the entire client area or will redraw that portion of the client area of its window which is specified in the first argument of `CWnd::InvalidateRect()`. The graphics program of Listing 3-3 will redraw all of the shapes in the shape list that are within the area being redrawn (the entire screen or the specified `CRect`). In this graphics program, the function `CWnd::Invalidate()` is used when redrawing the entire screen is desired, which is indicated when the `ROP2` code or the `m_BkMode` code is changed. Use the function `CWnd::InvalidateRect()` when you want to redraw only a portion of the screen, which is done when the new shape is being drawn and formed. The `OnPaint()` function loops over the elements of the array of pointers to shape objects (ellipses and rectangles) that the user has

created, calling the Draw() virtual member function on each shape. The virtual function mechanism built into C++ ensures that the corresponding Draw() function will be called in each case.

Previous	Table of Contents	Next

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

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

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

Keyword

☒ Brief
 ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Maintaining the Update Region

This section discusses in more detail the interactions between Windows OS, MFC, and the application that lead to the client area being erased and redrawn. This is the region of the window that currently is out of date and needs to be redrawn. The update region structure has an erase attribute, which is explored in the following discussion.

A call to `CWnd::InvalidateRect(&Rect, TRUE)` or `CWnd::InvalidateRgn(&Rgn, TRUE)` function adds a rectangle or a region respectively to the update region. If the second argument in the function call is `TRUE`, then the update region's erase attribute is set to `TRUE`. The update region's erase attribute is also set to `TRUE` for the first time the window is drawn, when portions of the window are uncovered, and when the window is resized or moved.

If the application does not want the client area of a window to be erased before `OnPaint()` is called, it should assign `FALSE` to the second parameter of `CWnd::InvalidateRect()`; then neither MFC nor Windows OS erases the background within the update region.

The Background Color

The function `CDC::SetBkColor(COLORREF)` sets the background color. This color is used for filling the update area between the hatch lines when using a hatched brush, for filling the background of character cells, and for filling the gaps in dashed lines. The system also uses this background color when converting bitmaps between color and monochrome device contexts. This fill is only done when the background mode is `OPAQUE`, as set by `CDC::SetBkMode()`. This background is more properly defined as the color of the interstices in brush patterns and non-solid pen lines.

Windows OS also uses the term “background” to refer to an unrelated concept, namely the background brush for the window. This brush, named `hbrBackground`, is used to erase the clipping region before drawing in the client area. The variable `hbrBackground` is essentially the background color and pattern of the client area. The default value that MFC assigns to `hbrBackground` is a solid white brush. The value of `hbrBackground` can be changed for a window, if desired, by assigning a new value to it in the `AfxRegisterWndClass()` function (as discussed in Chapter One).

These two types of background are not tied together in any way. For example, if the user changes the window background, the client area’s background will be the new color, but the interstices in hatched fills of rectangles, etc. will still be whatever color `CDC::SetBkColor()` has most recently specified for the device context.

The Handy Utility Classes: CRect, CPoint, and CSize

MFC provides the utility classes `CRect`, `CPoint`, and `CSize`, which are extremely handy when dealing with rectangles or drawing functions that have a bounding rectangle. These utility classes are interesting in that they are not derived from the class `CObject`, as are most other MFC classes. The `CRect`, `CPoint`, and `CSize` classes have a number of member functions and overloaded operators which make them very useful. The following (among other things) can be done with these utility classes:

- Add a `CSize` object to a `CPoint` object;
- Subtract a `CSize` object from a `CPoint` object;
- Subtract one `CPoint` object from another, yielding a `CSize` object;
- Add a `CPoint` object to a `CRect` object;
- Subtract a `CPoint` object from a `CRect` object;
- Determine if a `CPoint` object lies within a `CRect` object;
- Subtract one `CRect` object from another;
- Obtain the intersection of two `CRect` objects;
- Obtain the union of two `CRect` objects;
- Move the `CRect` object by an amount specified by a `CPoint` object or `CSize` object;
- Inflate/deflate a `CRect` object by an amount specified by a `CSize` object;
- Calculate and return the value of the width, height, size, top-left, and bottom-right of the `CRect`.

Using a Private Device Context

To create a window with its own private device context, a new window class that uses the `CS_OWNDC` window style must be registered. As discussed in Chapter One, the global function `AfxRegisterWndClass()` is used to register a window class. To get a window with its own private device context, use the

CS_OWNDC window style in the first argument. The CS_OWNDC style instructs the operating system to set aside a memory area large enough to hold the device context settings for each window created from this class. When creating a number of similar windows that can share the same device context, use the CS_CLASSDC style in place of CS_OWNDC. CS_CLASSDC tells the operating system to reserve only one memory area for a common device context, and let every window from the class share that group of device context settings. One last option is CS_PARENTDC, which can be used for child windows. Child windows, created from a class using the CS_PARENTDC style, use their parent's device context. This presumes the parent was created with either the CS_OWNDC or CS_CLASSDC style, so that the child has data to read.

Once a private device context has been created, all settings are saved in the device context, and the modified device context can be used whenever you choose. The device context can be initialized when the program first starts, in the C_MainFrame's constructor. The device context will continue to exist in memory until the window is destroyed.

Previous	Table of Contents	Next

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

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

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


```

//
class C_MainFrame : public CFrameWnd
{
    public:
        C_MainFrame();
        ~C_MainFrame();
    private:
        CPen* pMyPen;
        CBrush* pMyBrush;
        //{{AFX_MSG(C_MainFrame)
        afx_msg void OnPaint();
        afx_msg void OnDrawGraphics();
        //}}AFX_MSG

        DECLARE_MESSAGE_MAP()

};
//
//  mainfrm.cpp
//
#include "MyApp.h"
#include "mainfrm.h"
#include "resource.h"

BEGIN_MESSAGE_MAP(C_MainFrame, CFrameWnd)

    //{{AFX_MSG_MAP(C_MainFrame)
    ON_WM_PAINT()
    ON_COMMAND(ID_DRAW, OnDrawGraphics)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

C_MainFrame::C_MainFrame()
{
    CString MyClass;
    CBrush hYellowBrush( RGB(255,255,0));

    MyClass = AfxRegisterWndClass(
        CS_HREDRAW | CS_VREDRAW | CS_OWNDC,
        AfxGetApp()->LoadStandardCursor(IDC_ARROW),
        (HBRUSH)hYellowBrush.GetSafeHandle(),
        AfxGetApp()->LoadIcon(AFX_IDI_STD_FRAME));
    Create( MyClass, "Using an Owner Device Context",
        WS_OVERLAPPEDWINDOW, rectDefault,
        NULL, MAKEINTRESOURCE(IDR_MENU1));

    CClientDC dc(this);
    pMyPen = new CPen(PS_SOLID, 5, RGB(0, 0, 255));    // blue pen
    pMyBrush = new CBrush( RGB(128, 128, 0));          // olive brush
    dc.SelectObject(pMyPen);
    dc.SelectObject(pMyBrush);
}

C_MainFrame::~C_MainFrame()
{
    delete pMyPen;

```

```

        delete pMyBrush;
    }

void C_MainFrame::OnPaint()
{
    CPaintDC MyDC(this);
    MyDC.Ellipse(20, 20, 200, 100);
}

void C_MainFrame::OnDrawGraphics()
{
    CClientDC MyDC(this);
    MyDC.Rectangle(20, 20, 200, 100);
}

-----
                                resource files
-----
//
//      OwnerDC.rc
//
#include "afxres.h"
#include "resource.h"

AFX_IDI_STD_FRAME      ICON      "graphics.ico"
IDR_MENU1      MENU
BEGIN
    MENUITEM "&Draw",      ID_DRAW
END

//
//  resource.h
//
#define IDR_MENU1      101
#define ID_DRAW      102
-----

```

In the preceding program, the window class with a private device context is registered in the constructor. After the window object is created using the new window class MyClass, the device context can be initialized with all of its settings. In this program, the device context was initialized in the constructor. A new pen and brush are created and selected into the device context. Note that the pen and brush are created using the *new* operator so that the objects are stored on the heap. If the pen and brush were created on the stack as automatic variables, they would be destroyed automatically when the C_MainFrame constructor function returned. This is, of course, not acceptable since these objects remain selected in the device context for the duration of the program. The solution is to instantiate them on the heap as is done in the preceding listing and declare pointers to the CPen and CBrush objects as data members of the class C_MainFrame. This keeps them around for the duration of the program. Of course, a destructor must be defined, ~C_MainFrame(), to delete these pointers and their drawing objects. All of the settings that are placed into the window's device context at its initialization in the constructor are remembered for the duration of the program. This is because it is a private device context, and it remains available in memory for the entire duration of the program. When the program is terminated, the private device context is destroyed and released automatically.

At different points in the program's operations, the private device context is used to output graphics. An ellipse is drawn in the C_MainFrame::OnPaint() function. To do this, the private device context is first obtained as a CPaintDC device context and then used as is to paint an ellipse.

Note: If the device context's settings are changed anywhere within the program, they will remain changed unless another action is taken to again alter the settings in some way.

In the preceding program, anytime a WM_PAINT message is received, such as when the window is initially made visible or when the window is resized, the handler function OnPaint() is entered and the ellipse is drawn. The private device context, which has a five-pixel-wide blue pen and an olive brush, is used for drawing. The device context is also called from another function, the handler function OnDrawGraphics(), which is entered when the user selects the menu item " Draw." In this handler function, the device context is obtained as a CClientDC. Of course, it is the same private device context with the same settings as before. In this handler function the device context is used to draw a rectangle, which covers the previously displayed ellipse. As you can see, it draws with the same brush and pen.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

From this example the benefit of using a private device context is demonstrated. In each handler function that uses the device context, the program logic turns out to be much simpler. In these handler functions, no initialization activities, such as creating pens and brushes, are required. The logic ends up simpler and will execute faster than if all of the device context settings are initialized each time a handler function is entered.

The private device context will be destroyed automatically when the window that owns it is destroyed. Any drawing objects that were created and selected into the device context must be deleted before the program terminates.

Tip: A good place for deleting the drawing objects is the window's destructor, which is the chosen method in the preceding program. Otherwise, these objects will stay in memory after the program is terminated, taking up valuable memory space.

Summary

An application communicates to the user through drawing in the client area of its main window. The central concept for drawing is the device context. The device context, which is an object of the class CDC, maintains all the drawing attributes to be used for drawing. The drawing attributes are customized to the application's needs by CDC functions such as: SetTextColor(), SetBkMode(), SelectObject(), etc. The system has a limited number of device contexts; all users must get a device context, use it, and immediately return it to the system. Drawing objects (pens, brushes, fonts, bitmaps, regions, and palettes) are selected into the device context, the drawing is completed, and they are removed from the device context before the device context goes out of scope. There are a limited number of stock drawing objects, which are predefined and available in the system, that can be used.

The CDC class has numerous functions for customizing the device context as

well as for drawing. Device context classes are derived from CDC, of which the most useful are CPaintDC and CClientDC. CPaintDC device contexts are used to draw in the client area and must be used only in the OnPaint() function. CClientDC device contexts are used to draw in the client area, but cannot be used in the OnPaint() function.

CPens and CBrushes can be constructed in a variety of ways. Use the most convenient constructor for your application. CPens have various styles—solid, dashed, etc. and they can be of various widths and colors. Likewise, CBrushes have various styles: they can be solid, hatched, or patterned. CPens are used to outline figures and CBrushes are used to fill the figures. The RGB macro is used to specify CPen and CBrush colors.

The raster drawing mode, ROP2, applies to drawing to the screen. The default ROP2 mode (normally used) is R2_COPYPEN, which covers the old color on the screen with the new color being used for drawing. Other ROP2 modes of special interest are R2_XORPEN and R2_NOTXORPEN. Both of these have the special feature that when a shape is drawn twice in that mode, the shape disappears; hence these modes are useful in fast drawing techniques.

A window can have its own private device context, not shared with other applications. A window with its own private device context can be obtained by registering a new window class. If your window has its own private device context, you do not have to release the device context, which means that you can customize it once and use it throughout the program without further change. This increases the efficiency of your application.

Exercise

In this chapter you have learned how to draw with pens and brushes and how to draw text in the client area of the window. You now have the skills to draw the checkerboard and its pieces in the client area of the window. Draw each player's initials on his pieces. At this stage in the application development, the players are named Player A and Player B. Later we will be accepting input from the user and the first and last name of each player will be provided (see the Chapter Eight exercise). Draw the checkerboard with all its pieces, centering it in the client area. Draw the checkerboard, using a CPaintDC device context, in the OnPaint() handler function. When the user resizes the window, the size of the client area will change. Windows OS sends a WM_PAINT message when the window is resized. When the size of the client area changes, resize the checkerboard and all its pieces. Helpful hints are given in the section titled “Chapter 3 Exercise” of Appendix B.

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Chapter 4

Fast Drawing and Bitmap Graphics

This chapter presents fast drawing techniques, animation using bitmaps, and color palettes. First, fast drawing using the exclusive-or (R2_XORPEN) and the exclusive-nor (R2_NOTXORPEN) ROP2 codes is explored. Then, fast drawing using the “backing store” technique is discussed. Bitmaps are used in the backing store technique, and their use is introduced. After this, an example of creating animation using bitmaps is presented. Finally, the *device independent bitmap* (DIB) and *color palettes* are discussed.

In the graphics example of the previous chapter, the area enclosed in a rectangle surrounding the shape being formed is redrawn whenever the mouse moves. This causes the entire rectangular area to flicker while drawing. Drawing techniques can be used that only need to change the area within the shape being drawn. When the shape is an ellipse, the shape itself need only be erased and redrawn, rather than the entire rectangular area surrounding the ellipse. But a way to erase its previous shape from the screen needs to be provided prior to redrawing it in its new shape. In this chapter such techniques, which promote fast drawing and minimum screen flicker, are explained.

Using Exclusive-or and Exclusive-nor for Fast Redraws

The first fast drawing technique presented is the use of either of the fast drawing modes: exclusive-or or exclusive-nor. This technique is demonstrated by modifying the graphics program presented in the previous chapter in Listing 3-3. The running program is shown in Figure 4-1. The program called “FastDraw” looks a lot like the graphics program with the exception that the menu item “ROP2” has been replaced with a new menu item,

“FastRedrawMode,” which creates a popup menu with two choices: “XORMode” or “NotXORMode.” These menu choices, of course, are used to set the parameters of the fast drawing mode; “XORMode” gives the choice of drawing with the exclusive-or mode, and “NotXORMode” gives the choice of drawing with the exclusive-nor mode.

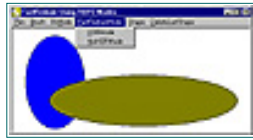


Figure 4-1: Executing FastDraw Program

This FastDraw program refreshes the screen faster by using a new drawing technique in the OnMouseMove() message handling function rather than using the call to InvalidateRect(), as was done in the graphics program of Chapter Three. When the mouse moves with the left button down, OnPaint() is no longer called. Instead, when the left button is down and the mouse has moved, the old form of the current shape is exclusive-or-ed (or exclusive-nor-ed) off the screen and the new form is exclusive-or-ed (or exclusive-nor-ed) onto the screen. This fast redraw technique depends on the property of exclusive-or (or exclusive-nor) or that a second exclusive-or (or exclusive-nor) removes the shape from the screen, restoring the screen to its state before the first exclusive-or (or exclusive-nor). Drawing a second time restores the colors on the screen to what they were before the first drawing, which in essence erases the shape.

Listing 4-1 shows the changes that must be made to the previous graphics program, given in Listing 3-3, to create this new program, “FastDraw.” (Of course, the code for the application class is the same, so it is not shown.) Some of the changes in the new program are due to the changes in the menu items. The resource files, fastdraw.rc and resource.h, are changed but are not shown because their changes are minor. All of the mainfrm.h file has been shown for clarity, with the new code in bold and deleted lines of code commented out. Since most of the code in mainfrm.cpp is the same as the program of Listing 3-3, where practical only the new or changed functions are shown. The new code is shown in bold, as well as preceded with an arrow to distinguish it from the code that was previously there. The code that was previously there is not bold. If a line of code that was previously there is no longer needed, it is commented out (preceded with a double slash (//)) and because it is a change from the previous code, it is also preceded with an arrow to highlight that a change has occurred.

The following changes can be seen in the file mainfrm.cpp:

- A new virtual function DrawNew(CClientDC*) of class C_Shape is defined. This new function draws the shape on a small segment of the screen, so it uses a CClientDC device context.
- Addition/deletion of variables of the class C_MainFrame, as highlighted in the code.
- Two new message map entries and their corresponding handler functions. (These result from the changed menu items and displace the three previous entries on the old menu.)

- Slight modifications to the OnPaint() function. (We no longer set the ROP2 mode for the entire screen area.)
- Substantial modifications to the functions OnMouseMove() and OnLButtonUp(), which are discussed in detail following the listing.

Previous	Table of Contents	Next

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

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

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

Keyword

☒ Brief
 ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Listing 4-1: Source Code for the FastDraw Program

```

-----
                                     mainframe class
-----
//
//  mainfrm.h
//
class C_Shape
{
    protected:
        CRect m_BoundingRect;
        LOGPEN  m_MyLogPen;
        LOGBRUSH m_MyLogBrush
    public:
        void SetBoundingRect(int left, int top, int right, int bottom);
        CRect GetBoundingRect();
        void SetRightBottom(int right, int bottom);
        void SetParams(LOGPEN*, LOGBRUSH*);
        virtual void Draw(CPaintDC*) = 0;
-->    virtual void DrawNew(CClientDC*) = 0;
};

class C_Rectangle : public C_Shape
{
    public:
        virtual void Draw(CPaintDC*);
-->    virtual void DrawNew(CClientDC*);
};

class C_Ellipse : public C_Shape
{
    public:
        virtual void Draw(CPaintDC*);
-->    virtual void DrawNew(CClientDC*);
};

class C_MainFrame : public CFrameWnd
{
    public:
        C_MainFrame();
        ~C_MainFrame();

```

```

private:
    int m_nShape;
    BOOL m_bButtonDown;
    int m_BkMode;
-->//    int m_ROP2;
        LOGPEN m_LogPen;
        LOGBRUSH m_LogBrush;
        enum EnumShape{eRectangle, eEllipse};
        EnumShape m_eShapeKind;
        #define MAX_SHAPES    100
        C_Shape* m_apShape[MAX_SHAPES];
-->//    CRect m_RepaintArea;
-->    int m_FastRedrawMode;
-->    BOOL m_bDrawingNewShape;

    //{AFX_MSG(C_MainFrame)
    afx_msg void OnPaint();
    afx_msg void OnLButtonDown(UINT nFlags, CPoint Mouse);
    afx_msg void OnMouseMove(UINT nFlags, CPoint Mouse);
    afx_msg void OnLButtonUp(UINT nFlags, CPoint Mouse);
    afx_msg void OnPenGreen();
    afx_msg void OnPenFuschia();
    afx_msg void OnPenDash();
    afx_msg void OnPenDot();
    afx_msg void OnBrushRed();
    afx_msg void OnBrushTeal();
    afx_msg void OnBrushOlive();
    afx_msg void OnBrushBlue();
    afx_msg void OnBrushSolid();
    afx_msg void OnBrushCross();
    afx_msg void OnBrushHatchDiag();
    afx_msg void OnOpaque();
    afx_msg void OnTransparent();
-->//    afx_msg void OnCopyPen();
-->//    afx_msg void OnXORPen();
-->//    afx_msg void OnNOTXORPen();
-->    afx_msg void OnRedrawXORMode();
    afx_msg void OnRedrawNOTXORMode();
    afx_msg void OnRectangle();
    afx_msg void OnEllipse();
    afx_msg void OnDelete();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

//
//    mainfrm.cpp
//
#include "MyApp.h"
#include "mainfrm.h"
#include "resource.h"

////////////////////////////////////
//    Message map
//
BEGIN_MESSAGE_MAP(C_MainFrame, CFrameWnd)
    //{AFX_MSG_MAP(C_MainFrame)

        (no change in any other message map entries)
-->//    ON_COMMAND(ID_COPYPEN, OnCopyPen)

```



```

-->//      ON_COMMAND(ID_XORMODE, OnXORPen)
-->//      ON_COMMAND(ID_NOTXORMODE, OnNOTXORPen)
-->      ON_COMMAND(ID_XORMODE, OnRedrawXORMode)
-->      ON_COMMAND(ID_NOTXORMODE, OnRedrawNOTXORMode)

    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////
//      Definitions of C_Shape, C_Rectangle, and C_Ellipse functions
//
        (no change in other function definitions)

-->void C_Rectangle::DrawNew(CClientDC* pMyDc)
-->{
-->    CPen NewPen;
-->    CBrush NewBrush;
-->    NewPen.CreatePenIndirect(&m_MyLogPen);
-->    NewBrush.CreateBrushIndirect(&m_MyLogBrush);
-->    CPen* pOldPen = pMyDc->SelectObject(&NewPen);
-->    CBrush* pOldBrush = pMyDc->SelectObject(&NewBrush);
-->    pMyDc->Rectangle(&m_BoundingRect);
-->    pMyDc->SelectObject(pOldPen);
-->    pMyDc->SelectObject(pOldBrush);
-->}

-->void C_Ellipse::DrawNew(CClientDC* pMyDc)
-->{
-->    CPen NewPen;
-->    CBrush NewBrush;
-->    NewPen.CreatePenIndirect(&m_MyLogPen);
-->    NewBrush.CreateBrushIndirect(&m_MyLogBrush);
-->    CPen* pOldPen = pMyDc->SelectObject(&NewPen);
-->    CBrush* pOldBrush = pMyDc->SelectObject(&NewBrush);
-->    pMyDc->Ellipse(&m_BoundingRect);
-->    pMyDc->SelectObject(pOldPen);
-->    pMyDc->SelectObject(pOldBrush);
-->}

////////////////////////////////////
//      Defintion of C_MainFrame functions
//
C_MainFrame::C_MainFrame()
{
-->    Create(NULL, "FastRedraw Using ROP2 Modes", WS_OVERLAPPEDWINDOW,
-->           rectDefault, NULL, MAKEINTRESOURCE(IDR_MENU1));

    m_bButtonDown = FALSE;
    m_BkMode = OPAQUE;
-->//m_ROP2 = R2_COPYPEN;
    m_LogBrush.lbColor = RGB(255, 0, 0);
    m_eShapeKind = eRectangle;
    m_nShape = 0;
-->    m_FastRedrawMode = R2_XORPEN;
-->    m_bDrawingNewShape = FALSE;
}

        (no change in destructor)

//----- handler functions for WM messages -----

```

```

void C_MainFrame::OnPaint()
{
    CPaintDC MyDc(this);
    MyDc.SetBkMode(m_BkMode);
    --> // MyDc.SetROP2(m_ROP2);
        for (int shape = 0; shape < m_nShape; shape++)
            m_apShape[shape]->Draw(&MyDc);
}

void C_MainFrame::OnLButtonDown(UINT nFlags, CPoint Mouse)
{
    if (m_nShape >= MAX_SHAPES || m_bButtonDown) return;
    switch(m_eShapeKind)
    {
        case eEllipse:
            m_apShape[m_nShape] = new C_Ellipse;
            break;
        default:
            m_apShape[m_nShape] = new C_Rectangle;
    }

    m_apShape[m_nShape]->SetParams(&m_LogPen, &m_LogBrush);
    m_apShape[m_nShape]->SetBoundingRect(Mouse.x, Mouse.y,
                                         Mouse.x, Mouse.y);

    m_nShape++;
    m_bButtonDown = TRUE;
    SetCapture();
}

void C_MainFrame::OnMouseMove(UINT nFlags, CPoint Mouse)
{
    if (!m_bButtonDown) return;

    --> CClientDC MyDC(this);
    --> MyDC.SetROP2(m_FastRedrawMode);

    --> if (m_bDrawingNewShape == TRUE)
    -->     m_apShape[m_nShape - 1]->DrawNew(&MyDC);           // undraws old

    --> m_apShape[m_nShape - 1]->SetRightBottom(Mouse.x, Mouse.y);
    --> m_apShape[m_nShape - 1]->DrawNew(&MyDC);             // draws new
    --> m_bDrawingNewShape = TRUE;
}

void C_MainFrame::OnLButtonUp(UINT nFlags, CPoint Mouse)
{
    --> CClientDC MyDC(this);
    --> m_apShape[m_nShape - 1]->DrawNew(&MyDC);             // draws with R2_COPYPEN
    --> m_bDrawingNewShape = FALSE;

        m_bButtonDown = FALSE;
        ::ReleaseCapture();
}

////////////////////////////////////
//          handler functions for menu items

(two new handler functions displacing previous menu item handler functions)
(no change in any other handler functions)
--> void C_MainFrame::OnRedrawXORMode()
--> {

```

```
-->    m_FastRedrawMode = R2_XORPEN;
-->}

-->void C_MainFrame::OnRedrawNOTXORMode()
-->{
-->    m_FastRedrawMode = R2_NOTXORPEN;
-->}
-----
```

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Details of the Exclusive-or (Exclusive-nor) Process

The mouse handler functions OnMouseMove() and OnLButtonUp() have substantial changes and will be discussed first.

While the mouse is down, OnMouseMove() gets a CClientDC and changes its ROP2 setting appropriately. Using the flag m_bDrawingNewShape, it determines if this is the first mouse move of the new shape; if so, it bypasses erasing the old shape. On subsequent mouse moves, it erases the old shape by redrawing it using the DrawNew() function. It then updates the shape's parameters and draws the new shape by calling the DrawNew() function.

Note: The shapes are drawn to the client area using a CClientDC device context. The member function DrawNew() of C_Shape handles the drawing using a CClientDC device context for which the ROP2 setting has been appropriately modified.

When the left mouse button is released, OnLButtonUp() gets a CClientDC device context which has the default setting for its ROP2 code, which is R2_COPYPEN, and calls the function DrawNew() to draw the shape in that mode. The shape then appears in its intended colors. Finally, the m_bDrawingNewShape parameter is set to FALSE, since drawing the new shape has been completed.

The previously defined function Draw() handles the drawing that occurs from OnPaint().

Limitations of the Exclusive-or (Exclusive-nor) Process

The exclusive-or (exclusive-nor) drawing modes work with pens and brushes; hence they are best suited for use with points, lines, and shapes (rectangles,

ellipses, etc.). To use the exclusive-or (exclusive-nor) technique with other graphic elements, such as characters, draw these to a memory bitmap and use the exclusive-or (exclusive-nor) drawing mode of BitBlt() to exclusive-or (exclusive-nor) these onto the screen. (The BitBlt() modes are discussed in the next section.) Because of this complexity, it is usually easier to use *backing store* when moving text around.

The exclusive-or technique has the fundamental limitation of drawing in the reversed color. The exclusive-or technique is best suited for rubberbanding, where just an outline is moved. An example of this is when a window is moved by clicking and dragging on its caption bar. Interestingly, the Windows operating system uses exclusive-or for the standard I-beam cursor but backing store for the standard arrow cursor. Using exclusive-or for the I-beam cursor ensures that it will be visible over light or dark backgrounds.

The exclusive-nor technique provides colors that are faithful, if the background is white (to be exact, if the background's pixels have bits that are all ones). However, the color changes if you are exclusive-nor-ing over other colors. For instance, exclusive-nor-ing a red object (pixel value 1) over a green object (pixel value 2) results in a light blue object (pixel value 12). The graphics behind the shape show through in an unusual way. For this reason, another slightly slower technique, called backing store, is often used instead to move shapes around the screen. Backing store is described next.

Using Backing Store for Fast Redraws

The backing store technique saves the graphic under the shape that is to be drawn before the shape is drawn. To undraw the shape, just put the saved graphics back onto the screen. Then, the new form of the shape is drawn. The backing store technique will be slower than the exclusive-or (exclusive-nor) technique, however; it produces no color distortions.

Figure 4-2 shows the output of the program “BackingStore.” As can be seen, the menu is changed, and the menu item “FastRedrawMode” has been eliminated, since the only redraw mode used is backing store.

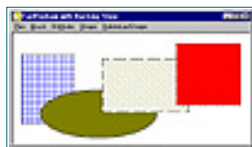


Figure 4-2: Executing BackingStore Program

The source code for the program “BackingStore” is produced by making changes to the FastDraw program.

Listing 4-2 presents the source code for the BackingStore program. The source code is very similar to that of Listing 4-1, which uses the exclusive-or (exclusive-nor) drawing mode to draw and redraw the shape as it is being formed. Listing 4-2 shows only the differences between this listing and Listing 4-1. The resource files are not shown, since their changes are minor. All of the mainfrm.h file has been shown to illustrate what has been eliminated and what has been added. Previous code that is no longer needed is commented out (preceded by double slashes (//)) and to highlight the change, the line of code

is preceded with an arrow in Listing 4-2. New lines of code that have been added are shown in bold as well as preceded with an arrow to highlight them. The code that was previously there is not bold. The portions of the mainfrm.cpp file that are changed are also shown.

The following changes should be noted in Listing 4-2:

- The menu item FastRedrawMode has been eliminated. The function prototypes, the message map entries, and the handler functions associated with this menu item's submenu are also eliminated.
- The variable m_FastRedrawMode in the C_MainFrame class has been eliminated.
- Two new functions have been defined in the class C_Shape which are: BackingStore(CClientDC*) and Undraw(CClientDC*). (These functions are discussed in detail following the listing.)
- The message handlers OnMouseMove() and OnLButtonUp() have been changed. (Discussed following the listing.)

Previous	Table of Contents	Next

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

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

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



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Go!

Keyword

☒ Brief ☐ Full[Advanced](#)[Search](#)[Search Tips](#)BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

Go!

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

Listing 4-2: Source Code for the BackingStore Program

```

-----
                                     mainframe class
-----
//
//  mainfrm.h
//
class C_Shape
{
    protected:
        CRect m_BoundingRect;
        LOGPEN  m_MyLogPen;
        LOGBRUSH m_MyLogBrush;
    public:
        void SetBoundingRect(int left, int top,
                               int right, int bottom);
        CRect GetBoundingRect();
        void SetRightBottom(int right, int bottom);
        void SetParams(LOGPEN*, LOGBRUSH*);
        virtual void Draw(CPaintDC*) = 0;
        virtual void DrawNew(CClientDC*) = 0;
--> void BackingStore(CClientDC*);
--> void Undraw(CClientDC*);

--> CDC* m_pMemDC;
--> CBitmap* m_pBitmap;
};

class C_Rectangle : public C_Shape
{
    public:
        virtual void Draw(CPaintDC*);
        virtual void DrawNew(CClientDC*);
};

class C_Ellipse : public C_Shape
{

```

```

        public:
            virtual void Draw(CPaintDC*);
            virtual void DrawNew(CClientDC*);
};

class C_MainFrame : public CFrameWnd
{
    public:
        C_MainFrame();
        ~C_MainFrame();
    private:
        int m_nShape;
        BOOL m_bButtonDown;
        int m_BkMode;
        LOGPEN m_LogPen;
        LOGBRUSH m_LogBrush;
        enum EnumShape{eRectangle, eEllipse};
        EnumShape m_eShapeKind;
        #define MAX_SHAPES 100
        C_Shape* m_apShape[MAX_SHAPES];
-->// int m_FastRedrawMode;
        BOOL m_bDrawingNewShape;

        //{AFX_MSG(C_MainFrame)
        afx_msg void OnPaint();
        afx_msg void OnLButtonDown(UINT nFlags, CPoint Mouse);
        afx_msg void OnMouseMove(UINT nFlags, CPoint Mouse);
        afx_msg void OnLButtonUp(UINT nFlags, CPoint Mouse);
        afx_msg void OnPenGreen();
        afx_msg void OnPenFuschia();
        afx_msg void OnPenDash();
        afx_msg void OnPenDot();
        afx_msg void OnBrushRed();
        afx_msg void OnBrushTeal();
        afx_msg void OnBrushOlive();
        afx_msg void OnBrushBlue();
        afx_msg void OnBrushSolid();
        afx_msg void OnBrushCross();
        afx_msg void OnBrushHatchDiag();
        afx_msg void OnOpaque();
        afx_msg void OnTransparent();
-->// afx_msg void OnRedrawXORMode();
-->// afx_msg void OnRedrawNOTXORMode();
        afx_msg void OnRectangle();
        afx_msg void OnEllipse();
        afx_msg void OnDelete();
        //}}AFX_MSG

        DECLARE_MESSAGE_MAP()

};
//
//  mainfrm.cpp
//
#include •MyApp.h•
#include •mainfrm.h•
#include •resource.h•

```



```

/////////////////////////////////////////////////////////////////
//    Message map
//
BEGIN_MESSAGE_MAP(C_MainFrame, CFrameWnd)
    //{AFX_MSG_MAP(C_MainFrame)

        (eliminate two message map entries)
        (no change in any other message map entries)

-->//    ON_COMMAND(ID_XORMODE, OnRedrawXORMode)
-->//    ON_COMMAND(ID_NOTXORMODE, OnRedrawNOTXORMode)

    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////
////
///    Definitions of C_Shape, C_Rectangle, and C_Ellipse functions
///
        (no change in other function definitions)

-->void C_Shape::BackingStore(CClientDC* pMyDC)
-->{
-->    m_pMemDC = new CDC;
-->    m_pMemDC->CreateCompatibleDC(pMyDC);
-->    m_pBitmap = new CBitmap;
-->    CRect r = GetBoundingRect();
-->    m_pBitmap->CreateCompatibleBitmap(pMyDC,abs(r.right - r.left) +1,
-->                                     abs(r.bottom - r.top) + 1);
-->    m_pMemDC->SelectObject(m_pBitmap);
-->    m_pMemDC->BitBlt(0, 0, abs(r.right -r.left) + 1,
-->                   abs(r.bottom - r.top) + 1, pMyDC,
-->                   min(r.left, r.right), min(r.top, r.bottom),SRCCOPY);
-->}

-->void C_Shape::Undraw(CClientDC* pMyDC)
-->{
-->    CRect r = GetBoundingRect();
-->    pMyDC->BitBlt(min(r.left, r.right), min(r.top, r.bottom),
-->                abs(r.right - r.left) + 1, abs(r.bottom -r.top) + 1,
-->                m_pMemDC, 0, 0, SRCCOPY);
-->    delete m_pMemDC;
-->    delete m_pBitmap;
-->}

/////////////////////////////////////////////////////////////////
//    Defintion of C_MainFrame functions
//

C_MainFrame::C_MainFrame()
{
-->    Create(NULL, •FastRedraw with Backing Store•,
-->          WS_OVERLAPPEDWINDOW, rectDefault, NULL,
-->          MAKEINTRESOURCE(IDR_MENU1));

        m_bButtonDown = FALSE;

```

```

        m_BkMode = OPAQUE;
        m_LogBrush.lbColor = RGB(255, 0, 0);
        m_eShapeKind = eRectangle;
        m_nShape = 0;
-->// m_FastRedrawMode = R2_XORPEN;
        m_bDrawingNewShape = FALSE;
    }

    (no change in destructor)
//----- handler functions for WM messages -----
void C_MainFrame::OnPaint()
{
    CPaintDC MyDc(this);
    MyDc.SetBkMode(m_BkMode);

    for (int shape = 0; shape < m_nShape; shape++)
        m_apShape[shape]->Draw(&MyDc);
}

void C_MainFrame::OnLButtonDown(UINT nFlags, CPoint Mouse)
{
    if (m_nShape >= MAX_SHAPES || m_bButtonDown) return;
    switch(m_eShapeKind)
    {
        case eEllipse:
            m_apShape[m_nShape] = new C_Ellipse;
            break;
        default:
            m_apShape[m_nShape] = new C_Rectangle;
    }

    m_apShape[m_nShape]->SetParams(&m_LogPen, &m_LogBrush);
    m_apShape[m_nShape]->SetBoundingRect(Mouse.x, Mouse.y,
                                         Mouse.x, Mouse.y);

    m_nShape++;
    m_bButtonDown = TRUE;
    SetCapture();
}

void C_MainFrame::OnMouseMove(UINT nFlags, CPoint Mouse)
{
    if (!m_bButtonDown) return;
    CClientDC MyDC(this);

-->// MyDC.SetROP2(m_FastRedrawMode);
    if (m_bDrawingNewShape == TRUE)
-->        m_apShape[m_nShape - 1]->Undraw(&MyDC);

    m_apShape[m_nShape - 1]->SetRightBottom(Mouse.x, Mouse.y);
--> m_apShape[m_nShape - 1]->BackingStore(&MyDC);
    m_apShape[m_nShape - 1]->DrawNew(&MyDC);
    m_bDrawingNewShape = TRUE;
}

void C_MainFrame::OnLButtonUp(UINT nFlags, CPoint Mouse)
{
-->// CClientDC MyDC(this);

```

```

--> // m_apShape[m_nShape - 1]->DrawNew(&MyDC);

--> delete m_apShape[m_nShape - 1]->m_pMemDC;
--> delete m_apShape[m_nShape - 1]->m_pBitmap;

    m_bDrawingNewShape = FALSE;
    m_bButtonDown = FALSE;
    ::ReleaseCapture();
}

// ----- handler functions for menu items -----
// (The two functions associated with the eliminated
// menu item have been eliminated; otherwise no change
// in the handler functions)
-----

```

Previous	Table of Contents	Next

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Bitmaps

A bitmap is a collection of color values for a rectangular region of an image. The actual data for the image is recorded one pixel at a time. Each pixel can have a different color, so if the color of each pixel is saved, the image will be captured. The amount of data it takes to store a bitmap depends on the size of the image and on the number of colors the bitmap uses. For black and white systems, only one bit is needed to represent the color for each pixel. The bit can be set to one (1) for white and to zero (0) for black. For color images more bits are required. A 16-bit VGA display requires four bits per pixel; these four bits are the binary values as shown on Table 3-8 in Chapter Three. Bitmaps are not efficient ways to store large images. For example, saving a VGA screen that can display 16 colors requires $640 \times 480 \times 4$ bits, a total of 806,400 bits, or about 100K of storage. A color display that can display 6 million colors needs 3 bytes (24 bits) per pixel to specify each color; to store the same screen area would require eight times more storage than the 16-color images.

In the preceding example program, bitmaps are used for smaller portions of the screen. Save the screen image as a bitmap of a rectangular area before drawing over it. Then draw over the rectangular area and, when you wish to erase the new drawing, restore the screen to what it was before drawing by putting the bitmap image stored in memory back onto the screen.

Using a Memory Device Context

In Listing 4-2 above, the graphics are saved as a bitmap in memory using a memory device context. Windows OS uses the concept of a memory device context to convert from the format of bitmap data to the physical format used by the screen display (or printer). A memory device context is just like the device context for the screen (or printer), except that it is not tied to a specific device. A memory block is created to hold the color values for each pixel in

the rectangular screen region that is saved as a bitmap. A bitmap must be selected into the memory device context before the bitmap can be displayed on a physical device. Selecting the bitmap into a memory device context gives the operating system a chance to figure out if the color data needs to be organized in color planes (like the VGA 16-color card uses) or organized by using adjacent color bits (which are used for higher color resolution video cards).

Listing 4-2 presents the new function `C_Shape::BackingStore (CClientDC*)`, which performs the function of getting the rectangular screen region stored in memory as a bitmap. Also presented is the new function `C_Shape::Undraw(CClientDC*)`, which performs the function of taking the bitmap from memory and putting it back onto the same rectangular screen region.

A four-step process is required to create an area of memory that will hold the bitmap and to copy pixels into this bitmap. This process (which can be seen in the code for `BackingStore()`) is:

1. `CDC::CreateCompatibleDC()` is called to create a memory device context that is “compatible” with the device context supplied as an argument. (The memory device context is really a data structure that holds among other things drawing attributes (discussed in Chapter Three) and a pointer to a bitmap area. The compatible device context will have the same number of bits per pixel as the display device context whose pointer is passed to `CBitmap::CreateCompatibleBitmap()`.)
2. `CBitmap::CreateCompatibleBitmap()` is called to create a memory block which will hold the bitmap. The bitmap consists of header information and the pixel values for the rectangular region of the image. (Arguments to the call to `CBitmap::CreateCompatibleBitmap()` specify the device context for which it must be compatible and the size (width and height) of the rectangular region. The created bitmap is a drawing object.)
3. `CDC::SelectObject()` is called to bind this bitmap drawing object, given as the argument, to the memory device context.
4. `CDC::BitBlt()` is then used to copy the pixels that will be under the form of the current shape into the bitmap memory block. (Actually, a rectangular area is copied; this rectangle is chosen to just cover the form of the current shape.)

The backing store method consists of three steps repeated for each mouse move, as follows:

1. First, before the shape is drawn, capture the rectangular image of the screen where the shape will be. This is done in the function `OnMouseMove()` with the line of code:

```
m_apShape[m_nShape - 1] ->BackingStore(&MyDC);
```

The result of this step is shown in Figures 4-3a and 4-3b.

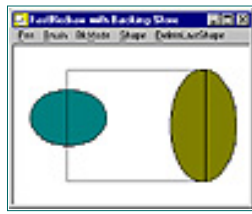


Figure 4-3a: Step One—Background Area to Capture with BackingStore()

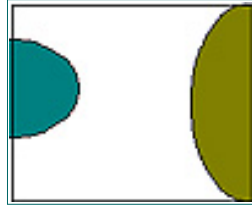


Figure 4-3b: The Bitmap Captured in Memory

2. Once the bitmap has been copied into memory, draw the newest form of the current shape. This is done in the function OnMouseMove() with the line of code:

```
m_apShape[m_nShape - 1] ->DrawNew(&MyDC);
```

The result of this step is shown on Figure 4-4.



Figure 4-4: Step Two—Drawing the New Image Using DrawNew()

3. Erase the newly drawn shape on the subsequent mouse move. This is done by the following line of code:

```
m_apShape[m_nShape - 1] ->Undraw(&MyDC);
```

This erases the newly drawn shape by taking the bitmap image from memory (shown on Figure 4-3b) and putting it back onto the screen in exactly the same position that it was in before it was captured in memory. The result of step 3, the undraw step, is shown in Figure 4-5.



Figure 4-5: Step Three—Erasing the New Image with Undraw()

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

The code for the member function `Undraw()` is much simpler than the code for `BackingStore()`. `CDC::BitBlt()` copies the bitmap back to the screen, restoring the screen to the way it was before the current shape was most recently drawn on it. It should be noted that the pointer to the memory device context, `m_pMemDC`, and the pointer to the bitmap, `m_pBitmap`, are data members of `C_Shape`; hence this information is preserved from the previous call to `OnMouseMove()`. After the bitmap has been used to undraw the shape, then the memory device context and the bitmap are deleted. This is needed to release their memory. Also, the subsequent mouse move will produce a different-sized rectangular region, which produces a different size bitmap. The rectangular area may be larger or smaller depending on whether the user is “growing” or “shrinking” the shape. A new memory device context and a new bitmap are obtained each time the mouse moves.

Now the backing store procedure is repeated for subsequent mouse moves. It is important to note that on a subsequent mouse move, the new rectangular region (either larger or smaller than the previous region) will be transferred to memory with the function `BackingStore()` before any drawing is done. The shape’s right-bottom is set to the current mouse position. The `BackingStore()` function will save a larger or smaller area of the screen (as shown in Figure 4-5), from which the new shape has been erased. It captures in memory the area of the screen without the new shape but with the newest rectangular area which may be larger or smaller than the previous rectangular area. Then the shape at the latest mouse move is drawn, repeating Step Two with a larger or smaller shape. Then Step Three is repeated. Step Three (as shown in the preceding figure, Figure 4-5) restores the screen to its original condition before any new shape is drawn.

The memory block created by `CBitmap::CreateCompatibleBitmap()` is called a *device dependent bitmap* (DDB). The DDB specifies a data format for storing bitmap information in memory. The DDB is device dependent because it holds actual hardware pixel values and does not indicate what color each pixel value

represents. Thus, the DDB is appropriate and efficient for moving data to and from a given display adapter (through a display device context), but cannot be used to accurately transfer colored images from one display device to another unless they happen to have the same palette and number of bits per pixel. *Device independent bitmaps* (DIBs) are used when transferring bitmaps from one display device to another. (DIBs are discussed later in this chapter.)

The CDC::BitBlt() Function

The CDC::BitBlt() function takes a number of parameters. Most of the parameters are associated with the location and dimension of the bitmap. There also is a raster operation code, dwROP, which is used much the same as the ROP2 code for the screen display is used. The dwROP code defines the logical combination of the colors in the destination device context, the source device context, and the brush that is currently selected into the destination device context. The meaning of the BitBlt() parameters are given in Table 4-1.

Table 4-1: Parameters of the BitBlt() Function **BOOL CDC::BitBlt**(int X, int Y, int nWidth, int nHeight, CDC* pSrcDC, int XSrc, int YSrc, DWORD dwRop);

<i>Parameter</i>	<i>Meaning</i>
X	The logical x-coordinate of the upper left corner of the destination rectangle.
Y	The logical y-coordinate of the upper left corner of the destination rectangle.
nWidth	The width in logical units of the destination rectangle.
nHeight	The height in logical units of the destination rectangle.
pSrcDC	A handle to the device context from which the bitmap will be copied. This is normally a memory device context created with CDC::CreateCompatibleDC() and with a bitmap that has been loaded into the memory device context using CDC::SelectObject().
XSrc	The logical x-coordinate of the upper left corner in the source bitmap.
YSrc	The logical y-coordinate of the upper left corner in the source bitmap.
dwRop	One of the raster operation codes. There are 256 such ROP codes, of which 15 have names defined in <windows.h>. The SRCCOPY code, which copies the source to the destination, is most often used for this parameter. A few of the others are: SCRINVERT: Combines the source and destination bitmaps using the Boolean XOR operator. (S ^ D) WHITENESS: Turns all output white. (This is a quick way to blank a device context.) BLACKNESS: Turns all output black. DSTINVERT: Inverts the destination bitmap. (~D)

SRCAND: Combines the source and destination bitmaps with the Boolean AND operator. (S & D)

NOTSRCCOPY: Inverts the source bitmap, then copies it to the destination. (~S)

SRCERASE: S & ~D

NOTSRCERASE: Inverts the result of combining the source and destination bitmaps using the Boolean OR operator. (~(S | D))

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

Using Bitmap Graphics for Animation

The most common situation that demands the use of memory bitmaps is animation. This section presents an example program which uses bitmaps to create animation. Prior to building this example program, ten bitmap resources were created. The ten bitmaps will give a motion effect when viewed successively, much like the frames of a motion picture. These are the files Frame0.bmp through Frame9.bmp. They show a ball bouncing.

When the program begins, it starts the ball bouncing, using a timer, and the ball will continue to bounce until the program ends. The executing program, called Animate, is shown in Figures 4-6a through 4-6c.



Figure 4-6a: Executing Animate Program With Ball at Bottom of Bounce



Figure 4-6b: Executing Animate Program With Ball Partway Through Bounce



Figure 4-6c: Executing Animate Program With Ball at Top of Bounce

The source code for the Animate program is given in Listing 4-3, which shows how to animate a set of bitmap resources into a moving picture. Each bitmap resource is a frame of the movie. In the animate.rc source file, each bitmap resource is identified with an ID and is associated with the file in which that bitmap resides. During compilation, these .bmp files are bound into the executable file. The resource IDs are defined as UINTs in the resource header file resource.h. The following

listing shows the mainframe class and the resource files but does not show the application class, which is the same as before.

Listing 4-3: Source Code for the Animate Program

```
-----
                                Application class
-----

    MyApp.h and MyApp.cpp have the same code as in Listing 1-1

-----
                                Mainframe class
-----
//
//  mainfrm.h
//
class C_MainFrame : public CFrameWnd
{
    public:
        C_MainFrame();
        ~C_MainFrame();
    private:
        CDC* m_apMemDC[10];
        CBitmap* m_apBitmap[10];
        int m_nCycleIndex;

        //{AFX_MSG(C_MainFrame)
        afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
        afx_msg void OnTimer(UINT nIDEvent);
        //}}AFX_MSG

    DECLARE_MESSAGE_MAP()
};

//
//  mainfrm.cpp
//
#include •MyApp.h•
#include •mainfrm.h•
#include •resource.h•

////////////////////////////////////
///      message map
///
BEGIN_MESSAGE_MAP(C_MainFrame, CFrameWnd)
    //{AFX_MSG_MAP(C_MainFrame)
    ON_WM_CREATE()
    ON_WM_TIMER()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////
///      Definition of C_MainFrame functions
///
```

```

C_MainFrame::C_MainFrame()
{
    Create(NULL, •Animation with Bitmaps•);
    m_nCycleIndex = 0;
}
C_MainFrame::~C_MainFrame()
{
    KillTimer(1);
    for (int i = 0; i < 10; i++)
    {
        delete m_apMemDC[i];
        delete m_apBitmap[i];
    }
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///handler functions for WM messages
///
int C_MainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    // load the bitmaps for animation

    CClientDC dcClient(this);

    for (int i = 0; i < 10; i++)
    {
        m_apBitmap[i] = new CBitmap;
        m_apBitmap[i]->LoadBitmap(ID_BMP_1 + i);
        m_apMemDC[i] = new CDC;
        m_apMemDC[i]->CreateCompatibleDC(&dcClient);

        m_apMemDC[i]->SelectObject(m_apBitmap[i]);
    }
    SetTimer(1, 250, NULL);
    return 0;
}

void C_MainFrame::OnTimer(UINT nIDEvent)
{
    int nBitmapIndex;
    m_nCycleIndex++;

    if (m_nCycleIndex >= 10)
        m_nCycleIndex = 0;

    nBitmapIndex = m_nCycleIndex;
    CClientDC dcClient(this);
    dcClient.BitBlt(50, 50, 320, 200, m_apMemDC[nBitmapIndex],
        0, 0, SRCCOPY);
}

-----
Resource files
-----
//
// animate.rc

```

```
//
#include "resource.h"

ID_BMP_1      BITMAP      "Frame0.bmp"
ID_BMP_2      BITMAP      "Frame1.bmp"
ID_BMP_3      BITMAP      "Frame2.bmp"
ID_BMP_4      BITMAP      "Frame3.bmp"
ID_BMP_5      BITMAP      "Frame4.bmp"
ID_BMP_6      BITMAP      "Frame5.bmp"
ID_BMP_7      BITMAP      "Frame6.bmp"
ID_BMP_8      BITMAP      "Frame7.bmp"
ID_BMP_9      BITMAP      "Frame8.bmp"
ID_BMP_10     BITMAP      "Frame9.bmp"

//
//  resource.h
//
//  the ID_BMP's must be numbered in sequential order, since
//  their ID number is used in a loop to display them
//  in their intended sequential order to achieve
//  the animated effect

#define ID_BMP_1      11
#define ID_BMP_2      12
#define ID_BMP_3      13
#define ID_BMP_4      14
#define ID_BMP_5      15
#define ID_BMP_6      16
#define ID_BMP_7      17
#define ID_BMP_8      18
#define ID_BMP_9      19
#define ID_BMP_10     20
-----
```

The program in Listing 4-3 illustrates the four steps needed to load and display bitmaps, which are as follows:

1. Load the bitmap data into memory with `CBitmap::LoadBitmap()`. (This function loads DDBs.)
2. Create the memory device context with `CDC::CreateCompatibleDC()`, which produces a memory device context with the same physical attributes as the device context given as its argument.
3. Select the bitmap into the memory device context with `CDC::SelectObject()`. When you select the bitmap into the memory device context, the operating system sets up the bitmap data with the exact sequence of bits needed to display the data on the physical device.
4. Copy the bitmap from the memory device context to the output device context (the screen) with `CDC::BitBlt()`.

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

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

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



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US



SEARCH

ITKNOWLEDGE

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

Go!

[Previous](#)

[Table of Contents](#)

[Next](#)



BROWSE

BY TOPIC

The Message Handler OnCreate()

The bitmaps are loaded into memory in the member function `C_MainFrame::OnCreate()` which responds to the message `WM_CREATE`. The operating system sends this `WM_CREATE` message when a window is about to be displayed for the first time. The `WM_CREATE` message is always sent as a notification, but does not necessarily expect the application to take action as a response. It is standard for applications to perform their custom initialization for a window when the operating system sends the `WM_CREATE` message to the window, since at this time, all data structures for the window are set up, but the window is not yet displayed. The bitmaps could be loaded in the constructor `C_MainFrame()`; in this example, however, they are loaded in `OnCreate()` to illustrate the use of the `OnCreate()` function.

`OnCreate()` first creates a `CClientDC` device context which is needed for the call to `CDC::CreateCompatibleDC()`. Then it loops over the ten bitmap resources. For each bitmap, it does the following three things: 1) Creates a `CBitmap` object and then calls `CBitmap::LoadBitmap()` to load the resource bitmap into accesible memory; 2) Creates a memory device context object and then uses `CDC::CreateCompatibleDC()` to make the memory device context compatible with the screen device context, `dcClient`; 3) Selects the bitmap into the memory device context using `CDC::SelectObject()`.

The device context object `dcClient` goes out of scope at the end of `OnCreate()`. The array of pointers to memory device context objects and the array of pointers to the bitmaps are member variables of `C_MainFrame`. These pointers and the objects they point to will be deleted when the main window closes and `C_MainFrame`'s destructor is called.

Note: There are no limits (except memory limits) on the number of memory device contexts that you may create and maintain. A memory device context is essentially a block of memory and will only be limited by the amount of available memory.

`OnCreate()` calls `SetTimer()` to start a timer. When the third parameter of `SetTimer()` is `NULL`, the operating system will send `WM_TIMER` messages to the main window at regular intervals. The timer will be “killed” when the main window closes and the `C_MainFrame`

destructor is called.

The Message Handler OnTimer()

`C_MainFrame::OnTimer()` is a message handler that responds to the `WM_TIMER` message. The code has a variable, named `m_nCycleIndex`, that increases by one each time that the `OnTimer()` function is entered (i.e., each time a timer message is received). The variable `m_nCycleIndex` is used to determine which bitmap to display on the screen. A `CClientDC` device context is created, and it then displays the bitmap by `BitBlt`-ing it onto the screen from its memory bitmap area. At each tenth timer message, `m_nCycleIndex` resets to zero. This provides the action of bouncing up and down.

Device Independent Bitmaps (DIBs)

Historically, device dependent bitmaps (DDBs) were the first kind of bitmap format that Windows OS used. The DDB just contains pixel data and a header structure. Windows OS then introduced the device independent bitmap (DIB). The DIB contains both pixel data and a palette, as well as a header structure. The palette is an array whose number of elements is the number of possible hardware pixel values. Array element i gives the red, green, and blue component intensity of that color for hardware pixel value i . Thus a program reading a DIB knows what the actual color of each hardware pixel value is. The DIB can have several modes: 1-bit (black and white), 4-bit (16 colors), 8-bit (256 colors), and 24-bit (16 million colors).

Note: Image files in either the DDB or the DIB format have the extension .bmp, so an application that reads these files must look at the header data in order to determine which format the file is in.

In the 1-bit (black and white), 4-bit (16 colors), and 8-bit (256 colors) modes, the DIB consists of a header (a struct of type `BITMAPINFOHEADER`), followed by a palette (2 to 256 `RGBQUAD`s), followed by a pixel section, which holds 1-, 4-, or 8-bit pixel values, packed into bytes. The number of possible hardware pixel values is 2, 16, and 256 respectively for the 1-bit, 4-bit, and 8-bit modes. The number of pixel values used may be less than the number of possible values. If the number used is less, then member `biClrUsed` of the `BITMAPINFOHEADER` struct is set to the number used. Otherwise, it is set to zero. The `BITMAPINFOHEADER` structure is shown in the following Listing 4-4.

Listing 4-4: The BITMAPINFOHEADER Structure

[illegible]

```
        DWORD biClrImportant; // normally 0
    } BITMAPINFOHEADER;
```

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Palettes

A palette is a table of color values for each hardware pixel value. In Windows OS each table entry of the palette specifies a color by containing the red, green, and blue component intensities of that color. The palette holds the RGB color of each hardware pixel value used. The palette is represented by an array of RGBQUADs, which have the format shown in Listing 4-5.

Listing 4-4: The RGBQUAD Structure

```

typedef struct _RGBQUAD{
    BYTE  rgbBlue;
    BYTE  rgbGreen;
    BYTE  rgbBlue;
    TYPE  rgbReserved;
} RGBQUAD;
  
```

A 24-bit DIB does not need a color palette because the pixel groups specify colors directly. In the 24-bit mode, there is no palette, and the pixel section holds the RGB values of each pixel as three consecutive bytes per pixel. This mode holds the most information about pixel color but uses the most space. If a 24-bit DIB is displayed with a 256-color display board, a 256-color palette can be defined that contains appropriate color values. Windows OS then matches the DIB's colors with the closest matching palette colors. If a palette is not defined, the DIB will be displayed with the 20 standard colors.

To understand the use of bitmaps in more detail, one must understand the use of Windows OS palettes. Figure 4-7 diagrams the Windows OS palettes and related hardware and software components. The image is produced on the monitor by reading hardware pixel values from the video RAM, translating them to RGB values with a lookup table (the hardware palette RAM), converting the digital RGB values to analog voltages with digital-to-analog

converters (DACs), and sending this information to the monitor along with synchronizing and blanking signals (not shown in Figure 4-7).



Figure 4-7: Windows OS Palettes

The System Palette

Older display adapters, such as EGA and the standard VGA, have four bits per pixel, and can therefore only display 16 colors. For these adapters, the system palette only needs to have 16 entries. The Windows operating system effectively only uses the first 16 values in the system palette.

Newer display adapters, such as SVGA have eight or more bits per pixel. Their pixels can therefore be 256 or more colors. For these adapters, Windows OS allows the corresponding number of entries in the system palette. Any running application can request the operating system to add additional colors to the system palette; the operating system provides a system that resolves conflicting requests from different applications.

The operating system maintains a table in memory which is the system palette. Table entry *i* gives the RGB value for hardware pixel value *i*. Twenty entries of the system palette are designated as the default palette. The default palette is normally set by the operating system and not changed by applications.

The default palette contains the standard 16 colors as given in Table 3-8 of Chapter Three, plus an additional 4 special colors which are: pale green, light blue, off-white, and medium gray. For a 256-entry system palette, the default palette occupies the first ten and the last ten entries. This arrangement assures that the ones-complement of a pixel value that represented a default palette entry also represents a default palette entry. To make this clear, the default palette indices are listed in two columns in the following table, Table 4-2. The number in the first column is the ones-complement of the number in the same row in the second column.

Note: It is important that the ones-complement of a palette entry also is a palette entry because pixels are often ones-complemented as a highlight. For example, the I-beam cursor inverts pixels under it.

Table 4-2: Default Palette Entries

Black	0	255	White
Dark Red	1	254	Aqua
Dark Green	2	253	Fuschia
Olive	3	252	Blue
Dark Blue	4	251	Yellow

Purple	5	250	Green
Teal	6	249	Red
Dark Gray	7	248	Gray
Special Color	8	247	Special Color
Special Color	9	246	Special Color

If the display adapter can display more than 20 colors, then the application may load additional color values into the system palette. Whenever the system palette changes, Windows OS loads the color values in the system palette into the hardware palette RAM of the display adapter. Thus the system palette will always correspond to the hardware palette in the display adapter. Although it is not often necessary, the application can use the function `CPalette::GetPaletteEntries()` to read the system palette.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Loading and Using the System Palette

To add specified colors to the system palette, the application uses `CPalette::CreatePalette()` to create a *logical palette* data structure that holds colors (RGB quads) that the application would like to have in the system palette. The application then uses `CDC::SelectPalette()` to select the logical palette into the device context for the display. The new palette becomes the object used by GDI to control the colors displayed in the device context. Finally, the application calls `CDC::RealizePalette()` to cause the operating system to try to add the logical palette's entries to the system palette. If there is an unused location in the system palette and a logical palette color is not already in the system palette, the operating system will fill it with the logical palette color. If there is no unused location, Windows OS will displace an existing system palette entry with the new color if the existing entry was placed there by an application having lower priority. The operating system has a complex priority scheme, but one aspect of it is that the active application has highest priority.

Windows OS does not tell the application which of its logical palette entries it has included in the system palette. Actually, it turns out that the application rarely needs this information, since the operating system will choose the closest color in the system palette when the application tries to draw a color that is not in the system palette.

How an application can put colors it wants into the system palette has been described. What the application must do to use those colors rather than the default palette colors is now described. To cause a pen or brush to use the closest color from the full system palette, rather than from the default palette, the color needs to be specified as `PALETTERGB(red, green, blue)`, rather than `RGB(red, green, blue)`. This not only works for pen and brush colors, but works any place that a color is specified. The differences between

PALETTERGB, which is somewhat imprecisely called “palette-relative color,” and RGB is: 1) An RGB color is always displayed using one of the default palette colors, or by a dithered combination of default palette colors when an exact match is not available in the default palette. However, when dithering is not allowed, then the closest default palette color is used; 2) A PALETTERGB color is displayed using the closest color in the full-system palette. (Dithering is never done for a PALETTERGB color.)

Displaying a DIB Using the System Palette

The function `CBitmap::LoadBitmap()` loads either a DDB resource or a DIB resource into memory and returns a pointer to this memory. When called for a DIB, `CBitmap::LoadBitmap()` converts the DIB to a DDB as it loads it into memory. Thus, for all bitmaps loaded with `CBitmap::LoadBitmap()`, only the default palette will be used for displaying the image. To load DIB data into memory use `::LoadResource()`, which does not convert the bitmap to the DDB format.

Summary

To move images on the screen or remove images from the screen, special drawing techniques are needed. Drawing techniques that can be used are drawing with the `R2_XORPEN` or the `R2_NOTXORPEN`. Both of these techniques involve drawing a shape in the specific mode, then redrawing that shape in that same specific mode to erase it. With either the `R2_XORPEN` (exclusive-or) or the `R2_NOTXORPEN` (exclusive-nor) modes, drawing the same shape a second time in the same mode causes the screen to be restored to what it was before the shape was drawn. These two drawing modes are very fast, but have some limitations for fast drawing use. They work with pens and brushes. The exclusive-or technique has the limitation of drawing in the reversed color. The exclusive-nor technique provides colors that are faithful, if the background is white; over other colors, the colors change. These techniques are best suited for rubberbanding, where just an outline is moved.

A fast drawing technique with fewer limitations is the backing store technique. To erase a shape from the screen, the backing store technique saves the graphic under the shape that is to be drawn before the shape is drawn. To erase (undraw) the shape, the graphic that was saved into the memory bitmap is put back onto the screen. The backing store technique is slower than the exclusive-or and exclusive-nor techniques, but it produces no color distortions.

Bitmaps are used for the backing store technique and for animation. Bitmaps store in memory the color value of each pixel in the rectangular region. Each pixel can have a different color; the number of colors used for a bitmap may be: 2 colors (black and white), represented by 1 bit in memory; 16 colors, represented by 4 bits in memory; or 256 colors, represented by 8 bits in memory. The function `CDC::BitBlt()` is used to move the pixel values from the screen to memory and from memory to the screen. Animation uses bitmaps that are loaded into memory from a file. These bitmaps are then successively moved onto the screen using the `BitBlt()` function to create the animated effect.

A device dependent bitmap (DDB) contains the pixel values for the current device that it is to be rendered upon. The output device could be a black-and-white display, an output device that displays 16 colors, etc. A device independent bitmap (DIB) contains sufficient information, which is palette information, that the bitmap can be transferred between different output devices, for example, from a black-and-white display to a 16-color display. A palette is a table of color values for each hardware pixel value. For display devices other than the black-and-white devices, each palette entry gives the RGB value for that entry. Twenty entries of the system palette are designated as the default palette; they are set by the operating system and not changed by applications. The default palette contains the standard 16 colors plus an additional 4 special colors: pale green, light blue, off-white, and medium gray.

Previous	Table of Contents	Next

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

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

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

 SEARCH
ITKNOWLEDGE

Go!

Keyword

☒ Brief ☐ Full

[Advanced Search](#)

[Search Tips](#)

 BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

Go!

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

Exercise

In this chapter you have learned how to move images on the screen using several techniques. You now have the skills to make a checker piece move. When the user presses the left mouse button down over a piece, it makes that piece move, following the mouse to wherever the user moves it. Then when the user releases the left mouse button, it draws the piece on that checkerboard square that it is over when the left mouse button is released. The user will be moving the pieces by pressing down on the mouse button over the piece to be moved, holding the mouse button down, and moving the mouse, effectively dragging the piece to its new position. When the new position is reached, the user lets up on the left mouse button and the piece is painted on the new checkboard square. Of course, the move must be a legal move. If it is not a legal move, then the piece should be returned to its original square. The section titled “Chapter 4 Exercise” of Appendix B provides hints and further discussion.

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

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

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

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



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

Go!

[Previous](#)

[Table of Contents](#)

[Next](#)

Chapter 5 Child Windows

Up to this point, only programs with a main window have been created. Child windows are introduced and discussed in this chapter. Child windows are created and managed by a parent window. In this chapter, we will first look at an ordinary child window that is physically attached to its parent and hence is visible only if it is within the client area of its parent. We will then look at what is known as a *popup window*, which is also a child window. Although created and managed by a parent, the popup window is visible anywhere on the screen, even outside the client area of its parent. Finally, we look at an ordinary child window whose location is fixed within the client area.

All subsequent chapters in this book use child windows. Chapters Six, Seven, and Eight deal with dialog boxes (which are popup windows) and controls (which are fixed child windows). Beginning with Chapter Nine and for the remaining chapters, we will be dealing with the CDocument and CView classes; the CView class provides C++ objects that are fixed child windows.

Child and popup windows typically process messages independently from their parent windows. They can communicate by sending each other *user messages*. The Windows OS allows user messages to be defined that can have any meaning desired. We will look at user messages in this chapter. (User messages are also used in Chapter Eight for communication between parent windows and their child windows.)

A Child Window

The first example in this chapter creates a single child window that can be moved within the parent's client area. Figure 5-1 shows this ChildWin

BROWSE
BY TOPIC

program in action. When the “CreateChild” menu item is selected, a child window appears inside the main program window. When the child window initially appears, it displays the string, “I am the child!” When the “OrdertoChild” menu item is selected, the parent causes the child to draw the text “Got Order From Parent” in the child’s window area. The main program window is created using the same registered window class that was used in Listing 1-3. The main window has the stock UpArrow cursor, a red background, and the stock exclamation icon.

The child window is created using its own registered class. It has the stock hourglass cursor, a yellow background, and a stock icon. When the cursor is in the main window client area, outside the child window, it becomes the UpArrow cursor. When it is inside the child window it becomes the hourglass cursor. The child window has a caption bar, a minimize box, a maximize box, and it is resizable. Since the child window has a caption bar, it is moveable but is only visible within the bounds of the parent’s client area; it is *clipped* when the child window is moved beyond the edge of the parent window’s border. The clipping of the child window is done automatically by the built-in logic of the Windows OS.

Figure 5-1 shows the bottom of the child window being clipped by the parent window. The child window is visible only if the parent window is visible. If the parent window is closed, the child window disappears. When the child window is maximized, it fills the entire client area, and when it is minimized, it appears in a launch bar at the bottom left of its parent.

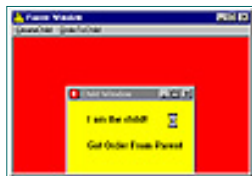


Figure 5-1: Executing ChildWin program

The ChildWin program is shown in Listing 5-1. In addition to the application class and the mainframe class, the listing has a child class and stdafx files. The child class contains the code for the child window. Since we now have more classes, we need a way for all classes to access the standard include files; the mechanism of adding stdafx files as shown in the listing solves this problem and is used in all tool-generated applications, so we use it here.

The mainframe class initiates the creation of the child window and manages the child. In the mainframe class a variable, `m_bChildOpen`, is used to keep track of whether or not a child window is open. An important thing to note in the following code is that an object, named `m_Child`, of class `C_ChildWindow`, is declared as a data member of `C_MainFrame`. This gives the mainframe the ability to communicate with and to manage the child. When the menu item “Create Child” is selected, the `OnCreateChild()` function is entered, if a child does not already exist. In this function, the Child’s function, `MakeChild()`, is invoked from the main frame class by the call, `m_Child.MakeChild()`. When the child’s function, `MakeChild()`, is invoked, it registers a new window class with an hourglass cursor, a yellow background, and a standard icon. Then it invokes the function `CWnd::Create()` to create the

child window of that registered class. When the menu item “OrderToChild” is selected, the OnOrderToChild() function is entered. If a child exists, then the parent causes the child to respond through the call m_Child.RespondToParent().

The function RespondToParent() is a member function of the class C_ChildWindow and it causes the text string “Got Order From Child” to be drawn in the client area of the child window. The mainframe is managing the child by accessing the child’s functions using the C_ChildWindow object, m_Child. An additional benefit of having the object m_Child as a data object of the main window is that it will automatically be deleted from memory and the screen when the main window is destroyed.

Previous	Table of Contents	Next

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

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

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

 SEARCH
ITKNOWLEDGE

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

 BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Listing 5-1: Source Code for ChildWin Program

```

-----
                                stdafx files
-----
//
//  stdafx.h  include file for standard system include files
//
#include <afxwin.h>

//
//  stdafx.cpp  source file for standard includes
//
#include "stdafx.h"

-----
                                application class
-----
//
//  myapp.h:  main header file for the MyApp application
//
class C MyApp : public CWinApp
{
public:
    BOOL InitInstance();
};
//
//  myapp.cpp:  defines the class behaviors for the application
//
#include "stdafx.h"
#include "myapp.h"
#include "mainfrm.h"

BOOL C MyApp::InitInstance()
{
    m_pMainWnd = new C_MainFrame;
    m_pMainWnd->ShowWindow(m_nCmdShow);
  
```

```

        return TRUE;
    }

    C MyApp  theApp;    // the object that runs the program

-----
                                mainframe class
-----
//
//  mainfrm.h:  interface of the C_MainFrame class
//
#include "child.h"

class C_MainFrame : public CFrameWnd
{
    public:
        C_MainFrame();
    private:
        BOOL m_bChildOpen;
        C_ChildWindow m_Child;

        //{AFX_MSG(C_MainFrame)
        afx_msg void OnCreateChild();
        afx_msg void OnOrderToChild();
        //}}AFX_MSG

        LONG OnCloseChild(UINT, LONG);

    DECLARE_MESSAGE_MAP()
};

//
//  mainfrm.cpp
//
#include "stdafx.h"
#include "mainfrm.h"
#include "resource.h"

BEGIN_MESSAGE_MAP(C_MainFrame, CFrameWnd)
    //{AFX_MSG_MAP(C_MainFrame)
    ON_COMMAND(ID_CREATECHILD, OnCreateChild)
    ON_COMMAND(ID_ORDERTOCHILD, OnOrderToChild)
    ON_MESSAGE(WM_USER, OnCloseChild)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

C_MainFrame::C_MainFrame()
{
    m_bChildOpen = FALSE;
    CBrush redBrush(RGB(255, 0, 0));
    CString MainWindowClass = AfxRegisterWndClass(
        CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW,
        AfxGetApp()->LoadStandardCursor(IDC_UPARROW),
        (HBRUSH)redBrush.GetSafeHandle(),
        AfxGetApp()->LoadStandardIcon(IDI_EXCLAMATION));
    Create(MainWindowClass, "Parent Window", WS_OVERLAPPEDWINDOW,

```

```

        rectDefault, NULL, MAKEINTRESOURCE(IDR_MENU1));
    }

void C_MainFrame::OnCreateChild()
{
    if (m_bChildOpen)
        return;

    CRect ChildRect(10, 30, 210, 180);
    m_Child.MakeChild("Child Window", WS_CAPTION | WS_THICKFRAME |
        WS_VISIBLE | WS_MINIMIZEBOX | WS_MAXIMIZEBOX |
        WS_SYSMENU, ChildRect, this, NULL);
    m_bChildOpen = TRUE;
}

void C_MainFrame::OnOrderToChild()
{
    if (!m_bChildOpen)
        return;
    m_Child.RespondToParent();
}

LONG C_MainFrame::OnCloseChild(UINT wParam, LONG lParam)
{
    m_bChildOpen = FALSE;
    return 1;
}

-----
                                child class
-----
//
//  child.h
//
class C_ChildWindow : public CWnd
{
public:
    C_ChildWindow();
    BOOL MakeChild(LPCSTR WindowTitle, DWORD dwStyle, RECT& rect,
        CWnd* pParentWnd, UINT ID = NULL);
    void RespondToParent();
private:
    //{{AFX_MSG(C_ChildWindow)
    afx_msg void OnPaint();
    afx_msg void OnDestroy();
    //}}AFX_MSG

    DECLARE_MESSAGE_MAP()
};

//
//  child.cpp
//
#include "stdafx.h"
#include "child.h"

```

```

BEGIN_MESSAGE_MAP(C_ChildWindow, CWnd)
   //{{AFX_MSG_MAP(C_ChildWindow)
        ON_WM_PAINT()
        ON_WM_DESTROY()
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()

C_ChildWindow::C_ChildWindow()
{
}

BOOL C_ChildWindow::MakeChild(LPCSTR WindowTitle, DWORD dwStyle,
                              RECT& rect, CWnd* pParent, UINT nID)
{
    CBrush yellowBrush(RGB(255, 255, 0));
    CString ChildWindowClass = AfxRegisterWndClass(
        CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW,
        AfxGetApp()->LoadStandardCursor(IDC_WAIT),
        (HBRUSH)yellowBrush.GetSafeHandle(),
        AfxGetApp()->LoadStandardIcon(IDI_HAND));
    CWnd::Create(ChildWindowClass, WindowTitle, dwStyle,
        rect, pParent, nID);
    return TRUE;
}

void C_ChildWindow::OnPaint()
{
    CPaintDC dc(this);
    dc.SetBkMode(TRANSPARENT);
    dc.TextOut(30, 20, "I am the child!");
}

void C_ChildWindow::RespondToParent()
{
    CClientDC dc(this);
    dc.SetBkMode(TRANSPARENT);
    dc.TextOut(30, 60, "Got Order From Parent");
}

void C_ChildWindow::OnDestroy()
{
    CWnd::OnDestroy();
    GetParent()->PostMessage(WM_USER);
}

-----
                                resource files
-----
//
//  ChildWin.rc
//
#include "resource.h"

IDR_MENU1    MENU    DISCARDABLE
BEGIN
    MENUITEM "&CreateChild",    ID_CREATECHILD
    MENUITEM "&OrderToChild",    ID_ORDERTOCHILD

```


END

```
//  
// resource.h  
//  
#define IDR_MENU1 101  
#define ID_CREATECHILD 40001  
#define ID_ORDERTOCHILD 40002  
-----
```

Now that we have explored the class CBrush in Chapter Three, we use a CBrush in the registration of the main window and the registration of the child window. Remember that in Chapter One, we created an HBRUSH object before calling the function AfxRegisterWndClass(). At this point, we use a CBrush object. We get its handle, and cast it to an HBRUSH, since the third parameter of AfxRegisterWndClass() must be an HBRUSH, with the following expression:

```
(HBRUSH)redBrush.GetSafeHandle()
```

Previous	Table of Contents	Next

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

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

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

Keyword

☒ Brief
 ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

The CWnd::Create() Function

Child windows are created using the CWnd::Create() function. A child window can be constructed in two steps. First, construct the m_Child object. Then, invoke the CWnd::Create() function, and the child window is attached to the invoking object, m_Child. MakeChild() is the function in which the child window class is registered. After the preceding steps are completed, the CWnd::Create() function can be called with the following line of code:

```
CWnd::Create(ChildWindowClass, WindowTitle, dwStyle, rect,
             pParent, nID);
```

Note: In the preceding line of code, the call CWnd::Create() was used to emphasize that it is the CWnd function which is being called. It should be noted that since the class C_ChildWindow is derived from CWnd, the preceding line of code could also be written as:

```
Create(ChildWindowClass, WindowTitle, dwStyle, rect,
       pParent, nID);
```

The CWnd::Create() function is used only for child windows. Remember main windows are created using a different function, CFrameWnd::Create(), which has been used in the previous chapters. These two functions look similar, but they are used differently and they have differing input parameters.

Table 5-1 gives the input parameters and their definitions for the function CWnd::Create(). Some of the input parameters are similar to the CFrameWnd::Create() function, such as the window class name, window name, and window style attributes. But the similarity ends there. For CWnd::Create(), the rect giving initial size and position must be in the parent window's client coordinates. As the preceding code demonstrates, we obtained the rect from within the parent window and then passed it to the MakeChild() function. The pointer to the parent window

(which is the keyword *this* from within the mainframe class) is also passed to the `MakeChild()` function. In this example the mainframe class has no need to establish an identifier for the child window, so we passed `NULL` for the `nID` parameter to the `MakeChild()` function.

Table 5-1: *CWnd::Create() Function Parameters*

BOOL CWnd::Create(LPCSTR lpszClassName, LPCSTR lpszWindowName, DWORD dwstyle, CWnd* pParentWnd, RECT& rect, UINT nID, CCreateContext* pContext = NULL);

<i>Parameter</i>	<i>Description</i>
<code>lpszClassName</code>	A pointer to a null-terminated character string that names the window class upon which the created window will be based. The window class name can be any name registered with the global <code>AfxRegisterWndClass</code> function or any of the predefined control-class names. If <code>NULL</code> , the default <code>CWnd</code> window class is used.
<code>lpszWindowName</code>	A pointer to a null-terminated character string that contains the window name. This string is used as text for the caption bar.
<code>dwStyle</code>	Specifies the window style attributes. This value can consist of a series of binary flag values that specify properties of the window. The properties can be properties that apply only to child windows (see Table 5-2) or window properties (see Table 1-5 in Chapter One).
<code>rect</code>	A reference to a <code>RECT</code> structure that contains the initial size and position of the window when it is created. The size and position must be provided in client coordinates of the parent window.
<code>pParentWnd</code>	A pointer to the parent window of the child window being created.
<code>nID</code>	The ID of the child window. Used if you wish to assign an ID to your window. If you do not wish to assign an ID, use <code>NULL</code> .
<code>pContext</code>	Specifies a pointer to the create context of the window. The pointer can be <code>NULL</code> .
Return Value:	Nonzero if creation is successful; otherwise, a zero is returned.

There are window style attributes that can be used for child windows only; these styles are given in Table 5-2. This first example, `WinChild`, creates a child window of style `WS_CHILD`. In the second example in this chapter the `WS_POPUP` style is used. Chapters Six and Seven will discuss the dialog styles used and the usage of the styles `WS_GROUP` and `WS_TABSTOP` for controls.

Table 5-2: *Window Style Attributes That Apply Only to Child Windows*

WS_CHILD	Creates a child window. This is a window that cannot be moved outside of its parent's frame and that moves with the parent window. Cannot be used with WS_POPUP style.
WS_CHILDWINDOW	Creates a child window that has the WS_CHILD style.
WS_CLIPSIBLINGS	Excludes sibling's areas when drawing in a child's client area. If this attribute is not set, then a child can draw in a sibling that is in front, but Windows OS then immediately redraws the sibling in front. This attribute is for use with WS_CHILD style only.
WS_DLGFRAME	Creates window with a modal dialog box frame but no title. This attribute cannot be used with WS_CAPTION style.
WS_GROUP	Specifies the first control of a group of controls in which the user can move from one control to the next by using the arrow keys. All controls defined with the WS_GROUP style after the first control belong to the same group. The next control with the WS_GROUP style ends the style group and starts the next group (that is, one group ends where the next begins). This style is valid only for controls.
WS_POPUP	Creates a popup window. This is a kind of child window that can be moved outside of the parent's client area. This attribute cannot be used with WS_CHILD style.
WS_POPUPWINDOW	A popup window that is the composite of the following individual styles: WS_POPUP WS_BORDER WS_SYSMENU. The WS_CAPTION style must be combined with the WS_POPUPWINDOW style to make the system menu visible.
WS_TABSTOP	Specifies one of any number of controls through which the user can move by using the Tab key. The Tab key moves the user to the next control specified by the WS_TABSTOP style. This style is valid only for controls.

Note: As you can note in the ChildWin example, you do not need to use the style WS_CHILD (but you can if you wish) because the CWnd::Create() creates only child windows.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Message Maps for Child Windows

A child window can have its own message map. Messages from the Windows OS to the child are not intercepted by the parent window. The child window's message map and message handler functions are completely independent of the parent, and they are included in the C_ChildWindow class. In Listing 5-1, the child window's implementation file, child.cpp, contains code which does the following:

- Creates the child window in the MakeChild() function by calling the CWnd::Create() function. It also provides other functions, as necessary. In this example, the child responded with the RespondToParent() function when the parent's menu item "OrderToChild" was selected;
- Provides a separate message map for the child window's messages, and also provides handler functions for the messages that the child receives.

In the ChildWin example, the child responds to the messages WM_PAINT and WM_DESTROY. When a WM_PAINT message is received by the child, its OnPaint() handler function is entered and the string "I am the Child!" is drawn in the child's client area. The WM_PAINT message is sent when the window first appears and when the window is resized.

The child also receives the WM_DESTROY message. A WM_DESTROY message is sent to the child window when the child window is to be destroyed. When the child window is destroyed, the parent must be notified so that it can note (using the variable m_bChildOpen) that the child no longer exists. However, the parent window and the child window are different C++ objects, are defined in different C++ classes, and have completely separate message maps. So how do you make the child window communicate with the parent window? Send a *user-defined message*.

User Messages

So far, the WM_ messages that the Windows OS sends have been demonstrated.

The operating system allows the user to define and use his own message, known as WM_USER. WM_USER is, of course, an ID (or preprocessor) constant. It is defined by the operating system and is the next constant beyond the usual messages defined in <windows.h>. When defining your own messages, you can assign the ID of WM_USER to your first message, the ID of WM_USER + 1 to your next message, and so on. The ID of your message must be equal to or above WM_USER, which is the lowest message number you can use to define your customized message.

In the ChildWin example, only one customized message is needed and it is identified as WM_USER. When the child is being destroyed, it sends the customized message to the parent with the following line of code:

```
GetParent() ->PostMessage(WM_USER);
```

The parent's message map includes an entry to route the WM_USER message to a handler function, which is:

```
ON_MESSAGE(WM_USER, OnCloseChild)
```

The ON_MESSAGE message map entry is a special one, in that it allows any customized message to be mapped. The general form of the message map entry for user messages is:

```
ON_MESSAGE(<message>, <member function>)
```

Note: The inclusion of the afx_msg brackets in the preceding code, illustrates the fact that a compiler tool was used to generate the message maps and to "stub out" the message handler functions. However, compiler tools cannot help with customized messages. All lines of code for a customized message must be entered into the program manually.

The message handler function for a customized message has the following prototype which must appear in the class' header file:

```
LONG memberFunction(UINT, LONG);
```

In the ChildWin example the line of code given below must be placed in the parent's (C_MainFrame) header file, mainfrm.h. It is placed outside the afx_msg brackets. (If it were placed inside the afx_msg brackets, it must be preceded with afx_msg, or the compiler tool will not be able to function.) The UINT parameter is normally used for a wParam and the LONG parameter is normally used for a lParam.

```
LONG OnCloseChild(UINT, LONG);
```

The handler function definition must also be coded manually; it is in the mainfrm.cpp file. The definition of the handler function in the ChildWin example is:

```
LONG C_MainFrame::OnCloseChild(UINT wParam, LONG lParam)
```

```
{  
    m_bChildOpen = FALSE;  
    return 1;  
}
```

When the user message is received by the parent, it causes the OnCloseChild() function to be exercised. The variable m_bChildOpen is set to FALSE, which denotes that the child window has been closed. The function must return a LONG, so the function returns one (1). Once the child window has been destroyed, the main frame window can create another child window.

Previous	Table of Contents	Next

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

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

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

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

[Previous](#)[Table of Contents](#)[Next](#)[BROWSE
BY TOPIC](#)

A Popup Window

The next type of window explored in this chapter is the popup window. Popup windows are related to a parent window; however, they are not physically attached to the parent window. Popup windows are not restricted to the parent window's client area. They can appear anywhere on the screen and can cover up portions of the parent window and other program windows. A popup window can never be covered by its own parent window; the parent window will always be "under" the popup window. This is true even if the parent window gains the focus. Both the popup window and its parent window can be covered and uncovered by other windows.

The following program, named PopUp, creates a popup window. Figure 5-2 shows this PopUp program in action; the child window is shown completely outside the area of the main window. The program basically is the same program as the ChildWin program, but it creates the child window using the WS_POPUP window style attribute. The popup window can be moved such that it extends well beyond the bounds of the parent window and can be completely separated from the parent on the screen, as shown in the Figure 5-2. The popup is activated whenever any area of its window is clicked on. The parent window can be reactivated by clicking anywhere on the parent window. When the popup window is maximized, it fills the entire screen, completely obscuring the parent window. When the popup window is minimized, it appears in the task bar at the bottom left of the desktop. If the parent window is minimized, the popup window automatically disappears from the screen. When the parent is restored, the popup automatically reappears at its previous screen location.

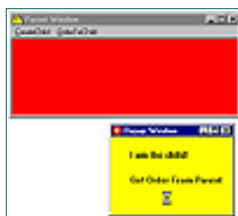


Figure 5-2: Executing PopUp Program

Creating popup windows is just like creating child windows, except that the WS_POPUP window style attribute is used in the CWnd::Create() function call. WS_CHILD and

WS_POPUP styles cannot be used at the same time to create a window because the two styles are mutually exclusive. The coordinates passed to CWnd::Create() will be translated to screen coordinates for popup windows; they are not used as client coordinates of the parent window, as they are for the ordinary child window in the ChildWin example. This makes sense, as popup windows can be located anywhere on the screen, while ordinary child windows are restricted to the parent window's client area.

More popup windows are presented in the following chapters: the modal dialog boxes used in Chapters Six and Seven are popup windows, and Chapter Eight introduces *modeless dialog boxes*, which are also popup windows.

Excerpt 5-1, which is taken from the PopUp program, demonstrates the changes necessary for the creation of a popup window. The remainder of the PopUp program is identical to the ChildWin program that has been given in Listing 5-1. Since the bulk of the PopUp program is the same as in the ChildWin example, only the changes that make the program obtain the popup window are shown. The only program changes are in the C_MainFrame::OnCreateChild() function and involve the parameters passed to the child's function, MakeChild(). Excerpt 5-2 shows the new OnCreateChild() function in its entirety. The lines of code that were changed from the ChildWin program are in bold and preceded by an arrow.

Excerpt 5-1: Excerpt from PopUp Program

```
void C_MainFrame::OnCreateChild()
{
    if (m_bChildOpen)
        return;
    CRect ChildRect(10, 30, 210, 180);
--> m_Child.MakeChild("PopUp Window", WS_POPUP | WS_THICKFRAME |
-->                               WS_CAPTION | WS_VISIBLE | WS_MINIMIZEBOX |
-->                               WS_MAXIMIZEBOX | WS_SYSMENU, ChildRect,
-->                               this, NULL);
    m_bChildOpen = TRUE;
}
```

In the PopUp program, the only changes made to the previous code were: changing the WindowTitle variable to "Popup Window," and adding the WS_POPUP style to the list of window style attributes in the second parameter of MakeChild().

Previous	Table of Contents	Next

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

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

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

Keyword

☒ Brief ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

A Fixed Child Window

Sometimes it is useful to have fixed child windows that cannot be moved. Controls are examples of small, fixed child windows that cannot be moved. The window of the CView class (*view window*) is another example. View windows are presented in Chapters Nine through Fourteen. The view window is a fixed child window that completely covers the parent window's client area. In this section we will explore how to create just such a fixed window—one that covers the parent window's client area completely.

This last program in this chapter creates a fixed child window that completely covers the parent's client area. Figure 5-3 shows the FixedChd executing program when it first opens up, before the child window has been created. Again, the ChildWin program introduced first in this chapter is used, with modifications. When the FixedChd program originally opens up, the parent window which has a red background and an UpArrow cursor is visible. When the user clicks on the menu item "C_createChild," the child window is created. The child window has no border and no caption bar. What is visible when the child takes over the parent's client area is the child's background color (yellow) filling the entire client area and the child's cursor, which is the hourglass. Also, the child writes the text "I am the child!" in its client area, which is another indication that the child has taken over the client area.

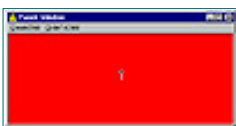


Figure 5-3: Executing FixedChd Program When First Opened

Figure 5-4 shows the FixedChd executing program after the child has been created and an order has been sent to the child. The fixed child moves with its parent when the parent is moved with the caption bar or re-sized using the thick border. The child always fills the entire client area of the parent. In this FixedChd example, the visual clues of the yellow-color and the text are the only way of knowing that the child occupies the parent's client area.



Figure 5-4: Executing FixedChd Program After Child is Created

Much of the FixedChd program is the same as in the ChildWin program. Since the child cannot be closed, the response to the message WM_DESTROY is absent from the code in the child class and the user message and its handler function are absent from the code in the mainframe class.

Code additions were made in the mainframe class only; both the mainfrm.h and mainfrm.cpp files have added code. The mainframe class of the FixedChd program is shown in Listing 5-2. The child is created and fills the entire client area of the main window. The mainframe class in the FixedChd program responds to the WM_SIZE message; it moves the child window such that it continuously occupies its entire client area. The lines of code that are added to the ChildWin program given in Listing 5-1 are in bold and preceded by an arrow.

Listing 5-2: The Mainframe Class of the FixedChd Program

```
-----
                                mainframe class
-----
//
//  mainfrm.h:  interface of the C_MainFrame class
//
#include "child.h"

class C_MainFrame : public CFrameWnd
{
    public:
        C_MainFrame();
    private:
        BOOL m_bChildOpen;
        C_ChildWindow m_Child;
        //{AFX_MSG(C_MainFrame)
        afx_msg void OnCreateChild();
        afx_msg void OnOrderToChild();
e      afx_msg void OnSize(UINT nType, int cx, int cy);
        //}}AFX_MSG

        DECLARE_MESSAGE_MAP()
};

//
//  mainfrm.cpp
//
#include "stdafx.h"
#include "mainfrm.h"
#include "resource.h"

BEGIN_MESSAGE_MAP(C_MainFrame, CFrameWnd)
    //{AFX_MSG_MAP(C_MainFrame)
    ON_COMMAND(ID_CREATECHILD, OnCreateChild)
    ON_COMMAND(ID_ORDERTOCHILD, OnOrderToChild)
-->    ON_WM_SIZE()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

```

C_MainFrame::C_MainFrame()
{
    m_bChildOpen = FALSE;
    CBrush redBrush(RGB(255, 0, 0));
    CString MainWindowClass = AfxRegisterWndClass(
        CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW,
        AfxGetApp()->LoadStandardCursor(IDC_UPARROW),
        (HBRUSH)redBrush.GetSafeHandle(),
        AfxGetApp()->LoadStandardIcon(IDI_EXCLAMATION));
    Create(MainWindowClass, "Parent Window", WS_OVERLAPPEDWINDOW,
        rectDefault, NULL, MAKEINTRESOURCE(IDR_MENU1));
}

void C_MainFrame::OnCreateChild()
{
    if (m_bChildOpen)
        return;
-->    CRect ClientRect;
-->    GetClientRect(ClientRect);
-->    m_Child.MakeChild("", WS_VISIBLE, ClientRect, this, NULL);
    m_bChildOpen = TRUE;
}

void C_MainFrame::OnOrderToChild()
{
    if (!m_bChildOpen)
        return;
    m_Child.RespondToParent();
}

-->void C_MainFrame::OnSize(UINT nType, int cx, int cy)
-->{
-->    CRect ClientRect;
-->    GetClientRect(ClientRect);
-->    m_Child.MoveWindow(ClientRect, TRUE);
-->}
-----

```

In this FixedChd program, the changes made to the previous code are:

- The parameters in the call to the MakeChild() function are changed. The WindowTitle is an empty string, the only window style attribute that is used is WS_VISIBLE, and the initial size and position of the child window when it is created is specified as the entire parent client area. A parameter named ClientRect which is obtained with the GetClientRect(ClientRect) function call contains the size and position of the parent window. The parameter ClientRect is used when the child window is created.
- The mainframe responds to the WM_SIZE message. Whenever the parent window is resized the C_MainFrame::OnSize() function is entered. This function gets the size and position of the parent window in the parameter named ClientRect. Then the following line of code is executed, which causes the child window to be moved to the parent window's size and position.

```

m_Child.MoveWindow(ClientRect, TRUE);

```

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

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

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

☒ Brief ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

 **Bookmark It**

Search this book:

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

Summary

Child windows are created and managed by a parent window. Child windows can be created with the WS_CHILD or WS_POPUP styles, which are mutually exclusive styles. A child window can have its own message map. Messages from the Windows OS to the child are not intercepted by the parent window. The child window's message map and message handler functions are completely independent of the parent.

Parent and child windows frequently employ user-defined messages to communicate with each other. A user-defined message is identified by an ID beginning with WM_USER. WM_USER is defined by the operating system and is the next constant beyond the usual messages defined in <windows.h>. A user-defined message has a specified prototype that is used in its message handler function. All code pertaining to user-defined messages must be entered manually; the compiler tools do not support user messages.

A child window created with the WS_CHILD style produces a child window, considered to be the ordinary style for child windows, that is confined within the borders of its parent window. When an ordinary child window is minimized, its icon is found in the lower left of the parent window. When it is maximized, it fills the client area of the parent window.

When a child window is created with the WS_POPUP style, its behavior is different. Its coordinates are defined in terms of screen coordinates; it is not confined to the boundaries of the parent window's client area. It can be moved entirely outside the boundaries of the parent window. When a popup child window is minimized, its icon appears in the desktop task bar (launch bar). When a popup child window is maximized, it fills the entire desktop area. When the parent window is closed, its child popup window is also closed. Modal dialog boxes and modeless dialog boxes are popup child windows.

Another version of the ordinary child window is a child window that is fixed to a specific area or location. Examples of this specialized child window are controls and the view window used in the document-view architecture. These ordinary child windows are specialized; they do not contain a caption bar and do not have a close feature—they cannot be closed. The view window does not have a border and always occupies the entire client area of the parent window. The example given in this chapter, FixedChd, does not have a border and always occupies the entire client area of its parent window.

Previous	Table of Contents	Next

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

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

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

Keyword

☒ Brief
 ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

Chapter 6

Dialogs, Common Dialogs, and Button Controls

In this chapter, the dialog box is introduced. The Common Dialogs are introduced and an example of their use is presented. An example program is given which has button and static controls attached directly to the mainframe window; this program also has a dialog box filled with button controls.

At this juncture, we once again will use the compiler tools to help write code for us. How the code is built is presented for the main window, which has controls directly attached. Building the dialog box using the Visual C++'s Dialog Editor is presented. Using Visual C++'s ClassWizard to encapsulate the dialog box in a C++ class derived from CDialog is also presented. Once the dialog has been encapsulated in its C++ class, access is gained (through inheritance) to many useful CDialog member functions.

Dialogs

Almost every program uses a *dialog window*, commonly called a “dialog box.” A dialog box is a window that is used to “hold” (or, rather, to provide a parent for) the controls, which are also small windows. These small control windows are children of the parent dialog box. It is through the control windows that the user makes selections or enters moderate amounts of text information. The dialog parent window receives messages from its child control windows, and sports a message map to handle these messages from its children. Messages from the child controls are called *control notification messages*. Drawing can be performed in the dialog window's client area, which is normally not done. Instead, child controls are specified to be in the dialog box's client area. The

controls display information or accept small amounts of text and the user interacts with these controls.

Dialog boxes usually exist for a short period of time, while the user is making selections, although they may exist for the life of the program. This depends on the type of dialog used. Dialog boxes come in two types: *modal dialog boxes* and *modeless dialog boxes*. Whether the dialog box is constructed and displayed in your program as a modal or a modeless box depends on the MFC functions called in the dialog class to construct and show the dialog window. (This chapter and Chapter Seven discuss modal dialog boxes. Chapter Eight discusses modeless dialog boxes.)

Modal vs. Modeless Dialog Boxes

Modal dialog boxes are generally more common than modeless ones. The CDialog base class supports both modal and modeless dialogs. To support either a modal or modeless dialog box, encapsulate your dialog in a class derived from CDialog.

When a modal dialog box is created, it deactivates all other windows of its application, so that the application will not respond to the mouse or keyboard; the user must complete the dialog before again working with the application. The user completes the dialog, then clicks on the OK or Cancel button. The dialog box then destroys itself and reactivates its parent window. A modal dialog box can be made *task modal*, so that no other windows in the application will respond while the dialog box exists, or it can be made *system modal*, so that no other interface elements in the entire Windows system will respond while the dialog box exists.

Note: In Win3.1x, the system modal dialog box behaves as just described. It should be noted that although system modal dialog boxes are supported by the Win32 operating systems, a Win32 system modal dialog box behaves like a task modal dialog box. As a result, it is recommended that you don't use system modal dialog boxes in Win32 applications.

A modeless dialog box does not deactivate its parent. An example of a modeless dialog box is a graphics drawing tool that lets the user change the color of a brush and continue to paint in the application while the graphics drawing tool remains open and available for the user to change selections as he chooses. The dialog box could be moved by the user to keep it from obscuring the user's drawing, and it would remain on the screen while the user draws, so that it is easy to access.

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

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

Keyword

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Common Dialogs

With the release of Windows 3.1, Microsoft introduced the *Common Dialogs*. These are dialog boxes that implement specific commonly used dialogs. Many Windows applications (especially Microsoft's applications) have already-implemented dialogs that are similar to or identical to the Common Dialogs. All applications should use the Common Dialogs as much as possible, so that Windows applications will have an interface that is as consistent as possible. MFC classes have been defined that make it simple to use these predefined dialog boxes by providing an MFC dialog class for each type of Common Dialog. To take advantage of these Common Dialog classes, include the <afxdlgs.h> header file in your program. The following paragraphs briefly discuss the MFC library class for each Common Dialog and the purpose of each Common Dialog. These classes produce modal dialog boxes with the single exception of the class CFindReplaceDialog, which produces modeless dialog boxes.

Class CFileDialog

The CFileDialog class encapsulates the Windows common file dialog box. Common file dialog boxes provide an easy way to implement "File|Open" and "File|Save As" dialog boxes in a manner consistent with Windows standards.

In the CFileDialog class, the creation function GetOpenFileName() creates a modal "File|Open" dialog box. This dialog box gets a path name of an input file from the user. This dialog would typically be used when the user selects the "File|Open" menu item to let the user specify the file that he wants to open. Of course, this dialog does not actually open the file; this requires additional code that must be supplied. This Common Dialog is illustrated in Figure 6-1, which is the only example of Common Dialog shown in this section. This Common Dialog should be familiar; you have used it often. You have also

used most of the other Common Dialogs which are discussed next, but not shown.



Figure 6-1: The GetOpenFileName Common Dialog

In the class CFileDialog, the creation function GetSaveFileName() creates a modal “File|Save As” dialog box. This dialog would typically be used when the user selects the “File|Save As” menu item. This dialog box gets the path name from the user of the file where the user wants to save data. This dialog, of course, does not actually write anything to the file; additional code must be supplied to do this task.

These Common Dialogs have been incorporated into the file saving (archiving) routines which are provided when you use the MFC CDocument class. You will find that you will be using the CDocument and CView classes when you need to store and retrieve files, which not only include these Common Dialogs, but also provide all of the other services needed to complete the saving and retrieving of files. (Using the CDocument class for storing and retrieving data is covered in Chapter Ten.)

Class CPrintDialog

The CPrintDialog class encapsulates Common Dialogs for printing. The common print dialog boxes provide an easy way to implement Print and Print Setup dialog boxes in a manner consistent with Windows standards. This Common Dialog is used with the Cview class, which provides the remainder of the built-in services required for printing. (Printing is covered in Chapter Ten.)

Class CPageSetupDialog

The CPageSetupDialog class encapsulates services provided by the Common Dialog for setting up a page with additional support for setting and modifying print margins. This Common Dialog was introduced with MFC version 4.0.

Class CFindReplaceDialog

The CFindReplaceDialog class allows you to implement standard string “Find|Replace” dialog boxes in your application. Unlike the other common dialog boxes, CFindReplaceDialog objects are modeless, allowing users to interact with other windows in their application while they are on screen. There are two kinds of CFindReplaceDialog objects: Find dialog boxes, created with the FindText() create function, and Find|Replace dialog boxes, created with the ReplaceText() creation function. Although the dialog boxes allow the user to input search and search/replace strings, they do not perform any of the searching or replacing functions. You must add these to the application.

Class CFontDialog

The CFontDialog class allows you to incorporate a font-selection dialog box into your application. A CFontDialog object is a dialog box with a list of fonts that are currently installed in the system. The user can select a particular font from the list, and this selection is then reported back to the application.

Class CColorDialog

The CColorDialog class allows the incorporation of a color-selection dialog box into your application. A CColorDialog object is a dialog box with a list of colors that are defined for the display system. The user can select or create a particular color from the list, which is then reported back to the application when the dialog window closes.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Using the ChooseColor Common Dialog

The need to use many of the Common Dialogs has been lessened by the built-in capabilities of the CDocument and CView classes. However, one Common Dialog which you will need to implement on your own is the ChooseColor dialog box of the CColorDialog class. This section covers the use of this Common Dialog, which is named the ChooseColor dialog after its creation function. The program, named "ColorDlg," is based on the "BackingStore" program from Chapter Four, which is modified such that when the user selects "Brush|Color" from the menu, the ChooseColor Common Dialog appears as shown in Figure 6-2.



Figure 6-2: The ChooseColor Common Dialog Used in Program ColorDlg

Once the common dialog appears, the user then selects a color, chooses the OK button, and the ChooseColor Common Dialog disappears. The chosen color can then be transferred to the application and will be the brush color for the next shape that is drawn. In this section, the changes that must be made to the "BackingStore" program to achieve this result are discussed.

The program and its files have been renamed from "BackingStore" to "ColorDlg." The .rc file is changed such that the entire submenu that appears when the user chooses "Brush|Color" is replaced with the single end menu item:

```
MENUITEM "&Color...",          ID_CHOOSECOLORDLG
```

The resource.h file must be similarly changed, removing the IDs that were eliminated when the submenu was replaced with the single item, and adding ID_CHOOSECOLORDLG. The mainfrm.h file is modified such that the message handlers for brush color—OnBrushRed(), OnBrushTeal(), OnBrushOlive(), and OnBrushBlue()—are replaced by one function named OnBrushSelectColor(). The mainfrm.cpp file must include the <afxdlgs.h> header file. It is also modified such that all of the preceding message handlers for brush color are removed from the message map and replaced by:

```
ON_COMMAND( ID_CHOOSECOLORDLG, OnBrushSelectColor )
```

All the handler functions for the deleted submenu items must be removed from the code. They are replaced with the single handler function, the entire code of which is:

```
void C_MainFrame::OnBrushSelectColor()
{
    COLORREF clrInit = RGB(255, 0, 0);
    CColorDialog BrushColor(clrInit, 0, this);
    if (BrushColor.DoModal() == IDOK)
        m_LogBrush.lbColor = BrushColor.GetColor();
}
```

clrInit is the color that is “selected” when the ChooseColor common dialog box opens up. In this function, we defined the clrInit color value to be red (RGB(255, 0, 0)). If the clrInit color is not defined, black is used as the default. We next instantiated an object (named BrushColor) of class CColorDialog. In this call to the constructor, the first argument is specified to be the default color of red and the second argument is specified to be 0, because we did not want to set flags to customize the dialog box. The third argument is specified to be the pointer to the dialog box’s parent. Then, the member function DoModal() is called, which causes the Common Dialog to appear on the screen as a modal dialog box. When the user closes the dialog box, we check to see if the return value was IDOK. If it was, we get the color that was returned by invoking the member function GetColor() and assign the color returned by this function to the variable m_LogBrush.lbColor.

As the previous discussion demonstrates, a lot of functionality can be contained in a few lines of code!

Note: You will find that you almost always use the Common Dialogs when they suit your needs, rather than building your own dialog boxes.

Designing Dialog Boxes

The standard way that applications specify the dialog box and its child controls is as a DIALOG resource in the resource file. As discussed in Chapter One, the Windows operating system defines a number of *resource types*, including ICON, CURSOR, MENU, ACCELERATOR, BITMAP, and DIALOG. The DIALOG resource is discussed in this chapter and in Chapter Seven. A

DIALOG resource is defined as a *resource script*. A resource script is in text form. It is the source code for a resource. This resource script is compiled into a binary file. Then, the linker appends the binary resource to the executable file. As discussed in Chapter One, the resource is not loaded into memory when the application starts execution. It is loaded later, when it is needed.

An application supplies the name of the DIALOG resource to MFC when it creates your dialog class object. The operating system will then load the dialog resource (i.e., read the binary resource block into memory) and use the DIALOG resource to create the dialog box with its child controls.

The DIALOG resource script is usually constructed by compiler tools, such as the “Dialog Editor,” although it can be constructed by hand. However, once you have constructed one DIALOG resource script by hand, you will appreciate the ease and efficiency of doing this with the tools, so you will thereafter probably use the compiler tools. The compiler tools also offer the service of *wrapping*, or encapsulating, the dialog resource with a class derived from CDialog, so it is recommended to always use the tools for dialog boxes.

We will look at generating the DIALOG resource script with the compiler tool. To generate a dialog box with the compiler tools, you make entries (choices) for each window (dialog box or control) on a *property page*. There are a lot of choices, and since the introduction of MFC 4.0 there is a new category called “extended styles.” Figure 6-3 shows the property page for a dialog box, which has four tabs; the choices for each tab are shown. Notice that a lot of choices for each dialog or control need to be made. These choices are based on styles, either window styles, dialog styles (for dialogs), or control styles (for controls).



Figure 6-3: Dialog Box Property Pages

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes
(Publisher: Wordware Publishing, Inc.)
Author(s): Shirley Wodtke
ISBN: 1556225121
Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

A dialog box has a special style parameter associated with it. This style parameter is a combination of applicable window styles and dialog box styles. The dialog box styles are given in Table 6-1. This table demonstrates the fact that MFC 4.0 introduced a number of new dialog styles, which are not available on Win3.1x operating systems. The dialog box default style is DS_MODALFRAME combined with the WS_POPUP, WS_CAPTION, and WS_SYSMENU styles. (Some compiler tools also use the WS_VISIBLE style.) The WS_POPUP style should generally be used for dialog boxes. The dialog box styles, the applicable window styles, the text for the dialog box caption, and the selected font will be coded into the DIALOG resource script. In this chapter, we will build a dialog box with the compiler tools and then examine the DIALOG resource script produced by the compiler tools.

Table 6-1: Dialog Box Styles

Style	Meaning
DS_ABSALIGN	Specifies a dialog box where the dialog coordinates for its upper left corner are relative to the screen origin instead of to the upper left corner of the parent window. Generally, use this style when you want the dialog box to start in a specific part of the display no matter where the parent window may be on the screen.
DS_LOCALEDIT	Specifies that edit controls in the dialog box will use memory in the application's data segment. By default, all edit controls in dialog boxes use memory outside the application's data segment. This feature may be suppressed by adding the DS_LOCALEDIT flag to the style for the dialog box. (Win3.1x only).

DS_MODALFRAME	Creates a dialog box with a modal dialog box frame that is usually combined with a title bar and a system menu by specifying the WS_CAPTION and WS_SYSMENU styles.
DS_NOIDLEMSG	Suppresses WM_ENTERIDLE messages that would otherwise be sent to the owner of the dialog box while the dialog box is displayed.
DS_SYSMODAL	Creates a system-modal dialog box.

Introduced with MFC 4.0

DS_CENTER	Centers dialog in the screen's working area when it opens up.
DS_CENTERMOUSE	Centers the mouse in the dialog box when it opens.
DS_CONTROL	Creates a dialog box that works as a child window of another dialog box.
DS_FIXEDSYS	Uses SYSTEM_FIXED_FONT instead of SYSTEM_FONT.
DS_NOFAILCREATE	Allows child controls to not notify their dialog parent on failure to create. The dialog box will create itself even if its child controls cannot be created.
DS_SETFOREGROUND	Brings the dialog to the foreground, placing it above windows with the WS_EX_TOPMOST style.
DS_3DLOOK	Draws three-dimensional borders around controls.

Any of the window styles discussed in Chapters One and Five can be applied to the dialog window. In addition, MFC 4.0 also introduced a number of extended window styles, some of which can be applied both to the dialog boxes and their controls. (These extended window styles are not available to Win3.1x users.) Table 6-2 explains the extended window styles that apply to dialog boxes and controls. Additional extended window styles that apply only to dialog boxes, but not to the controls, are listed in Table 6-3. One additional extended window style applies to all the controls, but not to a dialog box; this window style is noted later when the controls are discussed. Finally, the extended window style WS_EX_LEFTSCROLLBAR, which places a vertical scroll bar to the left of the client area, applies to dialog boxes, combo boxes, list boxes, and edit controls, but does not apply to the other controls discussed in this chapter.

Table 6-2: Extended Window Styles That Apply to Dialog Boxes and Controls

Style	Meaning

WS_EX_ACCEPTFILES	Specifies that a window created with this style accepts <i>drag-and-drop</i> files.

WS_EX_CLIENTEDGE	Specifies that a window has a 3D look—that is, a border with a “sunken” edge.
WS_EX_NOPARENTNOTIFY	Specifies that a child window created with this style will not send the WM_PARENTNOTIFY message to its parent window when the child window is created or destroyed.
WS_EX_RIGHT	Gives a window generic right-aligned properties. This depends on the window class.
WS_EX_RIGHTSCROLLBAR	Places a vertical scroll bar (if present) to the right of the client area. (This is the default.)
WS_EX_STATICEDGE	Creates a window with a three-dimensional border style intended to be used for items that do not accept user input.
WS_EX_TRANSPARENT	Specifies that a window created with this style is to be transparent. That is, any windows that are beneath the window are not obscured by the window. A window created with this style receives WM_PAINT messages only after all sibling windows beneath it have been updated.
WS_EX_CONTEXTHELP	Includes a question mark in the title bar of the window. When the user clicks the question mark, the cursor changes to a question mark with a pointer. If the user then clicks a child window, the child receives the WM_HELP message.
WS_EX_CONTROLPARENT	Allows the user to navigate among the child windows of the window by using the Tab key.
WS_EX_TOOLWINDOW	Creates a <i>tool window</i> , which is a window intended to be used as a floating toolbar. A tool window has a title bar that is shorter than a normal title bar, and the window title is drawn using a smaller font. A tool window does not appear in the task bar or in the window that appears when the user presses Alt+Tab.

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes
(Publisher: Wordware Publishing, Inc.)
Author(s): Shirley Wodtke
ISBN: 1556225121
Publication Date: 01/01/97

Bookmark It

Search this book:

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

Table 6-3: Extended Window Styles For Dialog Boxes

Style	Meaning
WS_EX_CONTEXTHELP	Includes a question mark in the title bar of the window. When the user clicks the question mark, the cursor changes to a question mark with a pointer. If the user then clicks a child window, the child receives the WM_HELP message.
WS_EX_CONTROLPARENT	Allows the user to navigate among the child windows of the window by using the Tab key.
WS_EX_TOOLWINDOW	Creates a <i>tool window</i> , which is a window intended to be used as a floating toolbar. A tool window has a title bar that is shorter than a normal title bar, and the window title is drawn using a smaller font. A tool window does not appear in the task bar or in the window that appears when the user presses Alt+Tab.

The types, styles, and placement of controls can also be defined using the compiler tools. The DIALOG resource script contains *control resource scripts* for each control. The choices made for a control will be reflected in the code in the control resource script for that control. This chapter focuses on *button controls* and *static controls* and builds an example illustrating these controls. (In Chapter Seven, list box, combo box, and edit controls are discussed.)

Overview of Common Controls

Common controls are Windows-defined interface elements. Certain common

controls, termed *Win3.1 common controls*, have always been an integral part of the Windows operating system. The Win3.1 common control, known as the custom control or user-defined control, is used as the base for CBitmapButton which is discussed in Chapter Fourteen. (The custom control will not be discussed in this chapter.)

Win 3.1x common controls are usually placed in dialog boxes and have an associated definition in the dialog's resource script. In this chapter and in Chapter Seven, when a control is referred to, it means a Win3.1 common control as defined here. The MFC classes associated with Win3.1 common controls are: CButton, CComboBox, CEdit, CListBox, CStatic, and CScrollBar. The class CScrollBar produces stand-alone scroll bars. Several of these control classes have optional built-in scroll bars. Also windows can have scroll bars built in. Thus, the need to use a stand-alone scroll bar control is limited, so they are not discussed in this book.

Additional common controls have been introduced with MFC 4.0. These are: animate control, tab control, tree control, list control, hot key control, slider control, progress indicator control, and the spin control. These new controls are collectively known as *Win95 common controls*. These newer Win95 common controls are the subject of Chapter Fourteen. The procedures to implement them are different from the Win3.1 common controls that are discussed in this chapter and in Chapter Seven.

Common controls are simple interface elements that display information and in general accept user input and user actions. Usually, common controls are placed in dialog boxes, but controls may also be placed directly in other windows. However, when placing controls directly in windows you do not have the benefit of using the Dialog Editor to define and position the controls.

The Win3.1 common controls are grouped into *control classes*. The use of the term "class" is not related to a C++ class, but rather simply means a "grouping." The control classes are: button control, static control, list box control, combo box control, edit control, and scroll bar control (not discussed).

Note: The term control class is used hereafter, because this term has always been used for the grouping of these common controls.

Within each control class, various Windows-defined styles can be specified that affect the appearance and behavior of the control. The control class styles are keywords that appear in the DIALOG resource script.

Window Styles for Win3.1 Common Controls

Certain window styles can be applied to Win3.1 common controls. Window styles that can be used with all the Win3.1 common controls classes are given in Table 6-4. In addition to these window styles one more choice in the "general" category is available, termed "Help ID" on the control's property page. MFC 4.0 introduced this new option for all controls (Win3.1x users do not have this option). When the "Help ID" option is selected, an ID, which can be linked to your help file, is coded into the resource script for that control.

Table 6-4: Window Styles for Win3.1 Common Controls

Style	Meaning
WS_CHILD	Creates a child window. Used when the control is placed directly on a window other than a dialog window.
WS_DISABLED	Creates a control that is initially disabled. To change to the <i>enabled state</i> after it is created, use the function <code>CWnd::EnableWindow()</code> .
WS_VISIBLE	Creates a control that is initially visible. To make the window visible after it is created, use the function <code>CWnd::ShowWindow()</code> .
WS_TABSTOP	Moves the focus to the next control that has this style when the user presses the Tab key. This window style is valid only for controls.
WS_GROUP	Specifies the first control of a group of controls in which the user can move from one control to the next by using the Arrow keys. All controls defined with the WS_GROUP style after the first control belong to the same group. The next control with the WS_GROUP style ends the group and starts the next group (that is, one group ends where the next begins). This window style is valid only for controls.
WS_EX_DLGMODALFRAME	Designates a window with a double border (like a modal dialog box frame).

Button Controls

A button control can be a *push button*, a *radio button*, a *check box*, an *owner-drawn button*, or a *group box*. Buttons can be used alone or in groups, and can either be labeled or appear without text. When the user clicks on a button control, it sends a `BN_CLICKED` message to its dialog box or parent window. Button controls typically change appearance when the user clicks on them.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes
(Publisher: Wordware Publishing, Inc.)
Author(s): Shirley Wodtke
ISBN: 1556225121
Publication Date: 01/01/97



Search this book:

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

Button Styles

All of the button styles given in the following paragraphs for each button type can be found in the control’s property page, which is used in the tool constructing the dialog box—the Dialog Editor. In addition, more button styles have been introduced with MFC 4.0. These new button styles are listed in Table 6-5 and are available only to Win95 and NT users. The style keyword is entered into the control’s script line within the DIALOG resource script, except when it is the default style for that control.

Table 6-5: Win95 Button Styles

Style	Meaning
BS_BITMAP	Displays a bitmap in the control’s text field.
BS_ICON	Displays an icon in the control’s text field.
BS_FLAT	Creates a control with a “flat” (two-dimensional) appearance.
BS_MULTILINE	Displays text in multiple lines, automatically wrapped to the width of the control.
BS_PUSHLIKE	Creates a control with a pushbutton-like appearance.
BS_NOTIFY	Sends the parent BN_DBLCLK, BN_KILLFOCUS, and BN_SETFOCUS notification messages from the control.
BS_RIGHTBUTTON	Positions the button’s circle, or check box’s square, on the right side.
BS_TEXT	Causes the button to display text.
BS_TOP	Places text at top.
BS_BOTTOM	Places text at bottom.
BS_LEFT	Left-justifies text.

itself and unchecks (removes the selection from) any other buttons in the same group of controls. Radio buttons are frequently used for a group of related but mutually exclusive choices.

Check Box Styles

In addition to the window styles and the Win95 button styles, the styles BS_CHECKBOX, BS_AUTOCHECKBOX, BS_3STATE, and BS_AUTO3STATE, described below, apply. Choices on the check box property page, shown in Figure 6-6, are based on the applicable window, extended window, button, and check box styles. (Win3.1x users will have fewer styles available for their property pages.)

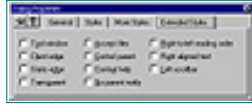


Figure 6-6: Check Box Property Pages

BS_CHECKBOX Creates a small square that has text displayed to its right, unless this style is combined with the BS_LEFTTEXT style.

BS_AUTOCHECKBOX Creates a check box, except a checkmark appears in the check box when the user selects the box and the checkmark disappears (is cleared) from other check boxes in the same group.

BS_3STATE Creates a check box that is the same as a check box, except that the box can be grayed as well as checked. A grayed check box appears as a gray square. The grayed state is used to show that the state of the check box is not determined.

BS_AUTO3STATE Creates a check box that is the same as the three-state check box, except that the box changes its state when the user selects it. The state cycles through checked, grayed, and normal.

Group Box Styles

In addition to the window styles and the Win95 button styles, the style BS_GROUPBOX, described below, applies. The choices that you make in the group box property page, shown in Figure 6-7, represent the pertinent window, extended window, button, and group box styles. (Win3.1x users will have fewer styles available in their property pages.)

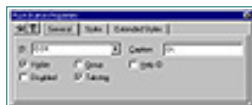


Figure 6-7: Group Box Property Pages

BS_GROUPBOX Creates a rectangle in which other controls can be grouped. Any text associated with this style is displayed in the rectangle's upper left corner. This is a visual element only, like a static control. The user cannot interact with this control and it generates no notification messages. Normally a group box is placed around a group of controls to let the user know what controls are in the group.

Owner Draw Style

BS_OWNERDRAW creates an owner-drawn button. The owner window receives a WM_MEASUREITEM message when the button is created, and it receives a WM_DRAWITEM message when a visual aspect of the button has changed. This style cannot be combined with any other button styles. This style is used as the base for the CBitmapButton class and is discussed in Chapter Fourteen.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97



Search this book:

Static Controls

Static controls are simple text fields, boxes, and rectangles that can be used to label, box, or separate other controls. The static control is most frequently used to display a line of text, left-, center-, or right-aligned according to the style. These choices produce a resource script that is either LTEXT, CTEXT, or RTEXT. Use the window style WS_BORDER to produce a static control with a border.

In addition to the window styles and the Win95 styles, the styles given in Table 6-6 apply. Figure 6-8 shows the choices that are available on the property page for static controls. (Win3.1x users will have fewer styles in their property pages.)



Figure 6-8: Static Control Property Pages

The styles for the static control are given in Table 6-6. Not all of these styles can be found in the static control’s property page due to the limited use of some of these styles. If you wish to use one not represented, then enter it manually directly into the DIALOG resource script.

Table 6-6: Static Control Styles

<i>Style</i>	<i>Meaning</i>
SS_SIMPLE	Single text line, left-aligned
SS_LEFT	Text left-aligned; words that would extend beyond the end of a line are automatically wrapped to the beginning of the next line.
SS_LEFTNOWORDWRAP	Text left-aligned; text that extends past the end of a line is clipped.
SS_CENTER	Centered text; words are wrapped to the beginning of the next line.

SS_RIGHT	Text right-aligned; words are wrapped to the beginning of the next line.
SS_ICON	An icon is displayed in the dialog box; the text is the name of the icon defined in the resource file. The icon automatically sizes itself.
SS_BLACKFRAME	Frame drawn in window frame color (default of black).
SS_BLACKRECT	Rectangle filled with window frame color.
SS_WHITEFRAME	Frame drawn in window background color (default of white).
SS_WHITERECT	Rectangle filled with window background color.
SS_GRAYFRAME	Frame drawn in desktop color.
SS_GRAYRECT	Rectangle filled with desktop color.
SS_NOPREFIX	Prevents interpretation of any & characters in the control's text as accelerator prefix characters.
SS_OWNERDRAW	Owner draws the static control.
Introduced with MFC 4.0	
SS_BITMAP	Display a bitmap. (Used like SS_ICON.)
SS_RIGHTJUST	Lower right of control with SS_ICON or SS_BITMAP style remains fixed when the control is resized.
SS_REALSIZEIMAGE	Prevent a control with SS_ICON or SS_BITMAP style from being resized as it is loaded or drawn.
SS_SUNKEN	Has a half-sunken border.
SS_CENTERIMAGE	If the bitmap or icon is smaller than the control, it is centered vertically.
SS_ETCHEDFRAME	Control's frame is in the EDGE_ETCHED style.
SS_ETCHEDHORZ	Control's top and bottom edges are drawn with the EDGE_ETCHED style.
SS_ETCHEDVERT	Control's left and right edges are drawn with the EDGE_ETCHED style.
SS_ENHMETAFILE	Displays an enhanced metafile.
SS_NOTIFY	Sends parent STN_CLICKED, STN_DBLCLK, STN_DISABLE, and STN_ENABLE notification messages.

Placing Controls on the Mainframe Window

Controls can be created as child windows of another window. This section explores creating controls as child windows of the mainframe window. This will be demonstrated in the upcoming example in which a push button control and a static control are placed directly onto the mainframe window. The role given to the push button control is that, when it is clicked, it will open the dialog box which contains all kinds of button controls, and the static control will hold an icon. Before exploring this type of example, let's look at how controls can be placed as child windows directly onto the mainframe window and at the messages that the control can send to its parent, whether that parent is a mainframe window or a dialog

window.

The process used is identical to that used in Chapter Five, which discussed child windows. However, in this case, we do not have to write our own child window class, instead, we conveniently have access to classes that are already written, which are the MFC CButton and CStatic classes. We declare objects of class CButton and CStatic in the mainframe's constructor. The following are declared as member objects of the mainframe class in the mainfrm.h file:

```
CButton  m_ButtonOpen;  
CStatic  m_StaticIcon;
```

And in the definition of the mainframe's constructor (in its implementation file, mainfrm.cpp), we will find the following code for the creation of each of these objects:

```
m_ButtonOpen.Create("Open Dialog", WS_CHILD | WS_VISIBLE |  
                        BS_PUSHBUTTON, ButtonRect, this,  
                        ID_BUTTON_OPENDIALOG);  
  
m_StaticIcon.Create("Smiley", WS_CHILD | WS_VISIBLE | SS_ICON,  
                    IconRect, this);
```

For the CButton object, m_ButtonOpen, we specified an ID argument, ID_BUTTON_OPENDIALOG. For the CButton object, an ID is necessary, because the button will send a message carrying that ID. The CStatic object, m_StaticIcon, has no use for an ID, so its final argument is omitted and will have the default value of NULL. The embedded objects, m_StaticIcon and m_ButtonOpen, are constructed and destroyed along with the mainframe window object.

The classes CButton and CStatic furnish us with member functions to use, such as the preceding Create() function, which creates each control. In the upcoming example, you will see some of these functions applied. CButton member functions are given in Table 6-7. CStatic member functions are given in Table 6-8. As demonstrated by these tables, we have access to convenient functions such as checking a button or setting an icon in the static control, etc. Table 6-9 gives the CButton Create() function. Table 6-10 gives the CStatic Create() function.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes
(Publisher: Wordware Publishing, Inc.)
Author(s): Shirley Wodtke
ISBN: 1556225121
Publication Date: 01/01/97

Bookmark It

Search this book:

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

Table 6-7: CButton Member Functions

Function	Meaning
Create()	Creates the button control and attaches it to the CButton object.
SetCheck()	Sets the check state of a button control.
GetCheck()	Retrieves the check state of a button control.
SetState()	Sets the highlighting state of a button control.
GetState()	Retrieves the check state, highlight state, and focus state of a button control.
SetButtonStyle()	Changes the style of a button.
GetButtonStyle()	Retrieves information about the button control style.

Table 6-8: CStatic Member Functions

Function	Meaning
Create()	Creates the static control and attaches it to the CStatic object.
SetIcon(hIcon)	Associates an icon with the CStatic object. hIcon identifies the icon. Used with a CStatic object created with the SS_ICON style.
GetIcon()	Returns the handle of the icon associated with the CStatic object. This function should be called only for CStatic objects that represent icons created with the SS_ICON style.

Table 6-9: The CButton Create() Function

BOOL Create(LPCSTR lpszCaption, DWORD dwStyle,
RECT& rect, CWnd* pParentWnd, UINT, nID)

<i>Argument</i>	<i>Meaning</i>
lpszCaption	Specifies the button control's text.
dwStyle	Specifies the button control's style.
rect	Specifies the button control's size or position. It can be either a CRect object or a RECT structure.
pParentWnd	Specifies the button control's parent window, usually a CDialog object. It must not be NULL.
nID	Specifies the button control's ID.

Table 6-10: CStatic Create() Function

BOOL Create(LPCSTR lpszText, DWORD dwStyle,
RECT& rect, CWnd* pParentWnd, UINT nID);

<i>Argument</i>	<i>Meaning</i>
lpszText	Specifies the text to place in the control. If NULL, no text will be visible.
dwStyle	Specifies the static control's window style.
rect	Specifies the position and size of the static control. It can be either a RECT structure or a CRect object.
pParentWnd	Specifies the CStatic parent window, usually a CDialog object. It must not be NULL.
nID	Specifies the static control's control ID.

Messages To and From Button Controls

Messages From the Button Control

A control sends messages, known as *control notification messages*, to its parent when the user interacts with it in certain ways. A button control sends a WM_COMMAND message with a BN_CLICKED notification value to the parent when it is clicked. In this case, the message map entry for the control notification message will have the form:

ON_BN_CLICKED(<control ID>, <handler function>)

The general form for the message map entry is:

ON_XXXX(<control ID>, <handler function>)

where XXXX is the notification message.

MFC 4.0 has introduced the button style BS_NOTIFY. If the control has this style, it will also send the parent BN_DOUBLECLICKED, BN_KILLFOCUS, and BN_SETFOCUS messages.

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

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

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

Keyword

☒ Brief
 ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Messages To the Button Control

While a control is displayed, messages can be sent to it from the application. In this way, you can find out current information about the control while your program is running. For example, the member function `GetCheck()` can be used to find out if the button is checked; this function will then send a message to the control to access the information it needs. Also, the application can change the control's attributes at run time. For example, the member function `SetButtonStyle()` can be used to change the button's style.

We also have `CWnd` functions, one of which is used in the upcoming example to directly send a message to the control. The function `CWnd::SendMessage()` sends a message to a control and has the form:

```
SendMessage(int nID, UINT message,
            WPARAM wParam, LPARAM lParam);
```

where: `nID` is the ID of the dialog control that will receive the message; `message` is the identifier of the message to be sent; and `wParam` and `lParam` specify additional message-dependent information.

The upcoming example has the following line of code:

```
SendMessage(IDC_RED, BM_SETCHECK, 1, 0);
```

This line of code sends to the control that has the ID of `IDC_RED`, the message known as `BM_SETCHECK`, with a `wParam` of 1 and a `lParam` of 0. When the control receives a message, it reacts in specified ways to the `wParam` and `lParam` values. For the `BM_SETCHECK` message, if the `wParam` is 1, the button control will check itself. If the `wParam` is 0, the button control will uncheck itself. The `BM_SETCHECK` message does not use `lParam`, so it is set to 0.

The `SendDlgItemMessage()` function can be used to send any of the predefined messages that are destined for a control. There are more of these predefined control messages; they are identified as `BM_XXXX` for button messages. If you wish to get a complete list of these messages, you will need to look them up in the reference material.

Other `CWnd` member functions that send data to the control are `CWnd::SetDlgItemInt()` and `CWnd::SetDlgItemText()`. `CWnd` functions can be used to get data from the control, which are: `CWnd::GetDlgItemText()` and `CWnd::GetDlgItemInt()`.

Example Program

This section presents an application that has one menu item and a push button, both labeled `OpenDialog`. When the menu item or the push button is chosen, the dialog box opens. The dialog box has all kinds of button controls and, as always, has its own message map. When the dialog box is open, we can dynamically create a static control which has the smiley icon by clicking on the push button labeled “Create Icon.”

Dynamically creating a control on a dialog box is very similar to placing a control on the main window. We will add an embedded data member of the button control class type to the constructor of the dialog class and call the `Create()` member function of class `CButton` when we wish to create the icon. The embedded `CButton` object is constructed and destroyed along with the dialog object; the operating system will destroy the embedded data member when it destroys the dialog window.

In this example the controls in the radio button group box, when clicked, send a message to their parent, the dialog box. The parent, in response, sends messages to the check box controls to do specific things (to check or uncheck themselves). We will look at three ways for the parent to send messages to operate its controls:

- Sending a message directly to the control to be operated using the function `SendDlgItemMessage()`.
- Declaring a member object of the `CButton` class and then using this member to invoke its inherited member function `SetCheck()`. (In this approach, the connection between the control and the C++ member object is made using `ClassWizard` which provides a `DoDataExchange` member function to do the job.)
- Using a defined function to access the control. (In this last method, we write an inline function named `GetGreenCheck()`, which is placed in the dialog class’s header file, to cast the return type of the `CWnd::GetDlgItem()` function (which provides a pointer to the control) to the `CButton` control type. This function `GetGreenCheck()` is then used to access the control’s member functions.

The executing `Buttons` program is shown in Figure 6-9. When the user clicks the radio button labeled “Check Red,” a check mark appears in the check box labeled “Red” and all other check boxes are unchecked. The remaining check boxes function in a similar manner. Finally, clicking the push button labeled

“Create Icon” will cause the smiley icon to appear on the dialog box.

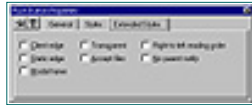


Figure 6-9: Executing Buttons Program with its Dialog Box Open

Previous	Table of Contents	Next

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

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

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

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

Programming the Buttons Example

To build this program, we will be working with the compiler tools, yet adding features outside their scope. Building such an application is tricky; therefore, detailed instructions have been included. The instructions to generate this application are given in two parts.

The first part covers all the tasks necessary to generate the code for the main window: creating a new project as a simple application (without using AppWizard); typing in all of its initial .h and .cpp files; creating the menu; adding the message map entries and the stubbed-out handler functions using ClassWizard; customizing the program by making some necessary modifications to the mainfrm.cpp, the resource.h and the Script1.rc file; and compiling the program.

The second part covers all the tasks necessary to generate the code for the dialog box: generating the resource script for the dialog box using the Dialog Editor tool; using ClassWizard to create a dialog class and to connect its controls to message handlers in the dialog class; making changes to your files to incorporate the new files that have been generated and to correct for assumptions made by ClassWizard; and completing the message handler code for both the dialog class and the main window's class. An overview of each part is given first, then each part is presented in detail.

Generating the Main Window's Code

The tasks to generate the main window program are outlined and discussed in the following nine steps:

1. Create a new project named Buttons as a simple application (without using AppWizard).
2. Type in all the .h and .cpp files and add all the .cpp files to your Buttons project. (Only the code that you should type in at this point is given.) The files are:

```

//
//  stdafx.h    include file for standard system include files
//
#include <afxwin.h>

//
//  stdafx.cpp    source file for standard includes
//
  
```

```

#include "stdafx.h"

//
// MyApp.h    interface of the C MyApp class
//
class C MyApp : public CWinApp
{
public:
    BOOL InitInstance();
};

//
// MyApp.cpp    implementation of the class C MyApp
//
#include "stdafx.h"
#include "MyApp.h"
#include "mainfrm.h"

BOOL C MyApp::InitInstance()
{
    m_pMainWnd = new C_MainFrame;
    m_pMainWnd->ShowWindow(m_nCmdShow);
    SetDialogBkColor();
    return TRUE;
}

C MyApp theApp;    // the object that runs the program
//
// mainfrm.h    interface of the C_MainFrame class
//
class C_MainFrame : public CFrameWnd
{
public:
    C_MainFrame();
    CButton m_ButtonOpen;
    CStatic m_StaticIcon;
private:
   //{{AFX_MSG(C_MainFrame)
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

//
// mainfrm.cpp    implementation of the C_MainFrame class
//
#include "stdafx.h"
#include "mainfrm.h"
#include "resource.h"

BEGIN_MESSAGE_MAP(C_MainFrame, CFrameWnd)
   //{{AFX_MSG_MAP(C_MainFrame)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
C_MainFrame::C_MainFrame()

```

```

{
    Create(NULL, "Button Controls", WS_OVERLAPPEDWINDOW,
           rectDefault, NULL, MAKEINTRESOURCE(IDR_MENU1));

    CRect ButtonRect(10, 10, 110, 50);
    m_ButtonOpen.Create("OpenDialog", WS_CHILD | WS_VISIBLE |
                        BS_PUSHBUTTON, ButtonRect, this,
                        ID_BUTTON_OPENDIALOG);

    CRect IconRect(10, 70, 0, 0);
    m_StaticIcon.Create("SmileyIcon", WS_CHILD | WS_VISIBLE |
                       SS_ICON, IconRect, this);
}

```

3. Create the menu resource. It will have only one item “Open Dialog” on the menu bar. The item will be an end menu item, not a popup. Give the end menu item an ID of ID_OPENDIALOG. The project’s script1.rc file and its resource.h file are automatically generated for you. Save the file as Script1.rc and make sure that it is added to your Buttons project. In subsequent steps, we add code to these resource files; however, it is important that none of the comment code is removed from the resource files, since we will re-enter the compiler tools and these files to continue generating the application.

4. Open ClassWizard and add the message map entry and handler function for the menu item.

5. Open the resource.h file and add the following line of code to the beginning of the #defines. By adding this line to the beginning of the #defines you do not have to worry about what number the compiler tool will use for its next entry. We will re-enter the compiler tool and generate a dialog box, which will add more IDs to this file. The line of code which you will now enter is the ID of the push button that is attached directly to the main window.

```
#define ID_BUTTON_OPENDIALOG      100
```

6. Open mainfrm.cpp and add the following message map entry, which maps the button click from the push button control to the message handler function OnOpenDialog which was added in step 4:

```
ON_BN_CLICKED(ID_BUTTON_OPENDIALOG, OnOpenDialog)
```

7. Copy the icon file into the working directory. You may want to close your project to do this. (If you create your own icon within your project, you must be aware that the CStatic::Create() function is not overloaded to take a UINT, so you will need to fix up your .rc and resource.h files accordingly.) The smiley icon from previous exercises can be copied.

8. Since, in this example, the smiley icon was copied into the .rc file, the following lines of code should be entered into the .rc file. Two lines of code are entered for the same icon in order to use the same icon in two different places: as the icon for the application and as the icon for the static control. (If your compiler has no direct way of opening the .rc file for editing, go to the MS-DOS prompt and use the edit script1.rc command from the DOS command line.)

Note: Within the .rc file, identifiers are not case sensitive.

The code to enter is as follows:

```
AFX_IDI_STD_FRAME      ICON      DISCARDABLE      "smiley.ico"
```

SmileyIcon

ICON

DISCARDABLE

"smiley.ico"

9. Compile and run the program and, at this point, it should look like the running program in Figure 6-10. The main window has its menu attached. The static icon and the push button that are directly attached to the main window are visible. However, neither the push button nor the menu item are functional at this point, since the handler function has not been completed. The main window's handler function to open the dialog box cannot be completed until the dialog box has been generated. Generating the code for the dialog box is discussed next.

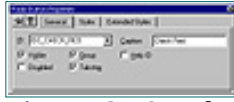


Figure 6-10: The Buttons Program After Completing Main Window Code

[Previous](#)

[Table of Contents](#)

[Next](#)

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Generating the Dialog Box Code

Generating the dialog box code for the Buttons example consists of the following steps each of which will later be explained in detail.

1. Generate the source code for the DIALOG resource script.
2. Generate the code for a dialog class (derived from CDialog) to which the DIALOG resource script will be attached.
3. Add message map entries and stubbed-out handler functions to the dialog class.
4. Establish a member object of the dialog class for the blue check box which is the only check box for which we use member functions to send messages to.
5. Customize our CDialog class by adding code and fix up our files.
6. Complete the message handler functions in the dialog class.
7. Complete the message handler function in the main window's class.

Creating the DIALOG Resource Script

In this step we build the dialog box by selecting “Insert|Resource” from the menu. Choose “Dialog” from the Insert Resource dialog box as shown in Figure 6-11 and then click “OK.”

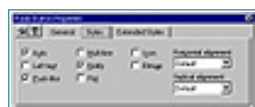


Figure 6-11: Inserting a Dialog Resource

The Dialog Editor will be called and the display shown in Figure 6-12 will be produced. A default dialog, with two buttons, is already in the editor window. Also, the Dialog Editor brings up a control palette, containing all the controls

which is labeled “Check Red.” This will identify this radio button as the beginning of the group. Do not mark the group property for any of the remaining radio buttons.

2. Set the tab order as shown in Figure 6-14, so that the radio button group is in order and is followed by the check boxes. To mark the end of the radio button group, assign the group and tabstop properties to the next control (in tab order) after the last of the radio buttons, which will be the check box labeled “Red.” You can activate the radio buttons in the group with the mouse or with keyboard inputs.



Figure 6-14: Setting the Tab Order

With keyboard input, the radio button group will function as follows:

- (a) Use the tabstop to get to the first radio button in the group.
- (b) Once in the group, use the Up or Down Arrow keys to traverse upward or downward, activating the radio buttons within the group.

Creating a Dialog Class

With our dialog box remaining open, open ClassWizard by selecting “View|ClassWizard” from the menu. The Add Class dialog shown in Figure 6-15 appears.



Figure 6-15: Adding a Class

The connection between the new dialog class that will be created and the DIALOG resource is made in this Create New Class step, shown in Figure 6-16. The dialog box ID, IDD_BUTTON_CONTROLS, is already in its edit box (if the dialog is open), and CDialog is also already in the edit box for the Base Class. The name C_BtnDialog is chosen for the new class. The ClassWizard-suggested names for the .h and .cpp files can be changed. To change file names choose the “Change” button. The underbar (_) preceding the file names has been eliminated, and a shorter name, BtnDlg (of eight characters only to benefit Win3.1x users), has been chosen. A class named C_BtnDialog and files named BtnDlg.h and BtnDlg.cpp were also chosen. To continue click on the “Create” button.

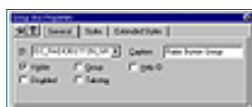


Figure 6-16: The Create New Class Query

The ClassWizard dialog box shown in Figure 6-17 now appears. It shows the added C_BtnDialog class and also shows that the member function DoDataExchange has been automatically included. We will use this dialog box to complete the task of adding the message map entries.

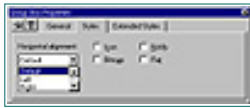


Figure 6-17: ClassWizard with Added BtnDialog Class and DoDataExchange

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

The Buttons Program Listing

Listing 6-1 gives the source code for the Buttons program. The compiler tools have inserted nonoperative code in the dialog class and also in the resource files. In order to focus discussion here on the relevant issues, much of the nonoperative code that was inserted by the compiler tools is not shown. The listing is followed by a discussion of functions used and relevant points to be made about this code.

Listing 6-1: Source Code for the Buttons Program

```

-----
                                stdafx files
-----

//
//  stdafx.h include file for standard system include files
//
#include <afxwin.h>

//
//  stdafx.cpp source file for standard includes
//
#include "stdafx.h"

-----
                                application class
-----
//
//  MyApp.h interface of the C MyApp class
//
class C MyApp : public CWinApp
{
public:
    BOOL InitInstance();
};
//
//  MyApp.cpp implementation of the class C MyApp
//
#include "stdafx.h"
#include "MyApp.h"
#include "mainfrm.h"

BOOL C MyApp::InitInstance()
  
```

```

{
    m_pMainWnd = new C_MainFrame;
    m_pMainWnd->ShowWindow(m_nCmdShow);
    SetDialogBkColor();
    return TRUE;
}

C MyApp  theApp;    // the object that runs the program

-----
                                mainframe class
-----
//
//  mainfrm.h    interface of the C_MainFrame class
//
class C_MainFrame : public CFrameWnd
{
    public:
        C_MainFrame();
        CButton m_ButtonOpen;
        CStatic m_StaticIcon;
    private:
       //{{AFX_MSG(C_MainFrame)
        afx_msg void OnOpenDialog();
       //}}AFX_MSG

        DECLARE_MESSAGE_MAP()
};

//
//  mainfrm.cpp  implementation of the C_MainFrame class
//
#include "stdafx.h"
#include "mainfrm.h"
#include "resource.h"
#include "BtnDialog.h"

BEGIN_MESSAGE_MAP(C_MainFrame, CFrameWnd)
   //{{AFX_MSG_MAP(C_MainFrame)
    ON_COMMAND(ID_OPENDIALOG, OnOpenDialog)
    ON_BN_CLICKED(ID_BUTTON_OPENDIALOG, OnOpenDialog)
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()

C_MainFrame::C_MainFrame()
{
    Create(NULL, "Button Controls", WS_OVERLAPPEDWINDOW,
        rectDefault, NULL, MAKEINTRESOURCE(IDR_MENU1));
    CRect ButtonRect(10, 10, 110, 50);
    m_ButtonOpen.Create("OpenDialog", WS_CHILD | WS_VISIBLE |
        BS_PUSHBUTTON, ButtonRect, this,
        ID_BUTTON_OPENDIALOG);
    CRect IconRect(10, 70, 0, 0);
    m_StaticIcon.Create("SmileyIcon", WS_CHILD | WS_VISIBLE |
        SS_ICON, IconRect, this);
}

void C_MainFrame::OnOpenDialog()
{
    C_BtnDialog myDialog;
    myDialog.DoModal();
}

```

```

-----
                                dialog class
-----
//
//  BtnDlg.h : header file
//

#include "resource.h"

class C_BtnDialog : public CDialog
{
//  Construction
public:
    C_BtnDialog(CWnd* pParent = NULL);    // standard constructor

// Dialog Data
   //{{AFX_DATA(C_BtnDialog)
    enum { IDD = IDD_BUTTON_CONTROLS };
    CButton    m_blue;
    //}}AFX_DATA

    CStatic m_CreateIcon;

    CButton* GetGreenCheck()
    { return (CButton*) GetDlgItem(IDC_GREEN);};

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(C_BtnDialog)
protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:
    // Generated message map functions
   //{{AFX_MSG(C_BtnDialog)
    afx_msg void OnCheckBlue();
    afx_msg void OnCheckGreen();
    afx_msg void OnCheckRed();
    afx_msg void OnCreateIcon();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

//
//  BtnDlg.cpp : implementation file
//
#include "stdafx.h"
#include "BtnDlg.h"

C_BtnDialog::C_BtnDialog(CWnd* pParent /*=NULL*/)
    : CDialog(C_BtnDialog::IDD, pParent)
{
    //{{AFX_DATA_INIT(C_BtnDialog)
    // NOTE: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT
}

void C_BtnDialog::DoDataExchange(CDataExchange* pDX)

```

```

{
    CDialog::DoDataExchange(pDX);
    //{AFX_DATA_MAP(C_BtnDialog)
    DDX_Control(pDX, IDC_BLUE, m_blue);
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(C_BtnDialog, CDialog)
    //{AFX_MSG_MAP(C_BtnDialog)
    ON_BN_CLICKED(IDC_CHECK_BLUE, OnCheckBlue)
    ON_BN_CLICKED(IDC_CHECK_GREEN, OnCheckGreen)
    ON_BN_CLICKED(IDC_CHECK_RED, OnCheckRed)
    ON_BN_CLICKED(IDC_CREATEICON, OnCreateIcon)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

void C_BtnDialog::OnCheckBlue()
{
    SendDlgItemMessage(IDC_RED, BM_SETCHECK, 0, 0);
    GetGreenCheck()->SetCheck(0);
    m_blue.SetCheck(1);
}

void C_BtnDialog::OnCheckGreen()
{
    SendDlgItemMessage(IDC_RED, BM_SETCHECK, 0, 0);
    GetGreenCheck()->SetCheck(1);
    m_blue.SetCheck(0);
}

void C_BtnDialog::OnCheckRed()
{
    SendDlgItemMessage(IDC_RED, BM_SETCHECK, 1, 0);
    GetGreenCheck()->SetCheck(0);
    m_blue.SetCheck(0);
}

void C_BtnDialog::OnCreateIcon()
{
    CRect IconRect(190, 195, 0, 0);
    m_CreateIcon.Create("SmileyIcon", WS_CHILD | WS_VISIBLE |
        SS_ICON, IconRect, this);
}

```

----- resource files -----

```

//
// resource.h
//
#define ID_BUTTON_OPENDIALOG          100
#define IDR_MENU1                     101
#define IDD_BUTTON_CONTROLS           102
#define IDC_RADIOBUTTON_GROUP         1000
#define IDC_CHECK_RED                  1001
#define IDC_CHECK_BLUE                 1002
#define IDC_CHECK_GREEN                1003
#define IDC_RED                        1004
#define IDC_GREEN                      1005
#define IDC_BLUE                       1006

```



```
#define IDC_CREATEICON          1007
#define ID_OPENDIALOG           40001
```

```
//
//  script1.rc
//
#include "resource.h"
#include "afxres.h"

AFX_IDI_STD_FRAME      ICON      DISCARDABLE      "smiley.ico"
SmileyIcon             ICON      DISCARDABLE      "smiley.ico"
```

```
IDR_MENU1 MENU DISCARDABLE
BEGIN
    MENUITEM "&OpenDialog",          ID_OPENDIALOG
END
```

```
IDD_BUTTON_CONTROLS DIALOG DISCARDABLE  50, 50, 186, 138
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Dialog-Button Controls"
FONT 8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON      "OK", IDOK, 129, 7, 50, 14
    PUSHBUTTON         "Cancel", IDCANCEL, 129, 24, 50, 14
    GROUPBOX           "Radio Button Group", IDC_RADIOBUTTON_GROUP, 17, 38, 77, 75
    CONTROL            "Check Red", IDC_CHECK_RED, "Button", BS_AUTORADIOBUTTON |
        WS_GROUP | WS_TABSTOP, 27, 55, 52, 10
    CONTROL            "Check Green", IDC_CHECK_GREEN, "Button",
        BS_AUTORADIOBUTTON, 27, 72, 58, 10
    CONTROL            "Check Blue", IDC_CHECK_BLUE, "Button", BS_AUTORADIOBUTTON,
        27, 89, 53, 10
    CHECKBOX           "Red", IDC_RED, 123, 55, 29, 10, WS_GROUP
    CHECKBOX           "Green", IDC_GREEN, 123, 72, 35, 10, NOT WS_TABSTOP
    CHECKBOX           "Blue", IDC_BLUE, 123, 89, 30, 10, NOT WS_TABSTOP
    PUSHBUTTON         "Create Icon", IDC_CREATEICON, 29, 127, 50, 14
END
```

Previous	Table of Contents	Next

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

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

Keyword

☒ Brief ☐ Full

☒ [Advanced Search](#)

☒ [Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

Discussion of the Buttons Program

Beginning with the last chapter, Chapter Five, we have reorganized our files such that we now use the stdafx files as given. At this juncture, we could have organized our files as we did in the first four chapters of this book. It would still work. However, when we use the document-view architecture, the stdafx file organization is absolutely necessary, so we have migrated to its usage. You will find that all of your files built by AppWizard will be organized with stdafx files.

In the application class, we have introduced a new function, `SetDialogBkColor()`, which must be called from within the `InitInstance()` function and which does just what its name implies. It has two arguments: the first argument is the RGB value of the dialog's background color, and the second argument is the RGB value of the text used in the dialogs. If no arguments are included, it uses the default color of gray for the background and black for the text for all of the dialogs in the application. If you insert arguments, then these RGB values will be the background and text colors of all of your dialogs. If you want to change the background or text color of individual dialogs or controls, you have to provide a special response function for the message `WM_CTLCOLOR` which is sent immediately before the dialog is displayed.

In the mainframe class, there are two interesting points not yet discussed in this chapter. First, note that two message map entries are going to the same message handler function, `OnOpenDialog()`. As many command messages as desired can be mapped to the same handler function. Secondly, note that the code in the `OnOpenDialog()` handler function looks a lot like the earlier example using the Common Dialog. An object, `MyDialog`, of class `C_BtnDialog` is instantiated on the stack. Then the object's `DoModal()` member function is called, which makes the dialog window visible. Modal dialogs are always created on the stack; therefore, when the handler function closes, not only does the dialog window disappear, but the C++ object, `MyDialog`, is also destroyed.

The dialog class was completely written by the compiler tool, except for the `GetGreenCheck()` inline function and the embedded `CStatic` object, `m_CreateIcon`, that we inserted ourselves.

Note: When the compiler tools generate code, the data members and member functions are all either public or protected.

As illustrated, the dialog class does several things:

- Its constructor passes to the `CDialog` base class the identification of the `DIALOG` resource script and the parent. The `DIALOG` resource script has been assigned an *enum value of* `IDD` (for the first dialog created). If a second `DIALOG` resource script is created in an application, it

will be assigned the value of IDD1, etc. If you write code manually for a dialog class, this enum convention must be used, since the MFC classes are relying on it.

- The virtual function DoDataExchange() is used almost without exception with a dialog box. The code to include it has been automatically written. When ClassWizard defines member variables, it does so by inserting specific DDX functions. In our example, we had only one item defined as a member object, m_blue. For this, ClassWizard inserted the DDX function:

```
DDX_Control(pDX, IDC_BLUE, m_blue);
```

This DDX function, when exercised, establishes the connection (pointer) between the object m_blue and the control identified as IDC_BLUE. We discuss more DDX functions and how they are used in Chapter Seven.

- The connection (pointer) for the green check box was established by inserting the following code in the dialog's header file; establishing a pointer in this way is an alternative to using the DDX mechanism:

```
CButton* GetGreenCheck()  
{ return (CButton*) GetDlgItem(IDC_GREEN);}
```

- The dialog class has its own message map which consists of responses to control notification messages from its child controls.

Note: An entry in the message map for IDOK was not asked for. Nevertheless, when we press the Enter key on the keyboard, the dialog box closes. The default action of the Enter key is a built-in member function of the CDialog class which is inherited by our dialog class. It operates whether we insert the OnOK() function in our dialog class or not. The same comment can be made for the Cancel button. The Cancel button will always be functional whether the handler function for IDCANCEL is in a dialog class or not, because the OnCancel() function is a member function of CDialog and is inherited by the dialog class. Both IDOK and IDCANCEL are reserved identifiers. Do not use them; if you do, you will get unusual run-time results.

The DIALOG resource script in the file Script1.rc and its entries in the resource.h file were written by the compiler tools as we placed controls and filled out property pages. It has many features to note:

- The first four lines define the dialog box. Its name, IDD_BUTTON_CONTROLS, appears first and is followed by the keyword DIALOG which defines it as a DIALOG resource. The final four numbers of the first line are the dialog box's dimensions in DLUs. The second line is the style of the dialog box itself, which is the standard style. The third line is the caption. The fourth line is the font, which is the system default font.
- Within the BEGIN and END statements that follow the first four lines of the dialog box's definition, we find a single script or line for each of the controls in the dialog box. Note that these lines appear in tab order. Each resource script line for a control contains: (1) the text, if any, associated with that control; (2) the ID value assigned to that control; (3) the style parameters, unless it is the default; and (4) four integers which define the area (in DLUs) occupied by the control window. There are two types of resource scripts for controls. One type begins with the keyword CONTROL, and its control class and style parameters are called out as the third and fourth parameters:

```
CONTROL    "Check Red", IDC_CHECK_RED, "Button", BS_AUTORADIOBUTTON |  
           WS_GROUP | WS_TABSTOP, 27, 55, 52, 10
```

The other type begins with the control's style parameter:

```
CHECKBOX    "Red", IDC_RED, 123, 55, 29, 10, WS_GROUP
```

You can choose to enter the Script1.rc file and modify these statements manually. In some cases it may be the only way to get what you want, for example, if you want to use a static control with the

style SS_ICON.

Previous	Table of Contents	Next

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

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

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

Keyword

☒ Brief
 ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Summary

Dialog boxes are one of the fundamental building blocks of graphical user interfaces. Most dialog boxes are modal, which means that the user must address the information presented on the dialog box and close the dialog box before access can be regained to the application. Modeless dialog boxes, which are less frequently used, allow access to the application while they are still open. Common dialogs are an implementation of specific frequently needed dialogs and are contained in the MFC classes: CFileDialog, which implements the dialog box for opening and saving files; CPrintDialog, which implements the dialog box for printing and print setup; CPageSetupDialog, which implements the dialog box for setting up a page; CFindReplaceDialog, which implements the “find” and “find and replace” dialog boxes; CFontDialog, which implements the font selection dialog box; and CColorDialog, which implements the dialog box for choosing colors.

A dialog box is a DIALOG resource and its source code is contained in a resource script file. A dialog box is a specialized window that can have specific window styles and dialog styles. The style most frequently used for a dialog box is the dialog style of DS_MODALFRAME combined with the window styles of WS_POPUP, WS_CAPTION, WS_SYSMENU, and WS_VISIBLE. Dialog boxes hold many kinds of controls.

Certain controls, termed Win3.1 common controls, have always been an integral part of the Windows operating system. These controls are defined by the operating system. These common controls are categorized into control classes. The MFC classes associated with each of the control classes are: CButton, CComboBox, CEdit, CListBox, CStatic, and CScrollBar.

Common controls are usually placed on dialog boxes, although they can be directly placed on any window. Windows-defined styles are specified that affect the appearance and behavior of the control. The button control class can

be a push button, a radio button, a check box, an owner-drawn button, or a group box. Each control class has its own styles and also can have various window and child window styles. These styles are represented by choices displayed on the control's property page. Controls send messages, known as control notification messages, to their parent and they also receive messages. The messages that a control can send and receive depend on the control class.

To place controls directly onto a window (other than a dialog box), the control is represented by an object of its class and the member function `Create()` is used. To place controls on a dialog box, a Dialog Editor is generally used, which generates the dialog resource script and the attendant entries in the resource.h file; the dialog resource script can be generated manually, although this is seldom done.

The most convenient method of creating a dialog box is to use a Dialog Editor, then to use the ClassWizard tool to encapsulate the dialog in its own class derived from `CDialog`. This is done so that the dialog inherits all of the functionality of the `CDialog` class. The ClassWizard tool also automatically incorporates the `DoDataExchange()` functions. A `DoDataExchange()` function, `DDX_Control()`, can be used to establish a pointer between an object of a specific control class and the control. As an alternative, a function which establishes the pointer can be added to your code manually.

Previous	Table of Contents	Next

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

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

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

☒ Brief ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Chapter 7

List Box, Combo Box, and Edit Controls and Data Transfer Functions

In the last chapter, the button and static common controls were discussed. This chapter completes the discussion of Win3.1 common controls. The Win3.1 common control classes of list box, combo box, and edit controls as well as their respective MFC classes: CListBox, CComboBox, and CEdit are covered in this chapter. The Do Data Exchange (DDX) and the Do Data Validation (DDV) functions have a larger role with these controls which take in user inputs. The DDX and DDV functions are covered in depth in this chapter. CString features are also discussed.

In the last chapter, an example with a dialog that has all kinds of controls of the button class was presented. This chapter presents an example which covers the remaining Win3.1 common control classes (except the scroll bar control class, which is a stand-alone scroll bar that is seldom used and is not supported by DDX). In the example presented in this chapter, three styles of the combo box—CBS_SIMPLE, CBS_DROPDOWN, and CBS_DROPLIST—as well as two styles of the edit control—single line and multiline (ES_MULTILINE)—are covered. In the example, note that scroll bars are attached to the edit controls, combo boxes, and list boxes, as needed. Scroll bars can be specified for these control classes.

Overview of List Box, Combo Box, and Edit Controls

The list box, combo box, and edit controls are much more complex than the buttons and static controls addressed in the last chapter. For example, combo

boxes contain an edit box that comes complete with limited built-in text editing capabilities. Scroll bars will automatically appear and disappear, as needed. For some controls, substantial amounts of data can be entered into the control by the user.

The window styles that can be used for these controls include the ones given for the button and static controls: WS_CHILD, WS_DISABLED, WS_VISIBLE, WS_TABSTOP, and WS_VSCROLL. In addition to these window styles, list box and edit controls can also have WS_HSCROLL, WS_VSCROLL, and WS_BORDER window styles, and combo box controls can have the WS_VSCROLL style.

All the extended window styles given in Table 6-2 of Chapter Six apply to the combo box, list box, and edit controls. In addition, the extended window style WS_EX_DLGMODALFRAME, which gives a window with a double border (like the dialog modal frame), and the extended window style WS_EX_LEFTSCROLLBAR, which places a vertical scroll bar to the left of the window, also apply to the combo box, list box, and edit controls.

The control styles for each class are given in the following discussion. Due to the complexity of these controls, there are many control styles.

Edit Control Styles

An edit control is a rectangular child window which has limited text editing capabilities. The user can enter text from the keyboard. The user selects the control, and gives it the input focus by clicking the mouse inside it or pressing the Tab key. The user can enter text when the control displays a flashing insertion point. The mouse can be used to move the cursor and select characters to be replaced, or to position the cursor for inserting characters. The Backspace key can be used to delete characters. Edit controls use the fixed-pitch font and display Unicode characters. Tab stops are assumed to be at every eighth character position. By default, an edit control has no border; to give it one, use WS_BORDER.

All of the edit control styles given are found on the control's property page, which is used by the Dialog Editor. The edit control's property page for MFC 4.0 is given in Figure 7-1. Win3.1x operating system users will have property pages with fewer choices. Edit control styles are described in Table 7-1.



Figure 7-1: Edit Control Property Pages

Table 7-1: Edit Control Styles

Style	Meaning
ES_LEFT	Justifies the text to the left.
ES_CENTER	Centers text in a multiline edit control.
ES_RIGHT	Aligns text flush right in a multiline edit control.

ES_LOWERCASE	Converts all characters to lowercase as they are being typed into the edit control.
ES_UPPERCASE	Converts all characters to uppercase as they are typed into the edit control.
ES_NUMBER	Accepts only numbers as inputs. (This is a new style introduced with MFC 4.0.)
ES_OEMCONVERT	Converts text entered in the edit control from the ANSI character set to the OEM character set and then back to ANSI. This ensures proper character conversion when the application calls the CharToOem function to convert an ANSI string in the edit control to OEM characters. This style is most useful for edit controls that contain filenames.
ES_AUTOVSCROLL	Automatically scrolls text up one page when the user presses Enter on the last line.
ES_AUTOHSCROLL	Automatically scrolls text to the right by 10 characters when the user types a character at the end of the line. When the user presses the Enter key, the control scrolls all text back to position zero.
ES_PASSWORD	(for single-line only) Displays all characters as an asterisk (*) as they are typed into the edit control. An application can use the EM_SETPASSWORDCHAR message to change the character that is displayed.
ES_NOHIDESEL	Overrides the default action, in which an edit control hides the selection when the control loses the input focus. Inverts the selection instead.
ES_WANTRETURN	Specifies that a carriage return be inserted when the user presses the Enter key while entering text into a multiple-line edit control in a dialog box. Without this style, pressing the Enter key has the same effect as pressing the dialog box's default push button. This style has no effect on a single-line edit control.
ES_READONLY	Prevents the user from entering or editing text in the edit control.
ES_MULTILINE	Designates a multiple-line edit control. (The default is single-lined.) If the ES_AUTOVSCROLL style is specified, the edit control shows as many lines as possible and scrolls vertically when the user presses the Enter key. If ES_AUTOVSCROLL is not given, the edit control shows as many lines as possible and beeps if Enter is pressed when no more lines can be displayed. If the ES_AUTOHSCROLL style is specified, the multiple-line edit control automatically scrolls horizontally when the caret goes past the right edge of the control. To start a new line, the user must press Enter. If ES_AUTOHSCROLL is not given, the control automatically wraps words to the beginning of the next line when necessary; a new line is also started

if Enter is pressed. The position of the wordwrap is determined by the window size. If the window size changes, the wordwrap position changes and the text is redisplayed. Multiple-line edit controls can have scroll bars. An edit control with scroll bars processes its own scroll-bar messages. Edit controls without scroll bars scroll as previously described and process any scroll messages sent by the parent window.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes
(Publisher: Wordware Publishing, Inc.)
Author(s): Shirley Wodtke
ISBN: 1556225121
Publication Date: 01/01/97



Search this book:

List Box Styles

List box controls consist of a list of character strings. The control is used whenever an application needs to present a list of names, such as filenames, that the user can view and select. The user can select a string by pointing to the string with the mouse and clicking a mouse button. When a string is selected, it is highlighted and a notification message is passed to the parent window. A scroll bar can be used with a list box control to scroll lists that are too long or too wide for the control window. All of these list box styles can be found on the list box’s property page. Figure 7-2 gives the list box property page for MFC 4.0. If you are using the Win3.1x operating system, fewer choices are available on the property page. A list box can have any of a large number of styles which are listed in Table 7-2. The default style is single-selection.



Figure 7-2: List Box Property Pages

Table 7-2: List Box Control Styles

Style	Meaning
LBS_STANDARD	Sorts strings in the list box alphabetically, and sends the parent window an input message whenever the user clicks or double-clicks a string. (The list box contains borders on all sides.)
LBS_MUTIPLESEL	Toggles string selection each time the user clicks or double-clicks the string; any number of strings can be selected.

LBS_EXTENDEDSEL	Allows the user to select multiple items using the Shift key and the mouse, or special key combinations.
LBS_SORT	Sorts strings in the list box alphabetically.
LBS_NOTIFY	Causes the parent window to receive an input message whenever the user clicks or double-clicks a string.
LBS_MULTICOLUMN	Specifies a multicolumn list box that is scrolled horizontally. The SetColumnWidth member function sets the width of the columns.
LBS_NOREDRAW	Causes the list box display to not be updated when changes are made. This style can be changed at any time by sending a WM_SETREDRAW message.
LBS_USETABSTOPS	Allows a list box to recognize and expand tab characters when drawing its strings. The default tab positions are 32 dialog units.
LBS_WANTKEYBOARDINPUT	Causes the owner of the list box to receive WM_VKEYTOITEM or WM_CHARTOITEM messages whenever the user presses a key while the list box has input focus. This allows an application to perform special processing on the keyboard input.
LBS_DISABLENOSCROLL	Allows the scroll bar to be displayed even when it does not contain enough items to scroll. With this style, the list box shows a disabled vertical scroll bar when the list box does not contain enough items. Without this style, the scroll bar is hidden when the list box does not contain enough items.
LBS_NOINTEGRALHEIGHT	Creates the size of the list box at exactly the size specified by the application when it created the list box. (Usually, Windows sizes a list box so that the list box does not display partial items.)
LBS_OWNERDRAWFIXED	Assigns responsibility to the owner of the list box for drawing its contents; the items in the list box are the same height.
LBS_OWNERDRAWVARIABLE	Assigns responsibility to the owner of the list box for drawing its contents; the items in the list box are variable in height.

LBS_HASSTRINGS

Specifies an owner-drawn list box that contains items consisting of strings. The list box maintains the memory and pointers for the strings so the application can use the `GetText()` member function to retrieve the text for a particular item.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes
(Publisher: Wordware Publishing, Inc.)
Author(s): Shirley Wodtke
ISBN: 1556225121
Publication Date: 01/01/97



Search this book:

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

Combo Box Styles

Combo box controls consist of a selection field, similar to an edit control, plus a list box. The list box may be displayed at all times or may be dropped down when the user selects a pop box next to the selection field. Depending on the style of the combo box, the user can or cannot edit the contents of the selection field. If the list box is visible, typing characters into the selection box will cause the first list box entry that matches the characters typed to be highlighted. Conversely, selecting an item in the list box displays the selected text in the selection field. All of the combo box styles are found on its property page. Figure 7-3 gives the combo box property page for MFC 4.0. Win3.1x operating system users will find fewer choices on their combo box property pages. Combo box control styles are described in Table 7-3.

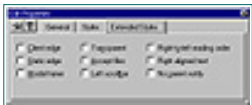


Figure 7-3: Combo Box Property Pages

Table 7-3: Combo Box Styles

Style	Meaning
CBS_SIMPLE	This style is an edit box with a list box below it. It displays the list box at all times. The current selection in the list box is displayed in the edit control. When this control has the focus, the user can type into the edit box. Thus the user can choose from among the listed strings or enter his own string.

CBS_DROPDOWN	Similar to CBS_SIMPLE except that it does not display the list box unless the user selects the pop box icon next to the selection field. This style has the same functionality as the CBS_SIMPLE Combobox. In Figure 6-1, the box below the text “Drives:” is a CBS_DROPDOWN combo box.
CBS_DROPDOWNLIST	Similar to CBS_DROPDOWN, except it replaces the edit control with a static-text item that displays the current selection in the list box. The user cannot type in another selection. This style is appropriate when the user is required to choose from among the listed strings.
CBS_SORT	Automatically sorts strings entered into the list box.
CBS_LOWERCASE	Converts all characters to lowercase as they are being typed into the edit box portion of the combo box. (This style was introduced with MFC 4.0.)
CBS_UPPERCASE	Converts all characters to uppercase as they are being typed into the edit box portion of the combo box. (This style was introduced with MFC 4.0.)
CBS_OEMCONVERT	Converts text entered in the combo box edit control from the ANSI character set to the OEM character set and then back to ANSI. This ensures proper character conversion when the application calls the AnsiToOEM Windows function to convert an ANSI string in the combo box to OEM characters. This style is most useful for combo boxes that contain filenames and applies only to combo boxes created with the CBS_SIMPLE or CBS_DROPDOWN styles.
CBS_AUTOHSCROLL	Automatically scrolls the text in the edit control to the right when the user types a character at the end of the line. If this style is not set, only text that fits within the rectangular boundary is allowed.
CBS_DISABLENOSCROLL	Allows the list box to show a disabled vertical scroll bar when the list box does not contain enough items to scroll. Without this style, the scroll bar is hidden when the list box does not contain enough items.

CBS_NOINTEGRALHEIGHT	Specifies that the size of the combo box is exactly the size specified by the application when it created the combo box. Normally, Windows sizes a combo box so that the combo box does not display partial items.
CBS_OWNERDRAWFIXED	Makes the owner of the list box responsible for drawing its contents; the items in the list box are variable in height.
CBS_OWNERDRAWVARIABLE	Makes the owner of the list box responsible for drawing its contents; the items in the list box are variable in height.
CBS_HASSTRINGS	Creates an owner-draw combo box containing items consisting of strings. The combo box maintains the memory and pointers for the strings so the application can use the GetText() member function to retrieve the text for a particular item.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Operations and Messages for Win3.1 Common Controls

The combo box, list box, and edit controls parallel the other Win3.1 common controls. They are encapsulated in the MFC classes CComboBox, CListBox, and CEdit, respectively. When they are wrapped they inherit many useful functions, including a Create() function. The Create() function is used exactly as shown in Chapter Six for the buttons example to dynamically create and attach each control to a window. Other useful functions are inherited from their respective MFC class. In fact, since each of these controls is so complex and there is such a large number of inherited functions, they are not listed here; they can be found in the reference material. Since we covered the simpler buttons class in the previous chapter, the general idea has been explained, so it is not repeated here for these controls.

These controls send control notification messages to their parents. The edit control notification messages are noted by EN_XXXX, and their message map entry has the form: ON_EN_XXXX(<control ID>,<handler function>). The combo box notification messages are noted by CBN_XXXX, and their message map entry has the form: ON_CBN_XXXX(<control ID>,<handler function>). Similarly, the list box control notification messages are noted by LBN_XXXX, and their message map entry has the form: ON_LBN_XXXX(<control ID>,<handler function>).

An Example Program

This example covers the remaining Win3.1 common control classes that were not covered in Chapter Six. The control classes covered are: the list box, the combo box, and the edit control. In this example, the user inputs data through these controls; the data is stored in a persistent database; the user can open and close the dialog box and the database persists. Modal dialog boxes are always

constructed on the stack, and frequently there is a need to provide more permanent storage for data entered into a dialog box. This example demonstrates how to move the data into a persistent database before the stack closes and the data is lost.

This example demonstrates the DDX (Do Data Exchange) mechanism, and a discussion follows. The DDX mechanism “captures” the user input from the controls. It stores the user input from the controls in a member variable attached to each control. However, these member variables belong to the class that is derived from CDialog, so when the stack on which the dialog box has been constructed is closed their values are lost. Their values can be transferred to a persistent database, the dialog box and the stack can close, and when the dialog is reopened, the database is still available. This is demonstrated in the example, which shows how to build and use a persistent database based on input from the dialog box. Global variables can be used for persistent data storage, as demonstrated in this example. Also, member variables of the mainframe class are candidates for persistent data storage, since the mainframe class persists for the duration of the application.

We will build the application, named `UsrInput`, which is shown in Figure 7-4. The main window has a menu with one main menu item, “O penDialog.” When the main menu “O penDialog” item is chosen, the dialog box as shown in Figure 7-4 opens. The dialog box contains controls in which the user can choose from a list or can make his own data entry. The dialog box also has push buttons labeled “Next,” “Previous,” and “Update.” These push button controls are used to build the database, cycle through it, and update it.



Figure 7-4: The `UsrInput` Program

Programming the `UsrInput` Example

We will build this program in two steps. First we program the main window, then the dialog box. The instructions to generate this application are given in two parts.

In this example, named `UsrInput`, generating the main window code is simpler, and the steps are more straightforward than the example given in the last chapter. The first part covers all the five tasks necessary to generate the code for the main window which are: creating a new project as a simple application (without using AppWizard); typing in all of its initial .h and .cpp files; creating the menu; adding the message map entry and the stubbed-out handler function; and compiling the program which will be the main window only.

The second part covers all the tasks necessary to generate the code for the dialog box: generating the resource script; creating a dialog class; adding message map entries; adding member variables; customizing the dialog class by adding code to suit our specific needs; completing the message handler code for the dialog class; completing the handler function for the main window class; compiling the completed program. An overview of each part is given

first, then the procedures for completing each part are given in detail.

Previous	Table of Contents	Next

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

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

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

☒ Brief ☐ Full

- Advanced Search
- Search Tips

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Generating the Main Window's Code

We have only five steps to create the main window program for this example. In this first part, all necessary discussion is added along with each step.

1. Create a new project named `UsrInput` as a simple application (without using AppWizard).
2. Type in all the .h and .cpp files, given as follows, and add all the .cpp files to your `UsrInput` project. Only the code that you should type in at this point is given; the files to type in are:

```
//
//  stdafx.h include file for standard system include files
//
#include <afxwin.h>
//
//  stdafx.cpp      source file for standard includes
//
#include "stdafx.h"
//
//  MyApp.h
//
class C MyApp : public CWinApp
{
public:
    BOOL InitInstance();
};
//
//  MyApp.cpp define the class behaviors for the class
//
#include "stdafx.h"
#include "MyApp.h"
#include "mainfrm.h"
BOOL C MyApp::InitInstance()
{
    m_pMainWnd = new C_MainFrame;
    m_pMainWnd->ShowWindow(m_nCmdShow);
    SetDialogBkColor();
    return TRUE;
}

C MyApp theApp; // the object that runs the program

//
//  mainfrm.h      interface of the C_MainFrame class
//
class C_MainFrame : public CFrameWnd
```

```

{
    public:
        C_MainFrame();

    private:
        //{AFX_MSG(C_MainFrame)
        //}AFX_MSG

    DECLARE_MESSAGE_MAP()
};

//
//  mainfrm.cpp      implementation of the C_MainFrame class
//
#include "stdafx.h"
#include "mainfrm.h"
#include "resource.h"
BEGIN_MESSAGE_MAP(C_MainFrame, CFrameWnd)
    //{AFX_MSG_MAP(C_MainFrame)
    //}AFX_MSG_MAP
END_MESSAGE_MAP()

C_MainFrame::C_MainFrame()
{
    Create(NULL, "User Input Controls", WS_OVERLAPPEDWINDOW,
        rectDefault, NULL, @program = MAKEINTRESOURCE(IDR_MENU1));
}

```

3. Create the menu resource. It will have only one item “Open Dialog” on the menu bar. The item will be an end menu item, not a popup. Give this end menu item an ID of ID_OPENDIALOG. The Script1.rc and its resource.h file entries are automatically generated. Save the file as Script1.rc and make sure that it is added to your project file. Do not remove any code from the Script1.rc and resource.h file. We will be re-entering these files in the next part to generate the dialog box.
4. Open ClassWizard and add the message map entry and the stubbed-out handler function for the menu item. At this point, we cannot complete the main window’s handler function to open the dialog box. We cannot do this until after the dialog box has been generated.
5. Compile and run the program; at this point, it should look like the running program on Figure 7-5. The main window has its menu attached. However, the handler function has not been completed, so the menu item does nothing. Next we will generate the code for the dialog box.



Figure 7-5: UsrInput Main Window Program

Generating the Dialog Box Code

Generating the dialog box code for the UsrInput example consists of the following steps, which are explained in detail in the paragraphs that follow this list of steps. The steps are:

1. Generate the source code for the DIALOG resource script.
2. Generate the code for a dialog class (derived from CDialog) to which the DIALOG resource script will be attached.
3. Add message map entries and stubbed-out handler functions to the dialog class.
4. Add member variables for each of the controls that will be receiving information either from the user or from the database.
5. Add additional code to the CDialog class to customize it as needed and also fix up files. (We have a function to override, which we must code ourselves, to initialize the lists of data in our combo boxes and list boxes, and we have our own functions to add that will perform the tasks of bringing the data to and from the persistent database.)
6. Complete the message handler functions in the dialog class.
7. Complete the message handler function in the main window’s class.

Next we will discuss the tasks listed in each of the preceding steps. Each of these seven tasks has a section covering it

in detail. In the last chapter introducing dialog boxes, each display that you would see as you completed a task was covered. In this chapter each task is discussed, but the displays are only shown if they are used differently than in the last chapter or there is some point to be made about the information on the particular display.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Creating the DIALOG Resource Script

In this step we build the dialog box and its associated DIALOG resource script with the Dialog Editor. The finished dialog box is given in Figure 7-6. Refer to this figure as you go through the instructions that follow.

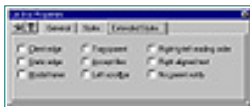


Figure 7-6: The Completed Dialog Box

In the Dialog Editor do the following steps to construct the dialog box:

- On the dialog property page, type in an ID of IDD_USER_INPUT. Give it the title "User Input Controls" and change its (XPos, YPos) to (20, 20).
 - Close the property page and change the size of the dialog to 200 x 150, so all the controls will fit.
- Drag and drop three push button controls. Label them "Next," "Previous," and "Update." Give them IDs of IDC_NEXT, IDC_PREVIOUS, and IDC_UPDATE. (For the push button controls, Visible should be checked, but uncheck Tabstop.)
- Do a static control label "Single Line Edit Box," as shown. (There is no need to change the ID on any of the static controls in this example.) Then drag and drop an edit box and enlarge the edit box as shown.
 - Give the edit control an ID of IDC_SINGLELINE. Make it Visible and with Tabstop on its property page. (We want tabstops on each control that accepts user input.) Go to the style page and make sure that AutoHScroll is unchecked.
- Do a static control label "Multi Line Edit Box" as shown. Then add another edit box and enlarge it to approximately the size shown.

a. Give the edit control an ID of IDC_MULTILINE. On the edit control's property page, make it Visible with Tabstop. Go to its style page: check Multiline, uncheck AutoHScroll, check Vertical Scroll, check Horizontal Scroll, and check Want Return.

Note: When you are using a multiline edit box on a dialog, you will usually want the style ES_WANTRETURN for the edit box, which gives your normal Enter key behavior for the edit box. If you do not use this style, you will find that using the Enter key triggers the default behavior of the OK button, and the dialog box closes. (If you don't use the ES_WANTRETURN style, Ctrl+Enter will give a normal carriage return in the edit box.)

5. Do a static control label "ListBox," as shown. Next add the list box control.

a. In this case, the IDC_LIST1 is fine and the styles you want are: Visible, Tabstop, Border, Sort, and Vertical Scroll. For the list box, you need to add code to initialize the list of choices in the function OnInitDialog().

6. Do a static control label "DropDown ComboBox," as shown. Next add the DropDown combo box under the list box. In this case, we do not alter the size of the combo box offered. This gives us a dropped list two items long with a vertical scroll bar.

a. Give it an ID of IDC_DROPDOWN_CBOX. For all the combo boxes in this example we want the general properties of Visible and Tabstop and the styles of Vert Scroll and Sort. On the style page, make sure that the Type of "Dropdown" is selected. In this case, we do not enter any list choices on the property page—we will add code to initialize the list choices in the function OnInitDialog(). (For combo boxes you have a choice of entering initial lists either way.)

7. Do a static control label "Simple ComboBox," as shown. Next add the Simple combo box. We will enlarge our simple combo box in the vertical direction to fit a larger list into it.

a. Give it the ID of IDC_SIMPLE_CBOX. We want the general properties of Visible and Tabstop and the styles of Vert Scroll and Sort. On the style page, make sure that the Type of "Simple" is selected.

b. Type the initial list into the property page, ending each line with Ctrl+Enter. (Note that Enter alone will not work—it causes the dialog box to close.) In the given example the initial list—Orange, Banana, Pear, Apple, Peach—is used. Figure 7-7 shows the combo box property pages, which have initialization lists entered on them for this example.



Figure 7-7: Initializing Lists on the Combo Box Property Pages

8. Do a static control label "DropList ComboBox," as shown. Next add

the DropList combo box. You will change the size of this combo box, but in this example it is enlarged so that it will drop down outside the boundaries of the dialog box.

- a. Give it an ID of IDC_DROPLIST_CBOX. We want the general properties of Visible and Tabstop and the styles of Vert Scroll and Sort. On the style page, make sure that the Type of “DropList” is selected.
- b. Type in the initial choices, ending each line with Ctrl+Enter. The entered choices are lengthy, but they demonstrate how the box is widened. The initial list—Strawberry Shortcake, Chocolate Ice Cream, Vanilla Pudding, Carrot Cake, and Pumpkin Pie—is used. (See Figure 7-7.)
- c. Test your dialog box to see if your list fits into the box’s dimensions. Your dialog box should now look like Figure 7-6. If it does not, go back and enlarge the box.

Note: In this example, the tab order is set so that it is convenient for the user to move from one control to the next to make selections. The tabstop property is set on all the controls that the user enters data into, but we also need to set the order of tabbing so that the tabbing progresses visibly from one control to the next.

9. Choose Set Tab Order from the Layout menu and set the tabbing order, as shown in Figure 7-8.

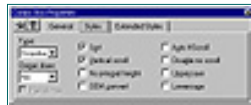


Figure 7-8: Setting the Tab Order

Creating a Dialog Class

Now it is time to use ClassWizard to create the dialog class. With the dialog box open, ClassWizard can be opened by selecting “View|ClassWizard” from the menu. The Add Class dialog appears. Since our dialog box was open, IDD_USER_INPUT is already in the appropriate edit box, and CDialog is also in its appropriate edit box. So all we need to do is to type in the name of our new dialog class—in this case C_UsrDialog is chosen. The filenames suggested by ClassWizard are changed to be “UsrDialog” (without the _ and with only 8 characters for the benefit of Win3.1x users). Next, you can click on the “Create” button.

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

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

SEARCH
 ITKNOWLEDGE

Keyword

☒ Brief
 ☐ Full

☒ Advanced

☒ Search

☒ Search Tips

BROWSE
 BY TOPIC

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes
(Publisher: Wordware Publishing, Inc.)
Author(s): Shirley Wodtke
ISBN: 1556225121
Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

[Previous](#)

[Table of Contents](#)

[Next](#)

Adding Message Map Entries to the Dialog Class

After creating a class, the MFC Class Wizard dialog box for Message Maps needs to be addressed.

1. Select C_UsrDialog in the Object ID list. Scroll in the “Messages” list until you get WM_INITDIALOG, which you then select and add the function.
2. Next select IDC_NEXT in the Object ID list. In the “Messages” list, select the BN_CLICKED message and add the function.
3. Select IDC_PREVIOUS. Select the BN_CLICKED message and add the function.
4. Select IDC_UPDATE. Select the BN_CLICKED message and add the function.

When you are finished the MFC ClassWizard box looks like Figure 7-9. If you do not click on “OK” at this point, you will be able to continue to the next step by choosing the tab “Member Variables.”

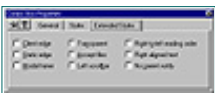


Figure 7-9: Adding Message Map Entries to the Dialog Class

Adding Member Variables to the Dialog Class

The MFC ClassWizard box now looks like the one in Figure 7-10; now you can add the member variables.



Figure 7-10: Adding Member Variables to the Dialog Class

1. Choose IDC_DROPDOWN_CBOX, then click on “Add Variable.” The Add Member Variable query box shown in Figure 7-11 pops up.

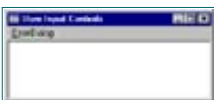


Figure 7-11: The Add Member Variable Query Box

2. Type in the variable name, in this case m_MainDish, as shown, making sure that Category is set on Value and that the Variable type is set on CString.
3. Continue adding a member variable for each of the user input boxes (list box, combo boxes, and edit boxes). You can use the names as shown in Figure 7-12. Specifying the maximum characters is

unnecessary (since we did not use Autoscroll each box is limited by its size), except in the multiline edit box, where you can set a limit (100 was chosen here).



Figure 7-12: The Finished Member Variable Box

4. Now choose “OK.”

Customizing the Dialog Class Code

Files now need to be fixed and the dialog class’s code needs to be customized. First, to fix our files, do the following:

1. Add the `UsrDialog.cpp` file to the project if it has not already been added automatically.
2. Replace the include files in the `UsrDialog.cpp` file to be just the following two:

```
#include "stdafx.h"
#include "UsrDialog.h"
```

3. Include the file “resource.h” in the `UsrDialog.h` file.
4. Include the file “`UsrDialog.h`” in the `mainfrm.cpp` file.

To customize the dialog class code to meet our needs, there are three tasks to perform. These are: providing the global database; completing the code in the overridden function that will initialize our lists; and providing functions to move data in and out of the global database.

1. Provide for a global database.
 - a. To do this define a struct with a member that corresponds to each member variable saved from the dialog box and provide the initialization of an array of three such structs. Add the following code to the beginning of the file “`UsrDialog.cpp`”:

```
int nDay;

struct{
    char Day[25];
    char Fruit[25];
    char Dessert[25];
    char Salad[25];
    char MainDish[25];
    char Comment[100];
} Holiday[] = { " ", " ", " ", " ", " ", " ", " ", " ", " ",
                " ", " ", " ", " ", " ", " ", " ", " ", " ",
                " ", " ", " ", " ", " ", " ", " ", " ", " "};
```

2. Add code which initializes the lists in the function `OnInitDialog()`.

Note: The function `OnInitDialog()` is a virtual function and has been added to our dialog class by Class Wizard when we requested a handler function for the message `WM_INITDIALOG`. So you might think of it as a special kind of handler function, although it is a virtual function.

3. In the file `UsrDialog.cpp`, complete the code in `C_UsrDialog::OnInitDialog()`, as follows:

```
BOOL C_UsrDialog::OnInitDialog()
{
-->    CListBox* pLB = (CListBox*)GetDlgItem(IDC_LIST1);
-->    pLB->AddString("Carrots");
-->    pLB->AddString("Celery");
```

```

-->    pLB->AddString("Lettuce");
-->    pLB->AddString("Tomato");
-->    pLB->AddString("Cucumber");
-->    pLB->AddString("Potato");

-->    CComboBox* pCBxDropDown =
        (CComboBox*)GetDlgItem(IDC_DROPDOWN_CBOX);
-->    pCBxDropDown->AddString("Salmon");
-->    pCBxDropDown->AddString("Steak");
-->    pCBxDropDown->AddString("Prawns");
-->    pCBxDropDown->AddString("Turkey");

    return CDialog::OnInitDialog();
}

```

4. Customize the dialog class to move data in and out of the global database. The code added in this section consists of defining two functions, `UpdateFromUser()` and `UpdateFromBase()`. `UpdateFromUser()` copies the values of the member variables to the global array; it uses `strcpy()` which can copy a `CString` to a char array. The function `UpdateFromBase()` takes the values from the global database and transfers them to the member variables, where they can then be displayed in the controls. In the handler functions, we will call the function `CWnd::UpdateData()` to move the data from the member variables into the controls and vice versa. `UpdateData()` works in both directions, as specified by the `BOOL` parameter passed to it. If `UpdateData(FALSE)` is called, the DDX functions transfer data from the data members to the dialog controls. If `UpdateData(TRUE)` is called, the DDX functions transfer data from the dialog controls to the data members. `UpdateData()` works by invoking the `DoDataExchange()` functions to move data between the member variables and the controls.

a. Insert the code shown below in bold and preceded by an arrow into the existing lines of code in the `UsrDlg.h` file. The existing lines of code are shown as not bold.

```

class C_UsrDialog : public CDialog
{
// Construction
public:
    C_UsrDialog(CWnd* pParent = NULL);    // standard constructor
-->    void UpdateFromBase(int n);
-->    void UpdateFromUser(int n);

```

b. Add the following code to the `UsrDlg.cpp` file:

```

void C_UsrDialog::UpdateFromBase(int n)
{
    m_Day = Holiday[n].Day;
    m_Fruit = Holiday[n].Fruit;
    m_Dessert = Holiday[n].Dessert;
    m_Salad = Holiday[n].Salad;
    m_MainDish = Holiday[n].MainDish;
    m_Comment = Holiday[n].Comment;
}

void C_UsrDialog::UpdateFromUser(int n)
{
    strcpy(Holiday[n].Day, m_Day);
    strcpy(Holiday[n].Fruit, m_Fruit);
    strcpy(Holiday[n].Dessert, m_Dessert);
    strcpy(Holiday[n].Salad, m_Salad);
    strcpy(Holiday[n].MainDish, m_MainDish);
    strcpy(Holiday[n].Comment, m_Comment);
}

```

c. Add initialization code to the `UsrDlg.cpp` file. When the dialog box opens up, initialize the member variables from the data in the first element of the global array. Delete all the member

variable initialization that has been automatically placed in the dialog's constructor by ClassWizard and replace it with the following line, given in bold and preceded by an arrow:

```
C_UsrDialog::C_UsrDialog(CWnd* pParent /*=NULL*/)
    : CDialog(C_UsrDialog::IDD, pParent)
{
    //{{AFX_DATA_INIT(C_UsrDialog)
    //}}AFX_DATA_INIT

    -->    UpdateFromBase(0);
}
```

Previous	Table of Contents	Next

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

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

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

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)**SEARCH**
ITKNOWLEDGE☒ Brief ☐ Full[Advanced](#)[Search](#)[Search Tips](#)**BROWSE**
BY TOPIC

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

Complete Message Handlers in the Dialog Class and the Main Window's Message Handler

The code in the dialog's handler functions, which are in the `UsrDialog.cpp` file, and the main window's message handler need to be completed, as follows:

1. Insert the following code, shown in bold and preceded by an arrow:

```
void CUsrDialog::OnNext()  
{  
-->    if (nDay >= 2) return;  
-->    UpdateFromBase(++nDay);  
-->    UpdateData(FALSE);  
}
```

```
void CUsrDialog::OnPrevious()  
{  
-->    if (nDay <= 0 ) return;  
-->    UpdateFromBase(-nDay);  
-->    UpdateData(FALSE);  
}
```

```
void C_UsrDialog::OnUpdate()  
{  
-->    if ( nDay < 0 ) return;
```

```

-->     if ( nDay > 2 ) return;
-->     UpdateData(TRUE);
-->     UpdateFromUser(nDay);
}

```

2. Complete the main window's message handler with the following lines of code:

```

        void C_MainFrame::OnOpenDialog()
        {
-->         CUsrDialog userDialog;
-->         userDialog.DoModal();
        }

```

Compile the Program

Now build and run your program.

1. Select “OpenDialog” from the menu, and the dialog box shown on Figure 7-13 appears with all the initial lists in the list box and the combo boxes. Note that the lists are sorted in alphabetical order.
2. Build a database of three items (an example entry of which is shown in Figure 7-13) and test its persistence by closing the dialog box, opening the dialog box again, and then cycling through your database.



Figure 7-13: The Executing Dialog Box

The UsrInput Program Listing

Listing 7-1 presents the source code for the UsrInput program. The source code includes the code generated by ClassWizard, as well as the code that you provide. In order to focus attention on the operative code, much of the tool-inserted code that is nonoperative (comments) is not shown. The listing is followed by a discussion of relevant points to be made about this code.

Listing 7-1: Source Code for the UsrInput Program

```

-----
                                stdafx files
-----
//
//  stdafx.h include file for standard system include files
//
#include <afxwin.h>

//
//  stdafx.cpp          source file for standard includes

```



```

//
#include "stdafx.h"

-----
                                application class
-----
//
//  MyApp.h
//
class C_MyApp : public CWinApp
{
    public:
        BOOL InitInstance();
};

//
//  MyApp.cpp  define the class behaviors for the class
//
#include "stdafx.h"
#include "MyApp.h"
#include "mainfrm.h"

BOOL C_MyApp::InitInstance()
{
    m_pMainWnd = new C_MainFrame;
    m_pMainWnd->ShowWindow(m_nCmdShow);
    SetDialogBkColor();
    return TRUE;
}

C_MyApp  theApp;  // the object that runs the program

-----
                                mainframe class
-----
//
//  mainfrm.h  interface of the C_MainFrame class
//
class C_MainFrame : public CFrameWnd
{
    public:
        C_MainFrame();

    private:

```

```

        //{{AFX_MSG(C_MainFrame)
        afx_msg void OnOpenDialog();
        //}}AFX_MSG

    DECLARE_MESSAGE_MAP()
};

//
//  mainfrm.cpp      implementation of the C_MainFrame class
//
#include "stdafx.h"
#include "mainfrm.h"
#include "resource.h"
#include "UsrDialog.h"

BEGIN_MESSAGE_MAP(C_MainFrame, CFrameWnd)
    //{{AFX_MSG_MAP(C_MainFrame)
    ON_COMMAND(ID_OPENDIALOG, OnOpenDialog)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

C_MainFrame::C_MainFrame()
{
    Create(NULL, "User Input Controls", WS_OVERLAPPEDWINDOW,
        rectDefault, NULL, MAKEINTRESOURCE(IDR_MENU1));
}

void C_MainFrame::OnOpenDialog()
{
    C_UsrDialog userDialog;
    userDialog.DoModal();
}

-----
                                dialog class
-----

//
//  UsrDialog.h : header file
//

#include "resource.h"

```

```

class C_UsrDialog : public CDialog
{
public:
    C_UsrDialog(CWnd* pParent = NULL);    // standard constructor
    void UpdateFromBase(int n);
    void UpdateFromUser(int n);

// Dialog Data
   //{{AFX_DATA(C_UsrDialog)
    enum { IDD = IDD_USER_INPUT };
    CString    m_MainDish;
    CString    m_Dessert;
    CString    m_Salad;
    CString    m_Fruit;
    CString    m_Day;
    CString    m_Comment;
    //}}AFX_DATA

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(C_UsrDialog)
protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:

    // Generated message map functions
   //{{AFX_MSG(C_UsrDialog)
    virtual BOOL OnInitDialog();
    afx_msg void OnNext();
    afx_msg void OnPrevious();
    afx_msg void OnUpdate();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

//
// UsrDialg.cpp : implementation file
//
#include "stdafx.h"

```

```

#include "UsrDialog.h"

int nDay;
struct{
    char Day[25];
    char Fruit[25];
    char Dessert[25];
    char Salad[25];
    char MainDish[25];
    char Comment[100];
} Holiday[] = { " ", " ", " ", " ", " ", " ", " ", " ", " ", " ",
                " / ", " / ", " / ", " / ", " / ", " / ", " / ",
                " ", " ", " ", " ", " ", " ", " ", " "};

C_UsrDialog::C_UsrDialog(CWnd* pParent /*=NULL*/ )
: CDialog(C_UsrDialog::IDD, pParent)
{
   //{{AFX_DATA_INIT(C_UsrDialog)
    //}}AFX_DATA_INIT

    UpdateFromBase(0);
}

void C_UsrDialog::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
   //{{AFX_DATA_MAP(C_UsrDialog)
    DDX_CBString(pDX, IDC_DROPDOWN_CBOX, m_MainDish);
    DDX_CBString(pDX, IDC_DROPLIST_CBOX, m_Dessert);
    DDX_LBString(pDX, IDC_LIST1, m_Salad);
    DDX_CBString(pDX, IDC_SIMPLE_CBOX, m_Fruit);
    DDX_Text(pDX, IDC_SINGLELINE, m_Day);
    DDX_Text(pDX, IDC_MULTILINE, m_Comment);
    DDV_MaxChars(pDX, m_Comment, 100);
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(C_UsrDialog, CDialog)
   //{{AFX_MSG_MAP(C_UsrDialog)
    ON_BN_CLICKED(IDC_NEXT, OnNext)
    ON_BN_CLICKED(IDC_PREVIOUS, OnPrevious)
    ON_BN_CLICKED(IDC_UPDATE, OnUpdate)

```

```
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

```
BOOL C_UsrDialog::OnInitDialog()
{
    CListBox* pLB = (CListBox*)GetDlgItem(IDC_LIST1);
    pLB->AddString("Carrots");
    pLB->AddString("Celery");
    pLB->AddString("Lettuce");
    pLB->AddString("Tomato");
    pLB->AddString("Cucumber");
    pLB->AddString("Potato");

    CComboBox* pCBxDropDown = (CComboBox*)GetDlgItem(IDC_DROPDOWN_CBOX);
    pCBxDropDown->AddString("Salmon");
    pCBxDropDown->AddString("Steak");
    pCBxDropDown->AddString("Prawns");
    pCBxDropDown->AddString("Turkey");

    return CDialog::OnInitDialog();
}

void C_UsrDialog::OnNext()
{
    if (nDay >= 2) return;
    UpdateFromBase(++nDay);
    UpdateData(FALSE);
}

void C_UsrDialog::OnPrevious()
{
    if (nDay <= 0) return;
    UpdateFromBase(--nDay);
    UpdateData(FALSE);
}

void C_UsrDialog::OnUpdate()
{
    if (nDay < 0) return;
    if (nDay > 2) return;
    UpdateData(TRUE);
    UpdateFromUser(nDay);
}
```

```

void C_UsrDialog::UpdateFromBase(int n)
{
    m_Day = Holiday[n].Day;
    m_Fruit = Holiday[n].Fruit;
    m_Dessert = Holiday[n].Dessert;
    m_Salad = Holiday[n].Salad;
    m_MainDish = Holiday[n].MainDish;
    m_Comment = Holiday[n].Comment;
}

void C_UsrDialog::UpdateFromUser(int n)
{
    strcpy(Holiday[n].Day,m_Day);
    strcpy(Holiday[n].Fruit,m_Fruit);
    strcpy(Holiday[n].Dessert,m_Dessert);
    strcpy(Holiday[n].Salad,m_Salad);
    strcpy(Holiday[n].MainDish,m_MainDish);
    strcpy(Holiday[n].Comment,m_Comment);
}

```

```

-----
                                resource files
-----

//
//  resource.h
//
#define IDR_MENU1                101
#define IDD_USER_INPUT          102
#define IDC_NEXT                1000
#define IDC_PREVIOUS            1001
#define IDC_UPDATE              1002
#define IDC_SINGLELINE          1003
#define IDC_MULTILINE           1004
#define IDC_LIST1               1005
#define IDC_DROPDOWN_CBOX       1006
#define IDC_SIMPLE_CBOX         1007
#define IDC_DROPLIST_CBOX       1008
#define ID_OPENDIALOG           40001

//
//  script1.rc

```

```

//
#include "resource.h"
#include "afxres.h"

IDR_MENU1 MENU DISCARDABLE
BEGIN
    MENUITEM "&OpenDialog",                ID_OPENDIALOG
END

IDD_USER_INPUT DIALOG DISCARDABLE  20, 20, 200, 150
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Dialog-User Input Controls"
FONT 8, "MS Sans Serif"
BEGIN
    LTEXT                "Single Line Edit Box:", IDC_STATIC, 2, 3, 65, 8, NOT WS_GROUP
    EDITTEXT             IDC_SINGLELINE, 1, 12, 62, 13
    LTEXT                "Simple ComboBox:", IDC_STATIC, 4, 30, 62, 8, NOT WS_GROUP
    COMBOBOX             IDC_SIMPLE_CBOX, 5, 40, 48, 42, CBS_SIMPLE | CBS_SORT |
                        WS_VSCROLL | WS_TABSTOP
    LTEXT                "DropList ComboBox:", IDC_STATIC, 1, 87, 65, 8, NOT WS_GROUP
    COMBOBOX             IDC_DROPLIST_CBOX, 1, 99, 86, 63, CBS_DROPDOWNLIST |
WS_VSCROLL | WS_TABSTOP
    LTEXT                "Listbox:", IDC_STATIC, 73, 3, 30, 8, NOT WS_GROUP
    LISTBOX             IDC_LIST1, 73, 12, 48, 40, NOT LBS_NOTIFY | LBS_SORT |
                        WS_VSCROLL | WS_TABSTOP
    LTEXT                "DropDown ComboBox:", IDC_STATIC, 63, 58, 82, 8, NOT WS_GROUP
    COMBOBOX            IDC_DROPDOWN_CBOX, 74, 68, 48, 30, CBS_DROPDOWN | CBS_SORT |
                        WS_VSCROLL | WS_TABSTOP
    LTEXT                "Multiline Edit Box:", IDC_STATIC, 126, 92, 60, 8, NOT
                        WS_GROUP
    EDITTEXT            IDC_MULTILINE, 106, 102, 93, 47, ES_MULTILINE | WS_VSCROLL |
                        WS_HSCROLL | ES_WANTRETURN
    DEFPUSHBUTTON        "OK", IDOK, 148, 3, 50, 14
    PUSHBUTTON           "Cancel", IDCANCEL, 148, 20, 50, 14
    PUSHBUTTON           "Next", IDC_NEXT, 148, 37, 50, 14, NOT WS_TABSTOP
    PUSHBUTTON           "Previous", IDC_PREVIOUS, 148, 54, 50, 14, NOT WS_TABSTOP
    PUSHBUTTON           "Update", IDC_UPDATE, 148, 71, 50, 14, NOT WS_TABSTOP
END

IDD_USER_INPUT DLGINIT
BEGIN
    IDC_SIMPLE_CBOX, 0x403, 7, 0

```

```

0x724f, 0x6e61, 0x6567, "\000"
    IDC_SIMPLE_CBOX, 0x403, 7, 0
0x6142, 0x616e, 0x616e, "\000"
    IDC_SIMPLE_CBOX, 0x403, 5, 0
0x6550, 0x7261, "\000"
    IDC_SIMPLE_CBOX, 0x403, 6, 0
0x7041, 0x6c70, 0x0065,
    IDC_SIMPLE_CBOX, 0x403, 6, 0
0x6550, 0x6361, 0x0068,
    IDC_DROPLIST_CBOX, 0x403, 21, 0
0x7453, 0x6172, 0x6277, 0x7265, 0x7972, 0x5320, 0x6f68, 0x7472, 0x6163,
0x656b, "\000"
    IDC_DROPLIST_CBOX, 0x403, 20, 0
0x6843, 0x636f, 0x6c6f, 0x7461, 0x2065, 0x6349, 0x2065, 0x7243, 0x6165,
0x006d,
    IDC_DROPLIST_CBOX, 0x403, 16, 0
0x6156, 0x696e, 0x6c6c, 0x2061, 0x7550, 0x6464, 0x6e69, 0x0067,
    IDC_DROPLIST_CBOX, 0x403, 12, 0
0x6143, 0x7272, 0x746f, 0x4320, 0x6b61, 0x0065,
    IDC_DROPLIST_CBOX, 0x403, 12, 0
0x7550, 0x706d, 0x696b, 0x206e, 0x6950, 0x0065,
0
END
-----

```

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Discussion of the UsrInputDialog

In the dialog class, ClassWizard is keeping track of the member variables in a “data list.” In the dialog class header file, we find no-op statements, and ClassWizard has placed all the member variable declarations between these no-op statements, which are:

```
//{{AFX_DATA(C_UsrDialog)
//}}AFX_DATA
```

Similar no-op statements are used by ClassWizard in the dialog class implementation (.cpp) file. These similar no-op statements are found in the dialog class constructor; ClassWizard has placed an initialization of each of the member variables between these no-op statments, which are:

```
//{{AFX_DATA_INIT(C_UsrDialog)
//}}AFX_DATA_INIT
```

In this example, we removed the ClassWizard member variable initialization and replaced it with our own initialization code in the constructor. If you need something different than the initialization that ClassWizard supplies, you will need to: (1) replace the code in the constructor with your own code or (2) assign values to the member variables later in the code so that the initial values get overwritten.

Note: The DoDataExchange() functions are placed by ClassWizard between the following “data map” no-op statements:

```
//{{AFX_DATA_MAP(C_UsrDialog)
//}}AFX_DATA_MAP
```

ClassWizard also provides a “virtual list” for all the virtual functions that it has added. The no-op statements that it uses for its “virtual map” are:

```
//{{AFX_VIRTUAL(C_UsrDialog)
//}}AFX_VIRTUAL
```

For our example, the resource script file, Script1.rc, has an additional script file, which is:

```
IDD_USER_INPUT DLGINIT
BEGIN
    .....
    (statements, mostly in octal)
    .....
END
```

This file was added when you entered the initialization lists for the controls IDC_SIMPLE_CBOX and IDC_DROPLIST_CBOX. The initialization strings are written in this file in octal. (Earlier versions of the compiler tool write this file slightly differently.)

Data Transfer

Like all other window objects, there are two dialog objects: the dialog window object, and the dialog C++ class object. The dialog window is the object which appears on the screen, and the dialog C++ class is the object in your code. Data is transferred between the dialog window and the corresponding dialog C++ class member variables by means of CDialog’s virtual function DoDataExchange(); however, you do not call this function in your code. DoDataExchange() is called as a result of a call to the UpdateData() function. When UpdateData() is called, it is passed a Boolean parameter that indicates the direction of the data transfer. UpdateData(TRUE) initiates a transfer of data from the dialog window controls to the associated dialog C++ class member variables. This retrieves information from the dialog window. Conversely, a parameter of FALSE would transfer data from the member variables to the dialog window controls. This updates the dialog controls with data from the dialog’s C++ class member variables. UpdateData() performs this transfer by passing the work on to the DoDataExchange() functions.

The DoDataExchange() function is the centralized location for data transfer in your dialog class. Within this function, data transfer is performed for each of the dialog controls individually. In order to support the bidirectional nature of data transfer promised by the UpdateData() function, DoDataExchange() also supports a bidirectional transfer mechanism. As in UpdateData(), DoDataExchange() is alerted to the data direction through a single parameter. However, unlike UpdateData(), DoDataExchange()’s parameter is much more complicated than a simple Boolean value. DoDataExchange() requires much more information, so its parameter is a pointer to a whole new class of object: CDataExchange.

CDataExchange is a class that contains information concerning a specific data

transfer. It holds a flag that indicates the data transfer direction, and a pointer to the parent window of the controls. Also, it encapsulates functionality that facilitates the data transfer process. The member functions defined for CDataExchange concern themselves with dialog control preparation. This preparation is a bunch of “housekeeping” busywork that each control must be subjected to in order for it to behave properly within the system.

Within the DoDataExchange() function, the CDataExchange object is passed only to the functions that perform the actual data transfers—the DDX functions. Although the DoDataExchange() function is the centralized data transfer function, the DDX functions are built with specific data types in mind. Looking at the DoDataExchange() functions from our previous two dialog examples illustrates this point.

In the buttons dialog example presented in Chapter Six, we look at its DoDataExchange() function with all comment lines removed:

```
void CBtnDialog::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    DDX_Control(pDX, IDC_BLUE, m_blue);
}
```

In the user input dialog example presented in this chapter, we look at its DoDataExchange() function with all comment lines removed:

```
void CUsrDialog::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    DDX_CBString(pDX, IDC_DROPDOWN_CBOX, m_MainDish);
    DDX_CBString(pDX, IDC_DROPLIST_CBOX, m_Dessert);
    DDX_LBString(pDX, IDC_LIST1, m_Salad);
    DDX_CBString(pDX, IDC_SIMPLE_CBOX, m_Fruit);
    DDX_Text(pDX, IDC_SINGLELINE, m_Day);
    DDX_Text(pDX, IDC_MULTILINE, m_Comment);
    DDV_MaxChars(pDX, m_Comment, 1000);
}
```

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes
(Publisher: Wordware Publishing, Inc.)
Author(s): Shirley Wodtke
ISBN: 1556225121
Publication Date: 01/01/97



Search this book:

Do Data Exchange Functions

From the preceding code, you can see that the DDX routines are used for both data transfer (DDX_CBString(), DDX_Text(), etc.) which are called *Value functions*, and for making the connection between a member object and its control (DDX_Control()), which are called *Control functions*. A listing of all the DDX functions is given in Table 7-4.

Table 7-4: DoDataExchange Functions

Function	Meaning
DDX_Control()	Gets a pointer to the control.
DDX_Radio()	Transfers radio button data.
DDX_Check()	Transfers check box data.
DDX_Text()	Transfers edit control data; works for many data types, including CString, int, long, float, double, etc.
DDX_LBString()	Transfers the list box's selected string.
DDX_CBString()	Transfers the combo box's selected string.
DDX_LBIndex()	Transfers the index of the list box's selection.
DDX_CBIndex()	Transfers the index of the combo box's selection.
DDX_LBStringExact()	Transfers the list box's exact selected string.
DDX_CBStringExact()	Transfers the combo box's exact selected string.

DDX_Text() represents many functions that allow transfer of various types of values to and from an edit control. When this function is overloaded, all these functions have the same name, DDX_Text(); they are distinguished by their argument types.

The difference between DDX_LBString() and DDX_LBStringExact() depends

on the direction of data transfer. If the data is coming from the controls, there is no difference between these two functions. When we are transferring information to the controls, they are different, and the difference stems from the nature of list box and combo box controls. Each of these controls has two means of finding a particular string in its list. One method is searching for a prefix of a string. This finds the first string that has the prefix. The other method is an exact-match method. When using this method, only exact matches of the string are found. The `DDX_LBStringExact()` and `DDX_CBStringExact()` functions select strings in the combo or list boxes only when an exact match is made. The `DDX_LBString()` and `DDX_CBString()` functions select the first element in the list that has the appropriate prefix.

DDX functions may be installed manually or via compiler tools, which will also let you create variables that work with the DDX functions. Once you have defined a DDX Value variable for a control, the framework can, under certain circumstances, automatically initialize and update the variable. This chapter presented an example and discussed when automatic initialization and updating occur and when you need to do it yourself. (Chapter Eight presents further examples.) Table 7-5 gives the type of DDX Value variables available:

Table 7-5: DDX Value Variable Types

<i>Control</i>	<i>Variable Type</i>
Edit box	CString, int, UINT, long, DWORD, float, double
Normal check box	BOOL
Three-state check box	int
Radio button (first in group)	int
Nonsorted list box	CString, int
Drop list combo box	CString, int
All other list and combo box types	Cstring

The following notes apply to using DDX Value variables:

- Possible values for three-state check boxes are 0 (off), 1 (on), and 2 (indeterminate).
- Values for a group of radio buttons range from 0 for the first button in the group to $n-1$ for a group with n buttons. A value of -1 indicates that no buttons are selected.
- When you are using a group of check boxes or radio buttons with a DDX variable, make sure the Auto property is set for all the controls in the group.
- Set the Group property for the first radio button in a group, and make sure all the other radio buttons immediately follow the first button in the tab order.
- To use an integer value with a combo box or list box, make sure the Sort property is turned off on the control's Style property page.

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes
(Publisher: Wordware Publishing, Inc.)
Author(s): Shirley Wodtke
ISBN: 1556225121
Publication Date: 01/01/97



Search this book:

Dialog Data Validation (DDV) Functions

The DDV functions check the range of input data. Table 7-6 lists the DDV functions:

Table 7-6: DDV Functions

Function	Meaning
DDV_MaxChars()	Verifies that the length of CString is less than or equal to the Max value.
DDV_MinMaxByte()	Verifies that the BYTE value is between Min and Max.
DDV_MinMaxDouble()	Verifies that the double value is between Min and Max.
DDV_MinMaxDWord()	Verifies that the DWORD value is between Min and Max.
DDV_MinMaxFloat()	Verifies that the float value is between Min and Max.
DDV_MinMaxInt()	Verifies that the int value is between Min and Max.
DDV_MinMaxLong()	Verifies that the long value is between Min and Max.
DDV_MinMaxUInt()	Verifies that the UINT value is between Min and Max.

Most of the DDV functions are for numeric range checking and check to see if a given number is within a specified range of values. The function DDV_MaxChars() is the exception, and it is used to check strings for specified length requirements. At run time, if the value entered by the user exceeds the range (or string length) specified during development, the framework automatically displays a message box asking the user to re-enter the value. The validation of DDX variables takes place all at once when the user chooses

“OK” to accept the entries in the dialog box.

DDV functions may be installed manually or via compiler tools. The compiler tool will install a DDV function in the code immediately after the DDX function corresponding to the same variable.

CString Features

The CString class is a significant improvement to the language. It is much easier to use a CString than the usual array of chars. The CString class has many useful operators and member functions, for which an overview is given in this section. It has the very important feature of dynamic memory allocation, which means that the programmer no longer has to be concerned about the size of the string; CString automatically handles it for you.

In the example in this chapter, structs are used which have char arrays for the global database. The member variables in the dialog class are of type CString. To transfer data from CString member variables to a char array, the function strcpy() must be used. However, to copy from a char array to a CString member variable, the assignment operator can be used.

The power of the CString class will become obvious as the operators and functions that it provides are reviewed. Constructing a CString object has been made very easy. There are various ways of constructing and initializing a CString object. The easiest way is to use the default constructor and do an assignment:

```
CString MyString = "This is easy!";
```

Another easy way is to use the string within the constructor:

```
CString MyString("Another easy string!");
```

The argument provided to the constructor could also be a reference to a CString or a pointer to a char array. A second parameter can be provided to the constructor, which must be an int.

Note: If you use the second int parameter, your string will be constrained to that int length.

Once you have your CString objects, access is gained to a large set of operators and member functions. You need only to review these to understand the power and ease of the CString class. Table 7-7 lists the most frequently used operators and member functions. There are numerous other functions for extraction, searching, etc., which are not listed. Even glancing at Table 7-7 will convey the importance of CString.

Table 7-7: CString Operators and Functions

<i>Operators/Functions</i>	<i>Meaning</i>
Operators	
==, !=, <=, >=, <, >	Conducts a case-sensitive comparison.

=	Assigns the new value (a char or a CString) to the CString.
+	Concatenates a CString with a char or a CString and returns a CString.
+=	Joins characters (CString or char) to the end of the CString.

Comparison Functions

Compare	Tests to see if the parameter is equal to the calling object's string.
CompareNoCase	Same as Compare() except case is not considered.
Collate	Same as Compare(), but is locale-specific.

Conversion Functions

MakeUpper	Converts all characters to uppercase.
MakeLower	Converts all characters to lowercase.
MakeReverse	Reverses the order of the string.
AnsiToOem	Converts all characters from ANSI to OEM.
OemToAnsi	Converts all characters from OEM to ANSI.

Get and Set Functions

GetLength	Returns the length of the string.
GetAt	Returns the character at the specified position.
SetAt	Sets the character at the specified position.
IsEmpty	Returns TRUE if the string is empty.
Empty	Empties the string.

Previous	Table of Contents	Next

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

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

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

Keyword

☒ Brief
 ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Summary

The common control classes of list box, combo box, and edit controls are encapsulated in the MFC classes CListBox, CComboBox, and CEdit, respectively. These controls are very complex. They have many styles, their MFC classes contain numerous member functions, and they send and receive numerous messages. All of the control's styles, which can be a style associated with that specific control class or a window style, are found on the control's property page. The property page of the list box, combo box, and edit controls contain numerous entries. The selections made on the control's property page will cause the Dialog Editor to enter the appropriate style ID in the control's resource script line of the DIALOG resource script. As before, the DIALOG resource script is encapsulated with a dialog class derived from CDialog.

The list box, combo box, and edit controls accept keyboard input from the user. The DoDataExchange() function is called to move the data out of the controls and into member variables of the dialog class. Each control is assigned a member variable that will be used to store the data that is moved out of the control. The DDX has a function for each control that associates that control with its member variable; when called, this function does the work of transferring the data from the control to the member variable or vice versa. DoDataExchange() has a function for each control. When the DoDataExchange() function is called, it calls the respective function for every control. The way to activate the DoDataExchange() function is to call the function CWnd::UpdateData(BOOL). If the function UpdateData(TRUE) is called, then the DDX initiates a transfer of data from the controls to the dialog class's member variables. If the function UpdateData(FALSE) is called, then the DDX transfers the data from the dialog class's member variables to the controls.

The DDX mechanism has a companion DoDateValidation (DDV) mechanism,

which can be used to automatically check the validity of the data entered by the user into a control. Data limits are associated with each control using a DDV function. The DDV functions are located within the DoDataExchange() function; when called, the DDV functions will check the data entered in the controls to see if the limits are exceeded. If so, it will automatically inform the user.

The C++ object of the dialog class is instantiated on the stack. When the stack closes, the values in the dialog class's member variables are lost. If this is not what you want, you must provide for more persistent storage of the data and move the data from the member variables to the persistent storage before the stack closes. Persistent storage can be global variables or member variables of the mainframe class, since the mainframe class persists throughout the lifetime of the application.

The most frequently used combo box styles are the simple combo box, the dropdown combo box, and the droplist combo box. Each of these has a list of potential selections associated with it and a box for the selection. The selection box is an edit box, which will accept keyboard input from the user, for the simple combo box and the dropdown combo box. For the droplist combo box, the selection box is a static control which merely displays the selection.

Combo boxes and list boxes have initial lists associated with them. These are the lists that initially appear when the control is opened. The initial list for a list box can be initialized in the OnInitDialog() function in the dialog's class or loaded from a string table in the OnInitDialog() function. The initial list for a combo box can be entered on the combo box's property page, or it can be initialized in the OnInitDialog() function either by code that adds the strings or by code that loads the strings from a string table. Strings can be entered on a string table, which is a resource contained in the .rc file, and later loaded from the string table resource.

String tables are used to centralize all of the strings used by an application. String tables are used to internationalize an application; to change to another language, one simply changes to a string table in the appropriate language.

The CString class is much easier to use than the usual array of chars. It has many useful operators and member functions. It has the very important feature of dynamic memory allocation, so the programmer no longer has to be concerned about the size of the string; CString automatically handles the size of the string.

Previous	Table of Contents	Next

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

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

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

Keyword

☒ Brief ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

Chapter 8

Communication Between Parent and Child for Modal and Modeless Dialogs

This chapter focuses on ways that the communication process between a parent and its child window is carried out. There are multiple ways of setting up the communication process, but the focus of this chapter is on which way works best in different situations. We will look at a modal dialog box and a modeless dialog box, focusing on the communication process between the parent, which is the main frame window, and its child, which is the dialog window. We also further examine the use of the DDX mechanism. In these examples, the MFC class CCmdUI is introduced.

Both examples discussed in this chapter involve a dialog box and, of course, a dialog class. Creating the dialog box's resource script and the dialog class were covered in the last two chapters, so these subjects are not discussed in this chapter. The code of the dialog box class in both of the examples discussed in this chapter has been created using compiler tools, but the creation process is not discussed in this chapter. Instead, the discussion is focused on the communication process between the main window's class and the dialog class.

The use of the MFC class CCmdUI is introduced in this chapter. For this new topic, a discussion is included which covers the ClassWizard procedures which implement the class CCmdUI. The CCmdUI class, of course, can also be implemented manually.

The Modal Dialog Example

Many applications allow the user to select from the menu or from within a dialog box for the same choice. In this modal dialog example, named ModalCom, we examine just such an application. The user is allowed to choose (via a popup menu) the color of the main window's client area. As shown in Figure 8-1, when the application first opens, the main window's client area has been colored yellow; this is noted by the checkmark next to the yellow selection in the popup menu. It also has an initial string, "Checking Menu Items and Radio Buttons," showing in the main window's title bar. Clicking the menu item "OpenDialog" causes a modal dialog box to open as shown in Figure 8-2. When the dialog box opens, the selected radio button is "Yellow," which is the same as shown on the main window's menu, and the text in the edit box is the same text showing in the main window's caption bar. Choices made in the main window have been passed to the dialog class and the information is presented in the dialog box when it opens.



Figure 8-1: Executing ModalCom Program



Figure 8-2: Opened Dialog Box Reflecting Main Window Selections

The user can change the selection for the main window's client area color and also enter a new main window title into the dialog box, as shown in Figure 8-3. When a modal dialog box is open, the code in the parent window that "owns" the dialog box, which is in class C_MainFrame, cannot be accessed until the dialog box is closed and the dialog window is no longer visible. Thus, when the dialog box is closed with the "OK" button, the radio button selection and the string entered in the edit box will be transmitted to the main window's code and the effect will be immediate, as shown in Figure 8-4. Note that the menu item checked as the current selection is "Red" which is the radio button selection that was made from the dialog box. Also note that the main window's title has been replaced by the text that was entered in the dialog's edit box. The communication process between the main window and the dialog box to achieve this is discussed following the code for this example.



Figure 8-3: Making New Selections in the Dialog Box



Figure 8-4: Main Window Reflecting Selections from Dialog

In the upcoming code, there is a data member of class `C_MainFrame`, named `m_nClientColor`, that holds an integer value used for setting the color of the main window's client area. This example demonstrates using member variables of the mainframe class for persistent data storage. The member variables of the mainframe class persist for the duration of the application. In the `C_ModalDlg` class, which is the class that encapsulates the dialog box, there is a data member named `m_nColor` which holds the information on which radio button was selected by the user. The information in `m_nColor` will be assigned to `m_nClientColor` and vice versa.

Note: The terms “member variable” and “data member” are used interchangeably.

As shown in Figure 8-3, the user selects from a group of radio buttons and has the choice of either “Red,” “Green,” or “Yellow.” As we learned in Chapter Seven, the DDX mechanism assigns a variable to the first radio button in the group; this is our variable `m_nColor`. When the DDX updates the value of the variable when `UpdateData(TRUE)` is called, it will assign 0 to the variable if the first radio button is selected, 1 to the variable if the second radio button is selected, and 2 to the variable if the third radio button is selected. In effect, this variable, `m_nColor`, describes the radio button group with just one variable. When the dialog box closes, we pass the value in `m_nColor` to `m_nClientColor` of the main window, and we use the knowledge of how DDX assigns values to `m_nColor` in designing the main window's code. The value of `m_nClientColor` is used to implement the class `CCmdUI` to check and uncheck the items in the popup menu.

Previous	Table of Contents	Next

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

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

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

Go!

Keyword

☐ Brief ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes
(Publisher: Wordware Publishing, Inc.)
Author(s): Shirley Wodtke
ISBN: 1556225121
Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

Go!

[Previous](#)

[Table of Contents](#)

[Next](#)

The MFC Class CCmdUI

Many times you want to change the appearance of a menu item to match the internal state of the application, as we do in this example. To do this, we can use the class CCmdUI which has member functions that operate on menu items. The CCmdUI member functions are listed in Table 8-1.

Table 8-1: CCmdUI Member Functions

Function	Meaning
SetCheck(0/1)	Sets the check state of the menu item.
Enable(BOOL)	Enables/disables the menu item.
SetText(LPCSTR)	Sets the text for the menu item.
SetRadio(0/1)	Like SetCheck(), but checks using a dot. Operates on items acting as part of a radio group.

In addition to the ON_COMMAND message map entry that we have been using thus far, there is a second message map entry, ON_UPDATE_COMMAND_UI, that can be entered for each command message. You can enter these by hand or you can use ClassWizard to create a function prototype, a message map entry, and a “stubbed-out” handler function for each such update handler function that you want. Figure 8-5 illustrates the use of ClassWizard to establish the update handler functions for this example.



Figure 8-5: Using ClassWizard for Update Handler Functions

The update handler functions are implemented like a second type of command handler function, each one having its own separate handler function. Taking as an example the OnUpdateYellow() handler function, we find that Class Wizard does the following things:

Enters the following function prototype into the mainfrm.h file:

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

Enters the following message map entry in mainfrm.cpp:

```
ON_UPDATE_COMMAND_UI(ID_YELLOW, OnUpdateYellow)
```

Stubs-out the following handler function in mainfrm.cpp:

```
void C_MainFrame::OnUpdateYellow(CCmdUI* pCmdUI)
```

```
{
}
```

The CCmdUI messages apply only to popup menus, not to top-level menu items that are permanently displayed. This message is sent for the popup menu item right before the popup menu becomes visible. The application must test on the state of some internal variable to determine how the menu item is to be drawn when it is displayed. In this example, the internal variable that serves this purpose is m_nClientColor. In each of the update handler functions, we test on this variable and determine how that menu item should be drawn. For example, the completed update handler function for OnUpdateYellow() is as follows:

```
void C_MainFrame::OnUpdateYellow(CCmdUI* pCmdUI)
{
    if (m_nClientColor == 2)
        pCmdUI->SetCheck(1);
    else
        pCmdUI->SetCheck(0);
}
```

The ModalCom Program Listing

Listing 8-1 gives the source code for the ModalCom program. Much of the tool-generated code is not shown in order to focus on the operative code. The listing is followed by a discussion of points that we have not yet covered about this code.

Listing 8-1: Source Code for the ModalCom Program

```
-----
                                stdafx files
-----
//
//  stdafx.h include file for standard system include files
//
#include <afxwin.h>

//
//  stdafx.cpp          source file for standard includes
//
#include "stdafx.h"

-----
                                application class
-----
//
//  MyApp.h
//
class C_MyApp : public CWinApp
{
public:
    BOOL InitInstance();
};

//
//  MyApp.cpp    define the class behaviors for the class
//
#include "stdafx.h"
#include "MyApp.h"
#include "mainfrm.h"

BOOL C_MyApp::InitInstance()
{
    m_pMainWnd = new C_MainFrame;
    m_pMainWnd->ShowWindow(m_nCmdShow);
}
```



```

        SetDialogBkColor();
        return TRUE;
    }

    C MyApp  theApp;  //  the object that runs the program

-----
                                mainframe class
-----
//
//  mainfrm.h    interface of the C_MainFrame class
//
class C_MainFrame : public CFrameWnd
{
public:
    C_MainFrame();
private:
    int m_nClientColor;
    CString m_sMainWindowTitle;

    //{AFX_MSG(C_MainFrame)
    afx_msg void OnPaint();
    afx_msg void OnGreen();
    afx_msg void OnUpdateGreen(CCmdUI* pCmdUI);
    afx_msg void OnOpenDialog();
    afx_msg void OnRed();
    afx_msg void OnUpdateRed(CCmdUI* pCmdUI);
    afx_msg void OnYellow();
    afx_msg void OnUpdateYellow(CCmdUI* pCmdUI);
    //}}AFX_MSG

    DECLARE_MESSAGE_MAP()
};

//
//  mainfrm.cpp    implementation of the C_MainFrame class
//
#include "stdafx.h"
#include "mainfrm.h"
#include "resource.h"
#include "ModalDlg.h"

BEGIN_MESSAGE_MAP(C_MainFrame, CFrameWnd)
    //{AFX_MSG_MAP(C_MainFrame)
    ON_WM_PAINT()
    ON_COMMAND(ID_GREEN, OnGreen)
    ON_UPDATE_COMMAND_UI(ID_GREEN, OnUpdateGreen)
    ON_COMMAND(ID_OPENDIALOG, OnOpenDialog)
    ON_COMMAND(ID_RED, OnRed)
    ON_UPDATE_COMMAND_UI(ID_RED, OnUpdateRed)
    ON_COMMAND(ID_YELLOW, OnYellow)
    ON_UPDATE_COMMAND_UI(ID_YELLOW, OnUpdateYellow)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

C_MainFrame::C_MainFrame()
{
    Create(NULL, "Checking Menu Items and Radio Buttons",
        WS_OVERLAPPEDWINDOW, rectDefault, NULL,
        MAKEINTRESOURCE(IDR_MENU1));
    m_sMainWindowTitle = "Checking Menu Items and Radio Buttons";
    m_nClientColor = 2;
}

```

```

}

void C_MainFrame::OnPaint()
{
    CPaintDC dc(this); // device context for painting
    CRect r;
    GetClientRect(&r);
    COLORREF brushColor;

    switch (m_nClientColor)
    {
        case 0:
            brushColor = RGB(255, 0, 0);
            break;
        case 1:
            brushColor = RGB(0, 255, 0);
            break;
        case 2:
            brushColor = RGB(255, 255, 0);
    }

    CBrush brush(brushColor);
    dc.FillRect(&r, &brush);
}

void C_MainFrame::OnGreen()
{
    m_nClientColor = 1;
    Invalidate();
}

void C_MainFrame::OnUpdateGreen(CCmdUI* pCmdUI)
{
    if (m_nClientColor == 1)
        pCmdUI->SetCheck(1);
    else
        pCmdUI->SetCheck(0);
}

void C_MainFrame::OnOpenDialog()
{
    C_ModalDlg myDialog;           // instantiates C++ myDialog object

    myDialog.m_sTitle = m_sMainWindowTitle; // Xfer to myDialog variables
    myDialog.m_nColor = m_nClientColor;      // before disp. dialog window

    int ReturnValue = myDialog.DoModal();    // captures return value
                                           // and destroys window

    if (ReturnValue == IDCANCEL) return;
    m_sMainWindowTitle = myDialog.m_sTitle; // captures dialog's variables
    m_nClientColor = myDialog.m_nColor;      // before C++ object is destroyed

    SetWindowText(m_sMainWindowTitle);      // resets window title
    Invalidate();                           // repaints
}                                           // C++ object, myDialog, is destroyed

void C_MainFrame::OnRed()
{
    m_nClientColor = 0;
    Invalidate();
}

```

```

void C_MainFrame::OnUpdateRed(CCmdUI* pCmdUI)
{
    if (m_nClientColor == 0)
        pCmdUI->SetCheck(1);
    else
        pCmdUI->SetCheck(0);
}

void C_MainFrame::OnYellow()
{
    m_nClientColor = 2;
    Invalidate();
}

void C_MainFrame::OnUpdateYellow(CCmdUI* pCmdUI)
{
    if (m_nClientColor == 2)
        pCmdUI->SetCheck(1);
    else
        pCmdUI->SetCheck(0);
}

```

```

-----
                                dialog class
-----
//
// ModalDlg.h : header file
//
#include "resource.h"

class C_ModalDlg : public CDialog
{
public:
    C_ModalDlg(CWnd* pParent = NULL);    // standard constructor
// Dialog Data
   //{{AFX_DATA(C_ModalDlg)
    enum { IDD = IDD_DIALOG1 };
    int         m_nColor;
    CString     m_sTitle;
    //}}AFX_DATA

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(C_ModalDlg)
protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL
};

//
// ModalDlg.cpp : implementation file
//
#include "stdafx.h"
#include "ModalDlg.h"

C_ModalDlg::C_ModalDlg(CWnd* pParent/*=NULL*/)
    : CDialog(C_ModalDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(C_ModalDlg)
    m_nColor = -1;

```

```

    //}}AFX_DATA_INIT
}

void C_ModalDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(C_ModalDlg)
    DDX_Radio(pDX, IDC_RED, m_nColor);
    DDX_Text(pDX, IDC_TITLE, m_sTitle);
    //}}AFX_DATA_MAP
}

```

resource files

```

//
// resource.h
//
#define IDR_MENU1 101
#define IDD_DIALOG1 102
#define IDC_RED 1000
#define IDC_GREEN 1001
#define IDC_YELLOW 1002
#define IDC_TITLE 1003
#define ID_RED 40001
#define ID_GREEN 40002
#define ID_YELLOW 40003
#define ID_OPENDIALOG 40004

//
// script1.rc
//
#include "resource.h"
#include "afxres.h"

DR_MENU1 MENU DISCARDABLE
BEGIN
    POPUP "&Color"
    BEGIN
        MENUITEM "&Red" ID_RED
        MENUITEM "&Green", ID_GREEN
        MENUITEM "&Yellow", ID_YELLOW
    END
    MENUITEM "&OpenDialog", ID_OPENDIALOG
END

IDD_DIALOG1 DIALOG DISCARDABLE 90, 40, 225, 150
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Modal Dialog"
FONT 8, "MS Sans Serif"
BEGIN
    GROUPBOX "Client Area Color", IDC_STATIC, 10, 6, 63, 68
    CONTROL "Red", IDC_RED, "Button", BS_AUTORADIOBUTTON |
        WS_GROUP, 20, 22, 35, 10
    CONTROL "Green", IDC_GREEN, "Button", BS_AUTORADIOBUTTON,
        20, 40, 35, 10
    CONTROL "Yellow", IDC_YELLOW, "Button", BS_AUTORADIOBUTTON,
        20, 58, 35, 10
    EDITTEXT IDC_TITLE, 73, 99, 144, 13, ES_AUTOHSCROLL | WS_GROUP
    LTEXT "Main Window Title:", IDC_STATIC, 2, 100, 66, 8, NOT WS_GROUP
    PUSHBUTTON "Cancel", IDCANCEL, 130, 23, 50, 14
    DEFPUSHBUTTON "OK", IDOK, 130, 6, 50, 14

```

END

Previous	Table of Contents	Next

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

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

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

Keyword

☒ Brief
 ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Data Transfer

In these paragraphs, we primarily discuss the data transfer occurring in the preceding code. Additionally, a few other points are made about this code. We first discuss what is happening in the main window class, `C_MainFrame`, and then turn our attention to the dialog class, `C_ModalDlg`.

In MFC programming the C++ object of the class that represents the window and the window itself must be distinguished from each other, because they are two distinct entities. The C++ object that represents the window is constructed. Then at some point, which does not need to occur immediately, the window based on that C++ object is made visible on the screen. The window can be destroyed, and its C++ object can still remain in memory. Understanding this distinction is important when you are doing data transfer between windows. In many cases, such as this example, what we are really doing is data transfer between the C++ objects that represent the windows, so we need to have an awareness of when these C++ objects exist and when they do not.

The Main Window Class

In the Listing 8-1 code, the data transfer between the parent and the child window is occurring in the main window's function, `OnOpenDialog()`. We will examine the code in the `OnOpenDialog()` function in detail here.

Modal dialog boxes are always created on the stack. When the `OnOpenDialog()` function is entered, the first thing that happens is that we construct a C++ object of the dialog class (representing the dialog window) on the stack with the following line of code:

```
C_ModalDialog myDialog;
```

Bear in mind that the dialog window is not yet visible. All that we have done is

to construct its C++ object. Since the C++ object of the dialog window is available in memory at this point, we can transfer data into its member variables from the member variables of the main window. With the following two lines of code, data is transferred between the main window and the dialog window's C++ member variables:

```
myDialog.m_sTitle = m_sMainWindowTitle;  
myDialog.m_nColor = m_nClientColor;
```

This assigns the values that were resident in the main window into the corresponding data members of the dialog's C++ object. The data transfer is now complete and we are ready to display the dialog window by calling the function DoModal(). The function DoModal()'s default behavior is to first call the function UpdateData(FALSE) and then to display the dialog window. When the function UpdateData(FALSE) is called, the values of the dialog's data members (which are now the same as the main window's values for the corresponding data members) are transferred into the controls. When the dialog window is displayed, the transferred values appear in the controls. This all occurs when the following expression is invoked:

```
myDialog.DoModal()
```

The statement in our code that invokes this call is more complex. The return value of the function DoModal() is the numeric value of the identifier IDOK if the user clicked the "OK" button or used the Enter key and is IDCANCEL if the user clicked the "Cancel" button. So we capture the value returned from the dialog window in a variable, named ReturnValue, in the following line of code:

```
int ReturnValue = myDialog.DoModal();
```

At this juncture, the dialog window has been displayed and subsequently closed by the user. The dialog window is destroyed, but its C++ object still remains on the stack. We need to know if the user cancelled the dialog window; if he did, we are not interested in doing data transfer from the dialog window. The next line of code tests to see if the user cancelled and, if he did, we leave the function (which destroys everything on the stack):

```
if (ReturnValue == IDCANCEL) return;
```

If there was no cancel, then the user probably made a new choice. When the user chooses the "OK" button, the inherited function OnOK() is invoked. The OnOK() function calls the function UpdateData(TRUE), which transfers the user's choices from the controls to the dialog's data members; subsequent to this transfer, it closes the dialog window. The dialog object's data members, containing the values of the user's choices, are still available on the stack. These values need to be transferred to our main window to implement them. The C++ object myDialog is still available, so we now do the transfer of these values into the main window's data members with the following two lines of code:

```
m_sMainWindowTitle = myDialog.m_sTitle;  
m_nClientColor = myDialog.m_nColor;
```

Since the values selected by the user may have changed, this is the juncture in our program when they must be implemented. We do so by invoking the CWnd function SetWindowText() which will set the caption text of the invoking window or C++ object. Also, we need to repaint the client area, so we cause a WM_PAINT message to be generated. These things are done with the following two lines of code:

```
SetWindowText(m_sMainWindowTitle);  
Invalidate();
```

At this juncture, we have completed everything and we are ready to close the function. At the close of the OnOpenDialog() function, the C++ object myDialog is destroyed.

Note: One final note about the C_MainFrame class's code that bears mention is the OnPaint() function. Note that the way we change the client area color is to simply find the rectangular area of the client area and then fill that rectangular area with the chosen color. Do not confuse this with changing the background color by registering a new window class, which, of course, was not done in this example.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

The Dialog Class

The data transfer that occurs in the dialog class is done by the inherited function OnOK(). Its default behavior is to call the function UpdateData(TRUE) and then to destroy the dialog window. The OnOK() function does not need to appear in your program; it is still there (inherited) and will be triggered by the "OK" button or the Enter key. Likewise the OnCancel() function does not need to appear in your program (and it does not in this example); it is still inherited and will be triggered by the "Cancel" button, which sends the IDCANCEL identifier.

The dialog class, when generated by ClassWizard, will have some data member initialization code automatically placed into the dialog's constructor. You may not want this code. If not, remove it. In this example, the automatic initialization code that we are given is the following:

```
//{{AFX_DATA_INIT(C_ModalDlg)
m_nColor = -1;
m_sTitle = _T("");
//}}AFX_DATA_INIT
```

Notice that the following line of code is removed:

```
m_nTitle = _T("");
```

This line of code is overwritten by values from the main window prior to the first time that the dialog box is opened. However, this line of code was written for a Win32 system and will only compile on a Win32 system. It will not compile on a Win3.1 system. Win3.1 ClassWizard writes the line as:

```
m_nTitle = "";
```

Note: The code generated by the compiler tools for a Win32 system will not necessarily compile on a Win3.1 system; items like this can prevent the source code from being backward compatible.

The remaining line of code in the data initialization section (written by Class Wizard) was left in because it is at least compatible with Win3.1, though it is ineffective because it too is immediately overwritten. The line of code is:

```
m_nColor = -1;
```

This line of code assigns a -1, which means that nothing will be checked in the radio button group, to the variable representing the radio group. ClassWizard writes this code which will initially assign these blanks to your data members. You may not want them; they may be immediately rendered ineffective. They may need to be deleted for any number of reasons.

Modeless Dialog Example

The dialog box resource script used for the modeless dialog box is the same as used in the previous modal dialog with only minimal, necessary changes. This allows you to focus on the new implementation required to make the same DIALOG resource script behave as a modeless dialog box.

Via a popup menu, the user is allowed to choose the color of the main window's client area. As shown in Figure 8-6, when the application first opens up, the color of the main window's client area has been initialized with the color green, which is noted by the checkmark next to the green selection, and it has the same initial string as used in the previous modal dialog example. Clicking the menu item "OpenDialog" causes a modeless dialog box to open as shown in Figure 8-7. When the dialog box is opened, the values showing have been coordinated with the current values from the main window. An "OK" button is not appropriate for a modeless dialog box, so it is removed. Clicking the "Close" button destroys the dialog window.



Figure 8-6: Executing Modeless Program



Figure 8-7: Opened Dialog Reflects Main Window Selections

The user can change the selection for the main window's client area color and also enter a new main window title into the dialog box, as shown in Figure 8-8. When a modeless dialog box is open, the code in the parent window can be accessed. The selections made in the dialog box can go into effect while the modeless dialog box remains open; the user can also return focus to the main

window and make changes from the menu while the modeless dialog box remains open. Choices made from the main window's menu while the dialog box is still open will be immediately reflected in the settings shown in the open dialog box, as shown in Figure 8-9.



Figure 8-8: Dialog Selections Reflect Immediately While Dialog Stays Open



Figure 8-9: Selection From Menu Reflects Immediately in Open Dialog

Discussion before and after the program listing covers the following issues brought up by this example:

- Instantiating an object of the dialog class as a modeless dialog
- Creating the two-way communication between the main window and the dialog
- Transferring the main window's data member values to the dialog class's data members
- Coordinating selections with the dialog box while it is still open
- Implementing data transfer from the dialog box to the main window to reflect the dialog box selections

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

Modeless Dialog Creation

Modeless dialog box creation is a two-stage process. First, the C++ object is instantiated on the heap. Then, in the second step, the dialog object's Create() function is called. The C++ object is instantiated on the heap using the following line of code, which is located in the C_MainFrame constructor:

```
m_pDlg = new C_Modeless(this);
```

It is, of course, on the heap since it needs to be around for the duration of the application.

Note: We now have a destructor, ~C_MainFrame, that deletes it. The pointer to the C++ dialog object, m_pDlg, is a data member of C_MainFrame and will be used for communication and data transfer to the dialog.

When the C++ object is instantiated on the heap, its constructor is called, which is:

```
C_Modeless::C_Modeless(CWnd* pParent /*=NULL*/ )
: CDialog(C_Modeless::IDD, pParent)
{
    m_nColor = -1;
    m_pMainWindow = pParent;
}
```

In the dialog's constructor, its data members are initialized, and a statement is included which assigns the pointer of the dialog's parent, pParent, to the dialog's data member m_pMainWindow. In this example, we know that the parent of the dialog class is the main window, so it is safe to make this assignment. Otherwise, we would have obtained the main window pointer using the expression: AfxGetApp()->m_pMainWnd. The pointer to the main window, m_pMainWindow, is a data member of the dialog class and is used for the dialog to communicate with the main window.

The second stage of the process occurs when we want to open (make visible) the dialog. This code is in the OnOpenDialog() function:

```
void C_MainFrame::OnOpenDialog()
{
```

```

        if (m_bDialogOpen == TRUE) return;

        m_pDlg->m_nColor = m_nClientColor;    // sets dialog's variables
        m_pDlg->m_sTitle = m_sMainWindowTitle;
        m_pDlg->Create();
    }

```

In this code, we first test to see if a dialog box is already open using our variable designed for this job, `m_bDialogOpen`. If a dialog box is not already open, the values in the main window's data members are transferred into the dialog's data members since the C++ object is available on the heap. Then, a call is made to the dialog's `Create()` function, which needs to be written manually, and is:

```

BOOL C_Modeless::Create()
{
    BOOL ReturnValue = CDialog::Create(C_Modeless::IDD,
                                      m_pMainWindow);

    UpdateData(FALSE);
    ShowWindow(SW_SHOWNA);
    return ReturnValue;
}

```

Within this function, `CDialog::Create()` is called. `CDialog::Create()` creates the dialog window and takes two parameters: the first is the dialog template (resource script identifier), and the second is the parent window object to which the dialog object belongs. The second parameter can be omitted; if it is, the pointer is `NULL`, and the dialog window will be created with its parent window set to the main application window. (In this example, we could have omitted the second parameter.) At this point, the window is created, but is not visible, and we call `UpdateData(FALSE)` to transfer the data from the dialog data members to the controls. Then the call to `ShowWindow()` is made.

If you do not use `WS_VISIBLE` as the dialog style in the resource script (which we have not done), then you must call `ShowWindow()` to make the window visible.

Note: `ShowWindow` can be called only once per application with `CWinApp::m_nCmdShow` (that is, `ShowWindow(m_nCmdShow)`), so we must call it with another value. We use the value `SW_SHOWNA`, which displays the window in its current state with the window that is currently active remaining active.

Up to this point, everything that you have learned in the two-class (application and mainframe) examples is exactly the same when applied in four-class examples (application, mainframe, document, and view). However, creating modeless dialog boxes is the exception. The creation process will vary slightly in the four classes, because you must obtain a pointer to the view class, not the mainframe class. In Chapter Ten, creating a modeless dialog box in the four-class setting is discussed.

User Messages

User messages are employed extensively in the following example. They are needed in this example to send messages from the dialog box to the main window in order to execute the main window's code to implement a selection made in the dialog box. (User messages were introduced and explained in Chapter Five.) We have the `C_MainFrame` functions `OnDlgRed()`, `OnDlgGreen()`, `OnDlgYellow()`, and `OnDlgText()`, which implement the respective choices made in the dialog box. We also have the function `OnDlgClose()` which sets our `BOOLEAN` flag, `m_bDialogOpen`, to `FALSE` when the dialog box is closed. This function is exercised when a `WM_DLG_CLOSE` user message arrives. The `WM_DLG_CLOSE` user message is sent when the dialog window is destroyed. Any method of closing the dialog box—the system menu, the close box, or the push

button labeled “Close” will cause the destruction of the dialog window, which will send a WM_DESTROY message. When a WM_DESTROY message is received, the dialog box will send the WM_DLG_CLOSE message to the main window.

In this example, user messages to the dialog window also need to be sent. You cannot set a control unless you have a valid pointer to the control, which means that the dialog window must be open. A message sent to a dialog window will not “arrive” unless the dialog window is open. The dialog class’s functions—OnMainRed(), OnMainGreen(), and OnMainYellow()—will cause the appropriate radio button to be checked when a selection is made from the main window’s menu. When a menu item is selected, a user message is sent to the dialog window. When the user message arrives, it is then mapped to the appropriate handler function which will check the appropriate radio button using the function SendDlgItemMessage().

As discussed in Chapter Five, you must manually enter all the code associated with user messages—the prototypes in the .h files, the message map entries ON_MESSAGE(<ID>,<MessageHandler>), and the handler functions are all done by hand.

Note: In this example all of the user messages are defined in the modeless.h file. The modeless.h file is visible to both mainfrm.cpp and to modeless.cpp. Do not use the file resource.h for defining the user messages if you use compiler tools. The compiler tools will rewrite the resource.h file every time you add to your resources and will overwrite the old file, essentially deleting all of the entries that you made by hand.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Modeless Program Listing

Listing 8-2 gives the source code for the Modeless program. As before, much of the tool-generated code is not shown in the listing. The listing is followed by a discussion of its features.

Listing 8-2: Source Code for the Modeless Program

```

-----
                                     stdafx files
-----

                                     same code as in Listing 8-1

-----
                                     application class
-----

                                     same code as in Listing 8-1

-----
                                     mainframe class
-----
//
//  mainfrm.h   interface of the C_MainFrame class
//
#include "Modeless.h"

class C_MainFrame : public CFrameWnd
{
public:
    C_MainFrame();
    ~C_MainFrame();

private:
    C_Modeless* m_pDlg;
    int m_nClientColor;
    CString m_sMainWindowTitle;
    BOOL m_bDialogOpen;

   //{{AFX_MSG(C_MainFrame)
    afx_msg void OnPaint();
  
```

```

afx_msg void OnGreen();
afx_msg void OnUpdateGreen(CCmdUI* pCmdUI);
afx_msg void OnOpenDialog();
afx_msg void OnRed();
afx_msg void OnUpdateRed(CCmdUI* pCmdUI);
afx_msg void OnYellow();
afx_msg void OnUpdateYellow(CCmdUI* pCmdUI);
//}}AFX_MSG

LONG OnDlgRed(UINT wParam, LONG lParam);
LONG OnDlgGreen(UINT wParam, LONG lParam);
LONG OnDlgYellow(UINT wParam, LONG lParam);
LONG OnDlgText(UINT wParam, LONG lParam);
LONG OnDlgClose(UINT wParam, LONG lParam);

DECLARE_MESSAGE_MAP()

};

//
// mainfrm.cpp      implementation of the C_MainFrame class
//
#include "stdafx.h"
#include "mainfrm.h"
#include "resource.h"

BEGIN_MESSAGE_MAP(C_MainFrame, CFrameWnd)
//{{AFX_MSG_MAP(C_MainFrame)
ON_WM_PAINT()
ON_COMMAND(ID_GREEN, OnGreen)
ON_UPDATE_COMMAND_UI(ID_GREEN, OnUpdateGreen)
ON_COMMAND(ID_OPENDIALOG, OnOpenDialog)
ON_COMMAND(ID_RED, OnRed)
ON_UPDATE_COMMAND_UI(ID_RED, OnUpdateRed)
ON_COMMAND(ID_YELLOW, OnYellow)
ON_UPDATE_COMMAND_UI(ID_YELLOW, OnUpdateYellow)
ON_MESSAGE(WM_FROMDLG_RED, OnDlgRed)
ON_MESSAGE(WM_FROMDLG_GREEN, OnDlgGreen)
ON_MESSAGE(WM_FROMDLG_YELLOW, OnDlgYellow)
ON_MESSAGE(WM_FROMDLG_TEXT, OnDlgText)
ON_MESSAGE(WM_DLG_CLOSE, OnDlgClose)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

C_MainFrame::C_MainFrame()
{
    Create(NULL, "Checking Menu Items and Radio Buttons",
        WS_OVERLAPPEDWINDOW, rectDefault, NULL,
        MAKEINTRESOURCE(IDR_MENU1));
    m_pDlg = new C_Modeless(this);
    m_sMainWindowTitle = "Checking Menu Items and Radio Buttons";
    m_nClientColor = 1; //initializes client bkg to green
    m_bDialogOpen = FALSE;
}

C_MainFrame::~C_MainFrame()
{
    delete m_pDlg;
}

void C_MainFrame::OnPaint()

```



```

{
    CPaintDC dc(this); // device context for painting
    CRect r;
    GetClientRect(&r);
    COLORREF brushColor;
    switch(m_nClientColor)
    {
        case 0:
            brushColor = RGB(255, 0, 0);
            break;
        case 1:
            brushColor = RGB(0, 255, 0);
            break;
        case 2:
            brushColor = RGB(255, 255, 0);
    }
    CBrush brush(brushColor);
    dc.FillRect(&r, &brush);
}

void C_MainFrame::OnGreen()
{
    m_nClientColor = 1;
    m_pDlg->PostMessage(WM_TODLG_GREEN);
    Invalidate();
}

void C_MainFrame::OnUpdateGreen(CCmdUI* pCmdUI)
{
    if (m_nClientColor == 1)
        pCmdUI->SetCheck(1);
    else
        pCmdUI->SetCheck(0);
}

void C_MainFrame::OnOpenDialog()
{
    if (m_bDialogOpen == TRUE) return;
    m_pDlg->m_nColor = m_nClientColor; // sets dialog's variables
    m_pDlg->m_sTitle = m_sMainWindowTitle;
    m_pDlg->Create();
}

void C_MainFrame::OnRed()
{
    m_nClientColor = 0;
    m_pDlg->PostMessage(WM_TODLG_RED);
    Invalidate();
}

void C_MainFrame::OnUpdateRed(CCmdUI* pCmdUI)
{
    if (m_nClientColor == 0)
        pCmdUI->SetCheck(1);
    else
        pCmdUI->SetCheck(0);
}

void C_MainFrame::OnYellow()
{

```

```

        m_nClientColor = 2;
        m_pDlg->PostMessage(WM_TODLG_YELLOW);
        Invalidate();
    }

void C_MainFrame::OnUpdateYellow(CCmdUI* pCmdUI)
{
    if (m_nClientColor == 2)
        pCmdUI->SetCheck(1);
    else
        pCmdUI->SetCheck(0);
}

LONG C_MainFrame::OnDlgRed(UINT wParam, LONG lParam)
{
    OnRed();
    return 1;
}

LONG C_MainFrame::OnDlgGreen(UINT wParam, LONG lParam)
{
    OnGreen();
    return 1;
}

LONG C_MainFrame::OnDlgYellow(UINT wParam, LONG lParam)
{
    OnYellow();
    return 1;
}

LONG C_MainFrame::OnDlgText(UINT wParam, LONG lParam)
{
    m_sMainWindowTitle = m_pDlg->m_sTitle;
    SetWindowText(m_sMainWindowTitle);
    return 1;
}

LONG C_MainFrame::OnDlgClose(UINT wParam, LONG lParam)
{
    m_bDialogOpen = FALSE;
    return 1;
}

-----
                                dialog class
-----
//
// Modeless.h : header file
//
#include "resource.h"

#define WM_TODLG_RED                WM_USER
#define WM_TODLG_GREEN              WM_USER + 1
#define WM_TODLG_YELLOW             WM_USER + 2
#define WM_FROMDLG_RED              WM_USER + 3
#define WM_FROMDLG_GREEN            WM_USER + 4
#define WM_FROMDLG_YELLOW           WM_USER + 5
#define WM_FROMDLG_TEXT             WM_USER + 6
#define WM_DLG_CLOSE                WM_USER + 7

```

```

class C_Modeless : public CDialog
{
public:
    C_Modeless(CWnd* pParent = NULL);    // standard constructor
    BOOL Create();
    CWnd* m_pMainWindow;

// Dialog Data
   //{{AFX_DATA(C_Modeless)
    enum { IDD = IDD_DIALOG1 };
    int         m_nColor;
    CString     m_sTitle;
    //}}AFX_DATA

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(C_Modeless)
protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:
    // Generated message map functions
    //{{AFX_MSG(C_Modeless)
    afx_msg void OnGreen();
    afx_msg void OnRed();
    afx_msg void OnKillfocusTitle();
    afx_msg void OnYellow();
    afx_msg void OnClose();
    virtual void OnOK();
    afx_msg void OnDestroy();
    //}}AFX_MSG
    LONG OnMainRed(UINT wParam, LONG lParam);
    LONG OnMainGreen(UINT wParam, LONG lParam);
    LONG OnMainYellow(UINT wParam, LONG lParam);
    DECLARE_MESSAGE_MAP()
};

//
// Modeless.cpp : implementation file
//

#include "stdafx.h"
#include "Modeless.h"

C_Modeless::C_Modeless(CWnd* pParent/*=NULL*/ )
    : CDialog(C_Modeless::IDD, pParent)
{
    //{{AFX_DATA_INIT(C_Modeless)
    m_nColor = -1;
    //}}AFX_DATA_INIT

    m_pMainWindow = pParent;
}

BOOL C_Modeless::Create()
{
    BOOL ReturnValue = CDialog::Create(C_Modeless::IDD, m_pMainWindow);
    UpdateData(FALSE);
    ShowWindow(SW_SHOWNA);
}

```

```

        return ReturnValue;
    }

void C_Modeless::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
   //{{AFX_DATA_MAP(C_Modeless)
    DDX_Radio(pDX, IDC_RED, m_nColor);
    DDX_Text(pDX, IDC_TITLE, m_sTitle);
   //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(C_Modeless, CDialog)
   //{{AFX_MSG_MAP(C_Modeless)
    ON_BN_CLICKED(IDC_GREEN, OnGreen)
    ON_BN_CLICKED(IDC_RED, OnRed)
    ON_EN_KILLFOCUS(IDC_TITLE, OnKillfocusTitle)
    ON_BN_CLICKED(IDC_YELLOW, OnYellow)
    ON_BN_CLICKED(IDCANCEL, OnClose)
    ON_WM_DESTROY()
    ON_MESSAGE(WM_TODLG_RED, OnMainRed)
    ON_MESSAGE(WM_TODLG_GREEN, OnMainGreen)
    ON_MESSAGE(WM_TODLG_YELLOW, OnMainYellow)
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()

void C_Modeless::OnGreen()
{
    m_pMainWindow->PostMessage(WM_FROMDLG_GREEN);
}

void C_Modeless::OnRed()
{
    m_pMainWindow->PostMessage(WM_FROMDLG_RED);
}

void C_Modeless::OnKillfocusTitle()
{
    UpdateData(TRUE);
    m_pMainWindow->PostMessage(WM_FROMDLG_TEXT);
}

void C_Modeless::OnYellow()
{
    m_pMainWindow->PostMessage(WM_FROMDLG_YELLOW);
}

void C_Modeless::OnClose()
{
    DestroyWindow();
}

void C_Modeless::OnOK()
{
    // do not call the base class CDialog::OnOK()
    // providing this OnOK function with no functionality
    // disables the <enter> key, which would otherwise
    // close the modeless dialog box
}

```

```

void C_Modeless::OnDestroy()
{
    CDialog::OnDestroy();
    m_pMainWindow->PostMessage(WM_DLG_CLOSE);
}

LONG C_Modeless::OnMainRed(UINT wParam, LONG lParam)
{
    SendDlgItemMessage(IDC_RED, BM_SETCHECK, 1, 0L);
    SendDlgItemMessage(IDC_GREEN, BM_SETCHECK, 0, 0L);
    SendDlgItemMessage(IDC_YELLOW, BM_SETCHECK, 0, 0L);
    Invalidate();
    return 1;
}

LONG C_Modeless::OnMainGreen(UINT wParam, LONG lParam)
{
    SendDlgItemMessage(IDC_RED, BM_SETCHECK, 0, 0L);
    SendDlgItemMessage(IDC_GREEN, BM_SETCHECK, 1, 0L);
    SendDlgItemMessage(IDC_YELLOW, BM_SETCHECK, 0, 0L);
    Invalidate();
    return 1;
}

LONG C_Modeless::OnMainYellow(UINT wParam, LONG lParam)
{
    SendDlgItemMessage(IDC_RED, BM_SETCHECK, 0, 0L);
    SendDlgItemMessage(IDC_GREEN, BM_SETCHECK, 0, 0L);
    SendDlgItemMessage(IDC_YELLOW, BM_SETCHECK, 1, 0L);
    Invalidate();
    return 1;
}

```

resource files

```

//
// resource.h
//
//
// same code as in Listing 8.1
//
// script1.rc
//
// same code as in Listing 8.1, except for the
// dialog resource script, which is given here in its entirety

```

```

IDD_DIALOG1 DIALOG DISCARDABLE 90, 40, 225, 150
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Modeless Dialog"
FONT 8, "MS Sans Serif"
BEGIN
    GROUPBOX        "Client Area Color", IDC_STATIC, 10, 6, 63, 68
    CONTROL          "Red", IDC_RED, "Button", BS_AUTORADIOBUTTON |
WS_GROUP, 20, 22, 35, 10
    CONTROL          "Green", IDC_GREEN, "Button", BS_AUTORADIOBUTTON,
20, 40, 35, 10
    CONTROL          "Yellow", IDC_YELLOW, "Button", BS_AUTORADIOBUTTON,
20, 58, 35, 10
    EDITTEXT         IDC_TITLE, 73, 99, 144, 13, ES_AUTOHSCROLL | WS_GROUP
    LTEXT            "Main Window Title:", IDC_STATIC, 2, 100, 66, 8, NOT WS_GROUP

```

PUSHBUTTON "Close", IDCANCEL, 130, 23, 50, 14
END

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Data Updating

When a modeless dialog box is used, the functions `UpdateData(TRUE)` and `UpdateData(FALSE)` must be called. You do not get the benefit of default behavior, as we did in the modal dialog box. In this modeless dialog example, `UpdateData(FALSE)` is called after the `Create()` function has been called and just before the dialog window is made visible.

The function `UpdateData(TRUE)` is needed only when we wish to take the text entered into the edit control and transfer it to the main window. The text is transferred when the edit box sends an `EN_KILLFOCUS` notification message. In the dialog class's function `OnKillfocusTitle()`, the function `UpdateData(TRUE)` is called, which transfers the data from the control to the dialog data member `m_sTitle`. Then a user message is sent to the main window. The main window gets the user message and goes to the function `OnDlgText()`, in which the variable `m_sMainWindowTitle` is updated to the current value of `m_sTitle`; then the text in the caption bar is set.

Special Virtual Functions

The `CDialog` class has two virtual functions that are important in every dialog box. They are `OnOK()` and `OnCancel()`. These functions are inherited by every class derived from `CDialog`.

The `OnOK()` function is exercised whenever either of two things happens: the Enter key is used or the user clicks on the default push button labeled "OK." The default push button sends `IDOK`. In this example, the default push button labeled "OK" is removed from the dialog resource script, so `IDOK` can never be sent. But this does not stop the user from hitting the Enter key. Remember that the dialog class has inherited the `OnOK()` function, so we always have it whether we want it or not. The way to disable this function is to override it in your code and then to not call the base class `CDialog::OnOK()`, so that the

base class functionality cannot be exercised.

An interesting point to be made is that if you are using ClassWizard for your message maps and map either of the special identifiers IDOK or IDCANCEL, ClassWizard enters these virtual functions as a message map function in the header file. In this example, ClassWizard was used to map the special identifier IDOK to OnOK() and then later the control was deleted from the dialog resource script. The text of the “Cancel” button is changed to “Close” and its response function OnClose() gets mapped to it with the special identifier IDCANCEL.

Summary

Transfer of data between a modal dialog box and the mainframe window can be accomplished by using member variables defined in the mainframe window class as persistent data storage. A modal dialog box is always instantiated on the stack. After the C++ dialog object has been instantiated on the stack, but before the dialog window is made visible, the data must be transferred from the mainframe’s member variables to the dialog’s member variables. The dialog is then made visible with the call DoModal(); the default behavior of DoModal() invokes the call UpdateData(FALSE), so the values in the dialog’s member variables are transferred into the controls. The user makes new choices in the dialog box. When the user presses OK, the default behavior invokes the call UpdateData(TRUE) and closes the dialog window. After the dialog window is closed, but before the stack closes, the data must be transferred from the dialog’s member variables to the mainframe’s member variables.

The class CCmdUI can be used to check submenu items. A CCmdUI message is sent to each item on the popup menu just before the popup menu is made visible. The application must test on the state of some internal variable to determine how the menu item is to be drawn (checked or unchecked) when it is displayed. ClassWizard can be used for inserting the CCmdUI message entries and response functions into the code. In addition to the ON_COMMAND message map entry, there is a second message map entry, ON_UPDATE_COMMAND_UI, that is entered for each command message. And there is also an update handler function.

Modeless dialog boxes are more complex to use. The application’s code can be accessed when a modeless dialog box is open. Choices made in the modeless dialog box can be invoked in the main window while the modeless dialog box stays open. Modeless dialog box creation is a two-stage process. First, the C++ object is instantiated on the heap. Then, in the second step, the dialog object’s Create() function is called. Just before the Create() function is called, the data transfer must be made from the mainframe’s member variables to the dialog’s member variables. Then, just before the dialog window is made visible, the call to UpdateData(FALSE) must be made which transfers the data from the dialog’s member variables into the controls. The modeless dialog becomes visible with the appropriate data showing in the controls. For modeless dialog boxes, the programmer must call UpdateData(FALSE) and UpdateData(TRUE) at the appropriate times; there is no default behavior to do

this for you. Whenever data is to be transferred into a control, the call to UpdateData(FALSE) must be inserted in the code manually. Whenever data is to be transferred out of a control, the call to UpdateData(TRUE) must be inserted in the code manually.

User messages are employed to communicate between the child dialog box and the parent. When a choice is made in the dialog box, it posts a message to the parent. When a choice is made from the mainframe's menu, a message is posted to the child dialog box.

Previous	Table of Contents	Next

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

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

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

Go!

Keyword

☒ Brief ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

Go!

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

Exercise

You now have the skills to add a modal dialog box to your checkers game. In Chapter Six, you learned how to design and use radio button controls. In Chapter Seven, you learned how to use controls that accept text input from the user.

Add a dialog box that has radio button groups and edit boxes to your checkers game. Open the dialog box with a menu choice. Make a radio button group for the choices of the checkerboard colors and another radio button group for the choices of the piece shapes. The radio button groups will allow the user a second way (in addition to the menu items) to make the choice of checkerboard color and piece shape.

Add edit controls to the dialog box. The edit controls will accept the first and last name of each player from the user. For example, the players may be Babe Ruth and Elvis Presley. When the dialog box closes, transfer this data to the mainframe class. Change the text in the caption bar to read “Babe Ruth vs Elvis Presley.” Also write the players’ initials on their checker pieces.

In this new checkers game, the user is now able to choose the color of the checkerboard and the shape of the pieces from submenu items of the main menu and the user can also make choices for checkerboard color and piece shape when the modal dialog box is open. Program your game so that when the dialog box opens up, the selections showing in the radio button groups are the most recent selections made from the menu items. Also, when your dialog box closes, the most recent selections made from the radio button groups are reflected as the checked menu items. See Appendix B, section “Chapter 8 Exercise,” for more explanation.

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

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

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

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

Keyword

☒ Brief ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

Chapter 9

The Document-View Architecture

Up to this point, we have been dealing with programs that have two classes: an application class derived from the MFC class `CWinApp` and a mainframe class derived from `CFrameWnd`. These two classes are still part of the *document-view architecture*, but these classes use additional functionality when they are integrated into the four-class, document-view structure. This chapter demonstrates how to use the additional functionality of the basic two classes that you have become accustomed to. Two new classes are added: the *view class* and the *document class*. You will learn how to use these two new classes. A four-class (application, mainframe, document, and view class) example is presented that customizes the features of its windows.

The Structure of the Four Classes

In this chapter two more classes are added to the basic structure that it takes to run a program. These are the document class, which is derived from the MFC class `CDocument`, and the view class, which is derived from the MFC class `CView` (or one of its derivatives). We turn our attention to learning the features of the four-class, document-view structure. For the remainder of this book, the structure of every program will contain four classes: the application class, the mainframe class, the document class, and the view class.

Every program has code that is common to any four-class program. As an example of common code recall that in the previous chapters, with the two-class structure that we have worked with, many times the application class could simply be copied from previous examples; the code was common and did not change from one example to the next.

Most compilers have introduced tools to help you set up the four-class, document-view structure. These tools generate all of the common code necessary to get a four-class application going. They also introduce optional code, such as functions that are usually overridden but do not always have to be overridden. These tools will automatically generate enough code (and usually more code than you need) to get a “generic” program going. You then need to introduce into this “generic” code structure all the code that is unique to your application. To do this, you need to understand which class your code goes into.

In this chapter, how the four class-structure operates is explained, which will give a basic understanding of what code goes where, when you are faced with adding your code to an already-generated “generic” application. The remaining chapters of this book will present example programs, and for each example, explain where the code is to be added to the “generic” tool-generated application. By the time you finish the remaining chapters, you will have a good understanding of where to add your code when you are working with the four-class, code-generating compiler tools.

The optional code introduced by compilers varies with the compiler (for example, Symantec differs from Microsoft), and there are also more minor variations in the versions (for example, Visual C++4.0 has some entries that are different from Visual C++ 1.5). Because of the fact that the tools give you optional code and there are variations on this optional code, this chapter presents a small example generated manually (but adhering to the style used by compiler tools) with only code that is essential to have for this example. (This helps you get used to the essential code before diving into the code generated by tools.)

In the two-class structure that we have looked at in previous chapters, the mainframe class contained all of the application’s screen “real estate” and the application class was a “background” class that provided other essential functions, as depicted in Figure 9-1. The four-class structure introduces the view class, which occupies all of the mainframe class’s client area “real estate.” The *view window* is a child window of the mainframe window. It has no border, no caption bar, and no visual elements. It is shrink-wrapped inside the mainframe client area such that it resizes itself as the mainframe window resizes (just like the fixed child window example in Chapter Five).

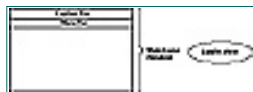


Figure 9-1: The Two-Class Mainframe-Application Structure

The four-class structure also introduces the document class which is “connected” to the view class and, like the application class, is a “background” class that has no allocation of screen “real estate.” This four class structure is depicted in Figure 9-2. As illustrated, the document class is central to the process of filing and retrieving documents. A document, as used here, means all the essential information of the application needed when you store the state of your application. When you start your application by loading a document from the file, it will have the appropriate settings for all the variables, such that your application appears on the screen exactly as you expected it to look. In

Chapter Ten, we create a four-class example in which we store documents to a file and restart the application from these stored files.

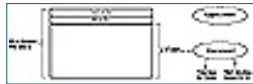


Figure 9-2: The Four-Class Document-View Structure

If a toolbar or status bar is present, the view window resizes itself and claims the remaining client area. This is shown in Figure 9-3, which depicts both a toolbar and status bar being present. The toolbar is a child window of the mainframe window and will normally be positioned just below the menu bar. The status bar is also a child window of the mainframe window and will normally be positioned along the bottom of the mainframe window. (Toolbars and status bars are addressed in Chapter Thirteen.)

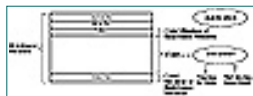


Figure 9-3: The Document-View with a Toolbar and Status Bar

Previous	Table of Contents	Next

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

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

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

Keyword

☒ Brief
 ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Message Routing

For the document-view architecture, the special WM_COMMAND message has a built-in routing mechanism. A WM_COMMAND message can go to any of the four classes. It is first routed to the view class. If a handler function for that WM_COMMAND message is found in the view class, it goes to that handler function, and the routing is complete because it does not go to any other classes. If no handler function is found in the view class, the WM_COMMAND message is next sent to the document class. If a response function is present, the routing ends. Otherwise, it proceeds to the mainframe class. If no handler function is found, it proceeds to the application class. In the document-view architecture, any of the four classes (application, mainframe, view, or document) can contain a message map for WM_COMMAND messages. All of the classes to which a WM_COMMAND message can be routed (or targeted) to are derived from the MFC class CCmdTarget.

The mainframe class receives more normal (not WM_COMMAND) WM_ messages than the view class does. For example, the mainframe can receive the WM_CLOSE message and will close. The view window cannot be closed; it does not receive the WM_CLOSE message. For the view class, you must make sure that it can receive the WM_ message before entering a handler function for that message. (The message-mapping compiler tools are a good guide for this.) The mainframe class also engages in other activities that the view class is not allowed to participate in such as: session management (e.g., enter idle, end session), menu management, and change management (e.g., font, palette, system color, and winini). The view class occupies the entire client area and is the class that, most of the time, will handle the keyboard and mouse input WM_ messages. For example, when you want to draw on the client area, the drawing functions will be located in the view class, and they will draw on the client area portion of the screen which, of course, is the view

window.

An Example Document-View Program

To illustrate the document-view architecture, a four-class example is manually constructed, so that the amount of code we need to examine is minimal. In this example, customizing the window is illustrated, and a discussion on how it is handled in the document-view, four-class architecture is presented.

Customized features of the window are shown, including: icon, menu, caption bar features, cursor, and background color. How to customize the window to have an owner device context and how to implement this feature are also shown. (In this example, the capabilities of the document class are not used—they are just there and not used. In Chapters Ten and Eleven, we explore the capabilities of the document class and examine the code for accessing the document’s capabilities.) The name of the program is “Custom,” and Figure 9-4 shows how it appears when it first opens up. A customized icon (you will recognize “smiley”) appears in the caption bar, the title is “Custom Window,” and we cannot minimize or maximize this window. The window has a yellow background and an UpArrow cursor. An ellipse is drawn in the client area when the window appears. The same example that was used in Chapter Three is used here to illustrate the operation of the owner device context, and we look at how to implement it in the four-class architecture.



Figure 9-4: Custom Program When Opened

The menu item choices are: “SetCaption,” which will change the text in the caption bar; “DrawRectangle,” which will draw a rectangle in the client area; and “QuitApp,” which will close the application. Figure 9-5 shows the “Custom” program after we have selected “SetCaption” and “DrawRectangle” from the menu.



Figure 9-5: Custom Program After Selecting Menu Items

The PreCreateWindow() Function

The function `PreCreateWindow()` is used in the four-class architecture to customize the window. The “window” actually consists of two windows, the mainframe and the view. As illustrated in Figure 9-2, the only visible portion of the mainframe is the caption bar and menu. When we wish to customize features showing on the caption bar, such as those that minimize or maximize the application, we will be using the `PreCreateWindow()` function of the `CFrameWnd` base class. When we wish to customize features that are clearly

in the purview of the view class, such as cursor and background color, and obtain an owner device context, we will use the PreCreateWindow() function of the CView base class. The CWnd::PreCreateWindow() function is a virtual function that is overridden by its derived classes CFrameWnd and CView, each of which adds its own functionality. The CWnd::PreCreateWindow() function has the following form:

```
virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
```

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97



Search this book:

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

We override this virtual function, provide our own inputs to the *CREATESTRUCT* structure, and then call the base class’s PreCreateWindow() function. The CREATESTRUCT structure defines the initialization parameters passed to the window procedure of an application.

Table 9-1: The *CREATESTRUCT* Structure and Its Members

CREATESTRUCTURE Members

lpCreateParams	Points to data to be used to create the window.
hInstance	Identifies the module-instance handle of the module that owns the new window.
hmenu	Identifies the menu to be used by the new window; if a child window, it contains the integer ID.
hwndParent	Identifies the window that owns the new window. This member is NULL if the new window is a top-level window.
cy	Specifies the height of the new window.
cx	Specifies the width of the new window.
y	Specifies the y-coordinate of the upper left corner of the new window. Coordinates are relative to the parent window if the new window is a child window; otherwise coordinates are relative to the screen origin.
x	Specifies the x-coordinate of the upper left corner of the new window. Coordinates are relative to the parent window if the new window is a child window; otherwise coordinates are relative to the screen origin.
style	Specifies the new window’s style.
lpszName	Points to a null-terminated string that specifies the new window’s name.
lpszClass	Points to a null-terminated string that specifies the new window’s Windows class name.
dwExStyle	Specifies the <i>extended style</i> for the new window.

```

typedef struct tagCREATESTRUCT {
    LPVOID    lpCreateParams;
    HANDLE    hInstance;
    HMENU     hMenu;
    HWND      hwndParent;
    int       cy;
    int       cx;
    int       y;
    int       x;
    LONG      style;
    LPCSTR    lpzName;
    LPCSTR    lpzClass;
    DWORD     dwExStyle;
} CREATESTRUCT;

```

Customizing the Mainframe Window

Features of the mainframe window are customized by overriding the `CFrameWnd::PreCreateWindow()` function and by assigning mainframe resources in the `.rc` file. We have used both of these methods in the Custom example.

Overriding the `CFrameWnd::PreCreateWindow()` Function

We override the `CFrameWnd::PreCreateWindow()` function to: set the window size to 200-pixels high and 300-pixels wide; set the x- and y- coordinates such that the window will appear in the middle of the screen; and set the window style such that the window will have a caption bar, a system menu, and will be resizable. It can not be minimized or maximized since these features were not included in the style specification. All this is done in the following overriding function:

```

BOOL C_MainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    cs.cy = 200;
    cs.cx = 300;
    cs.y = (::GetSystemMetrics(SM_CYSCREEN) - 200) / 2;
    cs.x = (::GetSystemMetrics(SM_CXSCREEN) - 300) / 2;

    cs.style = WS_CAPTION | WS_SYSMENU | WS_THICKFRAME;
    return CFrameWnd::PreCreateWindow(cs);
}

```

Mainframe Resources

We have used the resource script file for setting some of our mainframe window's features. These features are the icon, the menu, and the string which initially appears in the caption bar.

Any icon in the `.rc` file named `IDR_MAINFRAME` will be used as the icon of the mainframe window. The following line of code causes the smiley icon to be used as the mainframe's icon:

IDR_MAINFRAME ICON DISCARDABLE "res\\smiley.ico"

Any menu in the .rc file named IDR_MAINFRAME will be loaded as the mainframe window's menu. In the following line of code introducing our menu, we assign our menu as the mainframe's menu:

IDR_MAINFRAME MENU PRELOAD DISCARDABLE

In the string table, the string labeled IDR_MAINFRAME will be the string assigned to the mainframe's caption bar. We used the following string table entry:

IDR_MAINFRAME "Custom Window"

As an alternative, we could also have assigned the string "Custom Window," which we want to initially appear in the caption bar, to the CREATESTRUCT member lpszName in the PreCreateWindow() function.

Previous	Table of Contents	Next

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

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

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



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US



SEARCH

ITKNOWLEDGE

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

Go!

[Previous](#)

[Table of Contents](#)

[Next](#)

Customizing the View Window

Features of the view window are customized by overriding the `CView::PreCreateWindow()` function. In our example, we register a new window class that has a yellow background, an UpArrow cursor, and a class style that includes the `CS_OWNDC` style, which gives us a view window with its own device context. This is done with the following code:

```
BOOL C_CustomView::PreCreateWindow(CREATESTRUCT& cs)
{
    CBrush hYellowBrush( RGB( 255, 255, 0) );

    cs.lpszClass = AfxRegisterWndClass(
        CS_HREDRAW | CS_VREDRAW | CS_OWNDC,
        AfxGetApp()->LoadStandardCursor(IDC_UPARROW),
        (HBRUSH)hYellowBrush.GetSafeHandle());
    return CView::PreCreateWindow(cs);
}
```

In the upcoming example code, you will note that the drawing attributes of the device context, `pMyPen` and `pMyBrush`, are member variables of the `C_CustomView` class and they exist for the lifetime of the program. They are deleted by the destructor when the application is terminated and the view window and its attendant device context are destroyed. (As you remember, you do not delete pens and brushes while they are still selected into the device context.) The device context is customized in the `OnCreate()` function, which is the handler function for the `WM_CREATE` message generated when the view window is created. It is customized by the following code:

```
int C_CustomView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    CPaintDC dc(this);
    pMyPen = new CPen(PS_SOLID, 5, RGB(0, 0, 255)); // blue pen
    pMyBrush = new CBrush(RGB(128, 128, 0)); // olive brush
}
```



BROWSE

BY TOPIC

```

        dc.SelectObject(pMyPen);
        dc.SelectObject(pMyBrush);
        return 0;
    }

```

This code is entered only once during the window creation process. In this function we instantiate a device context object and customize its drawing attributes. The device context attributes are “permanently” set for the duration of this application. Thereafter, when we again instantiate a device context object, it will have the desired drawing attributes. We can use it as is, with no further modification.

The OnDraw() Function

In the upcoming code you will note that there is not an OnPaint() function or an entry in the message map for the WM_PAINT message. In the document view architecture, the OnPaint() function is replaced with the OnDraw() function; the framework will automatically map the WM_PAINT message to the OnDraw() function.

The CView::OnDraw(CDC* pDC) function is a more generalized function and is used for drawing to the screen, for printing, and for print preview. When drawing to the screen, the CPaintDC device context is used, but the framework takes care of obtaining the device context and will pass the pointer to the device context as the argument, pDC, of the OnDraw(CDC* pDC) function. If the framework is printing, then it will obtain a printer device context and pass its pointer as the argument to the OnDraw(CDC* pDC) function. It similarly supplies the appropriate device context for print preview. Its argument, pDC, is of class CDC*, so it can pass a pointer to any of the device contexts, which are all derived from CDC. Chapter Ten presents an example in which we do printing as well as print preview. (The OnDraw(CDC* pDC) function will be discussed further. For now, you only need to know that this is occurring and that the framework will pass you the correct device context pointer.)

In this example, our OnDraw(CDC* pDC) function is very simple. We use the pointer of the device context passed to us, which will be of class CPaintDC. The device context has already been customized to have the attributes that we want, so there is nothing left to do but to draw the ellipse:

```

void C_CustomView::OnDraw(CDC* pDC)
{
    pDC->Ellipse(20, 20, 200, 100);
}

```

Message Maps

The last item that we will discuss before looking at the source code is where (in which class) to respond to the WM_COMMAND messages from the menu items. A WM_COMMAND message is routed to each of the four classes until it finds its handler function. When it finds its handler function, no further routing occurs.

In this example, we had three menu items: “SetCaption,” “DrawRectangle,” and “QuitApp.” The WM_COMMAND generated from the menu item “DrawRectangle” must go to the view class, because it will draw on the view window with the device context owned by the view class.

The WM_COMMAND generated by the other two menu items, “SetCaption” and “QuitApp,”

could be located in any of the classes. In this example, the response to the menu item “SetCaption” is set in the mainframe class, and the response to the menu item “QuitApp” is located in the view class to illustrate the following point: these two response functions require that C_MainFrame functions be activated and that the C_MainFrame functions can be activated from any class.

This is illustrated by the handler function OnQuitApp() as shown here:

```
void C_CustomView::OnQuitApp()  
{  
    AfxGetApp()->m_pMainWnd->DestroyWindow();  
}
```

This handler function is located in the view class. To quit the application, the mainframe window needs to be destroyed. To do this, call the DestroyWindow() function that is a member of the C_MainFrame class. This is done by obtaining a pointer to the C++ C_MainFrame object and using this pointer to call the correct DestroyWindow() function. This handler function could also have been located in the application class or the document class, because we can obtain the C_MainFrame object’s pointer from any class using the expression:

AfxGetApp()->m_pMainWnd

The handler function that responds to the menu item “SetCaption” is, in this example, located in the C_MainFrame class. This is the least clumsy place to locate response functions that require C_MainFrame functions to be activated. The code for this handler function is:

```
void C_MainFrame::OnSetCaption()  
{  
    SetWindowText("Customizing the Window");  
}
```

The SetWindowText() function of the class C_MainFrame is called, and it sets the text of that window. Also, of course, no pointer is needed since we made the call from within the C_MainFrame class.

Previous	Table of Contents	Next

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

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

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

Keyword

☒ Brief
 ☐ Full

☒ [Advanced Search](#)

☒ [Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

The Custom Program Listing

Listing 9-1 gives the source code for the four-class “Custom” program. As shown, the services of ClassWizard can be accessed with this code. The resources were generated with tools, but the comment lines of the tool-generated code are not shown. The listing is followed by a discussion of additional features that have not yet been addressed.

Listing 9-1: Source Code for the Custom Program

```

-----
                                stdafx files
-----
//
// stdafx.h : include file for standard system include files
//
#include <afxwin.h>

//
// stdafx.cpp : source file for standard includes
//
#include "stdafx.h"

-----
                                application class
-----
//
//          Custom.h
//
#include "resource.h"

class C_CustomApp : public CWinApp
{
public:
    virtual BOOL InitInstance();
};
  
```



```

//
//      Custom.cpp : defines the class behaviors
//
#include "stdafx.h"
#include "Custom.h"
#include "MainFrm.h"
#include "CustDoc.h"
#include "CustView.h"

BOOL C_CustomApp::InitInstance()
{
    CSingleDocTemplate* pDocTemplate;
    pDocTemplate = new CSingleDocTemplate(
        IDR_MAINFRAME,
        RUNTIME_CLASS(C_CustomDoc),
        RUNTIME_CLASS(C_MainFrame),
        RUNTIME_CLASS(C_CustomView));
    AddDocTemplate(pDocTemplate);

    // enable DDE Execute open
    EnableShellOpen();
    RegisterShellFileTypes();

    // simple command line parsing
    if (m_lpCmdLine[0] == '\\0')
    {
        // create a new (empty) document
        OnFileNew();
    }
    else
    {
        // open an existing document
        OpenDocumentFile(m_lpCmdLine);
    }

    // enable drag/drop open
    m_pMainWnd->DragAcceptFiles();

    return TRUE;
}

C_CustomApp theApp; // instantiate the object that runs the program

-----
                                mainframe class
-----
//
// MainFrm.h : interface of the C_MainFrame class
//
class C_MainFrame : public CFrameWnd
{
    DECLARE_DYNCREATE(C_MainFrame)
public:
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
protected:

```

```

        //{{AFX_MSG(C_MainFrame)
        afx_msg void OnSetCaption();
        //{{AFX_MSG

        DECLARE_MESSAGE_MAP()
};

//
// MainFrm.cpp : implementation of the C_MainFrame class
//
#include "stdafx.h"
#include "Custom.h"
#include "MainFrm.h"

IMPLEMENT_DYNCREATE(C_MainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(C_MainFrame, CFrameWnd)
    //{{AFX_MSG_MAP(C_MainFrame)
    ON_COMMAND(ID_SETCAPTION, OnSetCaption)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

BOOL C_MainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    cs.cy = 200;
    cs.cx = 300;
    cs.y = (::GetSystemMetrics(SM_CYSCREEN) - 200) / 2;
    cs.x = (::GetSystemMetrics(SM_CXSCREEN) - 300) / 2;

    cs.style = WS_CAPTION | WS_SYSMENU | WS_THICKFRAME;
    return CFrameWnd::PreCreateWindow(cs);
}

void C_MainFrame::OnSetCaption()
{
    SetWindowText("Customizing the Window");
}

-----
view class
-----
//
// CustView.h : interface of the C_CustomView class
//
class C_CustomView : public CView
{
    DECLARE_DYNCREATE(C_CustomView)
public:
    CPen* pMyPen;
    CBrush* pMyBrush;
    virtual ~C_CustomView();
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    virtual void OnDraw(CDC* pDC);
protected:
    //{{AFX_MSG(C_CustomView)

```

```

        afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
        afx_msg void OnDrawRectangle();
        afx_msg void OnQuitApp();
        //}}AFX_MSG

    DECLARE_MESSAGE_MAP()
};

//
//      CustView.cpp : implementation of the C_CustomView class
//
#include "stdafx.h"
#include "Custom.h"
#include "CustDoc.h"
#include "CustView.h"

IMPLEMENT_DYNCREATE(C_CustomView, CView)

BEGIN_MESSAGE_MAP(C_CustomView, CView)
    //{{AFX_MSG_MAP(C_CustomView)
    ON_COMMAND(ID_QUITAPP, OnQuitApp)
    ON_WM_CREATE()
    ON_COMMAND(ID_DRAWRECTANGLE, OnDrawRectangle)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

C_CustomView::~C_CustomView()
{
    delete pMyPen;
    delete pMyBrush;
}

BOOL C_CustomView::PreCreateWindow(CREATESTRUCT& cs)
{
    CBrush hYellowBrush( RGB( 255, 255, 0));

    cs.lpszClass = AfxRegisterWndClass(
        CS_HREDRAW | CS_VREDRAW | CS_OWNDC,
        AfxGetApp()->LoadStandardCursor(IDC_UPARROW),
        (HBRUSH)hYellowBrush.GetSafeHandle());
    return CView::PreCreateWindow(cs);
}

int C_CustomView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    CPaintDC dc(this);
    pMyPen = new CPen(PS_SOLID, 5, RGB(0, 0, 255)); // blue pen
    pMyBrush = new CBrush(RGB(128, 128, 0)); // olive brush
    dc.SelectObject(pMyPen);
    dc.SelectObject(pMyBrush);
    return 0;
}

void C_CustomView::OnDraw(CDC* pDC)
{

```

```

        pDC->Ellipse(20, 20, 200, 100);
    }

void C_CustomView::OnDrawRectangle()
{
    CClientDC myDC(this);
    myDC.Rectangle(20, 20, 200, 100);
}

void C_CustomView::OnQuitApp()
{
    AfxGetApp()->m_pMainWnd->DestroyWindow();
}

-----
                                document class
-----
//
// CustDoc.h : interface of the C_CustomDoc class
//
class C_CustomDoc : public CDocument
{
    DECLARE_DYNCREATE(C_CustomDoc)
};

//
// CustDoc.cpp : implementation of the C_CustomDoc class
//
#include "stdafx.h"
#include "Custom.h"
#include "CustDoc.h"

IMPLEMENT_DYNCREATE(C_CustomDoc, CDocument)

-----
                                resource files
-----
//
//          resource.h
//
#define IDR_MAINFRAME                128
#define ID_SETCAPTION                32771
#define ID_DRAWRECTANGLE             32772
#define ID_QUITAPP                   32773

//
//          custom.rc
//
#include "resource.h"
#include "afxres.h"

IDR_MAINFRAME            ICON            DISCARDABLE            "res\\smiley.ico"

IDR_MAINFRAME MENU PRELOAD DISCARDABLE
BEGIN

```

```
        MENUITEM "&SetCaption",          ID_SETCAPTION
        MENUITEM "&DrawRectangle",       ID_DRAWRECTANGLE
        MENUITEM "&QuitApp",             ID_QUITAPP
END

STRINGTABLE PRELOAD DISCARDABLE
BEGIN
        IDR_MAINFRAME                    "Custom Window"
END
-----
```

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

The Document Template

In the four-class document-view architecture, the document template defines the relationships between the document, view, and mainframe classes. The document template is located in the `InitInstance()` overriding function in the application class. For a *Single Document Interface* (SDI) application, which is the case in Listing 9-1, the document template is of class `CSingleDocTemplate`. We find the following code in the `InitInstance()` function:

```

CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
    IDR_MAINFRAME,
    RUNTIME_CLASS(C_CustomDoc),
    RUNTIME_CLASS(C_MainFrame),
    RUNTIME_CLASS(C_CustomView));
AddDocTemplate(pDocTemplate);
  
```

The document template ties the `IDR_MAINFRAME` resources to the mainframe, and it creates and maintains a communication structure between the mainframe, document, and view classes. In a SDI application only one document is open at a time. The document functionality is specified by the `CSingleDocTemplate` object. You do not need to call any member function of `CSingleDocTemplate` (except the constructor), and as shown the constructor is called in the `InitInstance()` function of the application class. (The framework handles `CSingleDocTemplate` objects internally.)

A *Multiple Document Interface* (MDI) application is created using the document template class `CMultiDocTemplate`. An MDI application can have multiple documents, and each document can have multiple views attached to it. The structure of the MDI application is more complex, so we discuss MDIs

in Chapters Twelve and Thirteen.

Figure 9-6 depicts the document-view creation process. The application object has a pointer to its document template. The document template has a pointer to its document (only one) and also to the frame window. The frame window has a pointer to its current view window. The view is a child of the frame window, and it has a pointer to its document. The document has a pointer to the document template and also has a list of views, if more than one view is attached to the document. We can create multiple views attached to the document. In a SDI application, this is done by using *splitter windows*. (Chapters Ten and Eleven present examples of splitter windows used with SDI applications.)

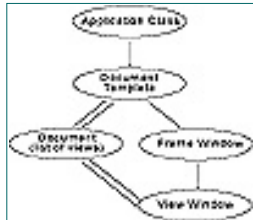


Figure 9-6: Document-View Creation

There are MFC functions that are available for traversing between the classes. These are shown on Figure 9-7. These functions are not used in the “Custom” example in this chapter, but are used in the examples given in Chapters Ten and Eleven.



Figure 9-7: MFC Functions for Traversing Between Classes (SDI Application)

The RUNTIME_CLASS Macro

Document templates use the RUNTIME_CLASS macro. This macro returns the CRunTimeClass structure for the named class. CRunTimeClass encapsulates needed services, including access to run-time class information, dynamic object creation, and *serialization*. When you use document templates, you must declare the level of run-time support needed for each of the classes for which the RUNTIME_CLASS macro has been used. Thus, we need to declare the level of run-time support needed in each of the classes: C_CustomDoc, C_MainFrame, and C_CustomView.

Note: In Listing 9-1, the header file for each of these classes contains the macro DECLARE_DYNCREATE(<classname>) and the corresponding implementation file contains the macro IMPLEMENT_DYNCREATE(<classname>, <base class>). This macro pair defines the level of run-time support requested.

Access to run-time class information is obtained through the function CObject::IsKindOf() which enables you to determine information about an

object’s class at run time. This is 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. (Run-time class information is not supported directly by the C++ language.)

Dynamic object creation enables you to create an object of a specified class at run time. Document, view, and frame classes must support dynamic creation because the framework needs to create them dynamically. The function `CRuntimeClass::CreateObject()` is used for dynamic object creation.

Serialization is the process of reading or writing an object’s contents to and from a file. You can: store an object’s contents to file, exit the application, restart the application, and read the object’s contents from the file. Such data objects are persistent. The class `CArchive`, which is used as an intermediary between the object to be serialized and the storage medium (usually a file), uses the overloaded insertion (`<<`) and extraction (`>>`) operators to perform reading and writing operations. (Examples in Chapters Ten and Eleven use and will clarify the serialization process for you.)

The macro pair `DECLARE_DYNAMIC(<class>)` used in the class declaration and `IMPLEMENT_DYNAMIC(<class>, <base class>)` used in the class implementation provides the service of run-time information only. The macro pair `DECLARE_DYNCREATE(<class>)` used in the class declaration and `IMPLEMENT_DYNCREATE (<class>, <base class>)` used in the class implementation provides the services of run-time information and dynamic class creation. The macro pair `DECLARE_SERIAL(<class>)` used in the class declaration and `IMPLEMENT_SERIAL(<class>, <base class>)` used in the class implementation provides all of the services: run-time information, dynamic object creation, and serialization. This is summarized in Table 9-2.

Table 9-2: Macros Used for Run-Time Services

Macro Pair	Run-Time Class Information	Dynamic Object Creation	Serialization
	CObject:: IsKindOf()	CRuntimeClass:: CreateObject()	CArchive:: >>and<<
DECLARE_DYNAMIC IMPLEMENT_DYNAMIC	Yes	No	No
DECLARE_DYNCREATE IMPLEMENT_DYNCREATE	Yes	Yes	No
DECLARE_SERIAL IMPLEMENT_SERIAL	Yes	Yes	Yes

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

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

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)**SEARCH**
ITKNOWLEDGE☒ Brief ☐ Full[Advanced](#)[Search](#)[Search Tips](#)**BROWSE**
BY TOPIC

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

The CView Class

We have seen that objects of class CView are child windows in the client area of the frame window, and they show and accept input data for their document. Chapters Ten and Eleven present examples where we derive our view class from the class CView. The CView class is a base class for other classes that are derived from it; here we will discuss the other classes derived from the CView class and their functionality.

The CScrollView class is derived from the CView class. The CScrollView class adds automatic scrolling capability to your window; derive your view class from CScrollView if you want scrolling capability. (In Chapter Thirteen an MDI example is presented that derives its view class from CScrollView and hence has all the automatic scrolling features.)

Views Based on a Dialog Template

If you need an application based on a simple modeless dialog, the class CFormView will save you a lot of work. CFormView is derived from CScrollView which is used as the base class, so your view derived from CFormView will support scrolling, as needed. Use CFormView as the base class for your view if you want a view that contains only controls. The CFormView class is associated with a dialog resource that defines and enumerates the controls. The controls are laid out on your window based on the dialog-template resource. In effect, your view window is a modeless dialog box with controls that accept input. A CFormView object receives notification messages directly from its controls, and it also receives command messages. (In Chapter Twelve an MDI example is presented which has its view derived from CFormView.)

Views Based on a Control

View windows can also be based on controls. The most notable example of this is CEditView. An object of a class derived from the CEditView class is like a CView object because it can process command messages and it is also like a CEdit object because it permits text editing and formatting. The CEditView object is an ordinary view object with the functionality of an edit control window. Its view window is essentially an edit control that is shrink-wrapped inside the mainframe's client area, so all you can do is text editing.

Tip: Do not try to build a full-featured word processor with CEditView. The class has many limitations that are imposed by the underlying edit control. You cannot mix fonts within the window and you are limited to a text buffer size of 64KB. CEditView does implement the clipboard Cut, Copy, and Paste commands. (In Chapter Seven we studied the edit control and its usage should be familiar, so an example of CEditView is not presented here.)

Prior to MFC 4.0, CEditView was derived directly from the CView class. With the advent of MFC 4.0, more controls can now be view windows. These controls are the *list control* and the *tree control*. In MFC 4.0, the class CCtrlView is derived from CView and all the control views are then derived from the class CCtrlView. The control views are: CEditView, which was previously discussed, CRichEditView, which encapsulates rich, or enhanced, edit-control functionality, CListView, which encapsulates list-control functionality, and CTreeView, which encapsulates tree-control functionality. The list control and tree controls are new MFC 4.0 controls; examples of these new controls are given in Chapter Fourteen.

Summary

Applications can be built using the four-class structure: the document class, the view class, the mainframe class, and the application class. These four-class applications have additional built-in capabilities, which we explore in the remainder of this book. Generally, the programmer uses compiler tools to get a “starter” application which consists of “generic” code which is common to most applications. These four-class applications can also be built manually.

The document-view architecture consists of a window of the view class, which occupies the mainframe class's entire client area. The view class has a document attached to it. The document class is used to store the data of the application. An application's data consists of all the essential information that an application needs to store when it stores data and then restarts the application from just that stored data. For the document-view architecture, the WM_COMMAND message is routed to each of the four classes. When the message's handler function is encountered, the handler function is executed and further routing is terminated.

The visual features of the application's window are customized using the PreCreateWindow() function. The PreCreateWindow() function is overridden; in the overriding function the variables of the CREATESTRUCT structure are assigned to be what you want; then the base class's PreCreateWindow() function is called to create the window. Features displayed on the caption bar can be customized using the CFrameWnd::PreCreateWindow() function.

Features of the client area which is occupied by the view window, can be customized using the `CView::PreCreateWindow()` function. Features are also customized by assigning mainframe resources in the .rc file.

The document-view architecture has an `OnDraw(CDC*)` function which must be used as the response function for the `WM_PAINT` messages. The `OnDraw()` function is used for drawing to the screen, for printing, and for print preview. The framework takes care of obtaining the appropriate device context and providing it as an input parameter to the `OnDraw(CDC*)` function.

The document-view architecture uses document templates, which specify the mainframe resources and create and tie together the mainframe, view, and document classes. Document templates use the `RUNTIME_CLASS` macro which returns a `CRunTimeClass` structure. The `CRunTimeClass` provides needed services including access to run-time information, dynamic object creation, and serialization which is used for storing to files and reading from files.

View classes can be derived from the `CView` class or one of its derived classes. Classes derived from `CView` include `CScrollView` and its descendant `CFormView` and `CCtrlView` and its descendants: `CEditView`, `CRichEditView`, `CListView`, and `CTreeView`.

Previous	Table of Contents	Next

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

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

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

Keyword

☒ Brief
 ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Chapter 10

Document-View Applications With Filing and Printing

This chapter introduces capabilities that are integral to the document-view four-class architecture, principally the capabilities of printing and filing. We look at a Single Document Interface (SDI) application first with a single view of the single document and then using splitter windows, we look at multiple views of the single document. We look not only at a modal dialog box, but also at a modeless dialog box that is managed by the view class. In Chapter Thirteen, we will look at the same capabilities for Multiple Document Interface (MDI) applications.

First shown here are the steps to build SDI applications where the starter application has been generated using AppWizard. You will create a starter application called “Elip” using AppWizard and it will contain the four classes: the application class derived from CWinApp, the mainframe class derived from CFrameWnd, the view class derived from Cview, and the document class derived from CDocument. You will learn how to add your application’s code into the document and view classes. You will learn which code of the application must be added to the document class and which code must be added to the view class. In this chapter, we focus on learning how to work with the view and the document classes and how to design the data such that you can file it from the document class and print it from the view class. You will design the application’s data and methods and locate them in the document class. You will then design the user interface which will be located in the view class.

After completing the discussion of the AppWizard-generated application,

manually generated code for the same application is introduced. A listing of this code is given, and you can review all the necessary code statements in the complete program.

Note: If you are a Borland C++ compiler user, you need to use the manually generated code.

In this chapter, you will develop a document-view application that you will use for filing and printing. The screen appearance of the final program is shown in Figure 10-1.



Figure 10-1: Finished Elip Application

The application has three main menu items that are specific to this application. They are: “BrushHatch” for the user to select the brush’s hatch, “Rotation” for the user to rotate the ellipse, and “Dimensions” which brings up a dialog box for the user to enter the width and height of the ellipse. Figure 10-2 shows the application with the ellipse rotated and with the brush hatch changed.



Figure 10-2: Changing the BrushHatch and Rotating the Ellipse

The other menu items are placed there by AppWizard. We do not use “Edit” or “Help” (you could remove them from the code if you wanted to). However, the “File” menu item is necessary and will be demonstrated when we are opening or filing the document’s data or printing the view.

Creating an AppWizard Project

We create an AppWizard *starter application*, then add our code to it in the appropriate places. Creating an AppWizard starter application is covered in detail in Appendixes D and E. (If you are using a compiler other than Microsoft, consult the appropriate appendix on your specific compiler.) The AppWizard discussion here is for Visual C++ 4; implementation details will vary for other compilers. Briefly, you will create the AppWizard application by selecting **File | New | ProjectWorkspace**. In the “**N**ew Project Workspace” query box, you will choose AppWizard and enter the location and name of your project. Six steps of AppWizard choices then appear. (These steps are depicted in Appendix D in the section titled “Creating a Starter AppWizard Project.”)

1. Choose the “Single Document” application (Multiple Document is

the default).

2. Choose “None” (the default).

3. Again, choose “None” (the default).

4. Choose only “Printing and print preview,” deselect all other choices, and then choose the “Advanced” button.

a. In the “Advanced Options” query box, choose the tab “Document Template Strings” and fill in the “File extension” that you wish your files to have.

5. Choose not to generate source comments, and to use the MFC library as a statically linked library.

6. The default of deriving our view class from CView is what we want. However, change the name of the four basic classes. Name them C_MainFrame, C_ElipApp, C_ElipDoc, and C_ElipView.

a. After you have finished creating the AppWizard starter application, compile it to see what has been automatically included. Before compilation, choose the project settings that you want (see Appendix D). Be sure to choose “Use MFC in a Static Library.” You may want to use release mode and precompiled headers since you will do several compiles as you incrementally add more code in this example.

Note: Precompiled headers save compilation time for the subsequent compilations, but they consume almost three megs of memory, so you only use them while you are developing your application.

Now you can build the application and execute it. Figure 10-3 shows how it looks at this stage.



Figure 10-3: AppWizard-Generated Starter Elip Application

AppWizard has provided a menu with the items “File,” “Edit,” and “Help.” We will be using the “File” menu item to save a document file, to initialize or load the application from a file, and to preview and print the document. The menu item “Edit” is used only for view classes derived from CEditView or CRichEditView. The menu item “Help” gives an “About” dialog box.

AppWizard has generated the starter MFC icon and provided a string for the caption bar. The icon is all right, but the contents of the caption bar are not, so we next modify the string in the caption bar. To do this, we need to change the contents of the IDR_MAINFRAME string. For this application, AppWizard has generated the following IDR_MAINFRAME string, which is known as the *document template string* because it is contained in the document template:

```
"Elip\n\nElip\nElip Files(*.ell)\n.ELL\n\nElip.Document\nElip Document"
```

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

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

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

SEARCH
 ITKNOWLEDGE

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

BROWSE
 BY TOPIC

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes
(Publisher: Wordware Publishing, Inc.)
Author(s): Shirley Wodtke
ISBN: 1556225121
Publication Date: 01/01/97

Bookmark It

Search this book:

[Previous](#)

[Table of Contents](#)

[Next](#)

This string is separated into seven substrings by newline characters (\n). If a substring is not wanted, you must still use the \n character as a delimiter. Each substring contains information specific to the document template. These substrings show up in various places when the application executes.

Note: For Win3.1x users, the substrings 1 and 2 are used slightly differently in the caption bar; their positions are reversed.

Table 10-1 gives the use for each substring in the IDR_MAINFRAME string.

Table 10-1: Usage of Each IDR_MAINFRAME Substring

Substring	Usage
1	Window Title, the name that appears in the window’s title bar for SDI applications. When an application is started this substring is used as the second string entry on the caption bar, after the root document name. Also, it appears in informative dialog boxes, etc.
2	Document Name, root for the default document name. When an application is first opened, it appears as the first string entry on the caption bar which gives the document that is open in the view. If this substring is empty, the string “Untitled -” is used as the document name. The first word of this substring is used as the default file name when a document is filed.
3	Name of this document type. In an MDI this string is displayed in the File New “New” dialog box.
4	Document name and filter, used in the “Open File” and “Save File” common dialogs as the document type description with its extension (filter).
5	File extension, used for the file extension for saving documents of this type.
6	Identifier of the document type to be stored in the registration database maintained by Windows. This string is for internal use only.
7	Name of the document type. This string will be displayed in certain dialog boxes.

Enter the string table and change the IDR_MAINFRAME string to be:

```
"Elip\nEllipse Doc-View App\nElip\nElip Files(*.ell)
\n.ELL\nElip.Document\nElip Document"
```

We have changed the caption title. Recompile the program and the application now has the title as shown in Figure 10-4.

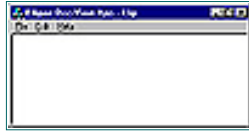


Figure 10-4: Starter Application After Altering IDR_MAINFRAME String

There are two main activities that need to take place when developing a document-view application; the first activity is to design the data that is specific to the application, and the other activity is to design the user interface for viewing and modifying this data. When designing the data, the emphasis is on the effectiveness of the data definition, the efficiency of the data manipulation routines and the data storage routines, whereas when designing the user interface, the emphasis is on ease of use and a pleasant and consistent appearance.

Designing the Application's Data

The principal object of our application is the ellipse object. The data of any application is kept in the document object. The ellipse object will be declared as an object data member of the document class; the ellipse object can be thought of as the document's data. We will be filing and retrieving from the file all the data necessary to generate an ellipse object, which consists of its brush hatch, its rotation, and its width and height. We define a class `C_Ellipse` that contains all the data members and the member functions needed by an ellipse object.

The class of the document object is derived from the `CDocument` class; this derived class is called `C_ElipDoc`. We could place our `C_Ellipse` class by itself in its own header and implementation files, but instead we place it within the file that contains the document class. The document class definition is kept in the file `ElipDoc.h`. We choose to place the definition of the `C_Ellipse` class in the `ElipDoc.h` file. Add to the `ElipDoc.h` file the following lines shown in bold and also marked with an arrow. (The code not in bold should already be in your file.)

Note: The added code can go anywhere in the `ElipDoc.h` file, but it is convenient to put it first.

```
//
//      ElipDoc.h : interface of the C_ElipDoc class
//
        . . . . .

-->class C_Ellipse : public CObject
-->{
```

```

--> DECLARE_SERIAL(C_Ellipse)
-->public:
-->    C_Ellipse();

-->    int m_nBrushHatch;
-->    int m_nWidth;
-->    int m_nHeight;
-->    CPoint m_EllipseCenter;    // center of client area
-->    CRect m_OriginalPos;       // original bounding rect
-->    CRect m_RotatedPos;        // rotated bounding rect
-->    int m_nRotation;           // current ellipse rotation
-->    // the two possible rotations of the ellipse
-->    #define ROTATED90    1
-->    #define ROTATED180   2

-->    void BoundingRect(CRect& r);
-->    void Rotate();
-->};

```

Notice that the class `C_Ellipse` was derived from `CObject`, and the macro `DECLARE_SERIAL(C_Ellipse)` was included in the class. These are necessary conditions to *serialize* objects of this class. Objects are serialized when they are filed or loaded from the file.

We must add the declaration (as a public data member) of a `C_Ellipse` object, named `m_Ellipse`, to the class `C_ElipDoc` declaration in the file `ElipDoc.h`. This makes `m_Ellipse` an object data member of the class `C_ElipDoc`. We will use the `m_Ellipse` object to access the member function `Serialize()` which has been inherited from `CDocument`. We also will use the `m_Ellipse` object to access the member variables and functions of its own class. In the following code excerpt, the code that is not bold indicates code that has already been placed there by AppWizard. The code in bold and preceded by an arrow is the code to add.

```

//
//    ElipDoc.h : interface of the C_ElipDoc class
//
    . . . . .

class C_ElipDoc : public CDocument
{
    . . . . .
// Attributes
public:
-->    C_Ellipse m_Ellipse;
    . . . . .

```

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

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



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Go!

Keyword

☒ Brief ☐ Full[Advanced](#)[Search](#)[Search Tips](#)BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

Go!

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

The implementation of the member functions of the class C_Ellipse will be placed in the implementation file for the C_ElipDoc class, namely ElipDoc.cpp. The data for our application should be initialized at run time; the place for this initialization is the constructor for the C_Ellipse class, so we add the constructor for the class C_Ellipse and in it place the code to initialize the C_Ellipse member variables. Following the constructor function code for the class C_Ellipse, enter the code for the definition of the other member functions of the class C_Ellipse. The following lines of code to add are marked with an arrow and shown in bold.

```
//
// ElipDoc.cpp : implementation of the C_ElipDoc class
//
. . . . .

-->IMPLEMENT_SERIAL(C_Ellipse, CObject, 1)

-->C_Ellipse::C_Ellipse()
-->{
-->    m_nBrushHatch = HS_CROSS;
-->    m_nWidth = 300;
-->    m_nHeight = 100;
-->    m_nRotation = ROTATED180;
-->}

-->void C_Ellipse::BoundingRect(CRect& r)
-->{
-->    m_EllipseCenter.x = r.Width() / 2;
-->    m_EllipseCenter.y = r.Height() / 2;
-->    CSize HalfSizeOrig(m_nWidth / 2, m_nHeight / 2);
-->    m_OriginalPos.TopLeft() = m_EllipseCenter - HalfSizeOrig;
-->    m_OriginalPos.BottomRight() = m_EllipseCenter + HalfSizeOrig;
-->}

-->void C_Ellipse::Rotate()
-->{
-->    switch (m_nRotation)
-->    {
-->        case ROTATED180:
```

```

-->         m_RotatedPos = m_OriginalPos;
-->         break;
-->     case ROTATED90:
-->         CSize HalfSizeRotated(m_nHeight / 2, m_nWidth / 2);
-->         m_RotatedPos.TopLeft() =
-->             m_EllipseCenter - HalfSizeRotated;
-->         m_RotatedPos.BottomRight() =
-->             m_EllipseCenter +HalfSizeRotated;
-->         break;
-->     }
-->}

```

The member variable and member functions of the class C_Ellipse do a number of things. They will store the current data for the ellipse: its brush hatch style, its height, its width, the current ellipse rotation, the center of the ellipse (which is always kept at the center of the client area by using the function BoundingRect()), the bounding rectangle based on its original rotation, m_OriginalPos, and the bounding rectangle based on its current rotation, m_RotatedPos. The rotation can be either horizontal or vertical. The member function Rotate() calculates the values of the member variables when a rotation occurs.

Note: The implementation file includes the statement IMPLEMENT_SERIALIZE(C_Ellipse, CObject, 1). This completes the conditions for making the class serializable. To summarize making a class serializable, the following three conditions are necessary:

- (1) The class must be derived directly or indirectly from CObject.
- (2) The class's declaration must contain the macro call DECLARE_SERIAL(<classname>).
- (3) The class's implementation file must contain the macro call IMPLEMENT_SERIAL(<classname>, <base classname>, wVer) where wVer is an UINT "version number" encoded in the archive to enable a deserializing program to identify and handle data created by earlier program versions.

Designing the User Interface

The class that takes care of the user interaction with the document is the view class, C_ElipView. It also takes care of drawing, which is the function that we now complete. When the application is first called, we want the ellipse with the initial values of its variables to be drawn in the client area. The member function that allows us to draw the ellipse in the client area of the view is OnDraw(). In order to keep the ellipse centered in the client area, we get the dimensions of the client area rectangle. We do this in the class C_ElipView since we use the function CWnd::GetClientRect(); we then pass the client area rectangle in the call to BoundingRect(). This code allows us to update the center of the ellipse as the client area is resized.

We access the member variables stored in the document class by using the pointer to the document class pDoc. We draw the ellipse in the client area by first customizing the device context and then making a call to the Ellipse() function. The Ellipse() function needs to know the bounding rectangle for the ellipse, which we provide with the expression pDoc->m_Ellipse.m_RotatedPos. pDoc is the pointer to the document class, and we access the CRect variable that we need, m_Ellipse.m_RotatedPos, using the pDoc pointer. In the following excerpt, the lines of code to add are marked with an arrow and shown in bold.

Note: AppWizard has provided not only the stubbed-out OnDraw() function, but also the line of code for obtaining the pointer pDoc as well as a debug statement (discussed later in this chapter).

```

//
//      ElipView.cpp
//
. . . . .

-->void C_ElipView::OnDraw(CDC* pDC)

```

```

-->{
    C_ElipDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
-->    CRect r;
-->    GetClientRect(&r);
-->    pDoc->m_Ellipse.BoundingRect(r);
-->    pDoc->m_Ellipse.Rotate();
-->    CBrush newBrush(pDoc->m_Ellipse.m_nBrushHatch, RGB( 0, 0, 0));
-->    CBrush* pOldBrush = pDC->SelectObject(&newBrush);
-->    pDC->Ellipse(pDoc->m_Ellipse.m_RotatedPos);
-->    pDC->SelectObject(pOldBrush);
-->}

```

At this stage, build the application and see what it looks like. After compiling your program, you should get the application as shown in Figure 10-5. Notice that the ellipse stays centered in the client area as you resize the window. At this stage in the development of our application, we have no way of changing the ellipse's parameters.

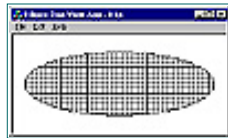


Figure 10-5: Elip Application With Initial Data

Next we design the menu, which is used for changing the ellipse's parameters.

In our Elip example, we have menu items that are specific to our application. One of the menu items will open a dialog box so that the user can enter the dimensions of the ellipse. The application-specific user interface consists of three application unique menu items and a dialog box. We first look at how to insert our application's menu items into the existing menu that AppWizard provided. We will complete the handler functions for all menu items except the one which opens the dialog box. Then we will create the dialog box and wrap it with a class derived from CDialog. At that point, we are able to complete the handler function in the C_ElipView class that opens the dialog box.

Previous	Table of Contents	Next

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

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

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

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

[Previous](#)[Table of Contents](#)[Next](#)[BROWSE
BY TOPIC](#)

The Application's Menu

To add new menu items, open the .rc file and then open the IDR_MAINFRAME menu. An entry box which is to be used for the next menu item entry is appended to the existing menu. We add the menu item “BrushHatch” and its submenu using the appended entry box. After our menu item is completed, we grab it and move it, positioning it between the existing menu items “File” and “Edit,” as shown in Figure 10-6.

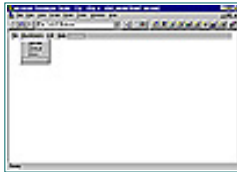


Figure 10-6: Inserting Menu Items

We next complete the other two application menu items, “Rotation” and “Dimensions,” moving them and inserting them as shown in Figure 10-7. We now have completed the menu, and it includes the three menu items and their submenus which are unique to this application.

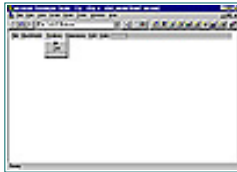


Figure 10-7: The Completed Menu

Next use ClassWizard to do the message map entries and the handler functions for each of the menu items. Add a command handler and also an update command handler for each menu item, except for the “Dimensions” menu item for which we only add a command. All handler functions are to be added to the view class. Complete the stubbed-out handler functions by adding the following code, shown in bold and preceded by an arrow, to the handler functions.

```
void C_ElipView::OnBrushhatchCross()
{
-->    C_ElipDoc* pDoc = GetDocument();
```



```

-->     pDoc->m_Ellipse.m_nBrushHatch = HS_CROSS;
-->     Invalidate();
}

void C_ElipView::OnUpdateBrushhatchCross(CCmdUI* pCmdUI)
{
-->     C_ElipDoc* pDoc = GetDocument();
-->     if (pDoc->m_Ellipse.m_nBrushHatch == HS_CROSS)
-->         pCmdUI->SetCheck(1);
-->     else
-->         pCmdUI->SetCheck(0);
}

void C_ElipView::OnBrushhatchUpward()
{
-->     C_ElipDoc* pDoc = GetDocument();
-->     pDoc->m_Ellipse.m_nBrushHatch = HS_FDIAGONAL;
-->     Invalidate();
}

void C_ElipView::OnUpdateBrushhatchUpward(CCmdUI* pCmdUI)
{
-->     C_ElipDoc* pDoc = GetDocument();
-->     if (pDoc->m_Ellipse.m_nBrushHatch == HS_FDIAGONAL)
-->         pCmdUI->SetCheck(1);
-->     else
-->         pCmdUI->SetCheck(0);
}

void C_ElipView::OnBrushhatchVertical()
{
-->     C_ElipDoc* pDoc = GetDocument();
-->     pDoc->m_Ellipse.m_nBrushHatch = HS_VERTICAL;
-->     Invalidate();
}

void C_ElipView::OnUpdateBrushhatchVertical(CCmdUI* pCmdUI)
{
-->     C_ElipDoc* pDoc = GetDocument();
-->     if (pDoc->m_Ellipse.m_nBrushHatch == HS_VERTICAL)
-->         pCmdUI->SetCheck(1);
-->     else
-->         pCmdUI->SetCheck(0);
}

void C_ElipView::OnRotation180()
{
-->     C_ElipDoc* pDoc = GetDocument();
-->     pDoc->m_Ellipse.m_nRotation = ROTATED180;
-->     Invalidate();
}

void C_ElipView::OnUpdateRotation180(CCmdUI* pCmdUI)

```

```

{
--> C_ElipDoc* pDoc = GetDocument();
--> if (pDoc->m_Ellipse.m_nRotation == ROTATED180)
-->     pCmdUI->SetCheck(1);
--> else
-->     pCmdUI->SetCheck(0);
}

void C_ElipView::OnRotation90()
{
--> C_ElipDoc* pDoc = GetDocument();
--> pDoc->m_Ellipse.m_nRotation = ROTATED90;
--> Invalidate();
}

void C_ElipView::OnUpdateRotation90(CCmdUI* pCmdUI)
{
--> C_ElipDoc* pDoc = GetDocument();
--> if (pDoc->m_Ellipse.m_nRotation == ROTATED90)
-->     pCmdUI->SetCheck(1);
--> else
-->     pCmdUI->SetCheck(0);
}

```

We will not be able to connect the OnDimensions() handler function to code until we have completed the dialog box, which we do in the next section.

The Dialog Box

It is now time to design the dialog box that will pop up when the user selects the “Dimensions” menu item. To add a new dialog box:

1. Go to the Microsoft Developer Studio menu item “Insert | Resource” and choose “Dialog” from the “New Resource” query box that appears. This opens the Dialog Editor. Design the dialog box to look like the one shown in Figure 10-8.

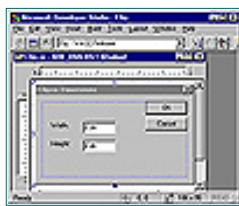


Figure 10-8: Designing the Ellipse Dimensions Dialog Box

2. Once the dialog box is designed, use ClassWizard to wrap it in a class named C_SizeDialog and specify that the header file be named SizeDlg.h and that the implementation file be named SizeDlg.cpp.
3. Use ClassWizard to add member variables.
 - a. Add the integer member variable m_nWidth for the first edit box and give it a validation range of 1 to 500.
 - b. Add the integer member variable m_nHeight for the second edit box and give it a validation range of 1 to 500.
4. At this point in the application development, add the changes shown in the following code.
 - a. Add the dialog class’s header file to the view class #includes.

(1) Open the file “ElipView.cpp.”

(2) Include the header file for the class C_SizeDialog, namely “SizeDlg.h.”

b. We have completed the dialog box, so now enter the code in the view’s handler function which will open the dialog box and update the ellipse data, based on the values entered by the user in the “Ellipse Dimensions” dialog box.

(1) Insert the following code shown in bold and preceded with an arrow into the OnDimensions() handler function.

```
//
// ElipView.cpp : implementation of the C_ElipView class
//

#include "stdafx.h"
#include "Elip.h"

#include "ElipDoc.h"
#include "ElipView.h"
#include "SizeDlg.h"

. . . . .

void C_ElipView::OnDimensions()
{
-->   C_ElipDoc* pDoc = GetDocument();
-->   C_SizeDialog size;
-->   //   set the current values in the dialog box
-->   size.m_nWidth = pDoc->m_Ellipse.m_nWidth;
-->   size.m_nHeight = pDoc->m_Ellipse.m_nHeight;
-->   if (size.DoModal() == IDCANCEL)
-->       return;
-->   //   retrieve the new values from the dialog box
-->   pDoc->m_Ellipse.m_nWidth = size.m_nWidth;
-->   pDoc->m_Ellipse.m_nH = size.m_nHeight;
-->   Invalidate();           //   draws new ellipse
}
```

The user interface is now complete. Rebuild the application, and run it. If you select the “Dimension” menu item, you get the dialog box shown in Figure 10-9.

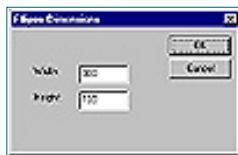


Figure 10-9: The Ellipse Dimensions Dialog Box

Note: If you enter values outside of the valid limits, you will be notified. The notification code has been added automatically, when you set validation limits on the variables. Enter a width of 600, and you will receive the information box shown in Figure 10-10.



Figure 10-10: The Validation Information Box

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

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

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

Keyword

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Printing the View

MFC provides built-in printing support for the view. The printing support that MFC provides to your application is device-independent. This means that the same code written for the OnDraw() function can be used to draw on the screen or to print to the printer. When you ask to print a document using the standard “File | Print” command, MFC calls the OnDraw() member function with a special device context that is aware of the current printer and knows how to translate your screen display into appropriate printed output. The function OnDraw() is called by the framework to render an image of the document. The framework calls this function to perform screen display, printing, and print preview, passing a different device context in each case. If you are printing, OnDraw() is called by another CView class, OnPrint(), with a printer device context. If you are displaying, OnPaint() calls OnDraw(), and the device context is of class CPaintDC, which is for the display context. In the print preview mode, the CDC object is linked to another device context object of class CPreviewDC, but that linkage is transparent to the user. The OnPrint() and OnDraw() functions work the same regardless of whether you are printing or previewing.

The Function OnPrepareDC()

At this point in the development of our application “Elip,” choose “File | Print” from the menu and you will get a very small ellipse. This is because our drawing units are display pixels (also known as device coordinates), and when we put these on a laser printer with 300 or 600 dpi it makes a very small figure indeed. To draw graphics, we need to provide scale factors, which means that we need to change the mapping mode from the default value, which is MM_TEXT.

To change the mapping mode for printing, we need to change the mapping mode in the CView::OnPrepareDC() function. The OnPrepareDC() function is called for OnPaint(), for OnDraw(), and for OnPrint(). The OnPrepareDC() function is called in OnPaint() immediately before the call to OnDraw(). The mapping mode is set before painting the view. If you are printing, the same OnPrepareDC() function is called, this time

immediately before the application framework calls OnPrint(). The mapping mode is set before the printing of the page. We want to change the mapping mode only if we are printing, so we will call the CDC::IsPrinting() function and if we are printing, we will then override the function and change the mapping mode.

The function OnPrepareDC() has two parameters. The first is the device context. The second parameter is a pointer to a CPrintInfo structure, which is valid only if OnPrepareDC() is being called prior to printing. Test for this condition by calling the CDC function, IsPrinting().

For our “Elip” application, we need to override the CView’s virtual function OnPrepareDC(). We will first insert the function’s prototype in the ElipView.h file, since AppWizard did not provide it. We add the following code, given in bold and preceded by an arrow, into the ElipView.h file:

```
protected:  
-->    virtual void OnPrepareDC(CDC* pDC, CPrintInfo* pInfo);
```

We will next discuss mapping modes, because we need to decide which mapping mode to use in our overriding function, OnPrepareDC().

Mapping Modes

Graphics and text output are passed coordinates in the device context that specify where they should be painted. We have been using the default system (MM_TEXT) where the units are called *device units* and we have one screen pixel, or printer dot, per unit. For the screen, device units are the number of pixels measured from the upper left corner of a window area. For a printer, device units are the number of dots measured from the upper left corner of the printed page. As previously demonstrated, when we translate screen pixels to printer dots, we have a very tiny picture.

Mapping modes permit the user to define *logical units*, and then the operating system figures out how to map it to your selected output device. Logical units can be in inches or millimeters, in which case the operating system figures out how big an inch or a millimeter is (in screen pixels or printer dots), and plots or prints the output there. A mapping mode is known as fixed scale when its logical units are specified in inches or millimeters. Table 10-2 gives the mapping modes and their meaning.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes
(Publisher: Wordware Publishing, Inc.)
Author(s): Shirley Wodtke
ISBN: 1556225121
Publication Date: 01/01/97

Bookmark It

Search this book:

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

Table 10-2: Mapping Modes

<i>Mapping Mode</i>	<i>Meaning</i>
MM_TEXT	Default mapping mode; each logical unit equals one device unit (screen pixel or printer dot). X increases to the right, Y increases downward.
Fixed Scale Mapping Modes: (For all fixed scale modes, X increases to the right, Y increases upward.)	
MM_LOENGLISH	Each logical unit is 0.01 inch (low resolution).
MM_HIENGLISH	Each logical unit is 0.001 inch (high resolution).
MM_LOMETRIC	Each logical unit is 0.1 millimeter (low resolution).
MM_HIMETRIC	Each logical unit is 0.01 millimeter (high resolution).
MM_TWIPS	Used with text fonts. Each logical unit is 1/20 point, or 1/1440 of an inch.
Variable Scale:	
MM_ISOTROPIC	Arbitrary scaling of the axes, but the X and Y scaling must be the same.
MM_ANISOTROPIC	Either axis can have any scaling factor. This is the most flexible mode.

Two mapping modes, MM_ISOTROPIC and MM_ANISOTROPIC, allow you to change the scale factor as well as the origin. When you change the scaling factor, you can think of it as stretching or compressing the coordinate system. With the MM_ISOTROPIC mode, a 1:1 aspect ratio is always preserved. In other words, a circle is always a circle as the scale factor changes. With the MM_ANISOTROPIC mode, the X and Y scale factors can change independently; circles can be “squished” into ellipses. For our purposes, we

will use the MM_ISOTROPIC mode to print graphics.

With the scaleable mapping modes, you use the two functions CDC::SetWindowExt(X,Y) and CDC::SetViewportExt(X,Y) together to set the scaling of the coordinate system. For example, to create a coordinate system where each logical unit is five times the default device unit coordinates (screen pixels or printer dots), use the code as shown in the following code excerpt. Open the file ElipView.cpp and add the following overriding function code:

```
-->void C_ElipView::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
-->{
-->    if (pDC->IsPrinting())
-->    {
-->        pDC->SetMapMode(MM_ISOTROPIC);
-->        pDC->SetViewportExt(5, 5);
-->        pDC->SetWindowExt(1, 1);
-->        pDC->SetViewportOrg(10, 10);
-->    }
-->}
```

The functions SetWindowExt(X,Y) and SetViewportExt(X,Y) work together to set the scale, based on the window's scale. SetWindowExt(X,Y) defines the window's coordinates. SetViewportExt(X,Y) defines the printer's coordinates. The scale factor is the ratio of the values provided in these two functions. This could have been done using floating point values, except Windows does not use floating point values for any function parameters. So Windows uses the ratios of the integers to pass the scaling factors. To stretch the output by 500 percent, we use the 5 to 1 ratio. To stretch the output by 66 percent, we could have used a 3 to 2 scaling. To compress the output by 66 percent, we could have used a 2 to 3 scaling. (A 20 to 30 scaling is identical to a 2 to 3 scaling.)

SetViewportOrg(X,Y) sets the upper left margin on the printed page; in this case, we wanted a (10, 10) margin. With this mapping mode in place, the printed figures will be larger and fill most of the page. You can experiment with your printer to determine which scaling factors work best for your output device.

Functions for Printing

The CView class defines several member functions that are called by the framework during printing. By overriding these functions in your view class, you provide the connections between the framework's printing logic and your view class's printing logic. Table 10-3 lists these member functions in the order in which they are called by the framework. AppWizard has automatically inserted the stubbed-out OnPreparePrinting(), OnBeginPrinting(), and OnEndPrinting() functions in your code.

Table 10-3: Cview Member Functions Overridden for Printing

Function	Reason for Overriding
OnPreparePrinting()	To insert values in the Print dialog box, especially the length of the document.
OnBeginPrinting()	To allocate fonts or other GDI resources.

OnPrepareDC()	To adjust attributes of the device context for a given page, or to do print-time pagination.
OnPrint()	To print a given page.
OnEndPrinting()	To deallocate GDI resources.

The framework stores much of the information about a print job in a CPrintInfo structure. Several of the values in CPrintInfo pertain to pagination and are accessible through the member function or member variables as shown in Table 10-4.

Table 10-4: Access to Page Number Information in CPrintInfo

Member Variable/Function Name	Reference to Page Number
GetMinPage()/SetMinPage()	First page of document
GetMaxPage()/SetMaxPage()	Last page of document
GetFromPage()	First page to be printed
GetToPage()	Last page to be printed
m_nCurPage	Page currently being printed

A maximum page number must be inserted in the print dialog box. If the maximum page is unspecified, it spools through endless pages. Since it is too easy to forget to enter the maximum page, this problem can be fixed by giving the framework the maximum page to enter into the print dialog box. To do this, use the overriding OnPreparePrinting() function, and within that function set the maximum page. This is done with the following code, where once again the code which is not bold has been already placed there by AppWizard and the bold code preceded by the arrow is code that you must add. Once this code is in your program, the print dialog box has disabled the “Print Range From and To,” and the endlessly spooling behavior is cured.

```

BOOL C_ElipView::OnPreparePrinting(CPrintInfo* pInfo)
{
-->    pInfo->SetMaxPage(1);
    return DoPreparePrinting(pInfo);
}

```

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

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

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

Keyword

☒ Brief ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

Print Preview and Print Setup

When you choose “File | Print Preview,” you are accessing a lot of already-coded capability, which the MFC framework supplies. AppWizard has inserted all the necessary code into your application to access the print preview capability. Print Preview shows a reduced image of either one or two pages of a document as it would appear when printed on the currently selected printer. It provides a standard user interface for navigating between pages, toggling between one- and two-page viewing, and zooming the display in and out to different levels of magnification.

Suppose we construct an ellipse that fills the window. Then, we select “File | Print Preview” and get the display shown in Figure 10-11.



Figure 10-11: Print Preview

When you choose “File | Print Setup,” the MFC framework automatically provides a dialog box from which you can make selections. You do not have to supply any of your own code for this; AppWizard has inserted all the code necessary for your application to access this capability. Figure 10-12 shows the “Print Setup” dialog box.



Figure 10-12: Print Setup

Data Persistence—Filing the Document’s Data

So far, the data belonging to the `m_Ellipse` object will vanish as soon as the application is closed. We need a way to save this data in files. The `Serialize()` function is the means to achieve this.

Since the document data consists of just one object `m_Ellipse` of class `C_Ellipse`, we need to call the `Serialize()` member function of the class `C_Ellipse`. We need to add the declaration for the `Serialize()` function to the `C_Ellipse` class definition. To do this, add the following lines of code preceded by arrows to the `C_Ellipse` class definition in the `ElipDoc.h` file:

```
//
//  ElipDoc.h : interface of the C_ElipDoc class
//
```

```
class C_Ellipse : public CObject
{
    . . . . .
//  Operations
public:
--> virtual void Serialize(CArchive& ar);
};
```

Modify the Serialize() function of the class C_ElipDoc which AppWizard has already introduced into the ElipDoc.cpp file as follows:

```
//
//  ElipDoc.cpp : implementation of the C_ElipDoc class
//
    . . . . .

void C_ElipDoc::Serialize(CArchive& ar)
{
--> m_Ellipse.Serialize(ar);
}
```

Serialization and CArchive

The process of saving objects to a file and restoring objects from a file is called *serialization*. All the data associated with a serializable object is sequentially read from or written to a single disk file. It is not possible to access individual data at random disk file addresses.

All you have to do in the Serialize() function is to load data from or store data in an archive object. The disk file on which the data will be stored is represented by an object of class CFile; between the Serialize() function and the CFile object is an archive object of class CArchive. The application's CArchive object is named "ar." The CArchive::IsStoring() member function tells us whether the archive is currently being used for storing to a file or loading from a file. The CArchive class has overloaded insertion operators (<<) and extraction operators (>>) for the data types: byte, word, long, dword, float, double, and CObject* (user-defined data type).

For Win32 applications, the data type int is also supported. For Win3.1 applications, the CArchive class does not support the int data type, because ints are different sizes on different hardware. If you have an int data type in a Win3.1 application, you cast it to a type that is supported prior to filing it and recast it from that type to an int when you retrieve it.

MFC library classes that are not derived from CObject, such as CString and CRect, have their own overloaded insertion and extraction operators so that they can be used with CArchive.

In our "Elip" example, we want to store the ellipse data to a file and retrieve it. To do this, add the following implementation of the Serialize() function to the implementation file "ElipDoc.cpp":

```
-->void C_Ellipse::Serialize(CArchive& ar)
-->{
-->    // C_Ellipse objects contain member variables that are of type
-->    // int the operators << and >> do not work in Win3.1 with
-->    // type int, so we need to convert these member variables to
-->    // type WORD
-->    //
```

```

--> // Note that Win32 apps do not need to use this conversion
--> // since their serialize function will support ints.
--> //
--> // The code given here will work for either a
--> // Win3.1 or Win32 application

--> WORD b, w, h, r; // temporary conversion variables

--> if (ar.IsStoring())
--> {
-->     b = m_nBrushHatch;
-->     ar << b;
-->     w = m_nWidth;
-->     ar << w;
-->     h = m_nHeight;
-->     ar << h;
-->     r = m_nRotation;
-->     ar << r;
--> }
--> else
--> {
-->     ar >> b;
-->     m_nBrushHatch = (int)b;
-->     ar >> w;
-->     m_nWidth = (int)w;
-->     ar >> h;
-->     m_nHeight = (int)h;
-->     ar >> r;
-->     m_nRotation = (int)r;
--> }
-->}

```

Now, rebuild the application and run it. Experiment with the use of the File menu items: “Open,” “Save,” and “Save As...”. For example, save an ellipse as the file Ellipse1.ell. Change the ellipse that is drawn in the client area. Then select “File | Open” and open the document stored in the file Ellipse1.ell; you will see the saved document reappear on the screen. All the storing to and from the disk is working properly, and this is all the result of the preceding simple implementation of the Serialize() function; all the other details are taken care of by the framework code.

Also, note that the “Most Recently Used” file list on the File menu is operational. The document files are all saved with the extension .ell, since that is the extension that we asked for. You must ask for a separate name for each file saved; otherwise, it will write over the default file, which is named Ellipse.ell.

Note: The association between the files with the extension .ell and the executable program has been established. What that means is that you can launch your program from the Windows Explorer or the File Manager by double-clicking on a file with the extension .ell. The program will open, and it will be initialized with the document stored in that file. Further, you can drag and drop files from the Windows Explorer or the File Manager.

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

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

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

Keyword

☒ Brief ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97



Search this book:

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

The OnNewDocument() Function

At this juncture, the menu selection “File | New” does not work as expected. The function `CDocument::OnNewDocument()` should always be used to initialize data in a new document; this function is called automatically when the user selects “File | New” from the menu. We have initialized the ellipse’s data members in the constructor, but the constructor gets called only once for the lifetime of the program. Yet the user can select “File | New” from the menu and expects to see the same ellipse that was initialized when the application opened. In an SDI application, the document object is being reused each time a different file is opened. Thus, when the user selects “File | New” we want to reinitialize the document’s contents. We do this in the function `OnNewDocument()`. AppWizard has already inserted this stubbed-out function into our code, so all we need to do is to customize it. This is done with the following code, where the non-bold is what is already in the code and

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

The Create() function for a CSplitterWnd object has five input parameters. The first parameter is the parent window, for which we use the keyword *this*, which establishes the MainFrame window as the parent of the splitter window. The second parameter is the number of rows (we asked for two). The third parameter is the number of columns (again, we asked for two). The fourth parameter is the minimum size to which any pane is allowed to shrink. The fifth parameter is a pointer to which is a pass-through. The pointer, pContext, was supplied as an input parameter to OnCreateClient(), and we pass it on to the Create() function.

Note: In Microsoft's Visual C++ 4.0, the AppWizard will add the splitter window code, plus it adds a menu item named "View" that you can use to split the window, if you specify the splitter window option at the time that you are creating the starter application (see Appendix D). In our case, we added it into our application afterwards. Most compilers do not support the automatic addition of splitter window code.

Recompile the program, run it, create two columns and two rows with the splitter bars. It looks like Figure 10-13, which depicts four views of the same document.



Figure 10-13: The Elip Program with Dynamic Splitter Windows

As previously discussed, the document keeps a list of pointers to all of its views. This process is depicted in Figure 10-14. Of course, the user is dynamically creating and decreasing the views. The user could have made only one row with two columns, in which case the document has only two views attached to it. The number of window panes attached to a document changes at

run time. The mechanism for creating, decreasing, and keeping track of the document's multiple views is all supplied by the framework.

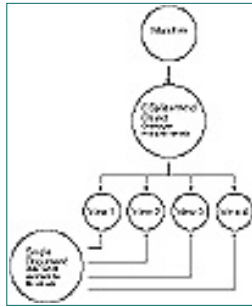


Figure 10-14: Multiple Views of Single Document

Now change the ellipse's brush hatch to upward and see what happens. Only pane 0 changes, as shown in Figure 10-15. This is because pane 0 is the original window and the command handler functions only update that window. So how do we fix this situation? The views all need to be synchronized.

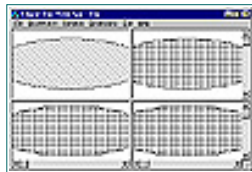


Figure 10-15: Pane 0 Updated by Selection From Menu

The function `CDocument::UpdateAllViews()` is designed just for synchronizing. Whenever you update anything in the document, tell the document to update all of its views. The document maintains a list of pointers to all of its views, and calling the function `UpdateAllViews()` causes it to cycle through all of its views and update them. In order to maintain the same view in all panes, the `UpdateAllViews()` function must be included in each command handler function that changes the document's data. So you must add this code to all of your command handler functions that change the document's data. An example of the code change follows. The non-bold code is code that you have previously added to your program. The code in bold and preceded by an arrow is the code that you want to add now.

```

void C_ElipView::OnBrushhatchCross()
{
    C_ElipDoc* pDoc = GetDocument();
    pDoc->m_Ellipse.m_nBrushHatch = HS_CROSS;
    - -> pDoc->UpdateAllViews(NULL);
        Invalidate();
}

```

ElipsMin Program with Minimum Code

In this section, an application is generated (not using AppWizard) which contains the document-view architecture with all the features of the "Elip" application in order to demonstrate the minimum amount of code necessary for this program. This manually generated program has the exact style of AppWizard-generated code, including all the names used in the previous

“Elip” program, so that you can make a one-to-one comparison with the AppWizard code. This allows you to see which class is participating in any given task; for the Borland C++ programmers, it represents the program that needs to be generated for this application, since Borland C++ does not have compiler tools for MFC.

In this book, three example programs of manually generated document-view applications are given. In Chapter Nine, the first one was presented in which the window features were customized. In this chapter, “ElipsMin” is presented, which covers printing and filing for an SDI application. In Chapter Twelve, the third manually generated document-view application is presented, which covers an MDI application.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

ElipsMin Program Listing

Listing 10-1 gives the “ElipsMin” program, which is followed by a discussion of important features to note about this code.

Listing 10-1: Source Code for the ElipsMin Program

```

-----
                                     stdafx files
-----
//
//   stdafx.h : include file for system include files,
//
#include <afxwin.h>           // MFC standard components
#include <afxext.h>           // MFC extensions

//
//   stdafx.cpp : source file for standard includes
//
#include "stdafx.h"

-----
                                     application class
-----
//
//   Elip.h : main header file for the application
//
#include "resource.h"

class C_ElipApp : public CWinApp
{
public:
    virtual BOOL InitInstance();

    DECLARE_MESSAGE_MAP()
};

//
//   Elip.cpp : Defines the class behaviors for the application.
//
#include "stdafx.h"
  
```

```

#include "Elip.h"
#include "MainFrm.h"
#include "ElipDoc.h"
#include "ElipView.h"

BEGIN_MESSAGE_MAP(C_ElipApp, CWinApp)
    // Standard file based document commands
    ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
    ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
    // Standard print setup command
    ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()

C_ElipApp theApp; // instantiate the object that runs the program

BOOL C_ElipApp::InitInstance()
{
    LoadStdProfileSettings(); //Load standard INI file options (incl. MRU)

    CSingleDocTemplate* pDocTemplate;
    pDocTemplate = new CSingleDocTemplate(
        IDR_MAINFRAME,
        RUNTIME_CLASS(C_ElipDoc),
        RUNTIME_CLASS(C_MainFrame),
        RUNTIME_CLASS(C_ElipView));
    AddDocTemplate(pDocTemplate);

    SetDialogBkColor();

    // enable DDE Execute open
    EnableShellOpen();
    RegisterShellFileTypes();

    // simple command line parsing
    if (m_lpCmdLine[0] == '\\0')
    {
        // create a new (empty) document
        OnFileNew();
    }
    else
    {
        // open an existing document
        OpenDocumentFile(m_lpCmdLine);
    }
    // enable drag/drop open
    m_pMainWnd->DragAcceptFiles();

    return TRUE;
}

-----
                                mainframe class
-----
//
//  MainFrm.h : interface of the C_MainFrame class
//
class C_MainFrame : public CFrameWnd
{
    DECLARE_DYNCREATE(C_MainFrame)
public:

```

```

        CSplitterWnd m_wndSplitter;
protected:
        virtual BOOL OnCreateClient(LPCREATESTRUCT lpcs,
                                    CCreateContext* pContext);
};

//
// MainFrm.cpp : implementation of the C_MainFrame class
//
#include "stdafx.h"
#include "Elip.h"
#include "MainFrm.h"

IMPLEMENT_DYNCREATE(C_MainFrame, CFrameWnd)

BOOL C_MainFrame::OnCreateClient(LPCREATESTRUCT lpcs,
                                CCreateContext* pContext)
{
    return m_wndSplitter.Create(this, 2, 2, CSize(1,1), pContext);
}

-----
                                view class
-----
//
// ElipView.h : interface of the C_ElipView class
//
class C_ElipView : public CView
{
    DECLARE_DYNCREATE(C_ElipView)
public:
    C_ElipDoc* GetDocument();
    virtual void OnDraw(CDC* pDC);
protected:
    virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
    virtual void OnPrepareDC(CDC* pDC, CPrintInfo* pInfo);
    //{AFX_MSG(C_ElipView)
    afx_msg void OnBrushhatchCross();
    afx_msg void OnUpdateBrushhatchCross(CCmdUI* pCmdUI);
    afx_msg void OnBrushhatchUpward();
    afx_msg void OnUpdateBrushhatchUpward(CCmdUI* pCmdUI);
    afx_msg void OnBrushhatchVertical();
    afx_msg void OnUpdateBrushhatchVertical(CCmdUI* pCmdUI);
    afx_msg void OnDimensions();
    afx_msg void OnRotation180();
    afx_msg void OnUpdateRotation180(CCmdUI* pCmdUI);
    afx_msg void OnRotation90();
    afx_msg void OnUpdateRotation90(CCmdUI* pCmdUI);
    //}}AFX_MSG

    DECLARE_MESSAGE_MAP()
};

#ifdef _DEBUG // debug version in ElipView.cpp
inline C_ElipDoc* C_ElipView::GetDocument()
    { return (C_ElipDoc*)m_pDocument; }
#endif
//

// ElipView.cpp : implementation of the C_ElipView class

```

```

//
#include "stdafx.h"
#include "Elip.h"
#include "ElipDoc.h"
#include "ElipView.h"
#include "SizeDlg.h"

IMPLEMENT_DYNCREATE(C_ElipView, CView)

BEGIN_MESSAGE_MAP(C_ElipView, CView)
   //{{AFX_MSG_MAP(C_ElipView)
    ON_COMMAND(ID_BRUSHHATCH_CROSS, OnBrushhatchCross)
    ON_UPDATE_COMMAND_UI(ID_BRUSHHATCH_CROSS, OnUpdateBrushhatchCross)
    ON_COMMAND(ID_BRUSHHATCH_UPWARD, OnBrushhatchUpward)
    ON_UPDATE_COMMAND_UI(ID_BRUSHHATCH_UPWARD, OnUpdateBrushhatchUpward)
    ON_COMMAND(ID_BRUSHHATCH_VERTICAL, OnBrushhatchVertical)
    ON_UPDATE_COMMAND_UI(ID_BRUSHHATCH_VERTICAL, OnUpdateBrushhatchVertical)
    ON_COMMAND(ID_DIMENSIONS, OnDimensions)
    ON_COMMAND(ID_ROTATION_180, OnRotation180)
    ON_UPDATE_COMMAND_UI(ID_ROTATION_180, OnUpdateRotation180)
    ON_COMMAND(ID_ROTATION_90, OnRotation90)
    ON_UPDATE_COMMAND_UI(ID_ROTATION_90, OnUpdateRotation90)
    //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()

void C_ElipView::OnDraw(CDC* pDC)
{
    C_ElipDoc* pDoc = GetDocument();
    CRect r;
    GetClientRect(&r);
    pDoc->m_Ellipse.BoundingRect(r);
    pDoc->m_Ellipse.Rotate();
    CBrush newBrush(pDoc->m_Ellipse.m_nBrushHatch, RGB(0, 0, 0));
    CBrush* pOldBrush = pDC->SelectObject(&newBrush);
    pDC->Ellipse(pDoc->m_Ellipse.m_RotatedPos);
    pDC->SelectObject(pOldBrush);
}

BOOL C_ElipView::OnPreparePrinting(CPrintInfo* pInfo)
{
    pInfo->SetMaxPage(1);
    return DoPreparePrinting(pInfo);
}

void C_ElipView::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
    if (pDC->IsPrinting())
    {
        pDC->SetMapMode(MM_ISOTROPIC);
        pDC->SetViewportExt(5, 5);
        pDC->SetWindowExt(1, 1);
        pDC->SetViewportOrg(10, 10);
    }
}

void C_ElipView::OnBrushhatchCross()
{

```

```

        C_ElipDoc* pDoc = GetDocument();
        pDoc->m_Ellipse.m_nBrushHatch = HS_CROSS;
        pDoc->UpdateAllViews(NULL);
        Invalidate();
    }

void C_ElipView::OnUpdateBrushhatchCross(CCmdUI* pCmdUI)
{
    C_ElipDoc* pDoc = GetDocument();
    if (pDoc->m_Ellipse.m_nBrushHatch == HS_CROSS)
        pCmdUI->SetCheck(1);
    else
        pCmdUI->SetCheck(0);
}

void C_ElipView::OnBrushhatchUpward()
{
    C_ElipDoc* pDoc = GetDocument();
    pDoc->m_Ellipse.m_nBrushHatch = HS_FDIAGONAL;
    pDoc->UpdateAllViews(NULL);
    Invalidate();
}

void C_ElipView::OnUpdateBrushhatchUpward(CCmdUI* pCmdUI)
{
    C_ElipDoc* pDoc = GetDocument();
    if (pDoc->m_Ellipse.m_nBrushHatch == HS_FDIAGONAL)
        pCmdUI->SetCheck(1);
    else
        pCmdUI->SetCheck(0);
}

void C_ElipView::OnBrushhatchVertical()
{
    C_ElipDoc* pDoc = GetDocument();
    pDoc->m_Ellipse.m_nBrushHatch = HS_VERTICAL;
    pDoc->UpdateAllViews(NULL);
    Invalidate();
}

void C_ElipView::OnUpdateBrushhatchVertical(CCmdUI* pCmdUI)
{
    C_ElipDoc* pDoc = GetDocument();
    if (pDoc->m_Ellipse.m_nBrushHatch == HS_VERTICAL)
        pCmdUI->SetCheck(1);
    else
        pCmdUI->SetCheck(0);
}

void C_ElipView::OnDimensions()
{
    C_ElipDoc* pDoc = GetDocument();
    C_SizeDialog size;

    // set the current values into the dialog box
    size.m_nWidth = pDoc->m_Ellipse.m_nWidth;
    size.m_nHeight = pDoc->m_Ellipse.m_nHeight;
    if (size.DoModal() == IDCANCEL)
        return;
}

```

```

        // retrieve the new values from the dialog box
        pDoc->m_Ellipse.m_nWidth = size.m_nWidth;
        pDoc->m_Ellipse.m_nHeight = size.m_nHeight;
        pDoc->UpdateAllViews(NULL);
        Invalidate();
    }

void C_ElipView::OnRotation180()
{
    C_ElipDoc* pDoc = GetDocument();
    pDoc->m_Ellipse.m_nRotation = ROTATED180;
    pDoc->UpdateAllViews(NULL);
    Invalidate();
}

void C_ElipView::OnUpdateRotation180(CCmdUI* pCmdUI)
{
    C_ElipDoc* pDoc = GetDocument();
    if (pDoc->m_Ellipse.m_nRotation == ROTATED180)
        pCmdUI->SetCheck(1);
    else
        pCmdUI->SetCheck(0);
}

void C_ElipView::OnRotation90()
{
    C_ElipDoc* pDoc = GetDocument();
    pDoc->m_Ellipse.m_nRotation = ROTATED90;
    pDoc->UpdateAllViews(NULL);
    Invalidate();
}

void C_ElipView::OnUpdateRotation90(CCmdUI* pCmdUI)
{
    C_ElipDoc* pDoc = GetDocument();
    if (pDoc->m_Ellipse.m_nRotation == ROTATED90)
        pCmdUI->SetCheck(1);
    else
        pCmdUI->SetCheck(0);
}

}

#ifdef _DEBUG
C_ElipDoc* C_ElipView::GetDocument() // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(C_ElipDoc)));
    return (C_ElipDoc*)m_pDocument;
}
#endif

-----
                                document class
-----
//
// ElipDoc.h : interface of the C_ElipDoc class
//
class C_Ellipse : public CObject
{
    DECLARE_SERIAL(C_Ellipse)
public:

```

```

    C_Ellipse();
    int m_nBrushHatch;
    int m_nWidth;
    int m_nHeight;
    CPoint m_EllipseCenter;    // center of client area
    CRect m_OriginalPos;       // original bounding rect
    CRect m_RotatedPos;        // rotated bounding rect
    int m_nRotation;           // current ellipse rotation
    // the two possible rotations of the ellipse
#define ROTATED90 1
#define ROTATED180 2

    void BoundingRect(CRect& r);
    void Rotate();
    virtual void Serialize(CArchive& ar);
};

class C_ElipDoc : public CDocument
{
    DECLARE_DYNCREATE(C_ElipDoc)
public:
    C_Ellipse m_Ellipse;
    virtual BOOL OnNewDocument();
    virtual void Serialize(CArchive& ar);
};

//
// ElipDoc.cpp : implementation of the C_ElipDoc class
//
#include "stdafx.h"
#include "Elip.h"
#include "ElipDoc.h"

//
// C_Ellipse
//
IMPLEMENT_SERIAL(C_Ellipse, CObject, 1)

C_Ellipse::C_Ellipse()
{
    m_nBrushHatch = HS_CROSS;
    m_nWidth = 300;
    m_nHeight = 100;
    m_nRotation = ROTATED180;
}

void C_Ellipse::BoundingRect(CRect& r)
{
    m_EllipseCenter.x = r.Width() / 2;
    m_EllipseCenter.y = r.Height() / 2;
    CSize HalfSizeOrig(m_nWidth / 2, m_nHeight / 2);
    m_OriginalPos.TopLeft() = m_EllipseCenter - HalfSizeOrig;
    m_OriginalPos.BottomRight() = m_EllipseCenter + HalfSizeOrig;
}

void C_Ellipse::Rotate()
{
    switch (m_nRotation)
    {
        case ROTATED180:

```



```

        m_RotatedPos = m_OriginalPos;
        break;
    case ROTATED90:
        CSize HalfSizeRotated(m_nHeight / 2, m_nWidth / 2);
        m_RotatedPos.TopLeft() = m_EllipseCenter - HalfSizeRotated;
        m_RotatedPos.BottomRight() = m_EllipseCenter + HalfSizeRotated;
        break;
    }
}

void C_Ellipse::Serialize(CArchive& ar)
{
    // C_Ellipse objects contain member variables that are of type
    // int; the operators << and >> do not work with type int for Win3.1,
    // so convert these member variables to type WORD

    WORD b, w, h, r;    // temporary conversion variables

    if (ar.IsStoring())
    {
        b = m_nBrushHatch;
        ar << b;
        w = m_nWidth;
        ar << w;
        h = m_nHeight;
        ar << h;
        r = m_nRotation;
        ar << r;
    }
    else
    {
        ar >> b;
        m_nBrushHatch = (int)b;
        ar >> w;
        m_nWidth = (int)w;
        ar >> h;
        m_nHeight = (int)h;
        ar >> r;
        m_nRotation = (int)r;
    }
}

//
// C_ElipDoc
//
IMPLEMENT_DYNCREATE(C_ElipDoc, CDocument)

BOOL C_ElipDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;
    m_Ellipse.m_nBrushHatch = HS_CROSS;
    m_Ellipse.m_nWidth = 300;
    m_Ellipse.m_nHeight = 100;
    m_Ellipse.m_nRotation = ROTATED180;

    return TRUE;
}

void C_ElipDoc::Serialize(CArchive& ar)

```

```

{
    m_Ellipse.Serialize(ar);
}

-----
                                dialog class
-----
//
//  SizeDlg.h : header file
//
class C_SizeDialog : public CDialog
{
public:
    C_SizeDialog(CWnd* pParent = NULL);
    enum { IDD = IDD_DIALOG1 };
    int      m_nWidth;
    int      m_nHeight;
protected:
    virtual void DoDataExchange(CDataExchange* pDX);
};

//
//  SizeDlg.cpp : implementation file
//
#include "stdafx.h"
#include "Elip.h"
#include "SizeDlg.h"

C_SizeDialog::C_SizeDialog(CWnd* pParent)
    : CDialog(C_SizeDialog::IDD, pParent)
{
}

void C_SizeDialog::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    DDX_Text(pDX, IDC_WIDTH, m_nWidth);
    DDV_MinMaxInt(pDX, m_nWidth, 1, 500);
    DDX_Text(pDX, IDC_HEIGHT, m_nHeight);
    DDV_MinMaxInt(pDX, m_nHeight, 1, 500);
}

-----
                                resource files
-----
//
//  resource.h
//
#define IDR_MAINFRAME                128
#define IDD_DIALOG1                  130
#define IDC_WIDTH                    1000
#define IDC_HEIGHT                   1001
#define ID_BRUSHHATCH_UPWARD         32771
#define ID_BRUSHHATCH_VERTICAL       32772
#define ID_BRUSHHATCH_CROSS          32773
#define ID_ROTATION_90                32774
#define ID_ROTATION_180               32775
#define ID_DIMENSIONS                 32776

//

```

```

//      Elip.rc
//
#include "resource.h"
#include "afxres.h"

IDR_MAINFRAME          ICON      DISCARDABLE      "res\\Elip.ico"

IDR_MAINFRAME MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "N&ew\tCtrl+N",          ID_FILE_NEW
        MENUITEM "&Open...\tCtrl+O",      ID_FILE_OPEN
        MENUITEM "&Save\tCtrl+S",          ID_FILE_SAVE
        MENUITEM "Save &As...",            ID_FILE_SAVE_AS
        MENUITEM SEPARATOR
        MENUITEM "&Print...\tCtrl+P",      ID_FILE_PRINT
        MENUITEM "PrintPre&view",          ID_FILE_PRINT_PREVIEW
        MENUITEM "P&rint Setup...",        ID_FILE_PRINT_SETUP
        MENUITEM SEPARATOR
        MENUITEM "Recent File",            ID_FILE_MRU_FILE1, GRAYED
        MENUITEM SEPARATOR
        MENUITEM "E&xit",                  ID_APP_EXIT
    END
    POPUP "&BrushHatch"
    BEGIN
        MENUITEM "&Upward",                ID_BRUSHHATCH_UPWARD
        MENUITEM "&Vertical",              ID_BRUSHHATCH_VERTICAL
        MENUITEM "&Cross",                 ID_BRUSHHATCH_CROSS
    END
    POPUP "&Rotation"
    BEGIN
        MENUITEM "&90",                    ID_ROTATION_90
        MENUITEM "&180",                   ID_ROTATION_180
    END
    MENUITEM "&Dimensions",                ID_DIMENSIONS
END
IDR_MAINFRAME ACCELERATORS PRELOAD MOVEABLE PURE
BEGIN
    "N",          ID_FILE_NEW,          VIRTKEY, CONTROL
    "O",          ID_FILE_OPEN,         VIRTKEY, CONTROL
    "S",          ID_FILE_SAVE,         VIRTKEY, CONTROL
    "P",          ID_FILE_PRINT,        VIRTKEY, CONTROL
END

IDD_DIALOG1 DIALOG DISCARDABLE  0, 0, 186, 95
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Ellipse Dimensions"
FONT 8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON "OK", IDOK, 129, 7, 50, 14
    PUSHBUTTON    "Cancel", IDCANCEL, 129, 24, 50, 14
    EDITTEXT      IDC_WIDTH, 55, 28, 40, 14, ES_AUTOHSCROLL
    EDITTEXT      IDC_HEIGHT, 55, 48, 40, 14, ES_AUTOHSCROLL
    LTEXT         "Width:", IDC_STATIC, 18, 28, 22, 8, NOT WS_GROUP
    LTEXT         "Height:", IDC_STATIC, 18, 48, 34, 8
END

```

STRINGTABLE PRELOAD DISCARDABLE

BEGIN

IDR_MAINFRAME "Elip\nEllipse Doc-View App\nElip\nEllipse(*.ell)
\n.ell\nElip.Document\nElip Document"

END

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Discussion of the “ElipsMin” Program

The application class contains the command handlers for opening files and the print setup. The ON_COMMAND statements with ID_FILE_NEW, ID_FILE_OPEN, and ID_FILE_PRINT_SETUP enable CWinApp::OnFileNew(), CWinApp::OnFileOpen(), and CWinApp::OnFilePrintSetup(), respectively. When enabled, these functions handle the execution of the “File | New,” the “File | Open,” and the “File | Print Setup” menu item commands. The application class also contains the function LoadStdProfileSettings(), which must be called from within InitInstance() to enable and load the current MRU file list and the last preview state.

The view class must contain a function to get the pointer to the document and cast it to the appropriate class. There is a debug version and a non-debug version of this function in the code.

The view class contains the command handlers for printing and print previewing. ID_FILE_PRINT activates CView::OnFilePrint(). ID_FILE_PRINT_PREVIEW activates CView::OnFilePrintPreview().

The menu item commands ID_FILE_SAVE and ID_FILE_SAVE_AS directly access the routine CDocument::DoSave(). The menu item command ID_APP_EXIT also directly accesses its routine CWinApp::OnAppExit().

Making the Dialog Box Modeless

This section presents a discussion on how to make the dialog box modeless. Modeless dialog boxes were covered in Chapter Eight; however, when you are dealing with the four-class document-view architecture making a dialog box modeless is different, so we discuss it again here. Particularly, we need to set up the two-way communication between the view class and the dialog class. The view class needs a pointer to the dialog, and the dialog class needs a pointer to the view class. A dialog box is always parented by the mainframe class, so getting a pointer to the dialog’s parent is of no help. So there

has to be a special way for the dialog class to get a pointer to the view class. In this section, how this is done is shown.

“ElipMdl” is the program that has all the same features as the “Elip” program, except that the dialog box is modeless. Figure 10-16 shows the modeless dialog box, which looks slightly different than the modal dialog box of the program “Elip,” and, of course, it behaves substantially differently. The “ElipMdl” program was created by first creating a project of the type application (non-AppWizard). Then all the source files (including the <res> subdirectory) were imported into this project from the “Elip” program. To make the dialog box modeless rather than modal, the necessary changes were then made in the view class and in the dialog class.



Figure 10-16: The Modeless Dialog Box

Changes need to be made both in the view class and in the dialog class. These changes are shown in the following discussion.

In the view class we need to make the following changes:

1. We will be declaring a pointer to the dialog object as a data member of the view class, so the view's header file needs to know about the dialog class. Remove the following line of code from the ElipView.cpp file and add it to the ElipView.h file:

```
#include "SizeDlg.h"
```

2. The view class maintains the pointer to the dialog object and also the view class will receive a user-defined message when the user has entered new data into the dialog box. In the ElipView.h file add the following data member and member function declaration:

```
C_SizeDialog* m_pDlg;
```

```
LONG OnChangeSize(UINT wParam, LONG lParam);
```

3. The response function to the user-defined message must be entered into the view's message map. In the ElipView.cpp file insert the following message map entry:

```
ON_COMMAND(WM_USER, OnChangeSize)
```

4. The modeless dialog box's C++ object is instantiated on the heap and its pointer is saved. This is all done in the view's constructor. The view must have a destructor to destroy the C++ object when the application ends; we define the handler function that responds to the user-defined message sent to the view when the user has entered data in the dialog box. In the ElipView.cpp file add the following code to the constructor and destructor, and add the OnChangeSize() handler function:

```

C_ElipView::C_ElipView()
{
    m_pDlg = new C_SizeDialog(this);
}
C_ElipView::~~C_ElipView()
{
    delete m_pDlg;
}

LONG C_ElipView::OnChangeSize(UINT wParam, LONG lParam)
{
    C_ElipDoc* pDoc = GetDocument();

    // set the new values into the document
    pDoc->m_Ellipse.m_nWidth = m_pDlg->m_nWidth;
    pDoc->m_Ellipse.m_nHeight = m_pDlg->m_nHeight;

    pDoc->UpdateAllViews(NULL);
    return 1;
}

```

5. The modeless dialog box is opened with a call to its Create() function. In the ElipView.cpp file, replace the OnDimension() handler function with the following code:

```

void C_ElipView::OnDimensions()
{
    C_ElipDoc* pDoc = GetDocument();
    if (m_pDlg->GetSafeHwnd() == 0)
    {
        // set the current values into the dialog box
        m_pDlg->m_nWidth = pDoc->m_Ellipse.m_nWidth;
        m_pDlg->m_nHeight = pDoc->m_Ellipse.m_nHeight;
        m_pDlg->Create();    // display the dialog box
    }
}

```

Note: Use the call to GetSafeHwnd() to determine if a dialog window is already open. If the handle returned from this call is NULL, the window does not exist and it is safe to open one.

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

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



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY IT KNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

[Go!](#)

[Previous](#)

[Table of Contents](#)

[Next](#)

BROWSE
BY TOPIC

In the dialog class, the following changes need to be made:

1. The dialog class maintains a pointer to the view class, so the pointer is declared as a data member. We add a second constructor (the dialog class constructor is overloaded) which the framework is smart enough to select when we instantiate the dialog box on the heap (modeless). We add the Create() function which is used in the modeless dialog box's two-stage construction. In the SizeDlg.h file, add the following data member and member function declarations:

```
CView* m_pView;
```

```
C_SizeDialog(CView* pView);  
BOOL Create();
```

2. We have additional handler functions in the dialog class. In the SizeDlg.h file, insert the following message map function declarations:

```
virtual void OnOK();  
afx_msg void OnClose();  
afx_msg void OnKillFocus();
```

3. Clicking the close button maps to the OnClose() function. When the edit boxes lose focus, the user has made an entry, so we map to its handler function. In the SizeDlg.cpp file, insert the following message map entries:

```
ON_BN_CLICKED(IDCANCEL, OnClose)  
ON_EN_KILLFOCUS(IDC_WIDTH, OnKillFocus)  
ON_EN_KILLFOCUS(IDC_HEIGHT, OnKillFocus)
```

4. When the dialog box is constructed, the framework provides a pointer to the view class as an input parameter, and we save the pointer. The Create() function creates the window and then shows it. We disable the OnOK() function. The OnClose() handler function destroys the window; messages are posted to the view when the user has made a new entry into the edit boxes. In the SizeDlg.cpp file, add the following function definitions:

```

C_SizeDialog::C_SizeDialog(CView* pView)
{
    m_pView = pView;
}

BOOL C_SizeDialog::Create()
{
    BOOL ReturnValue = CDialog::Create(C_SizeDialog::IDD);
    ShowWindow(SW_SHOWNA);
    return ReturnValue;
}

void C_SizeDialog::OnOK()
{
    // Do not call base class OnOK() function.
    // This disables the <enter> key from closing
    // the dialog box.
}

void C_SizeDialog::OnClose()
{
    DestroyWindow();
}

void C_SizeDialog::OnKillFocus()
{
    UpdateData(TRUE);
    m_pView->PostMessage(WM_USER);
}

```

In the resource file, make the following change:

1. The dialog box template was changed to initially appear at (200,200) so that it is not in the way. The tab order was changed so that the focus initially goes to the edit control for the width. The OK button was deleted, and the label on the cancel button was changed to read "Close." The dialog resource script is:

```

IDD_DIALOG1 DIALOG DISCARDABLE 200, 200, 186, 95
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Ellipse Dimensions"
FONT 8, "MS Sans Serif"
BEGIN
    EDITTEXT        IDC_WIDTH, 55, 28, 40, 14, ES_AUTOHSCROLL
    EDITTEXT        IDC_HEIGHT, 55, 48, 40, 14, ES_AUTOHSCROLL
    LTEXT            "Width:", IDC_STATIC, 18, 28, 22, 8, NOT WS_GROUP
    LTEXT            "Height:", IDC_STATIC, 18, 48, 34, 8
    PUSHBUTTON      "Close", IDCANCEL, 129, 24, 50, 14
END

```

Diagnostic Code

As you inspect the code that AppWizard has generated for you, you may notice that a lot of diagnostic code statements have been included. In this section we look at these diagnostic statements.

Any class derived from CObject has access to functions that provide diagnostic services for you. CObject provides two functions for diagnostic support: Dump() and AssertValid(). AppWizard inserts these into the classes that it generates.

The Dump() function dumps diagnostic information about the class to a specified output. These diagnostic dumps can be very useful in your debugging efforts. Dump() displays data about the class. If you do not override it, Dump() will display only the class name and object address. You can override the Dump() function to customize the data; however, you must provide your own detailed dumping mechanism to get more than the minimal data.

The AssertValid() function has a quite different role than Dump(). The AssertValid() function concerns itself with the internal consistency of the class data. If it determines that class data is invalid, inconsistent, or corrupt, the program halts with a message that lists the line number and filename where the assertion failed. AssertValid() does not provide any data unless something is very wrong. When you override AssertValid() for your own classes, you will use the ASSERT_VALID(pObject) macro to confirm invariant conditions associated with the class data. In the Release version, ASSERT_VALID() does nothing. In the Debug version, it displays an alert message if any test fails.

Diagnostic support is for the Debug version only, so the diagnostic code is enclosed in preprocessor directives that check for the debug mode. When you switch over to the Release version of the program, these calls will be invisible and ignored.

Another convenient macro to keep in mind is TRACE. The TRACE macro is separate from the diagnostic functions previously discussed. You will need to provide your own TRACE statements. The TRACE macro allows you to dump diagnostic information directly to the debug output. TRACE uses the same parameters as the printf() function. The TRACE macro is automatically ignored by the Release version of the compiler. To use the TRACE macro, your class must be derived directly or indirectly from CObject. The TRACE macro is used in Chapter Twelve.

Previous	Table of Contents	Next

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

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

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

Keyword

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Summary

The four-class document-view architecture provides important built-in capabilities. You can program your four-class document-view application entirely manually, or you may use a compiler tool to get a starter application to which you must then add your application's code. The IDR_MAINFRAME resource contains a document template string which is separated into seven substrings by newline characters; these substrings show up in various places when the application executes. You may wish to edit this string to get the results that you want.

The application's code must be separated into its data, which consists of all the information that you file and subsequently restore from file to start your application. The data of any application is kept in the document class. The document class takes care of storing to file and retrieving from file. The class that takes care of the user input and interaction with the document is the view class. The view class takes care of drawing to the screen and printing the view.

MFC provides built-in printing support for the view. The printing support is device-independent; that is, the OnDraw(CDC*) function is called by the framework to render an image of the document. The framework calls this function to perform screen display, printing, and print preview, passing a different device context in each case. Your view class inherits several member functions that are called by the framework during printing. By overriding these functions in your view class, you can customize the printing as needed. The function OnPreparePrinting() is overridden to insert values in the Print dialog box. The function OnBeginPrinting() is overridden to allocate GDI resources, such as fonts. The function OnPrepareDC() is overridden to adjust the attributes of the device context. The function OnEndPrinting() is overridden to deallocate the GDI resources.

The framework provides built-in support for the document class to file the data

and for the document class to retrieve the data from file. In order to file data, you must provide a Serialize() function. The process of saving objects to a file and restoring objects from a file is called serialization. All the data associated with a serializable object is sequentially read from or written to a single disk file. An archive object is used with the Serialize() function; all you have to do in the Serialize() function is to load data from or store data in an archive object.

Views can be made into multiple views (panes) during run time by the user; to do this you use a dynamic splitter window. CSplitterWnd is the class than manages split windows. CSplitterWnd provides pane-splitting controls and scroll bars and creates and de-creates the multiple panes. When the document's data is changed, the views in all panes must be synchronized to reflect the latest data; the function CDocument::UpdateAllViews() is used to synchronize the data displayed in all the panes.

Dialog boxes can be modeless, but their construction is different than that of a modal dialog; it is a two-stage construction. A modeless dialog is instantiated on the heap in the view's constructor and the pointer to the dialog is maintained as a member variable of the view class. The pointer to the dialog is used for transferring data to and from the dialog's member variables. Then the modeless dialog is displayed using a call to its Create() function, which calls the dialog class's constructor. When a modeless dialog is being constructed, the framework passes the pointer to the view class as an input to its constructor; the dialog class maintains the pointer to the view class as its member variable. The pointer to the view class is used for sending user-defined messages from the dialog to the view.

Previous	Table of Contents	Next

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

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

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

Keyword

☒ Brief
 ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Chapter 11

More About Splitter Windows and Filing

In the previous chapter, we first had only one view attached to a document, then we added a dynamic splitter which produced multiple views of the same document. In this chapter we do a tic tac toe game in a program named “Tac,” which has two different view classes where each view class displays different information. Both views are shown simultaneously on the screen using a static splitter window. MFC has a couple of ways to present multiple views: the splitter window and MDI child windows.

Splitter windows can be either dynamic where all panes display the same view, as we saw in Chapter Ten, or they can be static where each pane can display its own view class, as we will see in this chapter. In the next two chapters, which cover Multiple Document Interface (MDI) applications, you will see that it is easy to make multiple windows of the same view of the same data using the MDI child windows. What we do in this chapter is more difficult, which is to use two view classes, where each class displays different data and simultaneously displays both views using a static splitter window.

The example presented in this chapter, which is a tic tac toe game, also covers additional new subjects. We look at another method of allocating your application’s code between the document and the view classes. We learn how to create our own customized fonts and we use the MFC class CFont. Additionally, we learn how to use the *collection classes* for filing a document.

The example tic tac toe game that we build in this chapter uses a static splitter window that has one row and two columns, and a different view class is drawn in each pane. We build the tic tac toe game as an SDI application named

“Tac.” Figure 11-1 shows what the finished tic tac toe game will look like when it is first started and before any moves have been made.

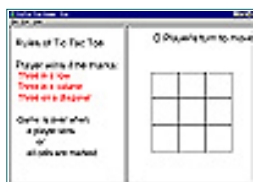


Figure 11-1: The Tic Tac Toe Game Example

The window is split into two static panes. The left pane displays the rules of the game. The right pane contains the 3 x 3 tic tac toe board. The play of the game proceeds when the user clicks the left mouse button on one of the cells of the tic tac toe board. The play of the game can be filed before it is finished, a new game can be started by choosing “New” from the “File” menu, and later the original game that was filed can be opened and finished.

In this “Tac” program, we associate a number with each square of the tic tac toe board like this:

1	2	3
4	5	6
7	8	9

We will use these numbers in drawing the board and in maintaining and filing the play of the game.

The Starter Application

A starter application is first created for this SDI application using AppWizard. Consult the appropriate appendix for your compiler. Visual C++ 4 is discussed here; details vary if you are using a different compiler. From the menu, choose “File | New | Project Workspace.” In the “New Project” query box (see Appendix D), choose AppWizard, enter the project’s location, and name your project “Tac.” The choices for the six steps for the starter application are:

1. Choose “Single Document” application.
2. Choose “None” (the defaults).
3. Choose “None” (the defaults).
4. Uncheck all features, even the printing since we will not be using printing in the tic tac toe example, but choose the “Advanced” button and fill in the “File extension” that you wish to be used for your files.
5. Choose no source comments and to use the MFC library as a statically linked library.
6. The default of deriving our view class from CView is what we want.

Note: All of the derived classes have been renamed to begin with C_ rather than C.

AppWizard automatically creates a directory to match the project name and then puts the .mak file in that directory.

Now we need to change the caption from “Untitled” to “Tic Tac Toe Game”:

1. Enter AppStudio by choosing the Tac.rc file from the list of “Tac” application files.
 - a. Open it by double-clicking.
 - b. Choose the “String Table” resource and the String Table opens up.
 - c. We want to change the string associated with IDR_MAINFRAME, so double-click on that ID, and its property page will appear.
 - (1) Edit the string so that it appears the way it is in Figure 11-2. This will give a caption title that does not contain the word “Untitled” and the first word of this string, which is TicTac, will be used as the root name of all the files that you save.

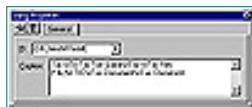


Figure 11-2: Editing the IDR_MAINFRAME String

Choose the compiler settings to be what you wish, then build the starter application and execute it. Figure 11-3 shows what it looks like at this stage.

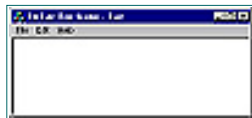


Figure 11-3: The Executing Starter “Tac” Application

The list of files that the “Tac” application has at this point is shown in Figure 11-4. We have a view class named C_TacView in the files named TacView.h and TacView.cpp. We also have a document class, named C_TacDoc, that is attached to the C_TacView class and is in the files named TacDoc.h and TacDoc.cpp.



Figure 11-4: The “Tac” Starter Project Files

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

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

[Previous](#)[Table of Contents](#)[Next](#)[BROWSE
BY TOPIC](#)

Multiple View Classes

For our tic tac toe game example, we want a view class that will draw the rules of the game in the left pane of the window. This class, which we will name C_RuleView, does not have a need to store any data, so we will not use a document for this class. As you can see from Figure 11-4, we already have one view class named C_TacView with its attached document C_TacDoc. We will use the C_TacView class for drawing the tic tac toe board, and we will use its attached document C_TacDoc for storing the data of the tic tac toe board. Our Tac application will show both views simultaneously using a static splitter window. The architecture of the Tac application is shown in Figure 11-5.



Figure 11-5: Two View Classes Managed by the Static Splitter

We will need a new view class, called C_RuleView, in which we will code all the functionality necessary to display the rules of the game. This view will occupy the left pane of the window. The best way to get another view class is to clone the existing view class.

1. Copy the two existing view files, TacView.h and TacView.cpp, in the Tac subdirectory and rename them as RuleView.h and RuleView.cpp.
2. Add the RuleView.cpp file to the project.
3. In the Microsoft Developer Studio:
 - a. Use the editor to replace all occurrences of “C_TacView” with “C_RuleView” and of “TacView” with “RuleView” in the file RuleView.cpp. Then the RuleView.h file will be shown as a dependency and you will be able to replace all of the occurrences of “C_TacView” with “C_RuleView” and of “TacView” with “RuleView” in the RuleView.h file.

At this point in the Tac application development, the modified list of files looks like

Figure 11-6.

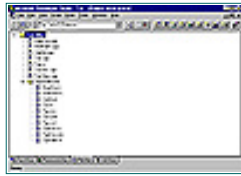


Figure 11-6: The Modified Tac Project Files

4. Include the two view header files `TacView.h` and `RuleView.h` in the `MainFrm.cpp` file.

Usually, the main frame implementation does not need to know about view classes, but in the case where the client area of the main frame is a splitter window, this knowledge is necessary. To make matters worse we need to add the document header file `TacDoc.h`, because the `C_TacView` class refers to its document class `C_TacDoc`. Making matters worse yet, the order in which you enter the `#includes` is important; the document header file must precede any view file that refers to it. So, here are all the files and the order in which they need to be included by `MainFrm.cpp`; the non-bold lines show what is already there, and the bold lines preceded by the arrow show what you must add:

```
// MainFrm.cpp : implementation of the C_MainFrame class
//
#include "stdafx.h"
#include "Tac.h"
#include "MainFrm.h"

--> #include "TacDoc.h"
--> #include "TacView.h"
--> #include "RuleView.h"
```

Our files are now fixed up, and we are ready to add the static splitter window, which is what we do in the following section.

Static Splitter Windows

A splitter window appears as a special type of frame window that holds several views in panes. A splitter window can split the window horizontally or vertically. There are dynamic and static splitter windows. Chapter Ten discusses dynamic splitter windows, which allow the user to split the window at any time by choosing a menu item or by dragging a splitter box located on the scroll bar.

In this tic tac toe example, we use a static splitter window that splits the window on creation. After the window is split, the user can move the splitter bars with the mouse to adjust the relative sizes of the panes. A static splitter window has a maximum limit of 16 rows and 16 columns. In this example, we use 1 row and 2 columns, and we have two panes with different views.

The panes of a static splitter window are defined when the window is first created, and they cannot be changed. The user can move the bars but cannot unsplit or resplit the window. Static splitter windows can accommodate multiple view classes, with the configuration set at create time.

To create a static splitter, do the following things:

1. Embed a `CSplitterWnd` object as an object data member in your view's parent, which

is the C_MainFrame class derived from CFrameWnd.

2. Override the CFrameWnd::OnCreateClient() member function.
3. Within the overridden OnCreateClient() function, call the CSplitterWnd object's CreateStatic() member function and specify the maximum number of rows and columns.
4. For each pane of the splitter window, call the CSplitterWnd object's CreateView() function and specify which view is to be displayed in that pane. Rows and columns are specified with zero-based numbers. The first pane is row 0, column 0.

Open the file MainFrm.h and add a splitter-window object to the MainFrame class and add the prototype of the OnCreateClient() function:

```
class C_MainFrame : public CFrameWnd
{
    . . . . .

    // Attributes
public:
    --> CSplitterWnd m_wndSplitter;

    . . . . .

    // Overrides
public:
    --> virtual BOOL OnCreateClient(LPCREATESTRUCT lpcs,
    --> CCreateContext* pContext);

    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);

    . . . . .
```

The implementation of the overridden OnCreateClient() function is added to the MainFrm.cpp file:

```
-->BOOL C_MainFrame::OnCreateClient(LPCREATESTRUCT lpcs,
-->                                CCreateContext* pContext)
-->{
-->    m_wndSplitter.CreateStatic(this, 1, 2);
-->    m_wndSplitter.CreateView(0, 0, RUNTIME_CLASS(C_RuleView),
-->                             CSize(300, 400), pContext);
-->    m_wndSplitter.CreateView(0, 1, RUNTIME_CLASS(C_TacView),
-->                             CSize(300,400), pContext);
-->    return TRUE;
-->}
```

The call to OnCreateClient() is done before the window becomes visible. The CreateStatic() function, a member of the CSplitterWnd class creates a splitter window with 1 row and 2 columns. The CreateView() function attaches the specified view to each pane in the splitter window. The left pane (0, 0) displays the class C_RuleView, while the right pane (1, 0) displays the class C_TacView. Build the application and execute it. It should look like Figure 11-7 (after some resizing).

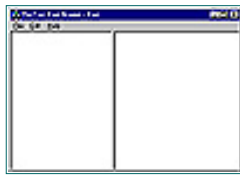


Figure 11-7: Executing Tac Program With Static Splitter Window

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

The special vertical splitter bar is actually a child window of the splitter window encapsulated by the `m_wndSplitter` object. Check to see that this special bar is working by dragging it left and right.

Now, we need to revisit the `OnCreateClient()` member function of `C_MainFrame` (`MainFrm.cpp`) and add the following line, just before the return statement. This will always show our views maximized, and we will design our views to fit nicely on a maximized screen:

```

BOOL C_MainFrame::OnCreateClient(LPCREATESTRUCT lpcs,
                                CCreateContext* pContext)
{
    // ...
    --> ShowWindow(SW_SHOWMAXIMIZED);
    return TRUE;
}
  
```

Collection Classes

MFC provides a number of ready-to-use arrays, lists, and maps that are referred to as *collection classes*. Using a collection allows the programmer to hold, process, or store groups of class objects or variables of certain standard types. You can use these collection classes to store objects in memory, and for some standard types they provide serialization support. The collection varies in size to accommodate as many objects as memory constraints will allow. A collection object appears as a single object. Class member functions can operate on all elements of the collection.

A collection class is characterized by its *shape* and by the types of its elements. The shape refers to the way the objects are organized and stored by the collection. MFC 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 need. Each of the three collection shapes is described briefly in the following paragraphs.

Array Collections

The array class provides a dynamically sized, ordered, and integer-indexed array. The array collection behaves like the standard array. It does everything a normal array does, with the added benefit of dynamic length. You no longer have to worry about the exact length of the array. The array collection internalizes all of these nasty details. The array collection classes are named for the objects that they hold and are: CByteArray (holds BYTE values), CWordArray (holds WORD values), CDWordArray (holds DWORD values), CPtrArray (holds void pointer values), CObArray (holds class object pointers), CStringArray (holds CString objects), and CUIntArray (holds UINT values). In our tic tac toe example, we will store the history of the moves using a CByteArray.

List Collections

The list collection classes provide an ordered, non indexed list of elements, implemented as a doubly linked list. A list has a head (the first element) and a tail (the remaining elements), and adding or removing elements from the head or tail, or inserting or deleting elements in the middle, is very fast. The head is an element of the list, and the tail is a linked list in its own right. The list class provides an iterator mechanism through which you may navigate the list. The list collection classes are named for the type of objects that they contain. The list collection classes are: CPtrList (holds void pointers), CObList (holds class object pointers), and CStringList (holds CString objects).

Map Collections

A map, which is also known as a dictionary, is a collection that associates a key object with a value object. Maps store logical connections between two objects of different types. The first object in the connection is a type that serves as a key (or search field) for the map. Each key is unique; the map does not allow duplicate key entries. The second object in the connection is the value object. Value objects hold the actual data. This combination of the small, unique key object and the value object allows for very fast storage and retrieval of data. Retrieval is very efficient because map searching is performed using a hashing algorithm. Maps, like lists, have iterators that allow you to move through the structure item-by-item. At each item, during the iteration, you perform whatever tasks you need for your program. The map collection classes are named for the key and the value that they support (all pointers are void pointers) and are: CMapWordToPtr, CMapPtrToWord, CMapPtrToPtr, CMapWordToOb, CMapStringToPtr, CMapStringToOb, and CMapStringToString.

Designing the Document's Data

This section covers designing the data for the document.

Caution: Designing the various objects that are part of the document and the various objects that are part of particular views is the most important and the most challenging task when developing an application with the MFC library.

The performance, simplicity, and maintainability of the application depend on the way data is organized between document and views. The document is the object that stores the application data, while the view is the object that allows the user to view the data in the document. In this tic tac toe example, the document contains only the data that will be filed; it is the minimum amount of data necessary to restart a game from a file. The rest of the application's code will be located in the view classes.

The document data consists of:

- A BOOL variable, m_bXPlayerTurn, which keeps track of the player's turn
- A single collection object of type CByteArray (a dynamic array of bytes) called m_byXHistory. This object stores the history of the X Player's moves; each byte in the array contains the value of each square that the X Player has marked.
- A single collection object of type CByteArray called m_byOHistory. This object stores the history of the O Player's moves; each byte in this array contains the value of each square that the O Player has marked.

The user may save the document in a file and may return to continue the game later on, so we will have to serialize the flag for players' turns and the history objects, so that the game can be stored and retrieved from a disk file. Suppose we have a game (in progress) as shown in Figure 11-8. We can file this game to disk storage and its file would contain the following values:

- a. m_bXPlayerTurn = TRUE
- b. The m_byXHistory array contains the values: 1, 9
- c. The m_byOHistory array contains the values: 2, 5, 7

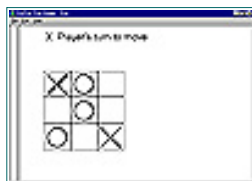


Figure 11-8: An Example Tic Tac Toe Game

Previous	Table of Contents	Next

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

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

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

Keyword

☒ Brief
 ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

Coding the Document Class

We need to add m_bXPlayerTurn, the player turn flag, and the m_byXHistory and m_byOHistory objects to the document class. We further need to override the functions CDocument::OnNewDocument() and CDocument::OnOpenDocument(). In the overridden functions we will clear the history objects from the document whenever a new document or an existing file is opened. We will also need to override the CDocument::Serialize() function. The prototypes for OnNewDocument() and Serialize() have already been added to the header file by AppWizard, but we need to add the prototype for the OnOpenDocument() ourselves.

1. In the TacDoc.h file, add the following lines:

```

. . . . .
// Attributes
public:
--> BOOL m_bXPlayerTurn;
--> CByteArray m_byXHistory;    // stores cells marked with X
--> CByteArray m_byOHistory;    // stores cells marked with 0
. . . . .

// Overrides
public:
--> virtual BOOL OnOpenDocument(const char* pszPathName);
    virtual BOOL OnNewDocument();
    virtual void Serialize(CArchive& ar);
    
```

2. In the TacDoc.cpp file, modify the constructor, and override OnNewDocument(), OnOpenDocument(), and Serialize().

- a. Add the following lines, shown in bold, to the TacDoc.cpp file:

```

C_TacDoc::C_TacDoc()
{
--> m_bXPlayerTurn = FALSE;
--> m_byXHistory.SetSize(9);
--> m_byOHistory.SetSize(9);
}
    
```



```

-->C_TacDoc::~~C_TacDoc()
-->{
-->    m_byXHistory.RemoveAll();
-->    m_byOHistory.RemoveAll();
-->}

-->BOOL C_TacDoc::OnOpenDocument(const char* pszPathName)
-->{
-->    m_byXHistory.RemoveAll();
-->    m_byOHistory.RemoveAll();
-->    CDocument::OnOpenDocument(pszPathName);
-->    return TRUE;
-->}

BOOL C_TacDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;
--> m_bXPlayerTurn = FALSE;
--> m_byXHistory.RemoveAll();
--> m_byOHistory.RemoveAll();
    return TRUE;
}

void C_TacDoc::Serialize(CArchive& ar)
{
--> int i;
--> long Xsize, Osize;
--> BYTE byte;
--> WORD w;

    if (ar.IsStoring())
    {
--> w = m_bXPlayerTurn;
--> ar << w;

--> Xsize = m_byXHistory.GetSize();
--> ar << Xsize;
--> for ( i = 0; i < Xsize; i++)
--> {
--> ar << m_byXHistory.GetAt(i);
--> }

--> Osize = m_byOHistory.GetSize();
--> ar << Osize;
--> for ( i = 0; i < Osize; i++)
--> {
-->     ar << m_byOHistory.GetAt(i);
--> }
    }
    else
    {
--> ar >> w;
--> m_bXPlayerTurn = (BOOL)w;

```

```

--> ar >> Xsize;
--> for ( i = 0; i < Xsize; i++)
--> {
-->     ar >> byte;
-->     m_byXHistory.Add(byte);
--> }

--> ar >> Osize;
--> for ( i = 0; i < Osize; i++)
--> {
-->     ar >> byte;
-->     m_byOHistory.Add(byte);
--> }
}

```

CByteArray Member Functions

The preceding code demonstrates a number of handy functions provided by the class CByteArray. Table 11-1 describes the CByteArray member functions.

Note: Before we used the arrays, we called the function SetSize() to establish their maximize size and allocate memory for them. If you do not use Set Size(), adding elements to your array causes it to be frequently reallocated and copied. Frequent reallocation and copying are inefficient and can cause fragmented memory.

Table 11-1: CByteArray Member Functions

Function	Meaning
Bounds	
GetSize()	Gets the number of elements in this array.
GetUpperBound()	Returns the largest valid index.
SetSize()	Sets the number of elements to be contained in this array.
Operations	
FreeExtra()	Frees all unused memory above the current upper bound.
RemoveAll()	Removes all the elements from this array.
Element Access	
GetAt()	Returns the value at a given index.
SetAt()	Sets the value for a given index; array not allowed to grow.
ElementAt()	Returns a temporary reference to the element pointer within the array.
Growing the Array	
SetAtGrow()	Sets the value for a given index; grows the array if necessary.
Add()	Adds an element to the end of the array; grows the array if necessary.
Insertion/Removal	
InsertAt()	Inserts an element (or all the elements in another array) at a specified index.
RemoveAt()	Removes an element at a specific index.

Operators

operator []

Sets or gets the element at the specified index.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Designing the View of the Rules

In the rules view window, we draw lines of text that describe the rules of the game. To draw a line of text use the `CDC::TextOut()` function. Provide the string that you wish to be drawn and select into the device context the font with which you wish the string to be drawn. In this section we will use logical fonts, where we can select numerous parameters describing the font. We will use the MFC class `CFont`, and we will create our logical font with the `CFont` function `CreateFont()`.

We will enter code in the rules view, using the function `C_RuleView::OnDraw()`, in which we draw lines of text (in pane 0,0) describing the rules of the game. In the view that displays the tic tac toe board, we will use the functions `C_TacView::OnDraw()` and `C_TacView::OnLButtonDown()`, to draw a line of text above the tic tac toe board (in pane 0,1) describing the player's turn. These character strings are ideal candidates for inclusion in the string table.

Now, add these strings to the string table. (Later discussions will describe how to code your program to load them from the string table.) In a string table, strings are grouped into segments, or blocks, of 16 strings each. The segment a string belongs to is determined by the value of its identifier. Strings with identifiers of 0 to 15 are in one segment; strings with identifiers of 16 to 31 are in the second segment, etc. To move a string from one segment to another you need to change its identifier. Individual string segments are loaded on demand in order to conserve memory. For this reason, programmers usually try to group strings into logical groups of 16 or less and then use each group only when it is needed.

Opening the string table is described in the appendix that covers the specific compiler that you are using. The "String Table" resource editor appears with "String Segment 0" highlighted as shown in Figure 11-9. "String Segment 0"

starts with IDR_MAINFRAME, and you will add your strings to this segment. The “String Table” resource editor that has been generated by AppWizard contains quite a few strings—some familiar and some not so familiar. Take a moment to scroll through the list of strings. Most of these strings that AppWizard generated for you are status bar prompts. (Status bars are covered in Chapter Thirteen.)

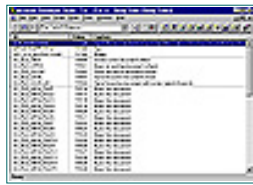


Figure 11-9: String Table for the Tac Program

To add strings:

1. Be sure that the string segment you wish to add your string to is selected—in this case String Segment 0.
2. Choose “Insert | New String” from the menu. The “String Properties” box will be displayed as a result.
3. Change the ID to the one that you want, and type the string into the edit box labeled “Caption.”
4. Then, to enter the string shown on the property page into the String Table, either press Enter to close the property page, or click on the highlighted row in the chosen String Segment.

The string will be entered into the table, and a new highlighted blank row will appear as shown in the Figure 11-10. This will be where the next string will be added. To add the next string, either double-click on the highlighted blank row, or, once again, select the menu items “Insert | New String” and the “String Property” page will again appear, ready for you to add your next string. Continue adding strings until your string table looks like Figure 11-10.

Note: It is a good idea to choose descriptive IDs wherever possible since they are easier to remember later on in your program when you need to load the string referenced in that ID.



Figure 11-10: The Finished String Table for the Tac Program

Using Logical Fonts

The Windows operating system comes with separate font files that define the characteristics of characters written in a number of different styles and sizes. You can find the font files in your Windows system directory, usually C:\WINDOWS\SYSTEM. Font files have the extension .FON, or .FOT for TrueType fonts.

The MFC classes provide the CFont class for creating and manipulating font objects. This class has few functions because its main purpose is to allow the creation of fonts. The most used function is CFont::CreateFont(), which creates a logical font of any size based on a long list of parameters passed to the function.

The sizing of the characters for both height and width is in logical units. With the default units of a device context, logical units are equal to pixels (dots on the screen). There are other systems of units which will allow you to size fonts and other graphical objects in inches, millimeters, or printer's units, but we will not cover them here.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

☒ Advanced

☐ Search

☐ Search Tips

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

Fourteen parameters are passed to the `CFont::CreateFont()` function. Table 11-2 describes these parameters. Figure 11-11 graphically defines the values defining the vertical character size, which are used in Table 11-2. You can be creative with `CFont::CreateFont()` and make fonts with characters lying on their sides, going upward instead of left-to-right, etc.

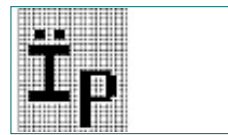


Figure 11-11: The Five Values Defining the Font Vertical Character Size

Table 11-2: The CFont::CreateFont() Parameters

`BOOL CFont::CreateFont(int nHeight, int nWidth, int nEscapement, int nOrientation, int nWeight, BYTE bItalic, BYTE bUnderline, BYTE bStrikeOut, BYTE bCharSet, BYTE nOutputPrecision, BYTE nClipPrecision, BYTE nQuality, BYTE nPitchAndFamily, LPCSTR lpszFacename);`

Parameter	Meaning
nHeight	The desired height of the characters, including the internal leading and excluding external leading, in logical units. To set the ascent size, rather than the total height, make this value negative. The absolute value will then be used to set the ascent size.
nWidth	The desired width of the characters in logical units. Normally set to 0, which allows the operating system to match the width to the height. Positive values force that width, changing the character's aspect ratio.
nEscapement	Specifies the orientation of the next character output relative to the previous one in tenths of a degree. Normally set to 0. Set to 900 to have all the characters go upward from the first character, 1800 to write backwards, or 2700 to write each character from the top down.
nOrientation	Specifies how much the character should be rotated when output in tenths of a degree. Set to 900 to have all the characters lying on their backs, 1800 for upside-down writing, etc.
nWeight	Sets the thickness of each character. The units are arbitrary, with the values of <code>FW_NORMAL</code> (400) for normal characters and <code>FW_BOLD</code> (700) for boldface defined in <code><windows.h></code> . <code><windows.h></code> has eight other weights defined.
bItalic	TRUE to specify italic characters, FALSE (zero) for normal.

bUnderline	TRUE to specify underlined characters, FALSE (zero) for normal.
bStrikeOut	TRUE to specify characters with a line through the center, FALSE (zero) for normal.
bCharSet	The character set of the font. This can be ANSI_CHARSET, SYMBOL_CHARSET, OEM_CHARSET or (with Japanese versions of the operating system) SHIFTJIS_CHARSET.
nOutputPrecision	Set equal to OUT_DEFAULT_PRECIS (zero). This parameter does not do anything.
nClipPrecision	Set equal to CLIP_DEFAULT_PRECIS.
nQuality	Can be either DRAFT_QUALITY, PROOF_QUALITY, or DEFAULT_QUALITY (the most common choice). PROOF_QUALITY forces the closest match to the loaded font data, which may change the font size if the specified size is not available.
nPitchAndFamily	Two values combined with the C language binary OR operator (). The two low-order bits specify the font pitch. This can be either DEFAULT_PITCH, FIXED_PITCH, or VARIABLE_PITCH. The four high-order bits specify the font family. This can be any of the following: FF_DECORATIVE, FF_DONTCARE, FF_MODERN, FF_ROMAN, FF_SCRIPT, or FF_SWISS. For example, use the combination DEFAULT_PITCH FF_ROMAN to create a Roman typeface, using the character pitch of the nearest matching font installed on the system. This will be a variable-pitch font if the normal Windows operating system Times Roman typeface is installed.
lpszFacename	A pointer to a null-terminated string that specifies the name of the font data. The maximum length of the name is LF_FACESIZE, which is defined in <windows.h> as 32.

Text Metrics

Because of the way the CFont::CreateFont() function generates a logical font using stored font data modified by the CFont::CreateFont() parameters, you will not know all of the dimensions of a font after you create it. The MFC classes provide the CDC::GetTextMetrics() function to find out details about a font. CDC::GetTextMetrics() determines the characteristics of the font currently selected into the device context, and copies the data into a data structure called TEXTMETRIC. The TEXTMETRIC structure is defined in <windows.h> and is shown in Listing 11-3.

Listing 11-1: The TEXTMETRIC Structure

```
typedef struct tagTEXTMETRIC
{
    int      tmHeight;           // character height
    int      tmAscent;          // ascent height
    int      tmDescent;         // descent height
    int      tmInternalLeading;   // internal Leading height
    int      tmExternalLeading;   // external Leading height
    int      tmAveCharWidth;     // average width of a character
    int      tmMaxCharWidth;     // widest character width
    int      tmWeight;           // weight (thickness) of the font
    BYTE     tmItalic;           // nonzero for italics
    BYTE     tmUnderLined;       // nonzero for underlined
    BYTE     tmStruckOut;        // nonzero for strike through characters
    BYTE     tmFirstChar;        // code value for first character defined
    BYTE     tmLastChar;         // code value for last character defined
    BYTE     tmDefaultChar;      // char to substitute for those missing
    BYTE     tmBreakChar;        // word break character--usually a space
}
```



```
    BYTE    tmPitchAndFamily; // pitch and family code
    BYTE    tmCharSet;        // either ANSI_CHARSET, SYMBOL_CHARSET,
                                // SHIFTJIS_CHARSET, or OEM_CHARSET
    int      tmOverhang;      // extra width allowed for bold, etc.
    int      tmDigitizedAspectX; // ratio of the X and Y aspects
    int      tmDigitizedAspectY; // is the aspect ratio for which the
                                // font was originally designed
} TEXTMETRIC;
```

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

☒ [Advanced Search](#)

☒ [Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

Coding the View Class Containing the Rules

In the preceding section we saw how to place the strings in the String Table. Once all of the strings are placed in the string table, they can be drawn on the screen in the font you choose when you create your CFont object.

We will draw the rules of the game in the C_RuleView class. They will be drawn when the window is created and will not be changed. We create two different fonts to use in drawing the text. The font is selected into the device context before drawing the text.

We also change the text color of the device context to red and draw three lines in red before we change the device context text color back to black. Each line is drawn using the function TextOut(). Each time a line is drawn, you must give it the x, y coordinates to begin that line. To do this, we establish a variable, nLineSpacing, which is based on the height parameters of the font selected into the device context.

1. Add the following code to the OnDraw() function in the C_RuleView class (in the RuleView.cpp file):

```
void C_RuleView::OnDraw(CDC* pDC)
{
    C_TacDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    --> TEXTMETRIC tm;
    --> CFont ArialFont30;
    --> CFont ArialFont25;
    --> CString text;
    --> ArialFont30.CreateFont( 30, 0, 0, 0, FW_NORMAL, 0, 0, 0,
    --> ANSI_CHARSET, OUT_DEFAULT_PRECIS,
    --> CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
    --> DEFAULT_PITCH | FF_SWISS, "Arial");
    --> ArialFont25.CreateFont( 25, 0, 0, 0, FW_NORMAL, 0, 0, 0,
    --> ANSI_CHARSET, OUT_DEFAULT_PRECIS,
    --> CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
    --> DEFAULT_PITCH | FF_SWISS, "Arial");
    --> text.LoadString(IDS_STRING129);
    --> CFont* pOldFont = --> (CFont*)pDC->SelectObject(&ArialFont30);
    --> pDC->GetTextMetrics(&tm);
    --> int nLineSpacing = tm.tmHeight + tm.tmExternalLeading;
    --> pDC->TextOut(15, nLineSpacing, text);
    --> text.LoadString(IDS_STRING130);
    --> pDC->TextOut(15, 3 * nLineSpacing, text);
}
```

```

--> pDC->SetTextColor(RGB(255, 0, 0));
--> pDC->SelectObject(&ArialFont25);
--> text.LoadString(IDS_STRING131);
--> pDC->TextOut(20, 4 * nLineSpacing, text);
--> text.LoadString(IDS_STRING132);
--> pDC->TextOut(20, 5 * nLineSpacing, text);
--> text.LoadString(IDS_STRING133);
--> pDC->TextOut(20, 6 * nLineSpacing, text);
--> pDC->SetTextColor(RGB(0, 0, 0));
--> text.LoadString(IDS_STRING134);
--> pDC->TextOut(20, 8 * nLineSpacing, text);
--> text.LoadString(IDS_STRING135);
--> pDC->TextOut(45, 9 * nLineSpacing, text);
--> text.LoadString(IDS_STRING136);
--> pDC->TextOut(70, 10 * nLineSpacing, text);
--> text.LoadString(IDS_STRING137);
--> pDC->TextOut(45, 11 * nLineSpacing, text);
--> pDC->SelectObject(pOldFont);
}

```

2. At this point compile and run your program, and it should look like Figure 11-12.



Figure 11-12: “Tac” Program with the Rules View Completed

Designing the View of the Game

The view that displays the tic tac toe board is `C_TacView`, and the board is displayed in pane (0, 1). In the `C_TacView` files we need to add code to do several things. We need to initialize the cells of the board from the data stored in the document, and we need to draw the board on the screen. We also must respond to the user if the left mouse button is pressed over a cell.

In order to do drawing in the `C_TacView` class, we introduce two new classes called `C_Cell` and `C_TicTacBoard`. These two classes will contain all the functionality we need to draw each cell and the complete board. The definition of our `C_Cell` and `C_TicTacBoard` classes could be placed in their own header and implementation files, but instead we place them within the files that contain the `C_TacView` class. The `C_TacView` class has an object data member of class `C_TicTacBoard`. The `C_TicTacBoard` class has an array of `C_Cell` objects as its data member.

In the following sections, the code for drawing the tic tac toe board and initializing it from the data stored in the document is first presented. Then the code for responding to the left button down message is presented; this is how the user claims a cell. The tic tac toe board can be a new board (no moves on it), where the user initiates a game, or a game in progress can be brought in from a file and the user can view or complete the game.

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



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Go!

Keyword

☒ Brief ☐ Full[Advanced](#)[Search](#)[Search Tips](#)BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

Go!

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

Drawing The Tic Tac Toe Board

The class C_Cell has two attributes: m_nState, which keeps track of which player has claimed the cell or if it is free; and m_Rect, which is the bounding rectangle of the cell. C_Cell has a member function Draw() which draws the cell.

The class C_TicTacBoard contains an array of C_Cell objects. The constructor calculates the m_Rect (location) of each cell. The C_TicTacBoard member function Initialize() initializes each cell's m_nState variable according to the document's history. C_TicTacBoard has a Draw() member function, which draws the tic tac toe board by iterating over its array of C_Cell objects and calling each cell's Draw() function.

To draw the tic tac toe board, add the following class declarations to the header file TacView.h:

```

-->class C_Cell
-->{
-->    public:
-->        int m_nState;          // status of the cell
-->        //possible states of the cell
-->        #define FREE          0
-->        #define XPLAYER       1
-->        #define OPLAYER       2
-->        CRect m_Rect;
-->        #define CELL_SIZE 70
-->        void Draw(CDC* pDC);
-->};

-->class C_TicTacBoard
-->{
-->    public:
-->        C_TicTacBoard();
-->        C_Cell m_Cell[10]; // 10 instead of 9 so the index is
-->                           // the same as the attached number
-->        void Initialize(C_TacDoc* pDoc); // initializes each
-->                                         // cell's m_nState to document's data
-->        void Draw(CDC* pDC);
-->};

```

The class C_TacView contains an object of the class C_TicTacBoard and a CFont object as data members. We override the OnInitialUpdate() function to call the C_TicTacBoard::Initialize() function, which will initialize each cell's m_nState according to the document's data. To do this, add the following lines to the definition of the class C_TacView, which is in the header file TacView.h:

```
class C_TacView : public CView
{
    . . . . .

// Attributes
public:
    C_TacDoc* GetDocument();
--> C_TicTacBoard m_TicTacBoard;
--> CFont m_MyFont;

    . . . . .

// Overrides
public:
--> virtual void OnInitialUpdate(); // initializes m_Cell array
    virtual void OnDraw(CDC* pDC); // override to draw this view
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    . . . . .
```

In the TacView.cpp file we must implement these functions. The definitions of the C_Cell function Draw() must be added, and the definition of the C_TicTacBoard constructor and its Initialize() and Draw() functions must all be added. The following code does all this, and must be added to the TacView.cpp file.

```
-->void C_Cell::Draw(CDC* pDC)
-->{
-->    pDC->Rectangle(m_Rect);

-->    CPen pen;
-->    CPen* pOldPen;
-->    pen.CreatePen(PS_SOLID, 3, RGB(0, 0, 0));
-->    pOldPen = pDC->SelectObject(&pen);

-->    if (m_nState == XPLAYER)
-->    {
-->        // draw two crossed lines
-->        pDC->MoveTo(m_Rect.TopLeft() + CSize(10, 10));
-->        pDC->LineTo(m_Rect.BottomRight() - CSize(10, 10));
-->        pDC->MoveTo(m_Rect.TopLeft() + CSize(60, 10));
-->        pDC->LineTo(m_Rect.BottomRight() - CSize(60, 10));
-->    }

-->    if (m_nState == OPLAYER)
-->    {
-->        // draw an ellipse
-->        CPoint p1 = m_Rect.TopLeft() + CSize(10, 10);
-->        CRect r(p1, CSize(CELL_SIZE - 20, CELL_SIZE - 20));
-->        pDC->Ellipse(&r);
-->    }
-->    pDC->SelectObject(pOldPen);
-->}
```

```

-->C_TicTacBoard::C_TicTacBoard()
-->{
-->    // calculate m_Rect of each cell
-->    CRect rectCorner(65,120,135,190);

-->    for (int i = 1; i < 10; i++)
-->    {
-->        CRect newRect = rectCorner;
-->        CPoint offset;
-->        int column = (i - 1) % 3;
-->        int row = (i - 1) / 3;
-->        offset.x = CELL_SIZE * column;
-->        offset.y = CELL_SIZE * row;
-->        newRect.OffsetRect(offset);
-->        m_Cell[i].m_Rect = newRect;
-->        m_Cell[i].m_nState = FREE;
-->    }
-->}

-->void C_TicTacBoard::Initialize(C_TacDoc* pDoc)
-->{
-->    // build each cell's m_nState, according to the document's data
-->    CByteArray* pXHistory = &pDoc->m_byXHistory;
-->    CByteArray* pOHistory = &pDoc->m_byOHistory;
-->    int i;
-->    BYTE byte;

-->    for ( i = 1; i < 10; i++)
-->    {
-->        m_Cell[i].m_nState = FREE;
-->    }

-->    for ( i = 0; i < pXHistory->GetSize(); i++)
-->    {
-->        byte = pXHistory->GetAt(i);
-->        m_Cell[byte].m_nState = XPLAYER;
-->    }

-->    for ( i = 0; i < pOHistory->GetSize(); i++)
-->    {
-->        byte = pOHistory->GetAt(i);
-->        m_Cell[byte].m_nState = OPLAYER;
-->    }
-->}

-->void C_TicTacBoard::Draw(CDC* pDC)
-->{
-->    for ( int i = 1; i < 10; i++)
-->    {
-->        m_Cell[i].Draw(pDC);
-->    }
-->}

```

We need to make further additions to the TacView.cpp file within the class C_TacView code. To do this, we add code to the C_TacView() constructor, which creates the CFont object, and we delete it in the destructor. The C_TacView::OnInitialUpdate() function is overridden to update the cells to the information in the document and then to call Invalidate() to repaint the screen. The

C_TacView::OnDraw() function draws the tic tac toe board. The following code should be added to these C_TacView functions:

```
C_TacView::C_TacView()
{
--> m_MyFont.CreateFont(30, 0, 0, 0, FW_NORMAL, 0, 0, 0,
-->                     ANSI_CHARSET, OUT_DEFAULT_PRECIS,
-->                     CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
-->                     DEFAULT_PITCH | FF_SWISS, "Arial");
}

C_TacView::~C_TacView()
{
--> m_MyFont.DeleteObject();
}

-->void C_TacView::OnInitialUpdate()
-->{
-->    // build up the moves, m_nState, on the tictac board
-->    // to reflect the history data in the document

-->    C_TacDoc* pDoc = GetDocument();
-->    m_TicTacBoard.Initialize(pDoc);
-->    Invalidate();
-->}

void C_TacView::OnDraw(CDC* pDC)
{
    C_TacDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

--> m_TicTacBoard.Draw(pDC);
--> TEXTMETRIC tm;
--> CString text;
--> pDC->GetTextMetrics(&tm);
--> int nLineSpacing = tm.tmHeight + tm.tmExternalLeading;
--> CFont* pOldFont = (CFont*)pDC->SelectObject(&m_MyFont);

--> if (pDoc->m_bXPlayerTurn == TRUE)
-->     text.LoadString(IDS_XTURN);
--> else

-->     text.LoadString(IDS_OTURN);

--> pDC->TextOut(70, nLineSpacing, text);
--> pDC->SelectObject(pOldFont);
}
```

At this point, compile and run your program and it looks like Figure 11-13.

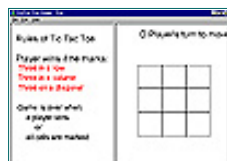


Figure 11-13: The Tac Program with the Tic Tac Toe Board Drawn

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

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

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



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

Go!

[Previous](#)

[Table of Contents](#)

[Next](#)

BROWSE
BY TOPIC

Drawing the Moves

The last thing to do is to draw the moves as the user clicks the left mouse button over the cells of the tic tac toe board. We want a handler function for the left mouse button down message. The handling of this message will be done in the C_TacView class. Using ClassWizard select the class name of C_TacView and add the WM_LBUTTONDOWN message. Class Wizard will automatically add the handler's prototype to the header file, and in the implementation file it adds the message map entry and the stubbed-out handler function. Now we can complete the code in the stubbed-out handler function of the WM_LBUTTONDOWN message:

```
void C_TacView::OnLButtonDown(UINT nFlags, CPoint point)
{
--> // check if the mouse was over any of the 9 cells
--> // if cell is occupied, beep. Otherwise
--> // complete the move.

-->C_TacDoc* pDoc = GetDocument();
-->for (int i = 1; i < 10; i++)
-->{
--> C_Cell* pCell = &m_TicTacBoard.m_Cell[i];
--> if (pCell->m_Rect.PtInRect(point))
--> {
-->     switch (pCell->m_nState)
-->     {
-->         case XPLAYER:
-->             ::MessageBeep(0);
-->             break;
-->         case OPLAYER:
-->             ::MessageBeep(0);
-->             break;
-->         case FREE:
-->             {
-->                 if (pDoc->m_bXPlayerTurn == TRUE)
-->                 {
```

```

-->         pCell->m_nState = XPLAYER;
-->         pDoc->m_bXPlayerTurn = FALSE;
-->         CByteArray* pXHistory = &pDoc->m_byXHistory;
-->         pXHistory->Add(i);
-->         pDoc->SetModifiedFlag();
-->     }
-->     else
-->     {
-->         pCell->m_nState = OPLAYER;
-->         pDoc->m_bXPlayerTurn = TRUE;
-->         CByteArray* pOHistory = &pDoc->m_byOHistory;
-->         pOHistory->Add(i);
-->         pDoc->SetModifiedFlag();
-->     }
-->     CClientDC dc(this);
-->     pCell->Draw(&dc);
-->
-->     TEXTMETRIC tm;
-->     CString text;
-->     dc.GetTextMetrics(&tm);
-->     int nLineSpacing = tm.tmHeight +
-->         tm.tmExternalLeading;
-->     CFont* pOldFont =
-->         (CFont*) dc.SelectObject(&m_MyFont);
-->     if (pDoc->m_bXPlayerTurn == TRUE)
-->         text.LoadString(IDS_XTURN);
-->     else
-->         text.LoadString(IDS_OTURN);
-->     dc.TextOut(70, nLineSpacing, text);
-->     dc.SelectObject(pOldFont);
--> }
--> }
--> }
--> }

```

At this point, your program is complete. Compile it, run it, and play a tic tac toe game. The completed tic tac toe program with a completed example game is shown in Figure 11-14.

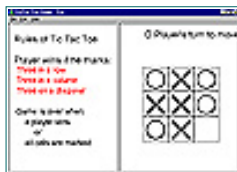


Figure 11-14: The Completed Tic Tac Toe Game

In the Tac application code, we have encountered two functions that we have not yet discussed, `OnInitialUpdate()` and `SetModifiedFlag()`. The function `CVew::OnInitialUpdate()` is called by the framework after a view is initially attached to a document. The function `CDocument::SetModifiedFlag()` can be called in your program any time you modify data in the document. This function sets a dirty bit in the document class, which then enables the MFC framework to notify the user that the document needs to be saved if the user tries to quit the application, close the document, or open another document. The dirty bit is cleared when the document is saved. These functions are studied further in the next chapter.

Summary

Static splitter windows split the window into panes on creation of the window. The number of panes must be specified in advance and cannot be changed during run time. The user can move the splitter bars, but cannot unsplit or resplit the window. A static splitter window has a maximum of 16 rows and 16 columns. They can be used to display a different view in each pane; the view assigned to each pane must be set in advance. When you create the splitter window you must assign which view is to be displayed in each pane.

MFC provides collection classes, which are ready-to-use arrays, lists, and maps (or dictionaries). Using a collection object allows the programmer to store groups of class objects or variables. The collection varies in size to accommodate as many objects as memory constraints will allow. Class member functions can operate on all elements of the collection. The array class provides a dynamically sized, ordered, and integer-indexed array. The programmer does not need to worry about the length of the array; these details are internalized by the array class. In the tic tac toe example, we stored the history of the moves using a CByteArray.

In the tic tac toe game, the document's data consists of a variable describing the player's turn, and a CByteArray for each player which stores the history of that player's moves. The CByteArray objects are serialized and stored on file. They are retrieved from file and the game shown in the view is reconstructed based on the information retrieved from the file.

To draw a line of text use the function CDC::TextOut(). You must provide the left-top co-ordinates and the string to draw, and you must select the desired font into the device context. The strings can be added to the string table. In the string table, strings are grouped into segments, or blocks, of 16 strings each. Individual string segments are loaded on demand in order to conserve memory.

The CFont class is used to create logical fonts using the class's CreateFont() member function. The main purpose of the CFont class is to allow the creation of fonts. Fourteen input parameters are passed to the CFont::CreateFont() function. You can be creative with this function and make fonts that have characters lying on their backs, going upward, going backward, etc.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

Chapter 12

An MDI Application

In this chapter, you will create a CFormView application as an MDI application. The executing program of the example, named “Form,” is shown in Figure 12-1, which shows three open child windows, two views of the document “Form1,” and a new document, “Form2.” When an MDI child window is minimized, its icon appears at the bottom of its parent, the main window, as shown in Figure 12-2, which shows all three of the child windows of Figure 12-1 minimized. Using this “Form” example, we will first look at the features of an MDI application.



Figure 12-1: The Executing “Form” Example



Figure 12-2: The MDI Child Windows Minimized

Next, generating the code for this program will be presented. Along the way, we add in TRACE statements to the code. TRACE statements are statements that you add to your code; they are only executed when an application is running in the debug mode, otherwise they are ignored. Then, running your code in debug mode to study when MFC calls various functions of the

CDocument class and of the CView class is examined. These functions are automatically called by the framework when specific events occur. The CDocument functions that we study are: OnNewDocument(), OnOpenDocument(), OnSaveDocument(), OnCloseDocument(), and DeleteContents(). These functions are used in the process of filing and retrieving data and opening and closing files. Also, you will add TRACE statements to the Serialize() function so that you can see when it is filing or retrieving from file. TRACE statements will also be used to study the MFC default calls to the CFormView functions: OnInitialUpdate() and OnUpdate(). It is important that you know all of the functions that are available to be overridden to tailor to your needs and it is equally important to understand when the framework calls each of these functions.

After completing the discussion of the program developed from the AppWizard starter application, a manually generated program, “FormMin,” for the same application, is introduced. This manually generated code contains the minimum amount of code necessary for this application. A listing of this code is given, and you can review all of the code that is necessary for this application.

Note:

Borland C++ users need to use manually generated code.

An MDI Application

An MDI application has a slightly different class structure than an SDI. Its document template is different than the SDI, and it has unique characteristics.

Class Structure

An MDI application has an MDI frame window, which occupies the entire client area of the mainframe window. The purpose of the MDI frame window is to hold and manage its child windows. The MDI frame window is derived from the class CMDIFrameWnd, which is derived from CFrameWnd. Each child window is created as an MDI child frame and is derived from the MFC class CMDIChildWnd. The Class CMDIChildWnd is also derived from CFrameWnd. The sole purpose of the MDI child frame is to hold the view. The MDI child frame can also hold a splitter window, which will manage the multiple views in the child window. (Chapter Thirteen presents an example using a splitter window in an MDI application.)

Figure 12-3 depicts where the MDI frame windows and MDI child windows fit on the mainframe window. It depicts the example shown in Figure 12-1, and it also shows the mainframe with a toolbar and a status bar. (Chapter Thirteen covers the toolbar and status bar.)



Figure 12-3: MDI Frame Window and Child Windows

Keyword

☒ Brief
 ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Characteristics

In an MDI application you can open multiple views of one document. Also, more than one document can be viewed at a time. The MDI has the feature of having two menus, and it has built-in keyboard accelerators. Each of these features is discussed in the following paragraphs.

Creating New Views

An MDI application allows the user to create multiple views of a specific document. Any MDI application will always have a menu item “Window.” When the “Window” menu item is selected, a drop-down menu appears with the choices “New Window,” “Cascade,” “Tile,” and “Arrange Icons.” These menu items are for interacting with the child frame windows.

To create a new view, select “Window | New Window.” A new view will then be made of the active document. When the “New Window” menu item is chosen, an additional child frame is constructed and a new view object is created to fill it. The new view is attached to its document just before it is displayed. As you will learn in this chapter, the function `OnInitialUpdate()` is called only once for this child frame, when the view is attached to its document.

Creating New Documents

New documents can be created at any time in an MDI application. A new document is created when you select “File | New” from the menu. The following steps occur in making a new document:

1. A document object is created.
2. A child frame (CMDIChildWnd) window object is created.
3. If this is the first document, the menu is switched from the

mainframe menu to the application menu. The application menu is the menu that is displayed when a document has been opened.

4. The view object is built and then the document template connects the classes.
5. The function `CDocument::OnNewDocument()` is called.
6. The function `CFormView::OnInitialUpdate()` is called.

Multiple Menus

An MDI application has two menus. The first menu, known as the mainframe menu, appears when no documents are open. This is shown in Figure 12-6. When the first document appears, the menu is switched to what is known as the application menu (refer to Figures 12-1 and 12-2 which display the application menu). The application menu has the “Window” menu item that allows the user to interact with the child windows in specific ways. Any menu items that your application might need to add will be added to the application menu, the menu that appears after a document is open.



Figure 12-6: The MDI Menu Prior to an Open Document

Keyboard Accelerators

An MDI application has built-in keyboard accelerators. These are available in any MDI, unless the code has been modified to change the built-in keyboard accelerator behaviors. (Commercially available MDI applications frequently override these built-in accelerators.) You will find that these built-in keyboard accelerators are functioning in the “Form” example given in this chapter. These built-in keyboard accelerators are given in Table 12-1.

Table 12-1: MDI Keyboard Accelerators

Key	Function
Ctrl+F6	Switches among the MDI child windows.
Alt+Spacebar	Invokes the system menu of the frame window.
Alt+minus	Invokes the system menu of the active MDI child.
Ctrl+F4	Closes the MDI child window.
Alt+F4	Closes the application window (main window).

The CFormView Class

The `CFormView` class is the base class used for views containing controls. Use a `CFormView`-derived class if you want form-based documents in your application. `CFormView` is derived from `CScrollView`, hence it supports scrolling, as needed. (`CScrollView` is discussed in Chapter Thirteen.)

The `CFormView` class has many of the characteristics of a modeless dialog

box. A CFormView-derived class is associated with a dialog resource that defines the frame characteristics and lists the controls.

When you use AppWizard to generate your starter application, it gives you the option of using CFormView as the base class for your view. When you select CFormView as the base class, AppWizard generates an empty dialog with the correct style properties set. You will use the Dialog Editor to complete the controls that you want in your view.

If you are not using AppWizard, you create a dialog template which is similar to creating a dialog box. If you use AppStudio to make a dialog for a form view, you must specify the properties for the dialog frame: WS_CHILD, no border, not visible, and no caption. These styles are necessary, because the form view is not a true dialog box. Then, with the dialog template open, invoke ClassWizard and choose CFormView as the class type when you are filling in the “Add Class” query box. ClassWizard creates a CFormView-derived class and connects it to the dialog template you just designed. This connection is established in the constructor for your class; ClassWizard generates a call to the base-class constructor, CFormView::CFormView(), and passes the resource ID of your dialog template.

ClassWizard is used for adding member variables and handler functions:

1. Choose the Member Variables tab of the ClassWizard query box and define member variables in your view class that correspond to the controls in your form view. ClassWizard will automatically add the DoDataExchange() function with the functions it contains. This allows you to use the DDX mechanism.

To define message handlers for control notification messages:

1. Choose the “Message Map” tab and then use the “Add Function” button in the ClassWizard query box to add the handler functions.

The CFormView object receives notification messages directly from its controls and it also receives command messages from the application framework. The ability of the CFormView object to receive command messages clearly separates CFormView from CDialog. The CFormView object can be controlled from the main menu.

Since the CFormView class is derived from CScrollView rather than CDialog, you cannot assume that the CDialog member functions are supported. CFormView does not have the virtual functions OnInitDialog(), OnOK(), or OnCancel(). CFormView does not call UpdateData(). You have to call these functions yourself. You will usually call these functions in response to the control notification messages or the command messages.

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

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

Keyword

☒ Brief
 ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

The CFormView class is not derived from the CDialog class, but it is built around a dialog. You can use many of the CDialog class member functions such as GetDlgItem() and GotoDlgCtrl(). All you have to do is to cast your CFormView pointer to a CDialog pointer. For example, within a handler function, you could have the following statement, which gets the pointer to the control IDC_NAME and then sets the focus on that control:

```
((CDialog*) this)->GotoDlgCtrl(GetDlgItem(IDC_NAME));
```

Creating the “Form” Program

The “Form” Starter Application

In this section you will generate an MDI CFormView starter application. The steps for the Visual C++ 4 AppWizard are given. (Consult the appropriate appendix if you have a different compiler.)

1. Select “New | Project Workspace.”
2. Select AppWizard.
 - a. Choose the path and directory in which the project is to be installed.
 - b. Name the project “Form.”

The six steps of the setup should be checked, as given:

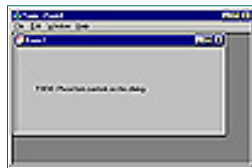
1. Check “Multiple documents.”
2. Check “None”; the defaults are OK.
3. Check “None”; the defaults are OK.
4. No features; all features should be unchecked.
 - a. Choose the “Advanced” button, and enter the file extension “mdi” for the document template string.
5. Check “No, thank you” for source file comments, and select the MFC library

6. Change each class name to `C_` rather than `C`.

- When you are finished, the “New Project Information” box should be filled out as shown in Figure 12-7.



1. Click “OK,” and the starter application will be generated.
2. Build the starter application and execute it. Figure 12-8 shows what the MDI application looks like at this stage. The application has a child window inside its client area. The AppWizard-generated icon named “FormDoc.ico” is placed on the child window, and the icon named “Form.ico” is the icon that is placed on the main window.



3. In a CFormView-derived application, the view class is a dialog box. It is named `IDD_FORM_FORM` in this example.

-

4. Now call ClassWizard and assign the names `m_sName`, `m_sBirthday`, and `m_sHobby` as the member variables for the corresponding edit boxes. These

member variables are automatically added to the view class by Class Wizard.

5. Make a message map entry in the view class for the EN_KILLFOCUS message for each edit control.

6. Save your file. Close ClassWizard and the Dialog Editor and you are ready to add code to finish your program.

The Multiple Document Template

The MDI document template, CMultiDocTemplate, has been automatically inserted by AppWizard into the application class. You will find the following code in the application's implementation file, which is named "Form.cpp":

```
CMultiDocTemplate* pDocTemplate;  
pDocTemplate = new CMultiDocTemplate(  
    IDR_FORMTYPE,  
    RUNTIME_CLASS(C_FormDoc),  
    RUNTIME_CLASS(C_ChildFrame), // custom MDI child frame  
    RUNTIME_CLASS(C_MyFormView));  
AddDocTemplate(pDocTemplate);
```

For an MDI application, CMultiDocTemplate plays a major role in coordinating the creation of the CDocument CMDIChildWnd, and CFormView classes. This document template "ties together" the document, the view, the MDI frame window, and the MDI child window, and it uses the referenced resources IDR_FORMTYPE.

IDR_FORMTYPE contains the application's resources, which are the document template string, the application menu, and the document icon. The remaining resources, known as IDR_MAINFRAME because they are linked to the main window, are a string for the mainframe caption bar, the mainframe menu, and the application icon. Accelerators and string tables are linked to the IDs that call them.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

The Trace Macro

The TRACE macro is a feature of all classes derived from CObject. It is used during the development of a program. The TRACE macro output goes to the debugging window when you are running in the Debug environment. When the program is executed in the Release mode (compiled without debugging features), the TRACE statements can be present, but are simply ignored and so do nothing. You do not have to remove them from your final code, and you do not have to define them for debug mode only. They are very convenient to use since you can leave them in your code.

The TRACE macro has the form:

TRACE (expression)

Where the expression specifies a variable number of arguments that are used in exactly the same way that a variable number of arguments are used in the run-time function printf().

Example:

```

int x = 1;
int y = 16;
float z = 32.0;
TRACE("This is a TRACE statement\n");
TRACE("x = %d and y = %d and z = %f\n",x,y,z);
  
```

We will be using the TRACE macro in our program “Form” to provide text output when we enter certain functions that we are interested in tracking.

The Document Class

We need to make some additions to the C_FormDoc class to provide essential services. Also, we need to store the data in the document class and then serialize it so that we can write this data to files and read it from files.

Member variables to store the data are defined in the document's header file, FormDoc.h. These variables will have the names m_sName, m_sBirthday, and m_sHobby, which are the same names that they are given in the view class. These variables provide the service of storing the data in the document class.

These member variables of C_FormDoc, m_sName, m_sBirthday, and m_sHobby, are written to file and retrieved from file, as required. This is done in the document's Serialize() function. We also add TRACE statements into the Serialize() function so that it will print text when it writes to file and reads from file.

In this example, we also override a number of the document's virtual functions so that we can follow when calls are being made to each of these functions. We will insert TRACE statements to print out text in each of these functions and then will follow these TRACE printouts as we run the program in the debug mode. The functions that we are overriding in the document class and a discussion of each of them is given in the following paragraphs.

The OnNewDocument() Function

The OnNewDocument() function is called by the framework as part of the “File | New” command. The default implementation of this function calls the DeleteContents() function to ensure that the document is empty and then marks the new document as “clean.” You can override this function to initialize the data structure for a new document.

In an SDI application, the document object gets used over and over again. In an MDI application, multiple document objects are created, as required. OnNewDocument() should always be used to initialize data in a new document rather than using the constructor. In an MDI application, you must use OnNewDocument(); do not use the constructor. The constructor will only get called once for the life of the program, yet the user might select “File | New” from the menu which calls OnNewDocument(). Using the constructor is adequate for an SDI, but to initialize each new document, you should get in the habit of using OnNewDocument() rather than the constructor.

The DeleteContents() Function

The DeleteContents() function is called by the framework to delete the document's data without destroying the document object itself. It is called by the OnNewDocument() function, by the OnOpenDocument() function, and by the OnCloseDocument() function. It is called to ensure that a document is empty before it is used. (This is particularly important for an SDI application, which uses only one document object.) It is also called just before the document is to be destroyed.

The OnCloseDocument() Function

This function is called by the framework when the document is closed, typically as part of the “File | Close” command. The default implementation of this function calls the DeleteContents() member function to delete the document’s data and then closes the frame windows for all the views attached to the document.

Override this function if you want to perform special clean-up processing when the framework closes a document. For example, if the document represents a record in a database, you may want to override this function to close the database.

The OnOpenDocument() Function

The OnOpenDocument() function is called by the framework as part of the “File | Open” command. The default implementation of this function opens the file and calls the DeleteContents() member function to ensure that the document is empty. If the user chooses the “File | Open” command in an SDI application, the framework uses this function to re-initialize the existing document object, rather than creating a new one. If the user chooses the “File | Open” command in an MDI application, the framework constructs a new document object each time and then calls this function to initialize it.

Override this function if you want to use something other than the archive mechanism for storing your data. For example, you might write an application where documents represent records in a database rather than separate files.

The OnSaveDocument() Function

The OnSaveDocument() function is called by the framework as part of the “File | Save” or “File | Save As...” command. The default implementation opens the file and then calls Serialize() to write the document’s data to the file.

Override this function if you want to perform special processing when the framework saves a document. For example, you might write an application where documents represent records in a database rather than separate files.

Previous	Table of Contents	Next

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

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

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

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

Code Additions to the Document Class

In the document's header file, we must declare the data members for storing the document's data. In the overrides, AppWizard has already added OnNewDocument() and the Serialize() function. We must add the prototype functions for the remainder of the functions that we are overriding. The code to add to the file FormDoc.h is shown below in bold and preceded by arrows. The non-bold code is what is already in your starter application's code.

```

// Attributes
public:
--> CString m_sName;
--> CString m_sBirthday;
--> CString m_sHobby;
. . . . .

// Overrides
. . . . .
public:
    virtual BOOL OnNewDocument();
    virtual void Serialize(CArchive& ar);
--> virtual BOOL OnOpenDocument(const char*);
--> virtual void DeleteContents();
--> virtual BOOL OnSaveDocument(const char*);
--> virtual void OnCloseDocument();
. . . . .
    
```

In the document's implementation file, FormDoc.cpp, we must complete all the overriding functions.

1. Add the following code, given in bold and preceded by an arrow, to the FormDoc.cpp code:

```

BOOL C_FormDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;
--> TRACE("OnNewDocument function entered\n");
--> m_sName = "Enter name here";
--> m_sBirthday = "Enter birthday here";
--> m_sHobby = "Enter hobby here";
    return TRUE;
}
-->BOOL C_FormDoc::OnOpenDocument(const char* pszPathName)
-->{
-->
-->    TRACE("OnOpenDocument function entered\n");
-->
    
```

```

        return(CDocument::OnOpenDocument(pszPathName));
-->}
--> void C_FormDoc::DeleteContents()
-->{
--> TRACE("DeleteContents function entered\n");
--> CDocument::DeleteContents();
-->}

--> BOOL C_FormDoc::OnSaveDocument(const char* pszPathName)
-->{
--> TRACE("OnSaveDocument function entered\n");
--> return(CDocument::OnSaveDocument(pszPathName));
-->}

-->void C_FormDoc::OnCloseDocument()
-->{
--> TRACE("OnCloseDocument function entered\n");
--> CDocument::OnCloseDocument();
-->}

. . . . .

void C_FormDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
--> ar << m_sName << m_sBirthday << m_sHobby;
--> TRACE("Storing data to file\n");
    }
    else
    {
--> ar >> m_sName >> m_sBirthday >> m_sHobby;
--> TRACE("Reading data from file\n");
    }
}

```

The View Class

In our example, “Form,” the data is input by the user into the view class; however, the data is then stored in the document class. The document class writes the data to file and reads the data from file, as required.

A view class derived from CFormView is chosen for this example because we could easily construct a small example for the specific purpose of studying the document-view architecture. The CFormView-derived class, C_MyFormView, is based upon a dialog template resource and is used almost like CDialog. There are some differences, however. DDX and DDV work, but you must take the responsibility of calling the CWnd::UpdateData() function manually at all the appropriate times. Also, the CDialog::OnInitDialog() function is not available for a CFormView-derived class. The functions CFormView::OnInitialUpdate() and CView::OnUpdate() must be used instead.

The CView::OnUpdate() Function

Since CDialog::OnInitDialog() is not available for the C_MyFormView class, we will use the virtual function CView::OnUpdate(). In the OnUpdate() function, which the C_MyFormView class inherits, the view’s data is updated with the document’s data. OnUpdate() is called by the framework after the view’s data has been updated. The default implementation invalidates the entire client area, marking it for painting when the next WM_PAINT message is received.

The default implementation of CFormView::OnInitialUpdate() always calls OnUpdate(); hence, the data will be initialized appropriately. OnUpdate() is also called by the CDocument::UpdateAllViews() function, which allows the view to update its display reflecting modifications.

A TRACE statement is added to the C_MyFormView::OnUpdate() function, so you can study the conditions under which this function is called by the MFC framework.

The CFormView::OnInitialUpdate() Function

CFormView::OnInitialUpdate() is called by the framework after the view is first attached to the document, but before the view is initially displayed. The default implementation of this function calls the OnUpdate() function. This function can be overridden to perform any one-time initialization that requires information about the document.

The function CFormView::OnInitialUpdate() has been overridden in the code, and a TRACE statement has been inserted in this function so you can study when the MFC framework calls this function. This function performs one-time initialization of the view. CFormView::OnInitialUpdate() overrides the CView::OnInitialUpdate() function to use DDX to set the initial values of the controls. Override CFormView::OnInitialUpdate() if you want to perform custom initialization. In this case, we only use it to include a TRACE statement to observe when it is called.

Previous	Table of Contents	Next

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

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

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

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[SEARCH
ITKNOWLEDGE](#)☒ Brief ☐ Full[Advanced](#)[Search](#)[Search Tips](#)[BROWSE
BY TOPIC](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

Data Transfer

CFormView is used almost like CDialog. The DDX and DDV work, but you must call UpdateData(TRUE) whenever you want the member variables updated with data from the controls. When data is input into any of the edit boxes, you must provide this code, which inserts the values from the controls into the view's member variables. You must also provide code to then transfer the data into the document class. In the "Form" program, when the user has completed typing the entry in the edit box and has then changed focus away from that edit box, an EN_KILLFOCUS message is sent, and we enter the handler function to update that data from the edit box. In each of the handler functions that respond to the EN_KILLFOCUS message, we manually call the UpdateData(TRUE) function.

You must call UpdateData(FALSE) when you want the data transferred from the view's member variables into the edit controls. In the OnUpdate() function, data is transferred from the document to the view's member variables. Then the data must be transferred into the edit boxes, so you must manually call the function UpdateData(FALSE).

In these handler functions, we must also keep track of when the data has been modified by the user. If at any time the data in the document is modified, we set a "dirty bit" in the document class by calling CDocument::SetModifiedFlag(). This will enable MFC to notify the user that the document needs to be saved if the user tries to quit the application, close the document, or open another document. This bit is cleared automatically when the document is saved.

Synchronizing Views

Since this is an MDI application, the user may have more than one view of the document open. When the document's data is changed, all the views must be synchronized. The function CDocument::UpdateAllViews() is called each time a handler function changes the document's data. If one of the views changes the data in the document, then all of the views must be notified of the change.

Code Additions to the View Class

1. Add the following code, shown in bold and preceded with arrows, to the FormView.h file. (The non-bold code is already there.)

```
// Overrides
protected:
    virtual void DoDataExchange(CDataExchange* pDX);
--> virtual void OnUpdate(CView* pSender, LPARAM lHint,
-->                      CObject* pHint);
--> virtual void OnInitialUpdate();
    . . . . .
```

2. Add the following code, shown in bold and preceded by arrows, to the FormView.cpp file. (The non-bold code is already in your file.)

```
-->void C_MyFormView::OnInitialUpdate()
-->{
-->    TRACE("OnInitialUpdate function entered\n");
-->    CFormView::OnInitialUpdate();
-->}

-->void C_MyFormView::OnUpdate(CView* pSender, LPARAM lHint,
-->                             CObject* pHint)
-->{
-->    TRACE("OnUpdate function entered\n");

-->    C_FormDoc* pDoc = GetDocument();
-->    m_sName = pDoc->m_sName;
-->    m_sBirthday = pDoc->m_sBirthday;
-->    m_sHobby = pDoc->m_sHobby;
-->    UpdateData(FALSE);
-->}

void C_MyFormView::OnKillfocusBirthday()
{
-->    UpdateData(TRUE);
-->    C_FormDoc* pDoc = GetDocument();
-->    pDoc->m_sBirthday = m_sBirthday;
-->    pDoc->SetModifiedFlag();
-->    pDoc->UpdateAllViews(NULL);
}

void C_MyFormView::OnKillfocusHobby()
{
-->    UpdateData(TRUE);
-->    C_FormDoc* pDoc = GetDocument();
-->    pDoc->m_sHobby = m_sHobby;
-->    pDoc->SetModifiedFlag();
-->    pDoc->UpdateAllViews(NULL);
}

void C_MyFormView::OnKillfocusName()
{
-->    UpdateData(TRUE);
-->    C_FormDoc* pDoc = GetDocument();
-->    pDoc->m_sName = m_sName;
```

```
--> pDoc->SetModifiedFlag();
--> pDoc->UpdateAllViews(NULL);
}
```

Running the “Form” Program in Debug Mode

You have been entering TRACE statements in the code, so that you can run the program in debug mode and tell when various CDocument functions have been entered. You also can tell when the CFormView functions OnInitialUpdate() and OnUpdate() have been entered.

1. Compile the program in debug mode.
2. Before running the program, go to the Microsoft Visual C++ program group.
 - a. Open “MFC Trace Options” and be sure that the “Enable Tracing” box is checked.
3. Open the debug output window that will have the trace statements from the menu by selecting “View | Output.” (For Win3.1 users, select “Window | Output.”)
4. Run your program by selecting “Build | Debug | Go.” (For Win3.1 users, select “Debug | Go.”)
5. Compile and run the program in debug mode. You will see when each function is called. An example output is shown in Figure 12-10. This program was run in debug mode with the output window open. When the program first opened up, the original view had no data in it and was named “Form1.” The data you see in Figure 12-10 was typed in and saved as Form1.mdi. “File | Open” was then chosen, and the file Form2.mdi, which had been previously saved to file, was then opened. The executing “Form” program is shown in Figure 12-10. Figure 12-11 shows the output window which contains the TRACE statements as they occurred for this sequence of operations.



Figure 12-10: The Executing “Form” Program



Figure 12-11: The TRACE Statements in the Debug Output Window

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

The “FormMin” Program With Minimum Code

In this section, a program named “FormMin” is generated without using AppWizard; it contains all the same features as the “Form” program. This code is used to show the minimum amount of code necessary for this application. This manually generated code has exactly the same style and names as the AppWizard-generated code. This allows an easy one-to-one comparison with the AppWizard code. And for Borland C++ programmers, it represents the program that they need to generate for this application; Borland C++ does not have compiler tools for generating a starter application.

“FormMin” Program Listing

Listing 12-1 gives the “FormMin” program. It is followed by a discussion of the important features to note about this code.

Listing 12-1: Source Code for the “FormMin” Program

```

-----
                                stdafx files
-----
//
// stdafx.h : include file for system include files,
//
#include <afxwin.h>      // MFC core and standard components
#include <afxext.h>      // MFC extensions

//
// stdafx.cpp : source file for the standard includes
//
#include "stdafx.h"

-----
                                application class
-----
//
// Form.h : main header file for the Form application
//
#include "resource.h"

class C_FormApp : public CWinApp
{
public:

```

```

        virtual BOOL InitInstance();

        DECLARE_MESSAGE_MAP()
    }

    //
    // Form.cpp : Defines the class behaviors for the application.
    //
    #include "stdafx.h"
    #include "Form.h"
    #include "MainFrm.h"
    #include "ChildFrm.h"
    #include "FormDoc.h"
    #include "FormView.h"

    BEGIN_MESSAGE_MAP(C_FormApp, CWinApp)
        // Standard file based document commands
        ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
        ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
    END_MESSAGE_MAP()

    C_FormApp theApp;

    BOOL C_FormApp::InitInstance()
    {
        LoadStdProfileSettings(); // Load INI file options (incl. MRU)
        CMultiDocTemplate* pDocTemplate;
        pDocTemplate = new CMultiDocTemplate(
            IDR_FORMTYPE,
            RUNTIME_CLASS(C_FormDoc),
            RUNTIME_CLASS(C_ChildFrame), // custom MDI child frame
            RUNTIME_CLASS(C_MyFormView));
        AddDocTemplate(pDocTemplate);

        // create main MDI Frame window
        C_MainFrame* pMainFrame = new C_MainFrame;
        if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
            return FALSE;
        m_pMainWnd = pMainFrame;

        // Enable drag/drop open
        m_pMainWnd->DragAcceptFiles();

        // enable DDE Execute open
        EnableShellOpen();
        RegisterShellFileTypes();

        // simple command line parsing
        if (m_lpCmdLine[0] == '\\0')
        {
            // create a new (empty) document
            OnFileNew();
        }
        else
        {
            // open an existing document
            OpenDocumentFile(m_lpCmdLine);
        }
    }

```



```

        pMainFrame->ShowWindow(m_nCmdShow);
        pMainFrame->UpdateWindow();

        return TRUE;
    }

-----
                                mainframe class
-----
//
//  MainFrm.h : interface of the C_MainFrame class
//
class C_MainFrame : public CMDIFrameWnd
{
    DECLARE_DYNAMIC(C_MainFrame)
};

//
//  MainFrm.cpp : implementation of the C_MainFrame class
//
#include "stdafx.h"
#include "Form.h"
#include "MainFrm.h"

IMPLEMENT_DYNAMIC(C_MainFrame, CMDIFrameWnd)

-----
                                MDI child class
-----
//
//  ChildFrm.h : interface of the C_ChildFrame class
//
class C_ChildFrame : public CMDIChildWnd
{
    DECLARE_DYNCREATE(C_ChildFrame)
};

//
//  ChildFrm.cpp : implementation of the C_ChildFrame class
//
#include "stdafx.h"
#include "Form.h"
#include "ChildFrm.h"

IMPLEMENT_DYNCREATE(C_ChildFrame, CMDIChildWnd)

-----
                                view class
-----
//
//  FormView.h : interface of the C_MyFormView class
//
class C_MyFormView : public CFormView
{
protected:
    C_MyFormView();
    DECLARE_DYNCREATE(C_MyFormView)

public:
    enum { IDD = IDD_FORM_FORM };

```

```

        CString    m_sBirthday;
        CString    m_sHobby;
        CString    m_sName;

        C_FormDoc* GetDocument();
protected:
        virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
        virtual void OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint);
        virtual void OnInitialUpdate();

        //{AFX_MSG(C_MyFormView)
        afx_msg void OnKillfocusBirthday();
        afx_msg void OnKillfocusHobby();
        afx_msg void OnKillfocusName();
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};

#ifdef _DEBUG // debug version in FormView.cpp
inline C_FormDoc* C_MyFormView::GetDocument()
    { return (C_FormDoc*)m_pDocument; }
#endif

//
// FormView.cpp : implementation of the C_MyFormView class
//
#include "stdafx.h"
#include "Form.h"
#include "FormDoc.h"
#include "FormView.h"

IMPLEMENT_DYNCREATE(C_MyFormView, CFormView)

BEGIN_MESSAGE_MAP(C_MyFormView, CFormView)
    //{AFX_MSG_MAP(C_MyFormView)
    ON_EN_KILLFOCUS(IDC_BIRTHDAY, OnKillfocusBirthday)
    ON_EN_KILLFOCUS(IDC_HOBBY, OnKillfocusHobby)
    ON_EN_KILLFOCUS(IDC_NAME, OnKillfocusName)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

C_MyFormView::C_MyFormView()
    : CFormView(C_MyFormView::IDD)
{
}

void C_MyFormView::DoDataExchange(CDataExchange* pDX)
{
    CFormView::DoDataExchange(pDX);
    DDX_Text(pDX, IDC_BIRTHDAY, m_sBirthday);
    DDX_Text(pDX, IDC_HOBBY, m_sHobby);
    DDX_Text(pDX, IDC_NAME, m_sName);
}

void C_MyFormView::OnInitialUpdate()
{
    TRACE("OnInitialUpdate function entered\n");
    CFormView::OnInitialUpdate();
}

```

```

void C_MyFormView::OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint)
{
    TRACE("OnUpdate function entered\n");
    C_FormDoc* pDoc = GetDocument();
    m_sName = pDoc->m_sName;
    m_sBirthday = pDoc->m_sBirthday;
    m_sHobby = pDoc->m_sHobby;
    UpdateData(FALSE);
}

void C_MyFormView::OnKillfocusBirthday()
{
    UpdateData(TRUE);
    C_FormDoc* pDoc = GetDocument();
    pDoc->m_sBirthday = m_sBirthday;
    pDoc->SetModifiedFlag();
    pDoc->UpdateAllViews(NULL);
}

void C_MyFormView::OnKillfocusHobby()
{
    UpdateData(TRUE);
    C_FormDoc* pDoc = GetDocument();
    pDoc->m_sHobby = m_sHobby;
    pDoc->SetModifiedFlag();
    pDoc->UpdateAllViews(NULL);
}

void C_MyFormView::OnKillfocusName()
{
    UpdateData(TRUE);
    C_FormDoc* pDoc = GetDocument();
    pDoc->m_sName = m_sName;
    pDoc->SetModifiedFlag();
    pDoc->UpdateAllViews(NULL);
}

-----
                                document class
-----
//
//  FormDoc.h : interface of the C_FormDoc class
//
class C_FormDoc : public CDocument
{
    DECLARE_DYNCREATE(C_FormDoc)

public:
    CString m_sName;
    CString m_sBirthday;
    CString m_sHobby;

    virtual BOOL OnNewDocument();
    virtual void Serialize(CArchive& ar);
    virtual BOOL OnOpenDocument(const char*);
    virtual void DeleteContents();
    virtual BOOL OnSaveDocument(const char*);
    virtual void OnCloseDocument();
};

```

```

//
// FormDoc.cpp : implementation of the C_FormDoc class
//
#include "stdafx.h"
#include "Form.h"
#include "FormDoc.h"

IMPLEMENT_DYNCREATE(C_FormDoc, CDocument)

BOOL C_FormDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;
    TRACE("OnNewDocument function entered\n");
    m_sName = "Enter name here";
    m_sBirthday = "Enter birthday here";
    m_sHobby = "Enter hobby here";

    return TRUE;
}

BOOL C_FormDoc::OnOpenDocument(const char* pszPathName)
{
    TRACE("OnOpenDocument function entered\n");
    return(CDocument::OnOpenDocument(pszPathName));
}

void C_FormDoc::DeleteContents()
{
    TRACE("DeleteContents function entered\n");
    CDocument::DeleteContents();
}

BOOL C_FormDoc::OnSaveDocument(const char* pszPathName)
{
    TRACE("OnSaveDocument function entered\n");
    return(CDocument::OnSaveDocument(pszPathName));
}

void C_FormDoc::OnCloseDocument()
{
    TRACE("OnCloseDocument function entered\n");
    CDocument::OnCloseDocument();
}

void C_FormDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        ar << m_sName << m_sBirthday << m_sHobby;
        TRACE("Storing data to file\n");
    }
    else
    {
        ar >> m_sName >> m_sBirthday >> m_sHobby;
        TRACE("Reading data from file\n");
    }
}

```

resource files

```

-----
//
//      resource.h
//
#define IDD_FORM_FORM                101
#define IDR_MAINFRAME                128
#define IDR_FORMTYPE                 129
#define IDC_STATIC_NAME              1000
#define IDC_STATIC_BIRTHDAY          1001
#define IDC_STATIC_HOBBY             1002
#define IDC_NAME                     1003
#define IDC_BIRTHDAY                 1004
#define IDC_HOBBY                    1005

//
//      Form.rc
//
#include "resource.h"
#include "afxres.h"

IDR_MAINFRAME          ICON      DISCARDABLE    "res\\Form.ico"
IDR_FORMTYPE           ICON      DISCARDABLE    "res\\FormDoc.ico"

IDR_MAINFRAME MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&New\tCtrl+N",          ID_FILE_NEW
        MENUITEM "&Open...\tCtrl+O",      ID_FILE_OPEN
        MENUITEM SEPARATOR
        MENUITEM "Recent File",            ID_FILE_MRU_FILE1, GRAYED
        MENUITEM SEPARATOR
        MENUITEM "E&xit",                  ID_APP_EXIT
    END
END

IDR_FORMTYPE MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&New\tCtrl+N",          ID_FILE_NEW
        MENUITEM "&Open...\tCtrl+O",      ID_FILE_OPEN
        MENUITEM "&Close",                ID_FILE_CLOSE
        MENUITEM "&Save\tCtrl+S",          ID_FILE_SAVE
        MENUITEM "Save &As...",            ID_FILE_SAVE_AS
        MENUITEM SEPARATOR
        MENUITEM "Recent File",            ID_FILE_MRU_FILE1, GRAYED
        MENUITEM SEPARATOR
        MENUITEM "E&xit",                  ID_APP_EXIT
    END
    POPUP "&Window"
    BEGIN
        MENUITEM "&New Window",            ID_WINDOW_NEW
        MENUITEM "&Cascade",                ID_WINDOW_CASCADE
        MENUITEM "&Tile",                  ID_WINDOW_TILE_HORZ
        MENUITEM "&Arrange Icons",          ID_WINDOW_ARRANGE
    END
END
END

```

```

IDR_MAINFRAME ACCELERATORS PRELOAD MOVEABLE PURE
BEGIN
    "N",          ID_FILE_NEW,          VIRTKEY, CONTROL
    "O",          ID_FILE_OPEN,        VIRTKEY, CONTROL
    "S",          ID_FILE_SAVE,        VIRTKEY, CONTROL
END

IDD_FORM_FORM DIALOG DISCARDABLE  0, 0, 167, 93
STYLE WS_CHILD
FONT 8, "MS Sans Serif"
BEGIN
    LTEXT          "Name:", IDC_STATIC_NAME, 28, 18, 22, 8, NOT WS_GROUP
    LTEXT          "Birthday:", IDC_STATIC_BIRTHDAY, 28, 36, 28, 8, NOT WS_GROUP
    LTEXT          "Hobby:", IDC_STATIC_HOBBY, 28, 55, 24, 8, NOT WS_GROUP
    EDITTEXT       IDC_NAME, 61, 18, 69, 14, ES_AUTOHSCROLL
    EDITTEXT       IDC_BIRTHDAY, 61, 36, 69, 14, ES_AUTOHSCROLL
    EDITTEXT       IDC_HOBBY, 61, 55, 70, 14, ES_AUTOHSCROLL
END

STRINGTABLE PRELOAD DISCARDABLE
BEGIN
    IDR_MAINFRAME          "Form"
    IDR_FORMTYPE           "\nForm\nForm\nForm Files (*.mdi)\n.MDI\nForm.Document\nForm Document"
END
-----

```

Previous	Table of Contents	Next

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

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

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

Keyword

☒ Brief
 ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Discussion of the “FormMin” Program

In this application you can launch a file by dragging it from the “Windows Explorer” (“Program Manager” in Win3.1) and dropping it on the running application. To have the drag-drop capability, you must call the function `CWnd::DragAcceptFiles()`, which has been inherited by the application’s mainframe window. The member variable, `m_pMainWnd`, of the application object points to the `CMDIFrameWnd` object. In `InitInstance()` the following line of code enables the drag-drop capability:

```
m_pMainWnd->DragAcceptFiles();
```

The mainframe class and the MDI child class have minimal amounts of code in them. This is due to the fact that the MDI functionality has been internalized (built in), both in the MFC classes and in the API functions that the MFC classes call.

The final item to note is that the program “FormMin” also has a minimal menu. The menu item “Edit” that appeared in the AppWizard-generated code is used only for `CEdit`-derived classes. We do not include the “Edit” menu item in the “FormMin” code. Likewise, we do not include the “Help” menu item in the “FormMin” code. The menus used in “FormMin” are simplified. They are shown in Figures 12-12 and 12-13.

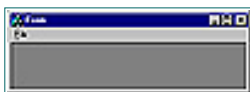


Figure 12-12: The “FormMin” Executing Program With Documents Closed

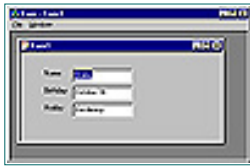


Figure 12-13: The “FormMin” Executing Program With an Open Document

Summary

MDI applications have a structure in which the MDI frame window object occupies the entire client area (except for the toolbar and status bar) of the mainframe window. The MDI frame window object parents the MDI child frames, where an MDI child frame is constructed to hold each view. Multiple views can be made of a document with the menu choice “Window | New Window.” New documents can be made with the menu choice “File | New.”

An MDI application has two menus. The IDR_MAINFRAME menu is attached to the mainframe window and is displayed when no documents are open. The second menu, known as the application menu, is displayed when a document is open. The application menu holds the menu item “Window” which opens a submenu of items with which the user can interact with the MDI child windows. An MDI application has built-in keyboard accelerators. Most of the code of an MDI frame window and its MDI child frame windows has been built in to the base MFC classes and the API functions that these MFC classes use; there is minimal code in your application.

Use CFormView-derived class if you want form-based documents. The CFormView class is derived from CScrollView and it supports scrolling. The CFormView class has many of the characteristics of a modeless dialog box; it is associated with a dialog resource that defines the frame characteristics and lists the controls. Since the CFormView is derived from CScrollView rather than CDialog, you cannot assume that the CDialog member functions are supported. CFormView does not have the virtual functions OnInitDialog(), OnOK(), or OnCancel() and it does not call UpdateData(). The CFormView object receives notification messages directly from its controls and it also receives command messages from the application framework; it can be controlled from the main menu.

TRACE macros can be used during the development of your program. When you run your program in debug mode, the TRACE statements are exercised and they print their output in the debugging window. These statements do not need to be removed from your code when you are finished debugging, because they are ignored when the program is run in the Release mode.

Using TRACE statements, we explored the various function calls made by the framework. The virtual CDocument functions that the framework calls when reading and writing files are: OnNewDocument(), DeleteContents(), OnCloseDocument(), OnOpenDocument(), OnSaveDocument(), and Serialize(). The virtual CView functions that the framework calls under specific circumstances are: OnInitialUpdate() and OnUpdate().

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

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

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

☒ Brief ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Chapter 13

Toolbars and Status Bars

This chapter presents an application with additional features that have not been discussed in previous chapters. In this chapter you will learn how to create a view that will scroll, how to customize the toolbar, how to customize the status bar, and how to use more than one document template in an application. These features will be illustrated using an MDI application. (They can also be applied to SDIs.) The view is derived from CScrollView, and scroll bars are added. In this example, a toolbar is included and customized. Also, a status bar is included and is customized to display specific messages when certain events occur. Two document templates are used. The first document template provides a view that can be scrolled. The second document template also provides a view that can be scrolled, but it has a dynamic splitter window added. The user chooses which document template to use.

The Bars Example

The “Bars” example that is developed in this chapter has scroll bars, a toolbar, a status bar, and uses a second document template. If you choose the second document template, you get a window that has splitter boxes, and you can dynamically split the window. When the application first opens up, the user sees a query box (shown in Figure 13-1) from which he can choose either the regular “Bars” application or the “Bars with Dynamic Splitters.”



Figure 13-1: Opening Selection for Document Template

When the first selection, “Bars,” is made for the document template, the regular view opens up with no splitter boxes. This is shown in Figure 13-2.



Figure 13-2: The Bars Regular View Based on First Document Template

The view is initialized with a yellow ellipse centered in the view’s client area. The application has a menu item “BrushColor,” which is used to change the color of the ellipse to yellow, green, or black, and a menu item “Rotation,” which is used to rotate the ellipse to a vertical position or to a horizontal position. Scroll bars will automatically be included when needed, which is when the window is resized, or the ellipse is rotated and part of it is no longer visible. The toolbar functions like the menu items. The status bar displays strings generated by the application.

The application is an MDI. Multiple views of a document as well as multiple documents can be open. When a new document is opened, the user must choose the document template to be used. Figure 13-3 illustrates multiple views and documents, by showing a view with a splitter window opened.



Figure 13-3: Bars Application with Multiple Views and Documents

Creating the Bars Starter Application

1. First, using AppWizard, build an MDI starter application. (We will briefly cover the steps for the Visual C++ 4 compiler. For the other compilers, consult the appendix covering your compiler and make the corresponding choices for your compiler.)
 - a. For the Visual C++4 compiler, choose “File | New | Project Workspace | AppWizard.”
 - b. Name this application “Bars,” since the toolbar and status bar features are illustrated in this example.
 - c. Make the following choices in the steps:
 - Step 1—choose “Multiple documents” (default settings).
 - Step 2—choose “None” (default settings).
 - Step 3—choose “None” (default settings).
 - Step 4—select toolbar, select status bar, deselect printing and print preview, deselect 3D Controls, choose the “Advanced button,” and enter the file extension of “bar.”
 - Step 5—choose “No, thank you” for source file comments and use MFC “As a statically linked library.”
 - Step 6—change the class names from C to C_ and set the base class for the view as CScrollView. When you are finished, your

“New Project Information” box should look like Figure 13-4.

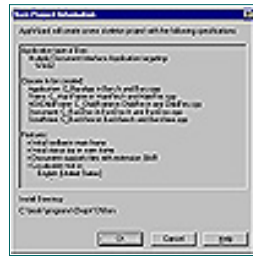


Figure 13-4: The AppWizard Starter Application information

d. Choose the OK button.

2. At this point, compile your application and it looks like Figure 13-5.



Figure 13-5: The Executing Starter Application

A toolbar and a status bar are attached to the mainframe window, and a menu item “View” is available to hide or show both the toolbar and the status bar. When either the toolbar or the status bar are hidden, the client area real estate is renegotiated and given to the MDI main frame.

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

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

Designing the Document Class

We use an ellipse class, named C_Ellipse, which is much like the one used in Chapter Ten. We place the C_Ellipse class in the document class. We add a serialize function for the C_Ellipse class, and in the C_BarsDoc serialize function we call the serialize function of m_Ellipse, the object of class C_Ellipse. Remember to add code to the CDocument's OnNewDocument() function so that when the user chooses "File | New" from the menu we have the document initialized. We enter additional code to the starter application to do all of this.

1. In the header file of the document class, BarsDoc.h, enter the following code:

```

-->class C_Ellipse : public CObject
-->{
-->    DECLARE_SERIAL(C_Ellipse)

-->public:
-->    COLORREF m_BrushColor;
-->    CPoint m_EllipseCenter;
-->    CRect m_OriginalPos;
-->    CRect m_RotatedPos;
-->    int m_nRotation;
-->    // the two possible rotations of the ellipse
-->    #define ROTATED90    1
-->    #define ROTATED180   2

-->    void BoundingRect(CRect& r);
-->    void Rotate();
-->    virtual void Serialize(CArchive& ar);
-->};
    . . . . .
  
```

2. In the BarsDoc.h file, add the following data member:

```

-->//Attributes
public:
-->C_Ellipse m_Ellipse;
  
```

3. In the implementation file of the document class, BarsDoc.cpp, enter the following code, which defines the C_Ellipse functionality:

```

-->IMPLEMENT_SERIAL(C_Ellipse, CObject, 1)
-->void C_Ellipse::BoundingRect(CRect& r)
-->{
-->    m_EllipseCenter.x = r.Width() / 2;
-->    m_EllipseCenter.y = r.Height() / 2;
-->    CSize HalfSizeOrig( 300 / 2, 100 / 2 );
-->    m_OriginalPos.TopLeft() = m_EllipseCenter - HalfSizeOrig;
-->    m_OriginalPos.BottomRight() = m_EllipseCenter + HalfSizeOrig;
-->}
-->void C_Ellipse::Rotate()
-->{
-->    switch (m_nRotation)
-->    {
-->        case ROTATED180:
-->            m_RotatedPos = m_OriginalPos;
-->            break;
-->        case ROTATED90:
-->            CSize HalfSizeRotated( 100 / 2, 300 / 2);
-->            m_RotatedPos.TopLeft() =
-->                m_EllipseCenter - HalfSizeRotated;
-->            m_RotatedPos.BottomRight() =
-->                m_EllipseCenter + HalfSizeRotated;
-->    }
-->}
-->void C_Ellipse::Serialize(CArchive& ar)
-->{
-->    // m_nRotation is an int, so to write code that will work
-->    // for either Win3.1 or Win32, we convert this int to type WORD
-->    // In Win3.1, ints are not supported for serialization.
-->    // In Win32, ints are supported for serialization.
-->    WORD r;
-->    if (ar.IsStoring())
-->    {
-->        ar << m_BrushColor;
-->        r = m_nRotation;
-->        ar << r;
-->    }
-->    else
-->    {
-->        ar >> m_BrushColor;
-->        ar >> r;
-->        m_nRotation = (int)r;
-->    }
-->}

```

4. In the document's implementation file, fix the Serialize() and OnNewDocument() functions that AppWizard has entered to look like the following:

```

BOOL C_BarsDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;

--> m_Ellipse.m_BrushColor = RGB(255, 255, 0);
--> m_Ellipse.m_nRotation = ROTATED180;

    return TRUE;
}

```

```

void C_BarsDoc::Serialize(CArchive& ar)
{
--> m_Ellipse.Serialize(ar);
}

```

Designing the View Class

In the view class, we need to do various things. We need to draw the ellipse in the OnDraw() function. We need to make sure that the document's size is described for the scroll bars. We need to write code that provides displays in the status bar, and we need to add our application's menu items and handler functions for changing the ellipse's color and rotation.

Drawing

First, we will discuss the overriding functions for this application, which includes the OnDraw() function. For setting the document size for the scroll bars, we are going to override the function OnUpdate(). So we need to add this function into the header file for the view class, named BarsView.h. After we are finished, the overriding functions in the .h file should look like the following code. (Some compilers do not add the PreCreateWindow() function, which we are not going to use anyway.) The functions that we will be overriding are shown in the following code and are OnDraw() and OnUpdate(). There are usually overrides added automatically, which will not be used; these functions are added in simply because it is likely that they will be overridden in an application. In this application we do not override the PreCreateWindow() function.

```

// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(C_BarsView)
public:
    virtual void OnDraw(CDC* pDC); // overridden to draw
                                   // this view
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
protected:
    virtual void OnInitialUpdate(); // called first after
                                   // construct
--> virtual void OnUpdate(CView* pSender, LPARAM lHint,
-->                      COleObject* pHint);
//}}AFX_VIRTUAL

```

We next discuss the overridden function for the drawing function. (In the next section, we will discuss the functions OnInitialUpdate() and OnUpdate() and how these are used for the document's size.) When you are finished with the OnDraw() function, its definition in the BarsView.cpp file should look like the following code:

```

void C_BarsView::OnDraw(CDC* pDC)
{
    C_BarsDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
--> CRect r;
--> GetClientRect(&r);
--> pDoc->m_Ellipse.BoundingRect(r);
--> pDoc->m_Ellipse.Rotate();
--> CBrush newBrush(pDoc->m_Ellipse.m_BrushColor);
--> CBrush* pOldBrush = pDC->SelectObject(&newBrush);
--> pDC->Ellipse(pDoc->m_Ellipse.m_RotatedPos);
--> pDC->SelectObject(pOldBrush);
}

```

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

Adding Scrolling

To add scrolling, the view class must be derived from CScrollView instead of CView, which we have done. We need to pass the document's size to the SetScrollSizes() member function of CScrollView whenever the ellipse is rotated. The SetScrollSizes() function has four arguments, as shown in Table 13-1; we use the first two arguments in this example and rely on the defaults for the last two arguments.

The ellipse size will change when the user rotates it. In this example, the document size is defined by the ellipse's bounding rectangle. The size of the document in the x direction and the size in the y direction are swapped when the user rotates the ellipse. Since we have variable-sized documents, we use the OnUpdate() function to update the scrolling limits. The OnUpdate() function will be called every time the document changes.

Table 13-1: The CScrollView::SetScrollSizes() Function

CScrollView::SetScrollSizes(int nMapMode, SIZE sizeTotal, SIZE sizePage, SIZE sizeLine).

Argument	Meaning
nMapMode	The mapping mode to set for the view.
sizeTotal	The total size of the scroll view. The cx member contains the horizontal extent. The cy member contains the vertical extent.
sizePage	The horizontal and vertical amounts to scroll in each direction in response to a mouse click in a scroll bar shaft. The cx member contains the horizontal amount. The cy member contains the vertical amount. The default values are sizeTotal.cx / 10 and sizeTotal.cy / 10.

sizeLine	The horizontal and vertical amounts to scroll in each direction in response to a mouse click in a scroll arrow. The cx member contains the horizontal amount. The cy member contains the vertical amount. The default values are sizeTotal.cx / 100 and sizeTotal.cy / 100.

The virtual function OnInitialUpdate() has been automatically inserted in the file BarsView.h. In the BarsView.cpp file the OnInitialUpdate() function has been automatically coded (by Visual C++ 4—earlier versions do not do this) to supply the SetScrollSizes() parameters. In our application, this will not suffice. We need to use OnUpdate() instead. We delete the OnInitialUpdate() function from both the .h and .cpp files. We add code to the OnUpdate() overriding function in BarsView.cpp, so that it reads as follows:

```
-->void C_BarsView::OnUpdate(CView* pSender, LPARAM lHint,
-->                          CObject* pHint)
-->{
-->    CSize sizeTotal;
-->    // calculate the total size of this view
-->    C_BarsDoc* pDoc = GetDocument();
-->    switch(pDoc->m_Ellipse.m_nRotation)
-->    {
-->        case ROTATED180:
-->            sizeTotal.cx = 300;
-->            sizeTotal.cy = 100;
-->            break;
-->        case ROTATED90:
-->            sizeTotal.cx = 100;
-->            sizeTotal.cy = 300;
-->    }
-->    SetScrollSizes(MM_TEXT, sizeTotal);
-->    Invalidate();
-->}
```

The framework's responsibilities in managing the scroll bars are as follows:

- Handles all WM_HSCROLL and WM_VSCROLL messages, scrolls the document in response, and moves the scroll box accordingly.
The positions of the scroll boxes reflect where the currently displayed portion of the document resides relative to the rest of the document. If the user clicks on a scroll arrow at either end of the scroll bar, the document is scrolled one “line.” If the user clicks on either side of the scroll box, the document is scrolled one “page.” If the user drags the scroll box itself, the document is scrolled accordingly.
- Calculates a mapping between the lengths of the scroll bars and the height and width of the document, adjusts this scaling factor when the window is resized or when the size of the document changes, and in turn removes or adds scroll bars as needed.

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

Customizing the Status Bar

The standard setup for the status bar has been provided by AppWizard. It consists of four panes as shown in Figure 13-6.



Figure 13-6: Status Bar Standard Setup

The status bar window displays text in panes under program control. It supports two types of text panes—message line panes and status indicator panes. AppWizard provides a standard setup for a status bar. It is found in the static indicators array in the MainFrm.cpp file; the indicators array for the standard framework status bar is:

```
static UINT BASED_CODE indicators[] =
{
    ID_SEPARATOR,           // status line indicator
    ID_INDICATOR_CAPS,
    ID_INDICATOR_NUM,
    ID_INDICATOR_SCRL,
};
```

The ID_SEPARATOR identifies a message line pane assigned to pane 0. The next three lines of code identify status indicator panes. ID_INDICATOR_CAPS assigns the status of the Caps Lock key to pane 1. ID_INDICATOR_NUM assigns the status of the Num Lock key to pane 2. ID_INDICATOR_SCRL assigns the status of the Scroll Lock key to pane 3. Each indicator pane is sized to exactly fit its expected printout. The length of the message line pane expands to consume the rest of the room available.

To redefine the status indicator panes, you must take control of the entire status bar, which means that you would need to define your own status bar with its own ID (and add that ID to the resource file). This example does not provide any interesting items for the status indicator panes to display, so we will not redefine the status indicator panes.

We will, however, provide application-generated text to display on the message line in pane 0. The message line pane displays strings that our application generates dynamically in its message handler functions. To set the value of the message line, we get a pointer to the status bar object, and then call the CStatusBar::SetPaneText() member function with a zero-based index parameter and the text to be displayed. This code will appear in any message handler from which we want to display a message in the status bar.

The ID of the AppWizard-generated status bar is AFX_IDW_STATUS_BAR. We use GetDescendantWindow(AFX_IDW_STATUS_BAR) to get the pointer to the CWnd descendant window matching the ID specified. The pointer must then be cast to be of class CStatusBar. An example of the code to add to any message handler that you wish to display text from is as follows:

```
CStatusBar* pStatus = (CStatusBar*)
    AfxGetApp()->m_pMainWnd->GetDescendantWindow(AFX_IDW_STATUS_BAR);
pStatus->SetPaneText(0, "Ellipse is colored black");
```

Adding the Handler Functions

1. The menu items “BrushColor” and “Rotation” must be added to the “application menu” named IDR_BARSTYPE. Insert them between existing menu items as was done in the example given in Chapter Ten.
2. Use ClassWizard to insert the handler functions into the message maps and to stub-out these handler functions.
3. At this juncture in the application development, add the following handler code to the implementation file of the view class, BarsView.cpp:

```
void C_BarsView::OnColorBlack()
{
--> C_BarsDoc* pDoc = GetDocument();
--> pDoc->m_Ellipse.m_BrushColor = RGB(0, 0, 0);
--> pDoc->SetModifiedFlag();
--> pDoc->UpdateAllViews(NULL);
--> CStatusBar* pStatus = (CStatusBar*)
--> AfxGetApp()->m_pMainWnd->GetDescendantWindow(AFX_IDW_STATUS_BAR);
--> pStatus->SetPaneText(0, "Ellipse is colored black");
--> Invalidate();
}
```

```
void C_BarsView::OnUpdateColorBlack(CCmdUI* pCmdUI)
{
--> C_BarsDoc* pDoc = GetDocument();
--> if (pDoc->m_Ellipse.m_BrushColor == RGB(0, 0, 0))
-->     pCmdUI->SetCheck(1);
--> else
-->     pCmdUI->SetCheck(0);
}
```

```
void C_BarsView::OnColorGreen()
{
--> C_BarsDoc* pDoc = GetDocument();
--> pDoc->m_Ellipse.m_BrushColor = RGB(0, 255, 0);
--> pDoc->SetModifiedFlag();
--> pDoc->UpdateAllViews(NULL);
--> CStatusBar* pStatus = (CStatusBar*)
--> AfxGetApp()->m_pMainWnd->GetDescendantWindow(AFX_IDW_STATUS_BAR);
--> pStatus->SetPaneText(0, "Ellipse is colored green");
--> Invalidate();
-->}
}
```

```
void C_BarsView::OnUpdateColorGreen(CCmdUI* pCmdUI)
{
--> C_BarsDoc* pDoc = GetDocument();
--> if (pDoc->m_Ellipse.m_BrushColor == RGB(0, 255, 0))
-->     pCmdUI->SetCheck(1);
--> else
-->     pCmdUI->SetCheck(0);
}
```

```

}

void C_BarsView::OnColorYellow()
{
--> C_BarsDoc* pDoc = GetDocument();
--> pDoc->m_Ellipse.m_BrushColor = RGB(255, 255, 0);
--> pDoc->SetModifiedFlag();
--> pDoc->UpdateAllViews(NULL);
--> CStatusBar* pStatus = (CStatusBar*)
--> AfxGetApp()->m_pMainWnd->GetDescendantWindow(AFX_IDW_STATUS_BAR);
--> pStatus->SetPaneText(0, "Ellipse is colored yellow");
--> Invalidate();
}

void C_BarsView::OnUpdateColorYellow(CCmdUI* pCmdUI)
{
--> C_BarsDoc* pDoc = GetDocument();
--> if (pDoc->m_Ellipse.m_BrushColor == RGB(255, 255, 0))
-->     pCmdUI->SetCheck(1);
--> else
-->     pCmdUI->SetCheck(0);
}

void C_BarsView::OnRotated180()
{
--> C_BarsDoc* pDoc = GetDocument();
--> pDoc->m_Ellipse.m_nRotation = ROTATED180;
--> pDoc->SetModifiedFlag();
--> pDoc->UpdateAllViews(NULL);
--> CStatusBar* pStatus = (CStatusBar*)
--> AfxGetApp()->m_pMainWnd->GetDescendantWindow(AFX_IDW_STATUS_BAR);
--> pStatus->SetPaneText(0, "Ellipse rotated to the horizontal position");
--> Invalidate();
}

void C_BarsView::OnUpdateRotated180(CCmdUI* pCmdUI)
{
--> C_BarsDoc* pDoc = GetDocument();
--> if (pDoc->m_Ellipse.m_nRotation == ROTATED180)
-->     pCmdUI->SetCheck(1);
--> else
-->     pCmdUI->SetCheck(0);
}

void C_BarsView::OnRotated90()
{
--> C_BarsDoc* pDoc = GetDocument();
--> pDoc->m_Ellipse.m_nRotation = ROTATED90;
--> pDoc->SetModifiedFlag();
--> pDoc->UpdateAllViews(NULL);
--> CStatusBar* pStatus = (CStatusBar*)
--> AfxGetApp()->m_pMainWnd->GetDescendantWindow(AFX_IDW_STATUS_BAR);
--> pStatus->SetPaneText(0, "Ellipse rotated to the vertical position");
--> Invalidate();
}

void C_BarsView::OnUpdateRotated90(CCmdUI* pCmdUI)
{
--> C_BarsDoc* pDoc = GetDocument();

```

```
--> if (pDoc->m_Ellipse.m_nRotation == ROTATED90)
-->     pCmdUI->SetCheck(1);
--> else
-->     pCmdUI->SetCheck(0);
}
```

4. Build the application, and at this stage, it looks like Figure 13-7. In Figure 13-7, notice that the scroll bars appear, as needed, and the status bar displays the text provided within the handler function. (In the case shown, the color green is selected for the ellipse in the document named “Bars2.”)

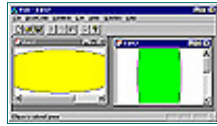


Figure 13-7: The Bars Application with its Menus and Scroll Bars

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Customizing the Toolbar

A toolbar is a CToolBar object. In its window it consists of a number of graphical buttons clustered in groups. AppWizard has provided the groupings and the buttons as shown in Figure 13-7. The graphical images for the buttons are stored in a single bitmap, toolbar.bmp, which AppWizard has provided in the RES subdirectory. This single bitmap is attached to the resource file IDR_MAINFRAME. When the buttons are clicked they send command messages. The CToolBar class incorporates *hit-testing* on regions of the toolbar to generate command messages. Hit-testing just means that the position of the cursor when the mouse is clicked is tested to determine which region the cursor is in. Update command UI message handlers are used to update the buttons' states, which, in turn, are used by the application framework to modify the buttons' graphical images.

Each button in a toolbar appears to have its own bitmap, but actually there is a single bitmap for the entire toolbar. The toolbar bitmap has a tile for each button which has the standard size of 15 pixels high and 16 pixels wide. The application framework supplies the button borders, and it modifies those borders, together with the button's bitmap tile color, to reflect the current button state.

We will first edit the toolbar that has been provided, adding five new tiles (or buttons) one at a time and inserting them between already existing tiles. Our new buttons will be used to select the brush color of yellow, green, or black and to select the rotation of 90 or 180. We will associate these new buttons with their command IDs.

The toolbar graphical editor differs between Visual C++ 4 and Visual C++ 1.5, so both of these will be covered.

Visual C++ 4 Toolbar Editing

1. To edit the toolbar, open its image editor. (For Visual C++ 4, it is listed as a toolbar resource.)
 - a. Open the IDR_MAINFRAME toolbar resource. An image window opens showing the bitmap. A graphics palette opens as well, showing the tool box, a color indicator, and the color palette.
 - b. Double-click on a bitmap button and its properties window opens.
 - c. To add new buttons (tiles), double-click on the tile at the end of the toolbar, and this tile opens up in the editing area. As soon as you edit this tile, a new “blank” tile opens up which you can use for your next addition.
 - d. Edit the tile until it looks like you want it to, then drag and drop it in the position that you want it to be in. It should then look like Figure 13-8.

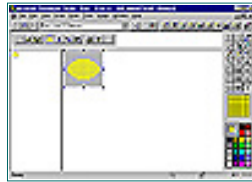


Figure 13-8: Editing a Toolbar Tile

2. To view its property page, double-click the button on the toolbar. We are adding the toolbar buttons after we have already established our menu and its message map entries. This means that the editor will not let us use the IDs that have already been used, so we will establish different IDs that will use the same UINT in the resource.h file. Upon examining the resource.h file, notice that ID_COLOR_YELLOW has been defined to be the UINT 32771. If we choose an ID of ID_BUTTON32771, it also will be defined to be an UINT of 32771 in the resource.h file. Thus ID_COLOR_YELLOW and ID_BUTTON32771 will be mapped to the same handler function.

Figure 13-9 shows the property page for the first tile that we just edited. *Tooltips* and status bar text are entered in the “Prompt” edit box.

Tooltips are rectangles with text that appear when the cursor is over the toolbar button. The first string is the string that will be displayed in the status bar’s pane 0. The delimiter \n follows the status bar string, and the text following the delimiter is the tooltip string.



Figure 13-9: The Toolbar Button Property Page

3. Continue adding and repositioning toolbar buttons.
 - a. To insert a space before a button that is not followed by a space, drag the button to the right or down until it overlaps the next button about halfway. When finished it looks like Figure 13-10.

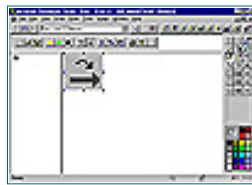


Figure 13-10: Completion of Toolbar Editing

4. Compile the program, and it looks like Figure 13-11. The toolbar buttons can be used in lieu of the menu items; the buttons are highlighted for the selected items. The toolbar tips work and the expected strings appear in the status bar's pane 0.

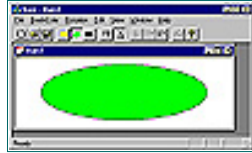


Figure 13-11: The Bars Application with Toolbar Completed

As toolbar editing was proceeding, the editor was building the toolbar resource in the .rc file. In the toolbar resource, the IDs must appear in exactly the same order that they appear in the toolbar bitmap. This code provides the one-to-one association between the button's position and the ID which will map to the handler function. The toolbar code, which the editor has written and located in the .rc file, is given in the following code:

```
IDR_MAINFRAME TOOLBAR DISCARDABLE    16,    15
BEGIN
    BUTTON            ID_FILE_NEW
    BUTTON            ID_FILE_OPEN
    BUTTON            ID_FILE_SAVE
    SEPARATOR
    BUTTON            ID_BUTTON32771
    BUTTON            ID_BUTTON32772
    BUTTON            ID_BUTTON32773
    SEPARATOR
    BUTTON            ID_BUTTON32774
    BUTTON            ID_BUTTON32775
    SEPARATOR
    BUTTON            ID_EDIT_CUT
    BUTTON            ID_EDIT_COPY
    BUTTON            ID_EDIT_PASTE
    SEPARATOR
    BUTTON            ID_FILE_PRINT
    BUTTON            ID_APP_ABOUT
END
```

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Visual C++ 1.5 Toolbar Editing

For the Visual C++ 1.5 compiler, the image editor contains the entire toolbar bitmap in its editing area.

1. Scroll the right-hand pane of the image window until the right-hand end of the bitmap is visible in the center of the pane.
 - a. Lengthen the bitmap by five tiles. This is done by dragging the resizing handle on the right-hand end of the bitmap to the right by the width of five tiles. The bitmap grows a full tile at a time. Notice that as you drag your bitmap, the Width text box in the properties window increments by 16 each time a new tile is created. When you have created five tiles the Width is 224 and it looks like the image shown in Figure 13-12.

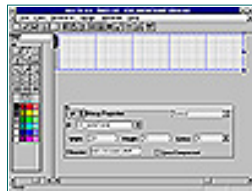


Figure 13-12: The Toolbar Bitmap Extended by Five Tiles

Now we want to place our newly created tiles such that they will become the fourth to the eighth tiles of the bitmap.

1. Enclose the five rightmost button images—the scissors to the question mark with the arrow—with the selection rectangle.
2. Drag the selected images five tiles to the right to open up the space for the new buttons. You will draw your bitmaps in these five tiles. When finished, it should look like Figure 13-13.

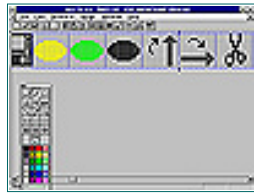


Figure 13-13: Positioning the New Button Tiles

After the bitmaps are drawn, you are ready to associate each bitmap button with its command ID. In MFC 2.5, this association is made in the mainframe class. (In MFC 4.0, the association is made in the .rc file.) The static buttons array, defined in the application's main frame class, associates commands with buttons. This array definition provides a 1-to-1 mapping based on the positions of the button tiles in the toolbar bitmap. The ID_SEPARATOR entries denote small amounts of extra space used to group the button tiles.

3. Open up the file mainfrm.cpp and add the following code given in bold and preceded by an arrow:

```
// toolbar buttons - IDs are command buttons
static UINT BASED_CODE buttons[] =
{
    // same order as in the bitmap 'toolbar.bmp'
    ID_FILE_NEW,
    ID_FILE_OPEN,
    ID_FILE_SAVE,
    ID_SEPARATOR,
    --> ID_COLOR_YELLOW,
    --> ID_COLOR_GREEN,
    --> ID_COLOR_BLACK,
    --> ID_SEPARATOR,
    --> ID_ROTATED_90,
    --> ID_ROTATED_180,
    --> ID_SEPARATOR,
    ID_EDIT_CUT,
    ID_EDIT_COPY,
    ID_EDIT_PASTE,
    ID_SEPARATOR,
    ID_FILE_PRINT,
    ID_APP_ABOUT,
};
```

4. When this code has been added, compile and run the program. The toolbar will be fully functional; the toolbar buttons will function just like the menu selections.



Figure 13-14: Adding a Splitter Window Class

Note: MFC 2.5 (Visual C++ 1.5) does not have tooltips.

Using Two Document Templates

Using two document templates in our application will be demonstrated. We create a splitter window, which will give us a window with dynamic splitter bars. We will add the document template with the splitter window, which means that we can open either a document without splitter windows (the one we have been using so far in this application), or we can open a document that has splitter windows. This section shows how all of this is done.

Adding a Dynamic Splitter To An MDI

To add dynamic splitter windows to an MDI application:

1. Derive a frame window class, which we will name `C_SplitterFrame`, from `CMDIChildWnd` and give this class a member variable of type `C_SplitterWnd`.
 2. Override the `OnCreateClient()` member function of the `C_SplitterFrame` window class to create a `C_SplitterWnd`.
 3. Use the new frame window class when opening documents—this is done by adding a new document template in the application class.
- Using ClassWizard to create the new class makes it easy to add splitter windows to an MDI application. ClassWizard provides an option that automatically derives a frame window class and overrides its `OnCreateClient()` member function. However, we will need to add the new document template manually (step 3). ClassWizard is used to do steps 1 and 2.

Note: There is very little difference between ClassWizard for Visual C++4 and Visual C++1.5 other than appearance, so only Visual C++4 is covered here.

We now use ClassWizard to do the above steps 1 and 2. Open ClassWizard and choose AddClass.

- a. In the query box that opens up, name the class `C_SplitterFrame` and set its base class to “splitter.”
- b. Name the header file `Splitter.h` and the implementation file `Splitter.cpp`. Your query box should have the entries as shown in Figure 13-14.
- c. Then choose “Create.”

When you choose the Create button to generate the new class, the ClassWizard dialog box regains the focus. The class is automatically created. As Figure 13-15 illustrates, ClassWizard has automatically overridden the OnCreateClient() function.

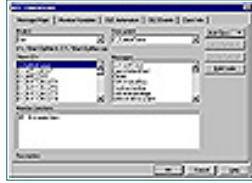


Figure 13-15: ClassWizard Adds OnCreateClient to Splitter Window

Choose the OK button to exit the ClassWizard dialog box. As before, the new files are automatically added to the project.

When you examine the code that ClassWizard has generated, you will see that it has written all the code that is needed. The overriding function `OnCreateClient()` that ClassWizard has created in the `Splitter.cpp` file is shown in the following code; you will not need to alter the code at all unless you want something other than a 2 x 2 splitter window.

```

        BOOL
C_SplitterFrame::OnCreateClient(LPCREATESTRUCT /*lpcs*/,
                                CCreateContext*
pContext)
    {
        return
m_wndSplitter.Create(this,
                      2, 2,          // TODO: adjust
the number of rows, columns
                      CSize(10, 10), // TODO:
adjust the minimum pane size
                      pContext);
    }

```

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

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

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

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



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Go!

Keyword

☒ Brief ☐ Full

+ Advanced

[Search](#)+ [Search Tips](#)BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

Go!

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

Adding a Document Template

C_SplitterFrame assumes the role that C_ChildFrame previously played in this application. Objects of three classes must cooperate in displaying a document: a C_SplitterFrame object, which manages the frame window; a CSplitterWnd object, which manages the document view's client area; and one or more C_BarsView objects, each of which manages a pane in the view window. The CSplitterWnd object is responsible for handling the C_BarsView objects as panes, managing their scroll bars, and drawing the splitter boxes and splitter bars.

Having defined a new frame window class, we must now use it when opening documents. This is done by adding a new document template in the implementation file of the application class.

1. We must open the Bars.cpp file and make some necessary code changes. We must include the header file Splitter.h; this is necessary to access the declaration of the C_SplitterFrame class. We need to make modifications in the InitInstance() member function of C_BarsApp. The added code calls AddDocTemplate() to register a document template which includes C_BarsDoc, C_SplitterFrame, and C_BarsView. Add the following code to the Bars.cpp file:

```
// Bars.cpp : Defines the class behaviors for the application.
//
```

```
#include "stdafx.h"
#include "Bars.h"
#include "MainFrm.h"
-->#include "Splitter.h"
#include "ChildFrm.h"
#include "BarsDoc.h"
#include "BarsView.h"
```

```
. . . .
```

```
BOOL C_BarsApp::InitInstance()
{
```



```

. . . . .

// Register document templates

CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CMultiDocTemplate(
    IDR_BARSTYPE,
    RUNTIME_CLASS(C_BarsDoc),
    RUNTIME_CLASS(C_ChildFrame), // custom MDI child frame
    RUNTIME_CLASS(C_BarsView));
AddDocTemplate(pDocTemplate);
--> AddDocTemplate(new CMultiDocTemplate(
--> IDR_BARSTYPE,
--> RUNTIME_CLASS(C_BarsDoc),
--> RUNTIME_CLASS(C_SplitterFrame), // mgs MDI child frame
--> RUNTIME_CLASS(C_BarsView)));

. . . . .

}

```

2. Compile the program and run it.

We have added a new document template, and the application will not know which one you want. The user will be asked to make the choice when the program begins or a new document is initiated by choosing “File | New.” When you initiate the program or a new document, the query box shown in Figure 13-16 is displayed. If the first string is chosen, the program runs without the splitter boxes; it chooses the first document template from the application class. If the second item is chosen, the window appears with the splitter boxes; for this choice it is using the second document template that was just added to the application class.

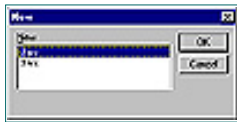


Figure 13-16: Query Box for Selecting Document Template

As the author of the application, you might not mind having the above query box that does not indicate (other than by position) which document template that you will get. However, a user of this program probably would not like this feature. So how do we fix this? We need to modify the resources used by the added document template, so that a different string appears in the query box shown in Figure 13-16.

Resources for Document Templates

As you saw in the second document template that we added above, we used the resource `IDR_BARSTYPE`, which was the same resource used in the first document template. This is why we have the same string in the document template query box shown in Figure 13-16. To have a different string appear for the second document template:

1. Define another resource with its own document template string.
 - a. Call it `IDR_SPLITTERTYPE` and enter the string that you want to appear in the appropriate place in this resource. This resource is then used in the definition of the second document template in the application class’s implementation file, which is shown in the following code:

```
AddDocTemplate(new CMultiDocTemplate(
--> IDR_SPLITTERTYPE,
    RUNTIME_CLASS(C_BarsDoc),
    RUNTIME_CLASS(C_SplitterFrame),
    RUNTIME_CLASS(C_BarsView)));
```

When we define the resource IDR_SPLITTERTYPE, we must define all the different types of resources that it needs. There will be an icon, a menu, and a string table entry which is the document template string. The .rc file (generated by AppWizard) has an icon, a menu, and a string table entry for the document template string defined for the resource IDR_BARSTYPE. When we generate the resource IDR_SPLITTERTYPE, we likewise need to define an icon, a menu, and a string table entry for the document template string. We do not want the icon or the menu to be different for the resource IDR_SPLITTERTYPE, so we will make a copy of and use the ones already defined for IDR_BARSTYPE.

2. Add the same icon defined for IDR_BARSTYPE as the icon for IDR_SPLITTERTYPE, as shown in the following code snippet from the .rc file:

```
        IDR_BARSTYPE        ICON DISCARDABLE    "res\\BarsDoc.ico"
-->  IDR_SPLITTERTYPE    ICON DISCARDABLE    "res\\BarsDoc.ico"
```

3. Add the same menu defined for IDR_BARSTYPE as the menu for IDR_SPLITTERTYPE, as shown in the following code snippet from the .rc file. (The menu is long and only the beginning and ending of each is shown.)

```
IDR_BARSTYPE        MENU        PRELOAD        DISCARDABLE
BEGIN
    &POPUP "&File"
        . . . . .
    POPUP "&Help"
    BEGIN
        MENUITEM "&About Bars...",        ID_APP_ABOUT
    END
END
-->IDR_SPLITTERTYPE        MENU        PRELOAD        DISCARDABLE
-->BEGIN
-->&POPUP "&File"
-->        . . . . .
--> POPUP "&Help"
--> BEGIN
-->        MENUITEM "&About Bars...",        ID_APP_ABOUT
--> END
-->END
```

4. Add a new string table entry for the IDR_SPLITTERTYPE resource which defines the document template string differently. We want to have separate string table entries for the document template string for the IDR_BARSTYPE resource and the IDR_SPLITTERTYPE resource. The document template strings will be defined differently. These string table entries will look like the following:

```
STRINGTABLE        PRELOAD        DISCARDABLE
BEGIN
    . . . . .
```

```

        IDR_BARSTYPE      "\nBars\nBars\nBars Files (*Bar)\n
        .BAR\nBars.Document\nBars Document
--> IDR_SPLITTERTYPE      "\nBars\nBars with Dynamic Splitter\n
--> Bars Files (*Bar)\n.BAR\nBars.Document\nBars Document
END

```

5. Although the preceding additions have fixed up the .rc file, we need to add IDR_SPLITTERTYPE to the resource.h file as shown in the following code:

```

//
//  resource.h
//
#define IDD_ABOUTBOX      100
#define IDR_MAINFRAME      128
#define IDR_BARSTYPE      129
-->#define IDR_SPLITTERTYPE  130
. . . . .

```

6. Having fixed the .rc file and the resource.h file, recompile and run the program. When this new program is run, the document template query box shown in Figure 13-17 appears. The rest of the program operates as before.



Figure 13-17: Query Box for Selecting Document Template

Previous	Table of Contents	Next

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

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

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

Keyword

☒ Brief
 ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Summary

To create a view that will scroll, your view class must be derived from CScrollView. The scrolling size parameters, which are the total size of the scroll view, the size of a scroll “page,” and the size of a scroll “line,” are set using the SetScrollSize() function. They are normally set in the OnInitialUpdate() function, but if the size of the document displayed in the view can change, they need to be set in the OnUpdate() function which gets called every time the document changes.

Status bars, which are windows of class CStatusBar, can be customized. The status bar is separated into a number of status indicator panes. Normally, there are four status bar panes. Pane 0 is referred to as the message line and is used to display a string, pane 1 is used to display the status of the Caps Locks key, pane 2 is used to display the status of the Num Lock key, and pane 3 is used to display the status of the Scroll Lock key. The message line of pane 0 can display a string from a string table, which is normally done, or it can display a string that is written by your application’s code. To display a string value in the message line, the function CStatusBar::SetPaneText() is used with its two input parameters, which are the zero-based index of the pane in which to display the text and the text that is to be displayed.

Toolbars, which are windows of class CToolBar, can be customized to suit your needs. The graphical images for the toolbar’s buttons are stored in a single bitmap, toolbar.bmp, which is attached to the resource file IDR_MAINFRAME. The toolbar bitmap has a tile for each button which has the standard size of 15 pixels high and 16 pixels wide. When the buttons are displayed, the framework supplies the button borders, modifying these borders and the button’s tile color to reflect the current button state (selected or not selected). Buttons can be edited and new buttons can be created and inserted between existing toolbar buttons. MFC 4.0 provides a toolbar resource in the

.rc file which maps the button to its handler function. MFC 4.0 associates an ID, a status bar string, and a tooltip string with each button; the status bar string and the tooltip string are displayed automatically by the framework. MFC 2.5 (for Win3.1 users) provides a static buttons array, defined in the mainframe class, that maps the button to its handler function. MFC 2.5 does not have tooltips.

Multiple document templates can be used in an application. As an example, a second document template that uses splitter windows can be added to the application class. When an application is started or a new document is opened, the framework supplies a query box for the user to select the document template. In order to display different resources for each document template, you must define a second resource type and provide all the resources it needs which are a menu, an icon, and a document template string.

Exercise

This exercise is to add to your checkers game the features that you have been learning since you began studying the document-view architecture. You have seen examples of how to separate the code into the document's data and the view. You have learned how to serialize the data and file it. You have learned the features of an MDI application. And in this chapter, you have learned how to customize toolbars.

Use an MDI application with a toolbar for your checkers game. The checkers game from the Chapter 8 Exercise can be used. You must separate the code into data that will be put into the document class, and the remainder of the code which will be put into the view class. The code separation can be much like the tic tac toe game of Chapter 11. In Chapter 11, you added history arrays and serialized them such that the play of the game could be stored to file and retrieved from file.

Customize the toolbar such that you have a toolbar button which can be used to make each of the color selections for the checkerboard and also to make each of the shape selections for the checkerboard pieces. Using the checkerboard game as an MDI application, you will be able to open different games in different windows and continue the play of multiple games by shifting the focus to a game's window and making a play, then shifting the focus to another window to make a play in the game displayed in that window, etc. Refer to the section "Chapter 13 Exercise" of Appendix B for more discussion of this assignment.

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

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

Keyword

☒ Brief
 ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Chapter 14

Custom Controls, New Common Controls, and Property Sheets

Controls are explored further in this chapter. First, we cover how to customize the standard Win3.1 common controls to suit the programmer's needs. The most commonly used customizing technique is to use *owner-draw* controls. The visual appearance and the behavior of an owner-draw control can be customized by the programmer. In this chapter we look at owner-draw controls and present an example using the CBitmapButton class. This custom control discussion applies to both Win3.1 and Win32 applications.

Next we cover the new common controls. These new common controls were introduced with MFC 4.0 and apply only to Win32 systems. This discussion of the new common controls does not apply to Win3.1 users. If you are using Windows 3.1x, you can ignore the contents of this chapter following the custom control discussion.

In the section covering the new common controls, examples are given of the frequently used new common control classes: CProgressCtrl, CSliderCtrl, CSpinButtonCtrl, CImageList, CListCtrl, and CTreeCtrl. The classes CPropertyPage and CPropertySheet are also presented in an example.

Custom Controls

A custom control acts like an ordinary control, but it is used to draw controls that are distinctly different than the ordinary controls. For example, an ordinary list control displays lists of text, but a customized list control can be made to display rectangles of color for the user to make a color selection. You

can paint anything you want in a custom control, which is also referred to as an owner-draw control. The Dialog Editor lets you position custom controls in dialog templates.

The operating system supports owner-draw controls by sending specific messages to the parent window of the control that allow you to customize the visual appearance and behavior of the control. These messages are handled in the MFC's message maps (in CWnd-derived classes) with: OnDrawItem(), OnMeasureItem(), OnCompareItem(), and OnDeleteItem(). These functions allow you access to data structures in which you can specify the drawing action that is required, specify the measurement and dimensions of your item, specify the position of the item, or specify the deletion of item-specific data. You override these functions in your control class to implement the owner-draw behavior, and your control can do its own drawing without any help from the parent.

MFC provides default implementation for the standard owner-draw messages. This default implementation will decode the owner-draw parameters and delegate the owner-draw messages to the controls. This is called *self-draw*, since the drawing code is in the class of the control, not in the owner window. This allows you to build reusable control classes that display the control, since the code for drawing the control is in the control class.

MFC 2.5 (and 4.0) provides the custom control class CBitmapButton derived from CButton. MFC 4.0 has added two more custom control classes: CCheckBox and CDragListBox, which are derived from CListBox. MFC 4.0 has introduced the button styles BS_BITMAP and BS_ICON which offer functionality similar to the CBitmapButton class.

We will look at the CBitmapButton class, which is the most typical example of a self-drawing control. A bitmap button is a button that shows one, two, three, or four bitmap images for the different states (up, down, focused, or disabled).

The MFC library's CBitmapButton class allows you to easily create buttons that are labeled with graphics instead of text. (And you don't have to call BitBlt.) The example given here shows you how to add bitmap buttons to a dialog. In this example, you will build a dialog with two bitmap buttons. The bitmaps are customized by dynamically subclassing them using the function SubclassDlgItem(). Subclassing is the term for replacing the WndProc of a window with a different WndProc and calling the old WndProc for default (superclass) functionality. A CWnd object gets attached to an existing window and you can modify the behavior in a derived class. This is called *dynamic subclassing*, since the behavior (and hence the class) of an object is changed at run time. Dynamic subclassing is done with either of the CWnd member functions:

```
CWnd::SubclassWindow( )
```

```
CWnd::SubclassDlgItem( )
```

These routines attach a CWnd object to an existing window. SubclassWindow() takes the window handle directly, and SubclassDlgItem() is

a helper that takes a control ID and the parent window (usually a dialog). SubclassDlgItem() is designed for attaching C++ objects to dialog controls created from a dialog template.

The CustCtrl Example

In the example given in this section, we build a dialog with two bitmap buttons. The “About” dialog box of any AppWizard-generated starter application is the most convenient dialog box in which to place the bitmap buttons and that is what is done in this example. The first button will have four states (up, down, focused, and disabled) and will be disabled when it initially appears. When the mouse is clicked on the first button, it will write a message line in the dialog box. The second button will have three states (up, down, and focused), and when the mouse is clicked on the second button, it will enable the first button and also write a message in the dialog box. When the “About” dialog box first appears, the first bitmap button is disabled, and it looks like Figure 14-1.



Figure 14-1: The CustCtrl Example When the About Dialog Opens

When the second bitmap button is clicked, the first button is enabled and a message is written on the dialog box. This is shown in Figure 14-2.



Figure 14-2: The Bitmap Buttons Functioning in Executable Program

Previous	Table of Contents	Next

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

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

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

Keyword

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Building the “CustCtrl” Program

To include a bitmap-button control in a dialog box, follow these steps:

1. Create one to four bitmap images for the button, which are the images for the different states a button can assume: up, down, focused, and disabled. Only the first bitmap is required; the others are optional. These bitmaps can be of any size, but all are treated as if they were the same size as the bitmap for the up state.
2. Create a dialog template (this example uses the application’s “About” dialog box). For each bitmap button, add an owner-draw button positioned where you want the bitmap button. The size of the button in the template does not matter.
3. Set each button’s caption to a value such as “Image1” and define a symbol for the button such as IDC_IMAGE1.
4. In the .rc file, give each of the bitmaps created for the button an ID constructed by appending one of the letters “U,” “D,” “F,” or “X” (for up, down, focused, or disabled) to the string used for the button caption in step 3. For the button caption “Image1,” for example, the IDs would be IMAGE1U, IMAGE1D, IMAGE1F, and IMAGE1X.
5. In the dialog’s class declaration, add a data member object for each CBitmapButton.
6. In the dialog’s constructor, call each CBitmapButton object’s LoadBitmap() function, using one to four parameters, where each parameter is the string identifying a bitmap for that button.
7. In the CDialog object’s OnInitDialog() routine, call each CBitmapButton object’s SubclassDlgItem() function, using as parameters the button’s control ID and the CDialog object’s *this* pointer. Then call the CBitmapButton object’s SizeToContent() function to resize the bitmap button to the size of the bitmap.
8. Since we want to handle the BN_CLICKED notification message sent by the bitmap-button control to its parent, use ClassWizard to add a message-map entry and a message-handler function for this message for each of the bitmap buttons. The notifications sent by a CBitmapButton object are the same as those sent by a CButton object.
9. In the OnImage2() handler function, enable the first bitmap button using the function CWnd::EnableWindow(). (In our example, the message-handler functions are used to write text into static controls, so that lines of text appear (or disappear) on the dialog box.)

Listing 14-1 shows the code that must be added to the CAboutDlg class, the code that must be added to the dialog resource script’s IDD_ABOUTBOX, the bitmaps that must be added to the .rc file, and the code that must be added to the resource.h file.

Listing 14-1: Source Code Excerpts for CustCtrl Example

```
-----
                                CAboutDlg in the application class
-----
//
//  application class implementation file
//
    . . . . .

////////////////////////////////////
//  CAboutDlg dialog used for App About
class CAboutDlg : public CDialog
{
protected:
--> CBitmapButton m_yellowMan;
--> CBitmapButton m_blueMan;

public:
    CAboutDlg();

// Dialog Data
    //{AFX_DATA(CAboutDlg)
    enum { IDD = IDD_ABOUTBOX };
    //}AFX_DATA

// Implementation
protected:
    virtual void DoDataExchange(CDataExchange* pDX);
--> virtual BOOL OnInitDialog();
    //{AFX_MSG(CAboutDlg)
--> afx_msg void OnImage1();
--> afx_msg void OnImage2();
    //}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
--> m_yellowMan.LoadBitmaps("Image1Up", "Image1Down", "Image1Focus",
-->                          "Image1Disabled");
--> m_blueMan.LoadBitmaps("Image2Up", "Image2Down", "Image2Focus");

    //{AFX_DATA_INIT(CAboutDlg)
    //}AFX_DATA_INIT
}
-->BOOL CAboutDlg::OnInitDialog()
-->{
--> m_yellowMan.SubclassDlgItem(IDC_IMAGE1, this);
--> m_yellowMan.SizeToContent();

--> m_blueMan.SubclassDlgItem(IDC_IMAGE2, this);
--> m_blueMan.SizeToContent();
--> return TRUE;
-->}
```

```

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CAboutDlg)
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
    //{{AFX_MSG_MAP(CAboutDlg)
    --> ON_BN_CLICKED(IDC_IMAGE1, OnImage1)
    --> ON_BN_CLICKED(IDC_IMAGE2, OnImage2)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

// App command to run the dialog
void CCustctrlApp::OnAppAbout()
{
    CAboutDlg aboutDlg;
    aboutDlg.DoModal();
}

////////////////////////////////////
// CCustctrlApp commands

-->void CAboutDlg::OnImage1()
-->{
-->    SetDlgItemText(IDC_LINE1, "I am m_yellowMan");
-->    SetDlgItemText(IDC_LINE2, " ");
-->    SetDlgItemText(IDC_LINE3, " ");
-->}

-->void CAboutDlg::OnImage2()
-->{
-->    SetDlgItemText(IDC_LINE1, " ");
-->    SetDlgItemText(IDC_LINE2, "I am m_blueMan");
-->    SetDlgItemText(IDC_LINE3, "I enable m_yellowMan");
-->    m_yellowMan.EnableWindow(TRUE);
-->}

-----
resource files
-----
//
//    excerpts from the .rc file

. . . . .

////////////////////////////////
// Dialog
//

IDD_ABOUTBOX DIALOG DISCARDABLE  34, 22, 255, 181
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "About Custctrl"

```

```

FONT 8, "MS Sans Serif"
BEGIN
    ICON                IDR_MAINFRAME, IDC_STATIC, 11, 17, 18, 20
    LTEXT                "Custctrl Application Version 1.0",
                        IDC_STATIC, 40, 10, 119, 8
    DEFPUSHBUTTON        "OK", IDOK, 206, 14, 32, 14, WS_GROUP
--> CONTROL            "Image1", IDC_IMAGE1, "Button", BS_OWNERDRAW |
-->                    WS_DISABLED | WS_TABSTOP, 54, 68, 50, 14
--> LTEXT                ". ", IDC_LINE1, 115, 73, 100, 8, NOT WS_GROUP
--> CONTROL            "Image2", IDC_IMAGE2, "Button", BS_OWNERDRAW |
-->                    WS_TABSTOP, 54, 127, 50, 14
--> LTEXT                ". ", IDC_LINE2, 115, 131, 102, 8, NOT WS_GROUP
--> LTEXT                ". ", IDC_LINE3, 115, 152, 109, 8, NOT WS_GROUP
END

//////////
// Bitmap
//

-->IMAGE1UP            BITMAP    MOVEABLE    PURE    "RES\\IMAGE1U.BMP"
-->IMAGE1DOWN          BITMAP    MOVEABLE    PURE    "RES\\IMAGE1D.BMP"
-->IMAGE1FOCUS          BITMAP    MOVEABLE    PURE    "RES\\IMAGE1F.BMP"
-->IMAGE1DISABLED       BITMAP    MOVEABLE    PURE    "RES\\IMAGE1X.BMP"
-->IMAGE2UP            BITMAP    MOVEABLE    PURE    "RES\\IMAGE2U.BMP"
-->IMAGE2DOWN          BITMAP    MOVEABLE    PURE    "RES\\IMAGE2D.BMP"
-->IMAGE2FOCUS          BITMAP    MOVEABLE    PURE    "RES\\IMAGE2F.BMP"

//
//      resource.h file
//
#define IDR_MAINFRAME                2
#define IDD_ABOUTBOX                100
-->#define IDC_IMAGE1                1000
-->#define IDC_LINE1                1001
-->#define IDC_IMAGE2                1002
-->#define IDC_LINE2                1003
-->#define IDC_LINE3                1006
-----

```

Previous	Table of Contents	Next

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

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

Keyword

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

The New Common Controls

The new common controls were introduced with MFC 4.0. Previous versions of MFC do not support the controls that we discuss in this section. The code for these new common controls is implemented in the Windows COMCTL32.DLL file. The new common controls can be used as child windows, but most frequently will be used in dialog boxes or other windows based on a dialog template. When your program initializes a dialog, it uses a symbolic class name in the dialog resource to connect to the window procedure in the DLL. Your program owns the control window, but the code resides in the DLL.

The new common controls are listed in Table 14-1 along with their MFC class and their WNDCLASS name, which resides in the COMCTL32.DLL file. In the next example we will use the first six common controls listed in Table 14-1, which are the ones used most often. The Tab control is useful; the last section of this chapter presents an example that shows you how to use a *tabbed dialog*, which has multiple pages that the user moves between by clicking on the tabs. You may never need a CStatusBarCtrl or a CToolBarCtrl because the CToolBar and CStatusBar classes (covered in the last chapter) are more useful. CAnimateCtrl is very simple once you have the appropriate animation clip to play. Headers and hotkeys are seldom used.

Table 14-1: The New Common Controls

Control	MFC Class	WNDCLASS
Progress	CProgressCtrl	"msctls_progress32"
Slider	CSliderCtrl	"msctls_trackbar32"
Spin Button	CSpinButtonCtrl	"msctls_updown32"
Image list	CImageList	N/A
List view	CListCtrl	"SysListView32"

Tree view	CTreeCtrl	"SysTreeView32"
Tab	CTabCtrl	"SysTabControl32"
Status bar	CStatusBarCtrl	"msctls_statusbar32"
Toolbar	CToolBarCtrl	"ToolbarWindow32"
Tooltip	CToolTipCtrl	"tooltips_class32"
Rich edit	CRichEditCtrl	"RICHEDIT"
Animation	CAnimateCtrl	"SysAnimate32"
Header	CHeaderCtrl	"SysHeader32"
Hotkey	CHotKeyCtrl	"msctls_hotkey32"

Creating New Common Controls

There are two ways to create a common control. You can instantiate the corresponding MFC control class and call the resulting object's `Create()` function, just as was done with the older (Win3.1) common controls. Or you can add a control resource line to the dialog resource script. Here we look at creating a *progress control* and attaching it directly to any window. First, the header file `<afxcmn.h>` must be included because it contains the declarations for `CProgressCtrl` as well as all of the other new common control classes. The following code will attach a progress control directly to a window:

```
#include <afxcmn.h>
. . .
. . .
CProgressCtrl ctrlProgressA;
CRect ctrlLocation(10, 10, 80, 15);
ctrlProgressA.Create(WS_CHILD | WS_VISIBLE | WS_BORDER,
                    ctrlLocation, this, IDC_PROGRESS);
```

The second method is to add control resource statements to the dialog template. This is what is done when you use the Dialog Editor to add these new common controls to a dialog box. In the upcoming example, we will be creating a dialog box and attaching numerous common controls to the dialog box.

The NewCmnCtrls Example

We will put a number of the most useful new common controls on a modal dialog box. The controls we will use are: the *progress control*, the *slider control*, the *spinner control* with its *buddy edit box*, the *list view control*, and the *tree view control*. From this example, which is shown in Figure 14-3, you will be able to see how to set up each of these controls.



Figure 14-3: The Executing NewCmnCtrls Example

A *progress control bar* is a window that an application uses to indicate the progress of a lengthy operation. It consists of a rectangle that is gradually filled from left to right with the system highlight color as an operation such as reading a file progresses.

A *slider control*, also known as a trackbar, contains a slider that the user moves with either the mouse or the direction keys. The control sends notification messages indicating the change. Slider controls can be used to select a discrete value or a set of consecutive values in a range.

A *spin button control* is a pair of arrow buttons that the user can click to increment or decrement a value; it is also referred to as an “up-down” control. The current position value can be displayed in a companion window. A spin button control is most often used with a companion control, called a buddy window.

A *list view control* displays a collection of items each consisting of an icon and a label. List views provide several ways of arranging items and displaying individual items.

A *tree view control* is a window that displays a hierarchical list of items. Each item consists of a label and an optional bitmapped image, and each item can have a list of subitems associated with it. By clicking an item, the user can expand and collapse the associated list of subitems.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes
(Publisher: Wordware Publishing, Inc.)
Author(s): Shirley Wodtke
ISBN: 1556225121
Publication Date: 01/01/97



Search this book:

New Common Control Styles

In addition to the window styles that apply to the new common controls, there are styles specific to each new common control. The slider control styles are given in Table 14-2. The spin button control styles are given in Table 14-3. The list view control styles are given in Table 14-4. The tree view control styles are given in Table 14-5.

Table 14-2: Slider Control Styles

Style	Meaning
TBS_HORIZ	Orients the slider horizontally (default style).
TBS_VERT	Orients the slider vertically.
TBS_NOTICKS	Displays a slider without tick marks.
TBS_ENABLESELRANGE	Displays a selection range. When a slider has this style, the tick marks at the starting and ending positions of a selection range are displayed as triangles (instead of vertical dashes) and the selection range is highlighted.
TBS_AUTOTICKS	Displays a slider with a tick mark for each increment in its range of values; marks are added automatically when SetRange() is called.
TBS_BOTH	Displays tick marks on both sides of the slider.
TBS_TOP	Displays tick marks on top of a horizontal slider.
TBS_BOTTOM	Displays tick marks on bottom of a horizontal slider.

TBS_LEFT	Displays tick marks on the left of a vertical slider.
TBS_RIGHT	Displays tick marks on the right of a vertical slider.

Table 14-3: Spin Button Control Styles

Style	Meaning
UDS_HORIZ	Arrows are horizontal rather than vertical.
UDS_ARROWKEYS	Responds to keyboard arrow keys when it has the focus.
UDS_AUTOBUDDY	Selects the previous control in the tab order as the control's buddy.
UDS_SETBUDDYINT	Causes the control to set the text of the buddy window when its position changes.
UDS_ALIGNRIGHT	Positions the spin button control next to the right edge of the buddy.
UDS_ALIGNLEFT	Positions the spin button next to the left edge of the buddy.
UDS_WRAP	Causes the position to wrap if it goes beyond the range.
UDS_NOTHOUSANDS	Does not insert a thousandths separator between every three decimal digits.

Table 14-4: List View Control Styles

Style	Meaning
LVS_ICON	Each item appears as a full-sized icon with a label below it. The user can drag the items to any location in the list control window.
LVS_SMALLICON	Each item appears as a small icon with the label to the right of it. The user can drag the items to any location.
LVS_LIST	Each item appears as a small icon with a label to the right of it. Items are arranged in columns and cannot be dragged to any arbitrary location by the user.
LVS_REPORT	Each item appears on its own line with information arranged in columns. The leftmost column contains the small icon and label, and subsequent columns contain subitems as specified by the application.
LVS_ALIGNLEFT	Left-aligns items in icon and small icon view.
LVS_ALIGNTOP	Top-aligns items in icon and small icon view.
LVS_AUTOARRANGE	Automatically arranges items in rows and columns in icon and small icon view.

LVS_SORTASCENDING	Sorts items in ascending order.
LVS_SORTDESCENDING	Sorts items in descending order.
LVS_SINGLESEL	Only one item can be selected (default is multiple selections).
LVS_SHAREDIMAGELISTS	Prevents the control from automatically deleting the image lists associated with it when it is deleted.
LVS_NOLABELWRAP	Displays item text on a single line in icon view (default allows text wrapping).
LVS_EDITLABELS	Allows item's text to be edited in place.
LVS_OWNERDRAWFIXED	In the report view, enables the owner window to paint items in response to a WM_DRAWITEM message.
LVS_NOSCROLL	Disables scrolling. (Default allows scrolling.)
LVS_NOCOLUMNHEADER	Does not display the header control that is by default displayed in the report view.
LVS_NOSORTHEADER	Specifies that column headers do not respond to clicks in the report view.

Table 14-5: Tree View Control Styles

Style	Meaning
TVS_HASLINES	Has lines connecting child items to parents.
TV_LINESATROOT	Has lines connecting child items to the root of the hierarchy.
TVS_HASBUTTONS	Adds a button to the left of each parent.
TVS_EDITLABELS	Allows user to edit the labels of the items.
TVS_SHOWSELALWAYS	The selected item remains selected when the control loses focus.
TVS_DISABLEDRAHDROP	Disables drag-drop notifications.

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

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

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

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

☒ Brief ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

Building the NewCmnCtrls Program

Getting the AppWizard Starter Application

We will get a starter application using AppWizard, add a menu item for opening the dialog box, and then create a dialog box filled with the new common controls.

1. Open up a project workspace. (For the Visual C++ 4 compiler, choose “File | New | Project Workspace | AppWizard.”) The project name is NewCmnCtrls.
2. Make the following choices in the steps.
 - a. Choose “Single document.”
 - b. Choose “None”; default settings are OK.
 - c. Choose “None”; default settings are OK.
 - d. Deselect “Toolbar,” “Status bar,” “Printing and Print preview”; you must select “3D controls.”
 - e. Choose “No, thank you” for source file comments and use MFC “As a statically linked library.”
 - f. Change the class names from C to C_.

When you are finished, your New Project Information box should look like Figure 14-4.



Figure 14-4: The AppWizard Starter Application

Add the Menu Item

Next add the menu item “OpenDialog.” In the view class, fill in the message map and stub-out its handler function using Class Wizard. Later we will finish the handler function after we have added the dialog class to the application.

Create the Dialog Template

1. Choose “Insert | Resources | Dialog” and name your new dialog IDD_NEWCMNCTRLS, giving it the caption as shown in Figure 14-5.
2. Enlarge the dialog. The palette appears if you right-click on the workspace toolbar and choose Controls.
3. Drag and drop the controls, giving them static control labels, as shown.

- a. Do nothing to the progress control; the defaults for the progress control are OK.
- b. The slider control has a static control to its left, in which we will write the value of the variable `m_nSlider`, so give this static control an ID of `IDC_M_NSLIDER`. The default ID for the slider (`IDC_SLIDER1`) is OK.

Note: The spinner control is preceded by a “buddy” edit box as its companion, its buddy must precede it in tab order.

- c. Choose the styles tab on the spinner control’s property page and select “Auto buddy,” “Set buddy integer,” and for “Alignment” choose “right.”
- d. The spinner control is followed by the static label “Spinner.” Another static control follows which is labeled “m_nSpin”; give the identifier `IDC_M_NSPIN` to this static control. Default IDs for the other spinner controls and labels are OK.
- e. The List View default ID is OK. On the property page of the list control:
 - (1) Choose the styles tab.
 - (2) Select “List” for the “View” style attribute.
 - (3) Select “Left” for the “Align” style attribute.
- f. The Tree View default ID is OK. On the property page of the tree control:
 - (1) Choose the styles tab.
 - (2) Select “Has buttons.”
 - (3) Select “Has lines.”
 - (4) Select “Lines at root.”

It is very helpful to look at the `DIALOG` resource script since these are different controls from any we have dealt with before. The `DIALOG` resource script that the Dialog Editor writes for this example is given in Listing 14-2.



Figure 14-5: The Finished Dialog Box

Listing 14-2: The `DIALOG` Resource Script

```
-----
                                excerpt from NewCmnCtrls.rc file
-----
IDD_NEWCMNCTRLS DIALOG DISCARDABLE 10, 10, 302, 178
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "New Common Controls Dialog"
FONT 8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON        "OK", IDOK, 171, 7, 50, 14
    PUSHBUTTON           "Cancel", IDCANCEL, 239, 7, 50, 14
    LTEXT                 "Progress", IDC_STATIC, 13, 23, 28, 8, NOT WS_GROUP
    CONTROL               "Progress1", IDC_PROGRESS1, "msctls_progress32",
        WS_BORDER, 49, 20, 80, 14
    LTEXT                 "Slider", IDC_STATIC, 13, 47, 18, 8, NOT WS_GROUP
    CONTROL               "Slider1", IDC_SLIDER1, "msctls_trackbar32", TBS_BOTH |
        TBS_NOTICKS | WS_TABSTOP, 40, 45, 100, 15
    LTEXT                 "m_nSlider", IDC_M_NSLIDER, 141, 49, 32, 8, NOT WS_GROUP
    LTEXT                 "Buddy", IDC_STATIC, 13, 75, 21, 8, NOT WS_GROUP
    EDITTEXT              IDC_EDIT1, 46, 75, 40, 14, NOT WS_TABSTOP
    CONTROL               "Spin1", IDC_SPIN1, "msctls_updown32",
        UDS_SETBUDDYINT | UDS_ALIGNRIGHT | UDS_AUTOBUDDY |
        UDS_ARROWKEYS | WS_TABSTOP, 95, 75, 11, 14

```

```
LTEXT      "Spinner", IDC_STATIC, 105, 75, 25, 8, NOT WS_GROUP
LTEXT      "m_nSpin", IDC_M_NSPIN, 141, 75, 28, 8, NOT WS_GROUP
LTEXT      "List View", IDC_STATIC, 15, 102, 29, 8, NOT WS_GROUP
CONTROL    "List1", IDC_LIST1, "SysListView32", LVS_LIST |
           LVS_ALIGNLEFT | WS_BORDER | WS_TABSTOP, 53, 98, 57, 73
LTEXT      "Tree View", IDC_STATIC, 187, 32, 33, 8, NOT WS_GROUP
CONTROL    "Tree1", IDC_TREE1, "SysTreeView32", TVS_HASBUTTONS |
           TVS_HASLINES | TVS_LINESATROOT | WS_BORDER |
           WS_TABSTOP, 187, 49, 108, 122

END
```

Previous	Table of Contents	Next

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

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

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

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

Creating the Dialog Class

1. Create the dialog class named C_NewCmnCtrlsDlg using ClassWizard.
2. Add a message map entry for WM_INITDIALOG to the dialog class; we will initialize the controls in the OnInitDialog() function.
3. Add a message map entry for the H_SCROLL message; in response to a horizontal scroll message, we will write the slider's value into a static control.
4. Add a message map entry for the V_SCROLL message; in response to a vertical scroll message, we will write the spin control's value into a static control.
5. Associate an integer member variable named m_nSpin with the spinner's buddy edit box.
6. Use Class Wizard to add Control variables (to have an object data member for each control) named m_myProgress, m_mySlider, m_mySpinner, m_myListView, and m_myTreeView.

Note: Class Wizard won't generate an integer member variable for new common controls. For these, you will add the integer member variables manually.

- a. For the progress control, add an integer member variable named m_nProgress to the dialog's header file.
 - b. For the slider control, add an integer member variable named m_nSlider to the dialog's header file.
7. Add the dialog class's header file to the #includes of the view's implementation file.

Complete the View's Handler Function

Now add the following code to initialize the dialog's member variables and to open the dialog in the OnOpenDialog() handler function:

```

C_NewCmnCtrlsDlg MyDialog;
MyDialog.m_nProgress = 60;
MyDialog.m_nSlider = 20;
MyDialog.m_nSpin = 50;
MyDialog.DoModal();
    
```

Program the Progress Control

We will use member functions inherited from the class `CProgressCtrl`. The range of the progress control is set using the function `SetRange()`. Its position is set to the `m_nProgress` value using the `SetPos()` function. These are done by adding the following code in the `OnInitDialog()` function:

```
m_myProgress.SetRange(0, 200);  
m_myProgress.SetPos(m_nProgress);
```

Program the Slider Control

In the `OnInitDialog()` function, we will use member functions inherited from the class `CSliderCtrl` to set its range and initial position. Also, we get its position and set that value in the static control identified with `IDC_M_NSLIDER`. The following code does this:

```
CString strText;  
m_mySlider.SetRange(0, 100);  
m_mySlider.SetPos(m_nSlider);  
strText.Format("%d", m_mySlider.GetPos());  
SetDlgItemText(IDC_M_NSLIDER, strText);
```

In the response function to the horizontal scroll, we must also get the slider's position and set that value in the static control. To do this, add the following code to the `OnHScroll()` handler function:

```
CString strText;  
strText.Format("%d", m_mySlider.GetPos());  
SetDlgItemText(IDC_M_NSLIDER, strText);
```

Program the Spinner Control

In the `OnInitDialog()` function, we use member functions inherited from the parent class `CSpinButtonCtrl` to set its range and initial position. Also, we get its position and set that value in the static control `IDC_M_NSPIN`. This is done with the following code:

```
m_mySpinner.SetRange(0, 100);  
m_mySpinner.SetPos(m_nSpin);  
strText.Format("%d", m_mySpinner.GetPos());  
SetDlgItemText(IDC_M_NSPIN, strText);
```

In the response function to the vertical scroll, we get the spinner's position and set that value into the static control identified with `IDC_M_NSPIN`. The spinner control has a buddy, the edit box, in which its position is automatically shown. In the static control, we see this number again; it is included to demonstrate using the response to the vertical scroll message. Add the following code to the `OnVScroll()` handler function:

```
CString strText;  
strText.Format("%d", m_mySpinner.GetPos());  
SetDlgItemText(IDC_M_NSPIN, strText);
```

Set Up an Image List

In the upcoming list view control and the tree view control, we will be using images. We need to declare two object data members of class `CImageList`, one for each image list, in the dialog's header file.

1. Add the following lines of code to the NewCmnCtrlDlg.h file:

```
CImageList m_ImageList1;  
CImageList m_ImageList2;
```

2. Create the icons you need.

You can use either 16 x 16 or 32 x 32 icons. The class CImageList's Create() function can accommodate either, and it can use the large icons as small 16 x 16 icons. In the list view control, we will display four icons: an icon of an apple identified as IDI_APPLE, an icon of a pear identified as IDI_PEAR, an icon of a banana identified as IDI_BANANA, and an icon of a lime identified as IDI_LIME. In the tree view control, we will use three icons: an icon of a gold star identified as IDI_COUNTRY since it will be used for countries, an icon of a red triangle identified as IDI_STATE since it will be used for states, and an icon of a city identified as IDI_CITY.

3. Load these icons into the image lists in the OnInitDialog() function in the dialog class's .cpp file. This is accomplished by adding the following code to OnInitDialog():

```
int i;  
HICON hIcon1[4];  
m_ImageList1.Create(16, 16, 0, 0, 4); // displays small icons  
hIcon1[0] = AfxGetApp()->LoadIcon(IDI_APPLE);  
hIcon1[1] = AfxGetApp()->LoadIcon(IDI_PEAR);  
hIcon1[2] = AfxGetApp()->LoadIcon(IDI_BANANA);  
hIcon1[3] = AfxGetApp()->LoadIcon(IDI_LIME);  
  
for (i = 0; i < 4; i++)  
{  
    m_ImageList1.Add(hIcon1[i]);  
}  
  
HICON hIcon2[3];  
m_ImageList2.Create(16, 16, 0, 0, 3);  
hIcon2[0] = AfxGetApp()->LoadIcon(IDI_COUNTRY);  
hIcon2[1] = AfxGetApp()->LoadIcon(IDI_STATE);  
hIcon2[2] = AfxGetApp()->LoadIcon(IDI_CITY);  
  
for (i = 0; i < 3; i++)  
{  
    m_ImageList2.Add(hIcon2[i]);  
}
```

Program the List View Control

We have chosen the LVS_LIST style for our list view control. We will add text to accompany each of the icons in the list and we will set the background color of the list to be light green. We do this by adding the following code to the function OnInitDialog():

```
m_myListView.SetImageList(&m_ImageList1, LVSIL_SMALL);  
static char* fruit[] = {"apple", "pear", "banana", "lime"};  
  
for (i = 0; i < 4; i++)  
{  
    m_myListView.InsertItem(i, fruit[i], i);  
}
```

```
m_myListView.SetBkColor(RGB(0,255,0)); // bkg color light green
```

Previous	Table of Contents	Next

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

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

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



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

Go!

Keyword

☒ Brief ☐ Full[Advanced](#)[Search](#)[Search Tips](#)BROWSE
BY TOPIC

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

Go!

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

Program the Tree Control

We plan on including images as well as text in our tree view, so we need to add the image list. We then add each item to the tree. To do this, we select one of the four forms of the CTreeCtrl::InsertItem() function that is most convenient for the example we have. We do this by adding the following code to the function OnInitDialog():

```

m_myTreeView.SetImageList(&m_ImageList2, TVSIL_NORMAL);

// Root items first, with automatic sorting
HTREEITEM hUnitedStates = m_myTreeView.InsertItem("United States",
    0, 0, TVI_ROOT, TVI_SORT);
HTREEITEM hMexico = m_myTreeView.InsertItem("Mexico",
    0, 0, TVI_ROOT, TVI_SORT);
HTREEITEM hCanada = m_myTreeView.InsertItem("Canada",
    0, 0, TVI_ROOT, TVI_SORT);

// United States subitems (no sorting)
HTREEITEM hNewYork = m_myTreeView.InsertItem("New York",
    1, 1, hUnitedStates);
HTREEITEM hCalifornia = m_myTreeView.InsertItem("California",
    1, 1, hUnitedStates);
m_myTreeView.InsertItem("New York City", 2, 2, hNewYork);
m_myTreeView.InsertItem("Syracuse", 2, 2, hNewYork);
m_myTreeView.InsertItem("Sacramento", 2, 2, hCalifornia);
m_myTreeView.InsertItem("San Francisco", 2, 2, hCalifornia);
m_myTreeView.InsertItem("Los Angeles", 2, 2, hCalifornia);

// Mexico subitems (no sorting)
HTREEITEM hMexicoState = m_myTreeView.InsertItem("Mexico"
    1, 1, hMexico);
HTREEITEM hQuintanaRoo = m_myTreeView.InsertItem("Quintana Roo",
    1, 1, hMexico);
m_myTreeView.InsertItem("Mexico City", 2, 2, hMexicoState);

```

```

m_myTreeView.InsertItem("Cuernavaca", 2, 2, hMexicoState);
m_myTreeView.InsertItem("Cancun", 2, 2, hQuintanaRoo);

// Canada subitems (no sorting)
HTREEITEM hBritishColumbia = m_myTreeView.InsertItem(
    "British Columbia", 1, 1, hCanada);
HTREEITEM hQuebec = m_myTreeView.InsertItem("Quebec",
    1, 1, hCanada);
m_myTreeView.InsertItem("Vancouver", 2, 2, hBritishColumbia);
m_myTreeView.InsertItem("Montreal", 2, 2, hQuebec);
m_myTreeView.InsertItem("Quebec", 2, 2, hQuebec);

```

In the preceding form of the `InsertItem()` function, the first parameter is the text of the new item to be added. The second and third parameters passed to this form of `InsertItem()` are image indexes. The second parameter specifies the image the tree view control will display when the item is not selected, and the third parameter specifies the image the control will display when the item is selected. Specifying the same index for an item's selected and nonselected states means that the same image will be used for both. The fourth parameter is the handle of the parent; this parameter is `TVI_ROOT` for root items.

Notification Messages

The new common controls use `WM_NOTIFY` messages to send notification messages to their parent. As you remember, the older (Win3.1) common controls used `WM_COMMAND` messages. A `WM_NOTIFY` message's `wParam` is the child window ID of the control that sent the message, and `lParam` is a pointer to an `NMHDR` structure. The `NMHDR` structure holds the pointer to the control sending the message, the ID of the control sending the message, and the control-specific notification code. Many controls send `WM_NOTIFY` messages with pointers to structures larger than `NMHDR` (a structure that is a superset of `NMHDR`). Those larger structures hold the three information items previously mentioned plus other control-specific items. When ClassWizard maps a `WM_NOTIFY` message, it generates a pointer to the appropriate structure.

There are numerous notification messages that can be sent from the control to its parent. Some notification messages can be used by all of the new common controls; these are denoted by the prefix of `NM_`. For example, the `NM_CLICK` message is sent when the control is clicked with the left mouse button. In addition to the `NM_` messages that apply to any of the new common controls, each control has control-specific messages that it can send. Slider notification messages are denoted by `TB_`. Spinner notification messages are denoted by `UDN_`. List view notification messages are denoted by `LVN_`. Tree view notification messages are denoted by `TVN_`.

The `WM_NOTIFY` messages are mapped to response functions by the `ON_NOTIFY()` and `ON_NOTIFY_RANGE()` message map macros. The `ON_NOTIFY_RANGE()` macro is used when you have a set of controls for which you want to perform the same action for a certain notification message; you specify a contiguous range of child identifiers for which to handle the notification message by specifying the beginning and ending child identifiers of the range.

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

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

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

Keyword

☒ Brief
 ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Property Sheets

Property sheets are in the new common controls library. Property sheets are tabbed dialog boxes that contain pages of controls. The user can switch among the tabbed dialog boxes with a mouse click on the tab. Adding a property sheet is very much like adding a dialog.

The PropertySheet Example

An example is presented of a modal property sheet. For this example, we have revisited the Chapter Eight program named “ModalCom.” We remove its dialog box and replace it with a property sheet that has two pages, one for the choice of the color and the second for entering the text. We name the new program “PropertySheet.” When the user chooses “OpenDialog” from the menu, a property sheet opens up as shown in Figure 14-6a. When the user clicks on the “Text” tab, the second page opens up as shown in Figure 14-6b. Otherwise, the program operates exactly as it did in the Chapter Eight example. Choices made from the main window’s menu will be transferred to the property sheet as it opens up. The user makes choices and clicks the OK button. The user’s choices will be transferred, when the property sheet closes, to the main window where they will take effect.



Figure 14-6a: The PropertySheet Program When OpenDialog is Chosen

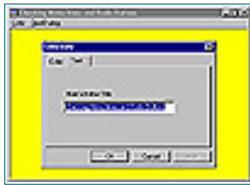


Figure 14-6b: The PropertySheet Program When the Text Tab is Chosen

The MFC classes `CPropertySheet` and `CPropertyPage` encapsulate all the functionality of the property sheets. A class needs to be derived from `CPropertySheet`, which is derived from `CWnd`, to represent the property sheet itself. Also, a class needs to be derived from `CPropertyPage`, which is derived from `CDialog`, to represent each of the property pages. Both `CPropertySheet` and `CPropertyPage` are defined in the header file `<afxdlgs.h>`. Property sheets can be modal or modeless. Just like with dialog boxes, a modal property sheet is displayed using the function `CPropertySheet::DoModal()`; a modeless property sheet is displayed using the function `CPropertySheet::Create()`. A modal property sheet has OK, Cancel, and Apply buttons automatically included. A modeless property sheet does not have these buttons; the user must create the desired buttons.

Creating a Modal Property Sheet

The general procedure for creating a modal property sheet is given in the following steps:

1. For each page of the property sheet, add a dialog template to the application's .rc file that defines the page's controls. You can use the Dialog Editor to do this. The caption statement is used to specify the title that will appear on the tab at the top of the page. The dialog templates for property sheet pages should not include OK and Cancel buttons as conventional dialog templates do, because the property sheet provides these buttons on its own.
2. For each page of the property sheet, derive a dialog-like class from `CPropertyPage` adding member variables for the page's controls. ClassWizard will write the `DoDataExchange()` function which is needed to transfer data between the member variables and the controls and optionally to validate the user's input.
 - a. For convenience, put all the classes that you create in a property file (`Property.h` and `Property.cpp`).
3. Derive a class from `CPropertySheet`.
 - a. Use ClassWizard to generate the class. Put this class in the `Property.h` and `Property.cpp` files.
 - b. In the property sheet class, instantiate property page objects for each of the property pages that you constructed in step 2.
 - c. Use `CPropertySheet::AddPage()` to add the pages to the property sheet in the order in which you want them to appear.
4. In the application's handler function that opens the property box, add code that looks very much like opening a modal dialog box.
 - a. Construct an object of your property sheet class.

(1) You must supply the first parameter (the caption of the property sheet) to the three parameter constructor; the last two parameters have default values. The constructor is overloaded and the first parameter can be a string or the ID of a string.

(2) You must specify a caption for the property sheet in the constructor; however, the initial caption can be changed later in the program by calling `CPropertySheet::SetTitle()`.

Note: Like dialog boxes, the property sheet is opened with the call `DoModal()`. If you are transferring data to the property sheet's member variables, you must do it before you make the call to `DoModal()`. If the property sheet is dismissed with the OK button, `DoModal()` will return `IDOK`. Otherwise, it will return `IDCANCEL`—just like a modal dialog box does. If you are transferring data from the property sheet's member variables, you must do it before the end of the function in which you called the property sheet.

Previous	Table of Contents	Next

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

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

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

☒ Brief ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

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

Building the “PropertySheet” Program

To build the new program “PropertySheet” complete the steps below:

1. Create an application project named “PropertySheet.”
 - a. You can use files from the “ModalCom” program of Chapter Eight.
 - b. You can use MyApp.h, Myapp.cpp, and mainframe.h as is, with no modifications.
 - c. You can use mainframe.cpp, resource.h, and script1.rc, but all require modifications.
 - (1) Mainframe.cpp must be modified; the dialog header file needs to be replaced with the property sheet’s header file in the #includes and, of course, the OnOpenDialog() handler function must be rewritten.
 - (2) In the resource files, delete the old DIALOG resource script, IDD_DIALOG1, and all references to it. You can use these modified files to continue building the PropertySheet program.
2. Use the Dialog Editor to add two property page dialog templates. The ID for the first template is IDD_COLOR and for the second template it is IDD_TEXT. These two finished dialog templates are shown in Figures 14-7a and 14-7b.

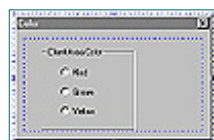


Figure 14-7a: The IDD_COLOR Dialog Template

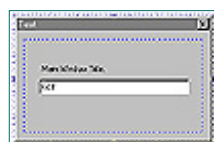


Figure 14-7b: The IDD_TEXT Dialog Template

3. Use ClassWizard to create the class C_Color derived from CPropertyPage to encapsulate the dialog template IDD_COLOR.
 - a. Put the new C_Color class in the files Property.h and Property.cpp.
 - b. Add the integer member variable, m_nColor, for the first radio button.
4. With ClassWizard also create the class C_Text derived from CPropertyPage to encapsulate the dialog template IDD_TEXT.
 - a. Put the new C_Text class in the files Property.h and Property.cpp.
 - b. Add the CString member variable, m_sTitle, for the edit box.
5. Using ClassWizard add another class, C_MySheet, derived from CPropertySheet.
 - a. Also add this new C_MySheet class to the files Property.h and Property.cpp.

b. In the class C_MySheet's header file declare the following two object data members:

```
C_Color m_ColorPage;  
C_Text m_TextPage;
```

c. In the class C_MySheet's constructor add the property pages with the following lines of code:

```
AddPage(&m_ColorPage);  
AddPage(&m_TextPage);
```

6. In this PropertySheet example, fix up your files.

a. Add the following line to the #includes of mainframe.cpp:

```
#include "Property.h"
```

b. Add the following code to the Property.h file:

```
#include "resource.h"  
#include <afxdlgs.h>
```

c. Fix up the file Property.cpp so that the #includes look like the following:

```
#include "stdafx.h"  
#include "Property.h"
```

Listing 14-3 contains the code for the “PropertySheet” program. Most of the code was generated by the Dialog Editor or ClassWizard and contains unnecessary lines of code, which are not shown. Only the necessary lines of code are shown in Listing 14-3. The stdafx files and the application class files are exactly the same as given in the ModalCom program of Chapter Eight (see Listing 8-1), so they are not shown in Listing 14-3.

Listing 14-3: Source Code for PropertySheet Program

```
-----  
                                mainframe class excerpts  
-----  
  
    mainfrm.h is exactly the same as given in Listing 8-1  
  
//  
//  mainfrm.cpp  
//
```

Note: only excerpts that contain changed code are shown here. The remainder of the code not shown here is exactly the same as given in Listing 8-1.)

```
#include "stdafx.h"  
#include "mainfrm.h"  
#include "resource.h"  
#include "Property.h"  
  
. . . . .  
  
void C_MainFrame::OnOpenDialog()  
{  
    C_MySheet propSheet("Selections");  
    propSheet.m_TextPage.m_sTitle = m_sMainWindowTitle;  
    propSheet.m_ColorPage.m_nColor = m_nClientColor;  
    int ReturnValue = propSheet.DoModal();  
    if (ReturnValue == IDCANCEL) return;  
    m_sMainWindowTitle = propSheet.m_TextPage.m_sTitle;  
    m_nClientColor = propSheet.m_ColorPage.m_nColor;  
    SetWindowText(m_sMainWindowTitle);  
}
```

```

        Invalidate();
    }

-----
                                property sheet files
-----
//
//  Property.h : header file
//
#include "resource.h"
#include <afxdlgs.h>

////////////////////
// C_Text dialog
class C_Text : public CPropertyPage
{
// Construction
public:
    C_Text();

// Dialog Data
    //{AFX_DATA(C_Text)
    enum { IDD = IDD_TEXT };
    CString    m_sTitle;
    //}}AFX_DATA

// Overrides
    // ClassWizard generate virtual function overrides
    //{AFX_VIRTUAL(C_Text)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL
};

////////////////////
// C_Color dialog

class C_Color : public CPropertyPage
{
// Construction
public:
    C_Color();

// Dialog Data
    //{AFX_DATA(C_Color)
    enum { IDD = IDD_COLOR };
    int    m_nColor;
    //}}AFX_DATA

// Overrides
    // ClassWizard generate virtual function overrides
    //{AFX_VIRTUAL(C_Color)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL
};

////////////////////
// C_MySheet

class C_MySheet : public CPropertySheet

```

```

{
// Construction
public:
    C_MySheet(LPCTSTR pszCaption, CWnd* pParentWnd = NULL,
              UINT iSelectPage = 0);

// Attributes
public:
    C_Color m_ColorPage;
    C_Text m_TextPage;
}
;
//
// Property.cpp : implementation file
//
#include "stdafx.h"
#include "Property.h"

////////////////////
// C_Text property page

C_Text::C_Text() : CPropertyPage(C_Text::IDD)
{
}

void C_Text::DoDataExchange(CDataExchange* pDX)
{
    CPropertyPage::DoDataExchange(pDX);
    //{AFX_DATA_MAP(C_Text)
    DDX_Text(pDX, IDC_TEXT, m_sTitle);
    //}AFX_DATA_MAP
}

////////////////////
// C_Color property page

C_Color::C_Color() : CPropertyPage(C_Color::IDD)
{
}

void C_Color::DoDataExchange(CDataExchange* pDX)
{
    CPropertyPage::DoDataExchange(pDX);
    //{AFX_DATA_MAP(C_Color)
    DDX_Radio(pDX, IDC_RED, m_nColor);
    //}AFX_DATA_MAP
}

////////////////////
// C_MySheet

C_MySheet::C_MySheet(LPCTSTR pszCaption, CWnd* pParentWnd, UINT iSelectPage)
: CPropertySheet(pszCaption, pParentWnd, iSelectPage)
{
    AddPage(&m_ColorPage);
    AddPage(&m_TextPage);
}

```

 excerpts from resource files

```
//
// resource.h
//
#define IDR_MENU1 101
#define IDD_COLOR 103
#define IDD_TEXT 104
#define IDC_RED 1004
#define IDC_GREEN 1005
#define IDC_YELLOW 1006
#define IDC_TEXT 1007
#define ID_RED 40001
#define ID_GREEN 40002
#define ID_YELLOW 40003
#define ID_OPENDIALOG 40004

//
// script1.rc
//
```

Note: only the dialog resource scripts for IDD_COLOR and IDD_TEXT are given here.

```
IDD_COLOR DIALOG DISCARDABLE 0, 0, 186, 95
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Color"
FONT 8, "MS Sans Serif"
BEGIN
    CONTROL "Red", IDC_RED, "Button", BS_AUTORADIOBUTTON | WS_GROUP,
        42, 32, 29, 10
    CONTROL "Green", IDC_GREEN, "Button", BS_AUTORADIOBUTTON, 42, 49, 35, 10
    CONTROL "Yellow", IDC_YELLOW, "Button", BS_AUTORADIOBUTTON,
        42, 66, 37, 10
    GROUPBOX "Client Area Color", IDC_STATIC, 24, 15, 89, 73, WS_GROUP
END

IDD_TEXT DIALOG DISCARDABLE 0, 0, 186, 95
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Text"
FONT 8, "MS Sans Serif"
BEGIN
    LTEXT "Main Window Title:", IDC_STATIC, 24, 28, 62, 8, NOT WS_GROUP
    EDITTEXT IDC_TEXT, 21, 42, 145, 14, ES_AUTOHSCROLL
END
-----
```

Previous	Table of Contents	Next

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

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

Keyword

☒ Brief
 ☐ Full

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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

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

Using the Apply Button

In the preceding “PropertySheet” program, the default behavior was sufficient, so we did not use the Apply button. So what does the Apply button do? If we had chosen to use it in this example, we would have used it to allow the user to update the member variables whenever he clicked on the Apply button, rather than waiting until the property sheet window is closed to update the member variables.

To use the Apply button, you would need to add code. First, you would need to code a response function for each choice the user made. For example, when a radio button is clicked you would need a handler function to call the SetModified(TRUE) function to set the modified flag for the active page.

When the user switches pages, the framework would check to see if the modified flag is set. If so, the Apply button would become active. If the user chooses, he could then click on the Apply button. You would write a handler function for the Apply button’s ID, which is IS_APPLY_NOW, by overriding the virtual CPropertyPage::OnApply() function. The code that you would put in the handler function is whatever would suit your needs.

When the user clicks on the Apply button, the framework would call the DDX code for the page; it then would call the virtual OnApply() function for all the pages, and it would reset the modified flag, which would disable the Apply button.

Summary

Customizable controls, which are known as owner-draw controls, can be placed on dialogs using the Dialog Editor. These controls can be customized to specify the drawing action required, the dimensions of the items in the control, the position of the items, and the deletion of item-specific data. You have a lot

of freedom to create what you want. The CBitmapButton class is an example of owner, or self-drawing, controls. The CBitmapButton class applies to both Win3.1 and Win32 applications.

MFC 4.0 introduced new common controls. These controls can be placed directly on a window or can be placed on a dialog box. Each of these new common controls encapsulates unique functionality. Frequently seen ones are the progress control bar, which is used to show the progress of a lengthy operation; the tree view control used in the Windows Explorer, which displays a hierarchical list of items; and the list view control also used in the Windows Explorer, which displays a list that can contain icons as well as text. Spin button controls and slider controls are frequently used to select numerical values. Each of these new common controls have styles specific to their class and each class has notification messages specific to its class. Notification messages for new common controls are sent as a WM_NOTIFY message, a new message defined in MFC 4.0. The WM_NOTIFY message allows more information to be sent than was possible in previous (Win 3.1x) common controls which used the WM_COMMAND message.

Property sheets are also in the new common controls library. Property sheets are tabbed dialog boxes containing pages of controls. They are based on the MFC classes CPropertyPage and CPropertySheet. Like dialogs, they can be modal or modeless and they operate very much like dialogs.

Previous	Table of Contents	Next

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes
(Publisher: Wordware Publishing, Inc.)
Author(s): Shirley Wodtke
ISBN: 1556225121
Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

[Table of Contents](#)

Appendix A

Class Hierarchy, Data Types, and Hungarian Notation Prefixes

Contents

Figure A-1 Class Hierarchy for MFC Classes Covered in This Book

Table A-1 Data Types

Table A-2 Hungarian Notation Prefixes



Figure A-1: Class Hierarchy for MFC Classes Covered in This Book

Data types are keywords that define the size and meaning of parameters. The following table gives the definition of data types used in this book.

Table A-1: Data Types

Type	Definition
BOOL	Boolean variable (should be TRUE or FALSE)
BOOLEAN	Boolean variable (should be TRUE or FALSE)
BYTE	Byte (8 bits)
CHAR	Character
COLORREF	Red, green, blue (RGB) color value (32 bits)
CONST	Variable whose value is to remain constant during execution

DWORD	Doubleword (32 bits)
FLOAT	Floating-point variable
HBITMAP	Handle of a bitmap
HBRUSH	Handle of a brush
HCURSOR	Handle of a cursor
HDC	Handle of a device context (DC)
HFONT	Handle of a font
HICON	Handle of an icon
HMENU	Handle of a menu
HPALETTE	Handle of a palette
HPEN	Handle of a pen
HWND	Handle of a window
INT	Signed integer
LONG	32-bit signed value
LPARAM	32-bit message parameter
LPCSTR	Pointer to a constant null-terminated character string
LPSTR	Pointer to a null-terminated character string
PSTR	Pointer to a null-terminated character string
PSZ	Pointer to a null-terminated character string
SHORT	Short integer
UINT	Unsigned integer
ULONG	Unsigned long integer (32 bits)
USHORT	Unsigned short integer (16 bits)
VOID	Any type
WORD	Unsigned word (16 bits)
WPARAM	32-bit message parameter

A naming convention called “Hungarian Notation” in honor of its inventor, Charles Simonyi, is an accepted standard. A variable’s name is preceded with key letters that describe what type of data the variable represents. Using this notation allows you to identify the type of your variables by the prefixes they sport. You can deduce considerable information about a variable just from its name.

These prefixes are used together as required. For example, a long pointer to a null-terminated string would have the prefix of `lpsz`.

Table A-2:Hungarian Notation Prefixes

<i>Prefix</i>	<i>Data Type</i>	<i>Example</i>
a	Array of variables	apShape[]
b	BOOL (int, use only TRUE and FALSE values, 0 or 1)	bButtonDown
by	BYTE (unsigned char)	byXHistory
c	char	cStatus

dw	DWORD (double word, four byte unsigned integer)	dwStyle
h	handle	hCursor
l	long integer type	lParam
m_	class member variable	m_pMainWnd
n	int	nBeeps
p	pointer	pPen
s	character string	m_sTitle
sz	character string terminated by zero	lpszClassName
w	unsigned word (two bytes)	wParam

Table of Contents

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

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

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

[HOME](#)[ACCOUNT INFO](#)[SUBSCRIBE](#)[LOGIN](#)[SEARCH](#)[MY ITKNOWLEDGE](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[SEARCH
ITKNOWLEDGE](#)☒ Brief ☐ Full[Advanced](#)[Search](#)[Search Tips](#)[BROWSE
BY TOPIC](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

[Table of Contents](#)

Appendix B Exercises

These exercises all involve the building of a checkers game program. The program is incrementally built as you learn the techniques of each chapter.

The Checkers Game

The game is played by two persons. Each player places his twelve “men” (checkers) on the dark squares of the first three rows on opposite ends of the board.

The object of the game is to capture all of the opponent’s checkers, or block them so they cannot be moved. The checkers are moved diagonally and each player alternately moves one of his men. In order to “capture” an opponent’s checker, a player must be able to jump (with his own man) over his opponent’s checker when there is a vacant square behind. Single men may move diagonally forward. When a checker has reached the last row of his opponent’s side, it becomes a “king” and then may move diagonally forward or backward. The “king” is crowned by placing another checker on top of it.

Note: In your computer game, you will need to change the appearance of a checker when it becomes a “king.”

You are allowed to jump as many of the opponent’s men on the same move if there are vacant squares diagonally behind each. When there is a “jump” available, a player must jump.

Computer Program Description

You will be designing and creating a basic drawing application, which is the game of checkers. Each assignment will add features to this drawing program. As in commercial Windows programs, some features will be invoked multiple ways; that is, users will invoke some features from either a menu or a dialog box. At the end of this project, your application will allow the user to choose the colors of the checkers board and its pieces, to choose different shapes for the checker pieces, to set the names of the players, and to allow the two players to compete by

moving the pieces with a mouse. The choice of board colors and piece shapes will be made either from a menu or from a dialog box. The players' names will be entered in an edit box control within the dialog box.

Please read Chapter 1 through 8 exercises before beginning so that you can spend time designing the layout of your application's window, analyzing the functional components and the flow of your program, and formulating the C++ classes best suited to the application. Chapter 3 includes a suggested C++ class formulation which could be very useful. You do not have to use this class formulation, you may choose your own.

Chapter 1 Exercise

The Chapter 1 exercise creates a main window with the title "Checkers Game—Player A vs. Player B" and adds a customized icon of your choosing. To get a customized icon, use AppStudio; design your icon and AppStudio will save it in the .rc file. You will be building the checkers game incrementally and adding more resources later, so it is necessary to start the AppStudio resource files here in Chapter 1. If you have an icon that you want to import from another file, you might question: "Why use AppStudio?" The answer is that you need to start working with AppStudio because you will add menus and dialog boxes to the same project (.mak file) in later chapters. For these upcoming resources, you will want to use AppStudio's services. You start here in Chapter 1 building the resource files with AppStudio.

Note: Do not remove any of the comment code that AppStudio generates for itself to re-enter its own resource files to generate more resources.

However, for the Chapter 1 exercise, we need to alter the operational code generated by AppStudio. (AppStudio was designed for use with a more advanced class structure, which we will get to in later chapters, so we need to alter its code for this specific use.)

AppStudio will give your icon an ID of IDR_ICON1; it will place it in the .rc file and it will generate a companion resource.h file. AppStudio will generate the following line in the .rc file:

```
IDI_ICON1            ICON        DISCARDABLE    "ICON1.ICO"
```

Change this line of code to read:

```
AFX_IDI_STD_FRAME    ICON        DISCARDABLE    "ICON1.ICO"
```

This is the only line that must be altered in the resource files.

Figure B-1 shows how a finished Chapter 1 exercise should look.



Figure B-1: Chapter 1 Exercise

Chapter 2 Exercise

The Chapter 2 exercise completes the application's menu. The main window's menu should have a popup menu for the board colors, a popup menu for the piece shape, a menu item that will open a dialog box, a menu item that starts the game after the choices have been made, and a menu item to close the application.

The popup menu for board colors should contain at least three pairs of board colors. The user

should be able to choose a color setting and the chosen menu item should be checkmarked when selected. The previous selection should be unchecked.

The popup menu for the piece shape should contain at least two shapes. When the user chooses a shape, the chosen menu item is checkmarked and the prior selection is unchecked.

You can create the menu using AppStudio. To connect it to the main window, you will declare an object of class CMenu in the mainfrm.h file (e.g., CMenu Menu). In the mainfrm.cpp file, in the constructor, you will add the following two lines of code:

```
Menu.LoadMenu( IDR_MENU1 );  
SetMenu( &Menu );
```

You can enlist the services of ClassWizard to add message map entries for the menu items. You must provide comment lines in your code for ClassWizard to operate.

Note: The lines of code that ClassWizard looks for are comment lines and the compiler will not detect any errors in them, but if there are any errors, ClassWizard will not open. You must provide the following lines of code within the body of the class C_MainFrame's declaration in the file mainfrm.h:

```
//{{AFX_MSG(C_MainFrame)  
//}}AFX_MSG  
DECLARE_MESSAGE_MAP( )
```

You must provide the following lines of code in the file mainfrm.cpp and they must be outside of any function block:

```
BEGIN_MESSAGE_MAP(C_MainFrame, CFrameWnd)  
   //{{AFX_MSG_MAP(C_MainFrame)  
   //}}AFX_MSG_MAP  
END_MESSAGE_MAP( )
```

You must include the file "resource.h" in the mainfrm.cpp #includes.

Figure B-2 shows a finished Chapter 2 exercise.

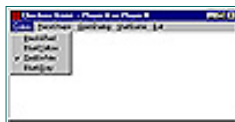


Figure B-2: Chapter 2 Exercisey

Chapter 3 Exercise

The screen should be blank until the menu item that starts the game is chosen. When the game is started, the checkerboard should be drawn with the chosen color selection and piece shape. Label each piece with the initials of the player. When a new selection is made, the selection takes effect immediately.

1. Draw the playing board with the pieces in their initial places. It should display the board with the chosen colors and piece shapes with each player's pieces in their initial positions. Using CRect is very handy!
2. Lay out your board in proportion to the size of the client area, and have the board resize as the window is resized. A WM_SIZE message will always generate a WM_PAINT message, so you can call a function which resizes the checkerboard from within the

OnPaint() function. Alternatively, you could have an OnSize() function that responds to the WM_SIZE message and resize the checkerboard from within the OnSize() function.

Formulation of C++ Classes

A two-dimensional array of size 8 x 8 of checkerboard square objects (of class C_Square) can be used. The two-dimensional array of size 8 x 8 of C_Square objects are data member objects of the mainframe class (C_MainFrame) and are declared in the mainframe's header file. You can then think of each square as containing a piece, where the piece for each square will have one of the following states: NOT_ALLOWED, EMPTY, A, B, KINGA, KINGB. A class C_Piece is defined which contains all the necessary data and functionality for the piece on the square. A class C_Square is derived from C_Piece and will contain additional data and functionality.

```
class C_Piece
{
    private data members:
        * the state of the piece
        * the piece color
        * the color of the piece's border
        * the player's initials
        * the bounding rectangle of this piece's checkerboard
          square

    public member functions:
        * a constructor for special initialization
        * set and get functions for the piece's data members
        * a draw function (the piece knows how to draw itself)
}

class C_Square : public C_Piece
{
    private data member:
        * the color of the checkerboard square

    public member functions:
        * a function to set the color of the checkerboard square
        * a draw function (the checkerboard square knows how
          to draw itself)
}
```

The C_Square draw function draws itself. It then checks the state of its piece, and if the state is NOT_ALLOWED or EMPTY no further drawing is required. Otherwise, it calls the C_Piece draw function and the piece draws itself.

Only one bounding rectangle—that of each checkerboard square—is used. The bounding rectangle of each piece can be set by a function that you will write. The bounding rectangle of each piece can be calculated by subtracting a “margin” from the dimensions of the window that you are drawing in, and, of course, calculating where each of the 64 squares is positioned.

Figure B-3 shows a finished Chapter 3 exercise.



Figure B-3: Chapter 3 Exercise

Chapter 4 Exercise

In the Chapter 4 exercise, you will make the checker pieces move. The user will move the pieces by pressing down on the mouse button over the piece to be moved, holding the mouse button down, and moving the mouse, effectively dragging the piece to its new position. If a game is in progress and the user chooses new colors, piece shapes, or player names, invoke the change without restarting the game. The game can be restarted by choosing the “Start Game” menu item.

Note: Declaring a “special” piece named `m_MovingPiece` of class `C_Piece` is a good idea. When a piece is to be moved off of a square, copy the appropriate data member information from that square (piece color, player, etc.) into the `m_MovingPiece` object and set the square to `EMPTY`. Use `m_MovingPiece` in the play of the game. At the end of each legal move, you would then copy the appropriate data member information from the `m_MovingPiece` object to the square on which the piece is being placed.

When the user clicks the mouse on a piece and holds the mouse button down while dragging it, the piece should move as well. When the mouse button is released, the piece should stay on the closest square. To accomplish this movement, you will add `WM_LBUTTONDOWN`, `WM_MOUSEMOVE`, and `WM_LBUTTONUP` handler functions. Your program should disallow placing a piece on top of another or on the wrong color square. It should also disallow players from moving out of turn or in the wrong direction. When an illegal move has occurred, the program should beep or a message box should appear and the piece should be restored to its position prior to the illegal move. The function `CRect::PtInRect()` is very useful.

You will need to select a technique for redrawing. Some choices are: exclusive or/nor, backing store, or drawing limited areas of the screen to memory. The exclusive or/nor technique is difficult to use if you have text (player initials are on each piece), so it is not well suited to this application. The backing store technique works well. Another technique, that of redrawing a specific region, works very well here because the squares that need to be redrawn can be easily defined. This technique of redrawing a specific region has the very desirable feature of graphical “stability”; the user notices no unusual graphical effects.

Tip: Hints to implement this redrawing technique are described in the next paragraph.

The Redraw Region for Moving a Piece

You need to design a function that redraws the region formed by the union of the bounding rectangle (the size of a checkerboard square) that is centered at the previous mouse position with the similar bounding rectangle centered at the current mouse position (call this region `UnionRect`). This region will always encompass all the changes when a piece is being moved. The implementation of this redraw technique described here draws to memory and then `BitBlt`s it to the screen. The checkerboard background is first drawn to memory using a memory device context followed by the moving piece being drawn with that same memory device context. This “off-screen” drawing is then “`BitBlt`-ed” onto the screen.

Draw into memory only that portion of the checkerboard that is inside `UnionRect`. To do this:

1. Set the viewport origin to `-UnionRect.left`, and `-UnionRect.top`. `CDC::SetViewportOrg()` sets the viewport origin of the device context. The viewport, along with the device context, defines how GDI maps points in the logical coordinate system to points in the device coordinate system of the actual device.
2. Check to see if any portion of the `UnionRect` is outside of the checkerboard.
 - a. If it is, then first fill the memory device context with white.
3. Check to see if a portion of any square is inside the `UnionRect`.
 - a. If it is, draw it onto the memory device context; the viewport origin setting described above will only draw the portion that is in the `UnionRect` to memory.
4. Then draw the moving piece onto that memory device context.
5. Set the Viewport Origin back to 0, 0 and `BitBlt` the image from the memory device context to the screen.

Note: This redrawing method is very fast and produces no noticeable visual effects.

Tip: The technique of drawing to memory to improve graphical stability can be used elsewhere in the checkers game. When the game is first started and the checkerboard is first being drawn to the screen, you can sometimes see the drawing as it occurs. To avoid this effect, you can draw the checkerboard to memory and then `BitBlt` it onto the screen. This will produce graphics with no noticeable effects to the user.

Figure B-4 shows a finished Chapter 4 exercise. In this example, customized cursors are used which denote whether the player whose turn it is has dark or light pieces.



Figure B-4: Finished Chapter 4 Exercise

Chapter 8 Exercise

This exercise adds a dialog box to the checkers game. Create a modal dialog box which contains choices for the players' names, the colors, and the shape of the pieces. This dialog is to be invoked from the "OpenDialog" menu item. The dialog box should contain radio buttons with the color and piece shape choices and edit control boxes to name the players, and OK and Cancel buttons. When the dialog first appears, these controls should be initialized to the current menu settings. The user can change any of these settings within the dialog. When OK is pressed, the dialog should close and the current dialog choices should take effect. That is, the colors of the board, the piece shapes, and the players' names will change and your menu items should be checked as if the user had made the choices directly from the menus. If Cancel is pressed, no changes should occur to your menus or the checkers game. At this juncture, you may wish to discontinue using a `CMenu` object to check the menu items; you can use the `CCmdUI` class and have `OnUpdate()` functions for each menu item.

The dialog box will contain edit boxes in which you can enter the players' names. The names of the players are to appear in the title bar. Use the `CWnd::SetWindowText()` function to insert text in the title bar. When the dialog is closed with the OK button, the board and pieces will have the colors specified and the pieces will have the specified shape. Also, the pieces will have the players' initials and the application's title bar will contain the players' names that were entered in the edit controls.

Figure B-5 gives a finished Chapter 8 exercise. In the example shown in Figure B-5 the user first started the game and then opened the dialog box in which he has made new choices for the game.



Figure B-5: Finished Chapter 8 Exercise

Chapter 13 Exercise

In this assignment you will do your checkers game as an MDI application.

1. Create a four-class application (document, view, application, and mainframe) for your checkers game.
2. Separate the code such that you can file the data and restore a game-in-play from a file and continue the play.
3. Add a toolbar such that choices for the checkerboard's color and piece shapes can also be made with toolbar buttons.

You can use the code that you have developed for the Chapter 8 exercise, but you must decide which code must be placed in the document and which code must be placed in the view. The code separation can be much like the tic tac toe game of Chapter 11. The data will include the current choices for the checkerboard colors and the piece shape. The data will also include the player's names, the player's turn, and the history of each player's moves. Using the technique that you learned in the Chapter 11 tic tac toe game you can store the play of the game in history arrays. You will need a history array for each type of piece: A pieces, kingA pieces, B pieces, and kingB pieces. Provide a `Serialize()` function for the data, so that it can be filed and retrieved from file.

In the view class, you will draw in the `OnDraw(CDC* pDC)` function whenever you need to use a `CPaintDC` device context, i.e., drawing in response to a `WM_PAINT` message. In the `OnDraw(CDC* pDC)` function, you must use the device context that is provided.

Note: Do not get your own device context. When you need to use a `CClientDC` device context, then that code must be located elsewhere (not in the `OnDraw(CDC* pDC)` function) and you must obtain the `CClientDC` device context yourself.

4. Customize the toolbar such that the user can make the checkerboard color choices and the piece shape choices using a toolbar button.

The idea of having the names of the players appear in the main frame's title bar cannot be applied in an MDI. (It can be applied in an SDI.) So ignore the code that sets the window's caption bar (it is ineffective) or remove it.

When you are through building your MDI checkers application, you will be able to open different games in different windows and continue the play of these multiple games by shifting the focus to a game's window and making a play, then shifting the focus to another window to make a play in the game displayed in that window, etc. Figure B-6 shows a finished Chapter 13 exercise with two in-progress games open in different windows. In this example program, the cursor indicates the player's turn for the game in that window; in the following example, it indicates that the "dark" colored player has the turn in the right window. When the cursor moves over the left window, it will indicate the player's turn in that window. The cursor points up and is "light" when the player with the turn has the light-colored pieces. The cursor points down and is "dark" when the player with the turn has the dark-colored pieces. Also note that a king piece has been denoted by writing the text "king" on the piece.

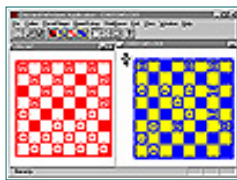


Figure B-6: Finished Chapter 13 Exercise

[Table of Contents](#)

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

[Table of Contents](#)

Appendix C Review of C++

The Microsoft Foundation Class (MFC) Library lets you write professional applications that run under Microsoft Windows. MFC makes many of the intricacies of the Windows Application Programming Interface (API) transparent to you, the programmer. MFC provides a C++ framework which lets you conveniently treat items such as windows and dialog boxes as objects, and still gives you access to all the capabilities of the Windows API.

This appendix briefly discusses the C++ concepts that must be understood when writing Windows applications using MFC. For readers not familiar with C++, this appendix will serve as a C++ crash course and will give the background needed in order to understand the programs described in this book. For others, this appendix will serve as a review of those C++ features needed to complete the programming exercises.

C++ adds a new set of features to the C language. These features both facilitate object-oriented programming and provide strong compile-time type-checking. The object-oriented style is particularly useful for organizing programs that have an extensive user interface. Strong type-checking greatly reduces the number of programming errors that are undetected by the compiler and linker.

This appendix presents the main new features of C++, which are the “class” concept and class inheritance. Some additional C++ concepts are also discussed.

The Class Concept

With C++, you define both the data type and its operations at once, by declaring a *class*. A class consists of data members and member functions that operate on that data.

In C++, if you have defined a class named “C_First,” you can create an *instance*, named “FirstObject,” of this class with the statement:

```
C_First FirstObject;    // object instantiation
```

Note: The keyword “class” does not appear in this statement. In C++ you do not precede the declaration of a class instance with the keyword “class.” The keyword class is only used when defining or declaring the class, which is discussed in the following paragraphs.

An instance of a class is called an *object*; it contains a block of memory that holds the values assigned to the object’s data members.

Note: C++ has a new comment delimiter, the double slash (//). The text following the //, up to the end of the line, is treated as a comment; it is ignored by the compiler.

The Class Definition

A class is defined when the closing brace of the class body is seen. Until a class is defined no objects of that class can be created. The following example of a class definition defines a class named “C_First,” which has a data member named m_nValue, which is an integer, and a member function named f(), which has no arguments and no return value.

```
#include <stdio.h>
//-----definition for class C_First
class C_First
{
    public:
        int m_nValue;        // data member definition
        void f()              // member function definition
            { printf ("The value of m_nValue is %d\n", m_nValue); }
};
```

A class definition always consists of the keyword “class” followed by the class name, then the brace-delimited *class definition block*. The class definition block contains definitions of all data members and declarations, or definitions, of all member functions. In the preceding example, the member function f() is defined within the class definition block.

A member function is specified to be a member of the class by placing either its prototype or its definition in the class definition block. (To review a familiar distinction, whereas a function declaration or a function prototype just names the function and specifies its arguments and return value, a function definition specifies the function’s code.) The prototype is usually placed in the class definition block (rather than the function definition) for all but very short functions. When you use the prototype, then you must define the function somewhere else in the program; this will be discussed later.

The formal specifications of the C++ language use the term *class declaration* for the code that is here called a class definition. The formal specifications do not use the term “class definition” at all. However, most books on C++ use the term “class definition”; therefore, the term “class definition” is used here in order to minimize confusion.

Forward Class Declarations

It is sometimes important to define a pointer to a class at an earlier place in the source file than where the class is defined. This is done using a *forward class declaration* ahead of the place where you define the pointer to the class. A forward class declaration consists simply of the keyword “class” followed by the class name and then a semicolon. For example:

```
class C_First;
```

A class is declared when its name is first encountered. Once a class is declared, references and pointers to that class can be defined.

Access Level

The members of a class may be given an *access level* of public, private, or protected. In order to do this, insert an access-specifier keyword— “public:”, “private:”, or “protected:” in the class definition; it applies to all members (data or functions) that follow it, up to the next access specifier. The access level controls access to the member from elsewhere in the program. If some members are declared in the class definition block before any access specifier appears, the access level defaults to private for those members.

In the preceding example, the access specifier “public:” appears before the declarations of the data member and the member function. Thus all members have *public access*. Public members have no restrictions on their access; they can be accessed by any function.

In subsequent examples, the access-specifier “private:” appears before the data members; thus they are *private data* members. Private data members can be accessed only by member functions of their own class. It is common to make data members of a class private, so that they are available only to the member functions of that class, but not to its “interface” (i.e., not available to the outside world). By restricting outside access to data members, you can keep outside code from arbitrarily altering an object’s data. The member functions can be coded to ensure that the object’s data is protected from corruption.

The access specifier “protected:” allows access only by member functions of the same class or a *subclass* (a derived class). The concepts of subclass and *protected access* will be discussed when inheritance is introduced later in this appendix.

Using Public Members

Once we instantiate an object FirstObject of class C_First as follows:

```
C_First FirstObject;
```

then we can refer to the public data member m_nValue of object FirstObject as FirstObject.m_nValue, and we can invoke the public member function f() for object FirstObject by the statement:

```
FirstObject.f();
```

To show how the execution of the code of members works, we now add a main function:

```
void main()
{
    C_First FirstObject;
    FirstObject.m_nValue = 4;
    FirstObject.f();
}
```

When main executes, its code does the following:

1. The object FirstObject is instantiated by the first statement in main.
2. The second statement sets data member m_nValue of object FirstObject to 4.
3. The third statement invokes member function f() for object FirstObject.
4. The code of f() prints the value of the variable referred to by the symbol “m_nValue.” Since f() is being invoked for FirstObject, the symbol “m_nValue” refers to data member m_nValue of object FirstObject. In other words, it refers to FirstObject.m_nValue. Thus the line “The value of m_nValue is 4” will be displayed.

This example has illustrated the use of *objectname.datamembername* to represent a data member of an object, and the use of *objectname.memberfunctionname()* to invoke a member function for an object.

Using Public Members Via the Object's Pointer

A pointer to the object is obtained by using the ampersand (&) operator. Using the preceding example program, we obtain a pointer to the object FirstObject with the statement:

```
C_First* pFirstObject = &FirstObject;
```

Note: The MFC coding convention for pointer declarations is that the asterisk always appears next to the type name rather than next to the variable name; this is different from the usual C convention, for example:

```
C_First* pFirstObject    // MFC coding convention
C_First *pFirstObject    // usual C convention
```

To refer to a data member when you have the object’s pointer *objectpointer*, use the syntax:

`objectpointer->datamembername`

For the preceding example, we have:

`pFirstObject->m_nValue`

Similarly, to invoke a member function when you use the object's pointer *objectpointer*, use the syntax:

`objectpointer->memberfunctionname( )`

For the above example, we have:

`pFirstObject->f()`

The “This” Keyword

The C++ language has a keyword named *this*, which is useable inside a member function. The keyword *this* represents the address of the invoking class object, which is illustrated in the following code. The `printf()` statement prints out the value of *this*, which is the location of the object `SecondObject`.

```
#include <stdio.h>
//-----Definition of class C_Second
class C_Second
{
    public:
        int m_nValue;
        void display()
        {
            printf ("I am an object of class C_Second.\n"
                    "My data member is %d \n"
                    "and my address is %p\n",
                    m_nValue, this);
        }
};

void main()
{
    C_Second SecondObject;
    SecondObject.m_nValue = 7;
    printf ("The address of SecondObject is %p\n",
            &SecondObject);
    SecondObject.display();
}
```

When `main()` executes, the `printf()` statement in `main` displays the address of object `SecondObject` of class `C_Second`. Then the statement `SecondObject.display()` invokes member function `display()` for object `SecondObject`. This member function displays 7 for the value of data member `m_nValue`, and displays the value of the *this* symbol after the text “...and my address is.” The displayed value of the *this* symbol will match the displayed address of `SecondObject`.

Invoking Member Functions

In C++, classes can have member functions as well as data members. Ordinary member functions (those that are not constructors or destructors) are always called in association with an object. It is said that the member function is invoked for the object. You invoke a member function for an object by using the name of the object, followed by a period, followed by the name of the member function, followed by its argument list (if any). A member function behaves differently from a plain function in two ways:

1. When the executing code of a member function uses the name of a data member, the value that is

referred to is that data member of the object that the member function is invoked for. (This is illustrated in the preceding code.)

2. When the executing code of a member function uses the name of another member function, that other member function is invoked for the object that invoked the first member function. (This is illustrated in the following code.)

You have just seen that a member function can refer to a data member of its invoking object by the name of the data member. A member function `h()` can similarly invoke another member function `g()` by using the name of the member function `g()` followed by its argument list (if any). When this syntax is used, `g()` is invoked for the same object that `h()` is invoked for. This is illustrated in the following example.

```
#include <stdio.h>
//-----definition of class C_Third
class C_Third
{
    public:
    int m_nValue;
    void g() { printf ("m_nValue=%d\n", m_nValue); }
    void h() { g(); }
};

void main()
{
    C_Third ThirdObject;
    ThirdObject.m_nValue = 10;
    ThirdObject.h();
}
```

The line `ThirdObject.m_nValue = 10;` in function `main()` sets data member `m_nValue` of object `ThirdObject` to 10. Then the statement `ThirdObject.h()` causes `h()` to be invoked for object `ThirdObject`. The body of `h()` contains the statement “`g();`”. When executed, this statement causes member function `g()` to be invoked for object `ThirdObject`. Member function `g()` prints the value of data member `m_nValue` of `ThirdObject`, which is the object that `h()` was invoked for. Thus the line “`m_nValue = 10`” is displayed.

Constructors

A *constructor* is a special member function. The constructor’s name is the same as the class name. No return type (not even `void`) is given in a declaration or definition of a constructor. A constructor may have zero or more arguments. Unlike an ordinary member function, a constructor cannot be explicitly called for an object. Rather, a constructor is automatically called when an object is instantiated. It is the job of the constructor to initialize the class’s data members, although a constructor can do other things as well.

The process of instantiation is simple for an object of a class that has no object data members (discussed later in this appendix) and no base class (also discussed later):

1. Space is allocated for the object if the object is being instantiated dynamically (i.e., with the `new` operator, which is discussed later).
2. The constructor whose signature matches the parameters supplied in the instantiation statement is executed.

For objects in the static and automatic storage class, space is preallocated. This is why space for the object only needs to be allocated at the time of instantiation for *dynamically allocated objects*. For the general case, the instantiation process is more complex. The general case is discussed later in this appendix.

For objects in the *static storage class*, space is allocated when the program is loaded. Objects that are instantiated outside of any function and objects that are instantiated within a function with the `static` keyword are in the static storage class.

For objects in the *automatic storage class*, space is preallocated on the stack when the function is entered. Objects that are instantiated nondynamically in a function without the keyword `static` are in the automatic

storage class.

Function Overloading

In C++, whenever two functions (or member functions) within a particular class have the same name but either a different number of arguments or at least one argument with a different type, they are said to be *overloaded functions* (or overloaded member functions). Overloaded functions (or member functions) within a class, are treated as distinct functions. The number of arguments and the types of the arguments of a function (or a member function) are called the function's signature. Thus two functions (or two member functions) of the same class with the same name but different signatures are overloaded functions.

Note: Overloading can apply to constructors as well as to ordinary member functions.

For an example of overloading see Listing C-1a—the constructor `C_Rectangle` is overloaded.

Destructors

Like constructors, the *destructor* is a special member function. The name of the destructor is the class name preceded by a `~`. You do not need to define a destructor for a class since the compiler will automatically supply one for you, but you may define one. There can never be more than one destructor defined for a class. A destructor has no arguments and no return value.

The destructor is automatically invoked for an object when the object ends its lifetime. One of the primary uses of destructors is to automatically return memory to the *free store*, which is also known as the heap.

The destructor performs cleanup; generally, it should deallocate resources that the constructor allocated. For example, if the constructor allocated memory (other than the memory allocated for the object's data members), it is usually the job of the destructor to deallocate it. If the object was dynamically allocated, then the storage for the object should be deallocated.

If the constructor invokes *new*, the destructor should have a corresponding *delete* to deallocate storage that *new* allocated (*new* is similar to the C `malloc` function). The C++ operator *delete* releases the storage pointed to by its argument. It is similar to the `free` function of C. If you have used *new* to allocate storage, you must use *delete* to deallocate that storage. You must use *delete* rather than `free` to deallocate storage that has been allocated by *new*. A convenient feature of *delete* is when called for a null pointer, it does nothing and returns.

A Basic Example—Example 1-1

The following paragraphs present a simple C++ program which further illustrates the concept of a class. This simple example is presented in four code sections: Listing C-1a gives the definition of class `C_Rectangle`; Listing C-1b gives the definitions of the member functions of `C_Rectangle`; Listing C-1c lists a main function that creates instances (objects) of the class in static memory and on the stack; and Listing C-1d, which is presented in later paragraphs, gives an alternative main function that creates class objects on the free store.

Each code section that is given is followed by a discussion of the concepts introduced in that code section. To form a complete program, you should combine Listings C-1a and C-1b, and add a main program from either Listing C-1c or C-1d.

Listing C-1a below defines a class named `C_Rectangle`. An object of this class represents a labeled right rectangle on a two-dimensional surface. The data members `m_nLeft` and `m_nTop` give the x- and y-coordinates of the upper left corner of the rectangle. Data members `m_nWidth` and `m_nHeight` represent the width and height of the rectangle. And data member `m_psName` points to a character string that represents the label.

Listing C-1a

```
#include <stdio.h>
#include <string.h>
```



```
// ----- Definition for class C_Rectangle
class C_Rectangle
{
    public:
        C_Rectangle (int nLeft, int nTop, int nWidth, int nHeight,
            char* psName);           // Constructor declaration
        C_Rectangle (char* psName); // Constructor declaration
        C_Rectangle ();              // Default constructor declaration
        ~C_Rectangle ();             // Destructor declaration
        void move (int nX, int nY);
        void print ();
    private:
        int m_nLeft, m_nTop;
        int m_nWidth, m_nHeight;
        char* m_psName;
};
```

The class C_Rectangle declares five private data members: m_nLeft, m_nTop, m_nWidth, m_nHeight, and m_psName. These five private data members can only be accessed by member functions of class C_Rectangle. When data members are declared as private, it is referred to as *data encapsulation*, since access to private members is denied to anything other than member functions of its class, which is C_Rectangle in this example.

The class C_Rectangle also declares six public member functions: C_Rectangle(int,int,int,int,char*), C_Rectangle(char*), C_Rectangle(), ~C_Rectangle(), move(int,int) and print(). These six member functions will be defined in the next code listing, Listing C-1b.

Note: The prototypes of several member functions including print() have empty parentheses. In C++, this means that the function has no arguments. On the other hand, in C, empty parentheses indicate that the arguments are unspecified, and you use (void) to specify no arguments. The notation (void) is not used in C++.

The Scope-Resolution Notation

When a member function is defined outside of the class definition block, the member function's name is not enough to unambiguously specify which member function it is. This ambiguity arises from the fact that member functions of different classes may have the same name.

In order to avoid this ambiguity, a special notation must be used to indicate the class of the member function when the member function's name is used at the top of its definition outside of the class definition block. This notation is called *scope-resolution notation*, and it consists of using the syntax *classname::memberfunctionname* to identify the member function. In Listing C-1b, the six member function definitions use this notation, since they are defined outside the class definition block.

Listing C-1b lists the definitions of the six member functions. A discussion of each of the member function definitions follows this listing.

Listing C-1b

```
// ----- Member function definitions
//
// Definition of constructor with 5 parameters
C_Rectangle::C_Rectangle(int nLeft, int nTop, int nWidth,
                        int nHeight, char* psName)
{
    m_nLeft = nLeft; m_nTop = nTop;
    m_nWidth = nWidth; m_nHeight = nHeight;
    if (m_nWidth < 0) m_nWidth = 0;
    if (m_nHeight < 0) m_nHeight = 0;
    m_psName = new char[strlen(psName) + 1];
    strcpy (m_psName, psName);
}
```

```

}
//
// Definition of constructor with one parameter
C_Rectangle::C_Rectangle (char* psName)
{
    m_nLeft = 1;  m_nTop = 1;
    m_nWidth = 10;  m_nHeight = 10;
    m_psName = new char[strlen(psName) + 1];
    strcpy (m_psName, psName);}
}
//
// Default constructor definition
C_Rectangle::C_Rectangle ()
{
    m_nLeft = 1;  m_nTop = 1;
    m_nWidth = 10;  m_nHeight = 10;
    m_psName = new char[strlen("NO NAME") + 1];
    strcpy (m_psName, "NO NAME");}
}

C_Rectangle::~C_Rectangle ()    // Destructor definition
{
    delete m_psName;
}

void C_Rectangle::move (int nX, int nY)
{
    m_nLeft += nX;
    m_nTop += nY;
}

void C_Rectangle::print ()
{
    if ( m_psName != NULL)
        printf (" %s:  Rectangle (%d,%d) to (%d,%d)\n\n",
                m_psName, m_nLeft, m_nTop,
                m_nLeft + m_nWidth, m_nTop + m_nHeight);
}

```

Listing C-1b defines two ordinary member functions (move and print), three constructors named C_Rectangle, and a destructor named ~C_Rectangle. These six member functions are discussed in the following sections.

The member function move(int,int)

The member function move(int,int) is called for objects of class C_Rectangle. An object of C_Rectangle represents a labeled rectangle. move(int nX, int nY) adds its arguments nX and nY to data members m_nLeft and m_nTop respectively, thereby moving the rectangle by the amount given in nX and nY.

The member function print()

The member function print() displays the characteristics of the labeled rectangle object that it is called for. It just uses the run-time library function printf() to convert the values of the object's data members to a text string and to write this string to the standard output device. It displays the coordinates of the lower right corner of the rectangle, rather than displaying the width and height of the rectangle, so it calculates these coordinates from the data members.

When member function print() is invoked for an object, we say that the object is printing itself. Thus (colloquially speaking) objects of class C_Rectangle know how to print themselves.

The constructor `C_Rectangle(int,int,int,int,char*)`

The code of constructor `C_Rectangle(int,int,int,int,char*)` initializes the data members `m_nLeft`, `m_nTop`, `m_nWidth`, and `m_nHeight`, to its argument values. These argument values are supplied at the time of instantiation.

Often, a class is responsible for storage in addition to that reserved for its data members. In this book, this additional storage is referred to as *indirect storage*. The constructors of `C_Rectangle` store the string label in indirect storage, as described next.

`C_Rectangle` is responsible for storing a string name, but it does not store the string directly in a data member. Rather, it finds the length of the string using the familiar `strlen()` function, allocates a block of memory that the string and its zero terminating byte will fit into, copies the string into this block, and sets its data member `m_psName` to point to this block.

The constructor uses the C++ operator *new* to allocate the block of memory. This operator is similar to the `malloc` function of C, but it is more general because, in addition to being able to allocate a basic block of memory like an array of chars, it can be used to allocate a block of memory for a class object and initialize the object all at once. The return value of *new* is a pointer to the type of variable that is its argument. In a C++ program, *new* should always be used to allocate storage; `malloc` and its related allocation functions should not be used at all. The *new* operator is described in more detail later in this appendix.

The various ways in which an object can be instantiated are discussed later in this appendix. There, you will see how arguments are supplied to the constructor when the object is instantiated.

The constructor `C_Rectangle(char*)`

This constructor with one argument creates a rectangle whose upper left corner has position (1,1), with a height of 10 and a width of 10, and whose label is the string supplied as the parameter when the object is instantiated.

The default constructor `C_Rectangle()`

A constructor with no arguments is called a *default constructor*. The default constructor in Listing C-1b, `C_Rectangle()`, creates a rectangle whose upper left corner has position (1,1), and which has a height of 10, a width of 10, and the label “NO NAME.”

If there are no constructors defined for a class, the compiler will automatically generate a constructor with no arguments. This compiler-generated constructor will call constructors on object data members of the class and, if the class is a derived class, will call constructors of base classes. (The concepts of derived and base classes and of object data members are described later in this appendix). However, the compiler-generated constructor will not initialize any non-object data members. Thus if the object is instantiated on the stack (as an automatic variable) or is instantiated dynamically (dynamic allocation is discussed later), its non-object data members will contain garbage.

The term default constructor is somewhat misleading. You would think that it means the constructor that the compiler generates when you do not supply your own. But it actually means a constructor with no arguments, whether you define it or let the compiler generate it.

Note: If you define a constructor with one or more arguments for a class, then the compiler will not automatically generate a constructor with no arguments even if one is called for. In this case, the compiler will produce an error message at compile time saying that a constructor with no arguments is missing.

The destructor `~C_Rectangle()`

In Listing C-1b, the class `C_Rectangle` defines the destructor `~C_Rectangle`. The destructor `~C_Rectangle()` uses the statement:

```
delete m_psName;
```

to deallocate the indirect storage space for the string that the constructors allocate with *new*. Note that this

statement deallocates the storage pointed to by `m_psName`. It does not deallocate the space for the pointer `m_psName`.

If an object has object data members and/or is derived from a base class, then additional actions are taken during destruction. This is discussed later in this appendix.

Syntaxes to Instantiate Static or Automatic Objects

You can supply zero or more parameters when you instantiate an object. Whenever you instantiate an object, a constructor of its class is automatically called. The constructor that is called is the one whose signature (i.e., its number of arguments and its argument types) matches the supplied instantiation parameters. There are various ways to write instantiation statements, and the available ways depend on whether you are supplying zero, one, or more parameters. These ways are described next.

Supplying No Parameters

To instantiate a static or automatic object without supplying any initialization parameters, the following syntaxes may be used:

```
classname objectname;
```

Example from Listing C-1c:

```
C_Rectangle SmallRect;  
    classname objectname = classname
```

Example:

```
C_Rectangle r = C_Rectangle();
```

The first is preferred because of its simplicity. When an object is instantiated with no parameters, the constructor that is invoked is the constructor with no arguments. As has been mentioned, the constructor with no arguments is called the default constructor.

Supplying One Parameter

To supply one parameter when you instantiate a static or automatic object, you can use any of the following statements:

```
classname objectname = value;
```

Example from Listing C-1c:

```
C_Rectangle SmallRect = "Small Rectangle";  
  
    classname objectname (value);
```

Example from Listing C-1c:

```
C_Rectangle StaticRect("Static Rectangle");  
  
    classname objectname = classname(value);
```

Example:

```
C_Rectangle StaticRect = C_Rectangle("Static Rectangle");
```

The first or second is preferred over the third, for simplicity.

Supplying More Than One Parameter

To supply more than one parameter when you instantiate a static or automatic object, you can use either of the following two kinds of statements:

```
classname objectname(value1, ..., valuen);
```

Example from Listing C-1c:

```
C_Rectangle MediumRect(10,15,300,300,"Medium Rectangle");
```

```
classname objectname = classname(value1, ..., valuen);
```

Example from Listing C-1c:

```
C_Rectangle BigRect = C_Rectangle(5,5,500,500,"Big Rectangle");
```

The first is preferred for simplicity.

Instantiating an Array of Objects

The following statement instantiates an array of three C_Rectangle objects:

```
C_Rectangle aRects[3];
```

Arrays of objects are declared in the same way that arrays of other data types are declared. When an array of objects is called, the constructor is called for each element in the array. If they are declared without initializing them, as given in the preceding statement, the class must have a default constructor.

Each element of the array can be initialized by calling the constructor with arguments for each element. If you do not provide enough initializers for the entire array, the default constructor is called for the remaining elements. For example, the statement:

```
C_Rectangle aRects[3] =  
    { C_Rectangle(1, 1, 20, 20, "Array: 1st-Rectangle") };
```

instantiates an array of three C_Rectangle objects. The first object is constructed by the constructor C_Rectangle(int, int, int, int, char*), and the other two are constructed by the default constructor.

Main Program That Instantiates Static and Automatic Objects

Listing C-1c below is a main program that instantiates eight objects, and then invokes their member functions.

Object GlobalRect is instantiated outside of any function, so it is a variable of the static storage class.

The next two objects, SmallRect and MediumRect, are instantiated within a function (the function main), without the static keyword and without the *new* keyword. They are therefore variables of the automatic storage class; they are stored on the stack.

The fourth object, StaticRect, is instantiated within the main function, but the keyword “static” is used. This keyword causes StaticRect to be a variable of the static storage class, even though it is instantiated within a function.

The fifth object, BigRect, is instantiated a little later, after several invocations of the print member function on the other objects. BigRect is a variable of the automatic storage class.

Note: An instantiation statement appears after member function call statements of other objects appear. C++ is liberal in this regard. Instantiation statements can appear anywhere. In fact in C++ all variable definition statements can appear anywhere. This contrasts with C, which requires that all variable definitions precede any so-called executable statements in a function. In C++ the distinction between statements that define or instantiate variables and executable statements is dropped. After all, constructors are called when objects are instantiated, and constructors can contain executable code.

The last three objects are instantiated in the array `aRects[3]`, which is of the automatic storage class.

Listing C-1c

```
// ---A main function instantiating static & automatic objects
//

C_Rectangle GlobalRect;

void main ( )
{
    C_Rectangle SmallRect = "Small Rectangle";

    C_Rectangle MediumRect(10, 15, 300, 300, "Medium Rectangle");

    static C_Rectangle StaticRect("Static Rectangle");

    GlobalRect.print();
    SmallRect.print();
    MediumRect.print();
    StaticRect.print();

    C_Rectangle BigRect = C_Rectangle(5,5,500,500,"Big Rectangle");

    BigRect.print();

    C_Rectangle aRects[3] =
        { C_Rectangle ( 1, 1, 20, 20, "Array: 1st-Rectangle") };

    for (int i = 0; i < 3; ++i)
        aRects[i].print();

    MediumRect.move(50, 40);
    MediumRect.print();
}
```

Object instantiation statements that are outside of any function body, and object instantiation statements that are within a function code body but contain the `static` keyword, create objects that are allocated in static memory. These are sometimes called *static objects*. In Listing C-1c, `GlobalRect` and `StaticRect` are static objects. All the static objects are constructed just before the program's main function starts execution. They are constructed in the order that they appear in the source file. The lifetime of the static objects is the full program's lifetime. They are destructed after the main function returns, or after the run-time-library function exit is called. The static objects are destructed in the opposite order to the order they were constructed. All static objects that are constructed are eventually destructed before the program terminates execution.

Instantiation statements within a function but without the `static` specifier create objects that are allocated on the stack. Such objects are automatic objects. In Listing C-1c, `SmallRect`, `MediumRect`, `BigRect`, and `aRects[3]` are automatic objects. Automatic objects are instantiated when their instantiation statement is encountered during program execution. An automatic object ends its lifetime when execution leaves the block that it is instantiated in. It always ends its lifetime at the latest, when the function it is instantiated in returns or calls `exit` to end program execution. An automatic object is destructed when the object ends its lifetime. If an automatic object is instantiated in a repeating block, such as the controlled block of a `for` loop, the same automatic object will be instantiated and end its lifetime and be destructed for each pass through the block.

Calling the Ordinary Member Functions

The statement `GlobalRect.print();`, in Listing C-1c, produces the line:

```
NO NAME:  Rectangle (1,1) to (11,11)
```

since `GlobalRect` was instantiated with no parameters. The default constructor initialized `GlobalRect` to have 1 in `m_nLeft`, 1 in `m_nTop`, 10 in `m_nWidth`, 10 in `m_nHeight`, and it set the pointer `m_psName` to point to the string “NO NAME.”

The statement `SmallRect.print();` produces the line:

```
Small Rectangle: Rectangle (1,1) to (11, 11)
```

since `SmallRect` was instantiated with one parameter, a string, and it set the pointer `m_psName` to point to the input parameter. This constructor also initialized `m_nLeft` to 1, `m_nTop` to 1, `m_nWidth` to 10, and `m_nHeight` to 10.

The statement `MediumRect.print();` produces the line:

```
Medium Rectangle:  Rectangle (10,15) to (310,315)
```

The statement `StaticRect.print();` produces the line:

```
Static Rectangle: Rectangle (1,1) to (11, 11)
```

The statement `BigRect.print();` produces the line:

```
Big Rectangle:  Rectangle (5,5) to (505,505)
```

The for loop printing out `aRects[i]` produces:

```
Array: 1st-Rectangle: Rectangle (1,1) to (21,21)
NONAME: Rectangle (1,1) to (11,11)
NONAME: Rectangle (1,1) to (11,11)
```

The statements `MediumRect.move(50, 40);` and `MediumRect.print();` produce the line:

```
Medium Rectangle:  (60, 55) to (360,355)
```

Syntaxes to Dynamically Instantiate and Destantiate an Object

The last section showed how to instantiate objects in the static and automatic storage classes. This section shows you a third way to instantiate objects. This is *dynamic allocation* using the operator *new*. Dynamic instantiation allocates storage for an object during program execution. To this extent, it is similar to the use of `malloc` in C which allocates heap memory space. However, rather than allocating just a block of memory, dynamic instantiation using the *new* operator allocates memory for an object and initializes the object by calling its constructor. The *new* operator must be used for dynamic instantiation of objects. The argument of the *new* operator is a type specifier which represents a variable or an array of variables. The term “variable” can be a simple variable, a struct instance or a class instance, but we focus here on the class instance (object).

The operator *new* allocates a block of memory that is the size that is occupied by the object. It allocates this block of memory from the *free store*. Free store is the term used in C++ for a reservoir of uninitialized memory. If an object (or an array of objects) is being instantiated, then *new* initializes the object by using the constructor whose signature matches the supplied parameters. Finally, *new* returns a pointer to the object (or array of objects) that it created. You can supply zero or more parameters when you dynamically instantiate an object.

The *new* operator normally takes one argument, which is a type specifier that gives the type of object (or array of objects) that is being instantiated. The *new* operator and its argument constitute an expression. The return value of this expression is a pointer to the instantiated object (or array of objects). The various forms

that this expression can take are given in the paragraphs below.

Just as the `malloc` function has the `free` function as its counterpart, the `new` operator has the `delete` operator as its counterpart. The `delete` operator deallocates blocks of memory, returning them to the free store. The `delete` operator automatically calls the destructor for the object before it deallocates the memory. You can only apply `delete` to pointers that were returned by `new`. Deleting a pointer not obtained from `new` causes your program to behave strangely (this is an error that the compiler cannot detect). However, you can delete a null pointer (a pointer with value 0) with no adverse effects.

It is best for an application to deinstantiate each dynamically allocated variable or array when the application is done using it. This minimizes the use of memory. However, there is a safety valve: any storage that is still allocated to dynamic instances is automatically deallocated when the program terminates. The following paragraphs show how deinstantiation is done.

Instantiating an Object Supplying No Parameters

To dynamically instantiate an object without supplying any initialization parameters, either of the following syntaxes may be used.

```
new classname
```

Example:

```
new C_Rectangle;  
new classname
```

Example:

```
new C_Rectangle();
```

When an object is instantiated with no parameters, the constructor that is invoked is the constructor with no arguments (the default constructor).

Instantiating an Object Supplying One or More Parameters

To supply one or more parameters when you dynamically instantiate an object, you use:

```
new classname objectname(value1, ..., valuen)
```

Example from Listing C-1d:

```
new C_Rectangle(nLeft, nTop, nWidth, nHeight, psName);
```

Instantiating an Array of Objects

An array of objects can be dynamically instantiated. However, you cannot supply any initialization parameters; thus the default constructor is always used to construct objects that are elements of an array that is dynamically instantiated. Code from Listing C-1d exemplifies this:

```
C_Rectangle* aRects = new C_Rectangle[3];
```

This code instantiates an array of three `C_Rectangle` objects and constructs each of the three with the default constructor `C_Rectangle()`. The pointer `aRects` is set to point to the first of these objects. `aRects[i]` represents the *i*-th object, where *i* is between 0 and 2.

Deinstantiating Dynamically Instantiated Objects

The operator `delete` must be used to deinstantiate any object (or array of non-objects) that has been dynamically instantiated using `new`. `delete` deallocates the storage that was allocated for the variable. The syntax is simply


```
delete pointer;
```

where *pointer* points to the object (or array of non-objects). As an example, the following pair of statements from Listing C-1d loops over the dynamically instantiated rectangle objects and deinstantiates each of the objects. *apRects[i]* holds the address of the *i*-th object.

```
for ( i = 0; i < nRect; ++)  
    delete apRects[i];
```

When deinstantiating an object, *delete* destructs it just before deallocating the storage.

Deinstantiating Dynamically Instantiated Arrays of Objects

A slightly different syntax is used to destantiate an array of objects. This syntax is

```
delete[] pointer;
```

Where *pointer* points to the array of objects. For example, from Listing C-1d:

```
delete[] aRects;
```

Main Program That Instantiates and Deinstantiates Objects Dynamically

Listing C-1d below is a main function that dynamically instantiates and deinstantiates objects of the class *C_Rectangle*. When added to Listings C-1a and C-1b, Listing C-1d forms a program. This program lets the user create several rectangles that contain attribute values that the user supplies. It then prints out the attribute values for each rectangle, moves each rectangle 20 units along each axis, and prints out the rectangle's attribute values again.

The user first indicates how many rectangles he wants to create. Then those rectangle objects are dynamically instantiated one at a time using the *new* operator in the statement,

```
apRects[i] = new C_Rectangle(nLeft, nTop, nWidth,  
                             nHeight, psName);
```

Values supplied by the user (in variables *nLeft*, *nTop*, *nWidth*, *nHeight*, and *psName*) are used as instantiation parameters. Since these parameters match the arguments of the constructor *C_Rectangle(int,int,int,int,char*)*, this constructor will be used to construct the instantiated object. The pointer to the instantiated object is assigned to an element of the array *apRects*. This array of pointers holds a pointer to each dynamically instantiated rectangle object.

Listing C-1d

```
//--Main function that dynamically instantiates and  
// deinstantiates objects  
#define MAX_RECTS 50  
main()  
{  
    // Create an array of three rectangles and print them  
    C_Rectangle* aRects = new C_Rectangle[3];  
    for ( int j = 0; j < 3; j++ )  
        aRects[j].print();  
    // Destruct the array  
    delete [] aRects;  
  
    // Create a set of rectangles specified by the user.  
    int nRect;
```

```

static C_Rectangle* apRects[MAX_RECTS];
printf ("\nHow many rectangles do you want");
scanf ("%d", &nRect);
if (nRect > MAX_RECTS) nRect = MAX_RECTS;

int i;
for ( i = 0; i < nRect; i++)
{
    int nLeft, nTop, nWidth, nHeight;
    char psName[64];
    // Get data for rectangle.
    printf ("\n");
    printf ("rectangle %d:  enter  Left  Top  Width  Height: ",
            i + 1);
    scanf ("%d%d%d%d", &nLeft, &nTop, &nWidth, &nHeight);
    // Make name that reflects rectangle's number
    sprintf (psName, "Rectangle %d", i + 1);
    // Dynamically instantiate a rectangle object
    apRects[i] = new C_Rectangle(nLeft, nTop, nWidth,
                                nHeight, psName);
}

// For each rectangle:
// Print rectangle, move it, then print moved rectangle.
for (i = 0; i < nRect; i++)
{
    printf ("\nOriginal  ");
    apRects[i]->print();
    apRects[i]->move(20, 20);
    printf ("Rectangle moved by 20,20:  New");
    apRects[i]->print();
}

// Destruct each rectangle
for ( i = 0; i < nRect; i++)
    delete apRects[i];
}

```

Inheritance

Inheritance is used to define a new class (called the *derived class*) that inherits the attributes and functionality of another class (called the *base class*), and defines additional attributes and/or functionality. A derived class is also called a *subclass*, while the base class is also called a *superclass*. The terms subclass and superclass are more general in that the base class of a base class is also called a super class. And the derived class of a derived class is also called a subclass.

Note: In C++, the terms “parent” and “child” are sometimes used for base and derived classes, but we will avoid this usage because parent and child have an entirely unrelated meaning in Windows. In this book, the use of the words “parent” and “child” as well as the words “ancestor” and “descendant” are reserved to denote the Windows meaning.

To specify that class ***derivedclass*** inherits class ***baseclass***, the name of the base class ***baseclass*** is specified in the header of the class definition of the derived class: A base class can be inherited privately or publicly. If the base class is inherited privately, then its data members and member functions cannot be accessed by code that is outside of the subclass hierarchy. (What is meant by “code outside the subclass hierarchy” is code that is not in a member function of the derived class or of a subclass of the derived class.) The following syntax specifies a private inheritance.

```

derivedclass : baseclass
{
    .... Declarations and/or definitions of members
}

```

If the base class is inherited publicly, then its public data members and member functions are accessible by code that is outside of the subclass hierarchy. Public inheritance is more useful and more commonly used than private inheritance. Public inheritance is used throughout this book. To specify public inheritance, you must use the following syntax:

```

derivedclass : public baseclass
{
    .... Declarations and/or definitions of members
}

```

Inheritance is a formalized method of reusing executable code. If one class is inherited from another, then the inheriting class (the derived class) inherits (shares) the functions of the class being derived from (the base class). A derived class effectively contains the data members and member functions of the base class. However, the rules governing access to these members are complex. These rules are described in the following paragraphs.

Access By Code in a Member Function of the Derived Class

Access to a Data Member of the Base Class

The storage for an object of a derived class includes storage for the data members of its base class. Thus an object of the derived class contains the data represented by the data members of the base class.

Protected and public data members in the base class are accessible by code in the derived class.

Private data members in the base class are not accessible to the code of member functions in the derived class. Although they are not accessible to the code of derived-class member functions, these data members are still present in objects of the derived class. These base-class data members are accessible as usual to member functions of the base class.

Access to a Member Function of the Base Class

A private member function of the base class is not accessible to code in the member functions of the derived class. Protected and public member functions in the base class are accessible.

The member functions of the base class are considered to be member functions of the derived class, with one exception. If you define a member function in the derived class with the same name and signature as a base class member function, then the one defined in the derived class is used when an object of the derived class invokes the member function. In this instance, the member function in the derived class *overrides* the member function in the base class.

Overriding Functions

When there is a subclass hierarchy, and there is a member function that has one or more overriding versions in different classes in the hierarchy, then when that member function is invoked for an object, the compiler chooses the version that is defined in the object's class. If the object's class does not have a version of the member function, then the compiler looks in the base class of the object's class, then in the base class's base class, etc. until it finds a version of the desired member function.

If the object is represented by a pointer, then the same process occurs. A pointer to an object is always defined to be a pointer to objects of a particular class. The compiler starts looking for the desired member function at the class to which the pointer is defined to point.

There are times when you might want a pointer to a class to actually point to an object of a derived class or subclass. If you invoke a member function for one of the pointer elements, the wrong version of the member function could be called. After all, the compiler does not know the class of the object that the

element points to at execution time; it only knows what class the pointer element is defined to point to at compile time.

The following program illustrates the use of an overriding function not using the “virtual” keyword in the base class member function.

```
#include <stdio.h>
class C_Fourth {
    public:
        void print() { printf (". . . .A. . . .\n"); }
};
// Class C_Override is derived from class C_Fourth
class C_Override : public C_Fourth {
    public:
        void print() { printf (". . . .B. . . .\n"); }
};
void main()
{
    C_Fourth* aObjects[2];
    aObjects[0] = new C_Fourth;
    aObjects[1] = new C_Override;
    aObjects[0]->print();
    aObjects[1]->print();
}
```

This program displays the following lines:

```
. . . .A. . . .
. . . .A. . . .
```

Since aObjects[1] points to an object of class C_Override, we would have liked the last statement to display the line “....B.....” The reason it does not is that the compiler generates a call to C_Fourth::print() rather than C_Override::print(). The compiler does so because aObjects is defined to be an array of pointers to class C_Fourth.

This problem is a common one in object-oriented programming in general. C++ provides a mechanism that solves it. This mechanism is called *virtual functions* which is described next.

Virtual Functions

The virtual functions mechanism ensures that the proper member function is called when a member function is invoked for a pointer to an object, and the object pointed to is of a subclass of the class that the pointer is defined to point to. The virtual function mechanism is applied to a member function by prepending the keyword “virtual” to the function’s declaration (or definition) in the class definition block of the most superclass class that the member function is declared or defined in. All of the member functions of the same name in the subclasses (the overriding member functions) are thus considered to be virtual.

When a virtual function is invoked for a pointer to an object, the compiler generates code that checks the class of the actual object pointed to at execution time and calls the correct member function for that class. It is that simple. You should use virtual functions whenever you have overriding functions, and these may be invoked by pointer.

There are just a few disadvantages to using virtual functions, and these disadvantages are usually slight: two to four bytes are added to the size of the object of any class that has one or more virtual functions (this holds a virtual table pointer); space for a virtual table is required, typically four bytes per entry, where the number of entries is the number of virtual functions times the number of classes with virtual functions in the subclass hierarchy; it takes a few extra instruction executions to invoke a virtual function, but probably less than 20.

The following program illustrates the use of an overriding function using the “virtual” keyword in the base

class member function.

```
#include <stdio.h>
class C_Fifth {
public:
    virtual void print() { printf ("....A....\n"); }
};
// Class C_Override is derived from class C_Fifth
class C_Override : public C_Fifth {
public:
    void print() { printf ("....B....\n"); }
};

void main()
{
    C_Fifth* aObjects[2];
    aObjects[0] = new C_Fifth;
    aObjects[1] = new C_Override;
    aObjects[0]->print();
    aObjects[1]->print();
}
```

This program displays the following lines:

```
....A....
....B....
```

Virtual Destructors

The problem that virtual member functions solve for ordinary member functions also exists for destructors of dynamically instantiated objects. To avoid this problem, you should declare the destructor of a derived class virtual if the class has any virtual member functions. (If you have not defined a destructor for the class, you should define a minimal one and declare it virtual.)

Example 2

Example 2, presented in Listing C-2, defines two classes, C_Box and C_NamedBox. Objects of class C_Box are simple rectangles. Class C_NamedBox is derived from class C_Box. Objects of class C_NamedBox are rectangles that also have a name.

In this example, the concept of *access functions* is introduced. The data members of C_Box and C_NamedBox are declared private. Private data members can only be accessed by member functions of their own class or derived class. Access functions can be defined which can “set” or “get” the values of the data members. In this example, access functions are defined to “set” the data members.

Writing access functions might seem like a lot of needless work. However, there is an important advantage to using private data members and their associated access functions. Access functions ensure that your objects never contain invalid values. By using member functions to control access to private data, you hide the representation of your class. Access functions let you change the implementation of a class without affecting any of the programs that use it. This convention is known as *encapsulation*, which is one of the most important principles of object-oriented programming.

C_Box objects have the attributes of position and size. The values of these attributes are stored in data members m_nLeft, m_nTop, m_nWidth, and m_nHeight. C_Box has a member function print() that prints the attributes.

C_NamedBox objects have, by inheritance, the same data members and member functions that C_Box objects have, and they have an additional data member, which is a name string. The data member m_psBoxname points to this string. Like objects of class C_Box, objects of class C_NamedBox also can print their attributes. Since they have a name string, they must be printed differently from objects of the

base class C_Box. To do this, an overriding print() function is provided.

Listing C-2

```
#include <stdio.h>
#include <string.h>
//-----Class definition for C_Box-----
class C_Box
{
    public:
        C_Box(int nLeft, int nTop, int nWidth, int nHeight);
        virtual void print();          // virtual function declaration
        void printKeywordThis(){ printf("Object is at:  %p\n\n", this ); }
// Definitions of access functions
        void setLeft(int nLeft) { m_nLeft = nLeft ; }
        void setTop(int nTop) { m_nTop = nTop ; }
        void setHeight(int nHeight) { m_nHeight = nHeight ; }
        void setWidth(int nWidth) { m_nWidth = nWidth ; }
    private:
        int m_nLeft, m_nTop;
        int m_nWidth, m_nHeight;
};

//-----Class definition for C_NamedBox-----

class C_NamedBox : public C_Box
{
    public:
        C_NamedBox(char* psBoxname, int nLeft, int nTop,
                    int nWidth, int nHeight);
        virtual ~C_NamedBox() { delete m_psBoxname; }
        void setBoxname(char* psBoxname); // Access function declaration
        void print();          // overriding function declaration
    private:
        char* m_psBoxname;
};

//-----Member function definitions for class C_Box-----

C_Box::C_Box(int nLeft, int nTop, int nWidth, int nHeight)
{
    m_nLeft = nLeft; m_nTop = nTop;
    m_nWidth = nWidth; m_nHeight = nHeight;
    if (m_nWidth < 0) m_nWidth = 0;
    if (m_nHeight < 0) m_nHeight = 0;
}

void C_Box::print()    // Prints values of object's data members
{
    printf("Box (%d,%d) to (%d,%d)\n", m_nLeft, m_nTop,
          m_nLeft + m_nWidth, m_nTop + m_nHeight);
}

//-----Member function definitions for class C_NamedBox-----

C_NamedBox::C_NamedBox(char* psName, int nLeft, int nTop, int nWidth,
                        int nHeight) : C_Box( nLeft, nTop, nWidth, nHeight)
{

```

```

        m_psBoxname = new char[strlen(psBoxname) + 1];
        strcpy( m_psBoxname, psBoxname);
    }

void C_NamedBox::print()    // overriding function definition
{
    printf("  box name = %s\n", m_psBoxname);
    C_Box::print();
}

void C_NamedBox::setBoxname (char* psBoxname)
{
    delete m_psBoxname;
    m_psBoxname = new char[strlen(psBoxname) + 1];
    strcpy( m_psBoxname, psBoxname);
}

//-----main function-----

void main()
{
    // Instantiate two objects:  a box and a namedbox
    C_Box box( 10, 10, 100, 40 );
    C_NamedBox namedbox( "A NAMED BOX", 40, 60, 400, 100 );

    // Print the objects and their location
    box.print();
    box.printKeywordThis();
    namedbox.print();
    namedbox.printKeywordThis();

    // Use access functions to change attributes of the objects
    box.setLeft(50);
    box.setTop(80);
    box.setHeight(200);
    box.setWidth(500);

    namedbox.setLeft(100);
    namedbox.setTop(150);
    namedbox.setHeight(200);
    namedbox.setWidth(300);
    namedbox.setBoxname("A NEW NAME FOR BOX");

    // Print the objects
    box.print();
    namedbox.print();

    C_Box* apBox[2];
    apBox[0] = &box;
    apBox[1] = &namedbox;

    // Print the objects using the pointer values apBox[]
    printf("\n\n");
    for (int i = 0; i < 2; i++)
        apBox[i]->print();
}

```

There are four access functions in class C_Box. These are setLeft(), setTop(), setHeight(), and setWidth(),

which can reset the values of the private data members m_nLeft, m_nTop, m_nHeight, and m_nWidth, respectively. They provide access to change the values of the private data members from anywhere in the program. There is one access function in class C_NamedBox which is setBoxname().

Constructors of Derived Classes

The constructor of a derived class is responsible for initializing all of the data members of an object of the derived class. It is responsible for initializing the data members inherited from the base class. As you would expect, it uses the constructor of the base class to initialize the data members that are inherited from the base class, and explicitly initializes its own additional data members.

The derived class constructor therefore must call the base class constructor. It does this by using a *base class initializer*. The base class initializer is placed in an *initialization list* for the constructor. The base class initializer consists of the name of the base class followed by a parentheses-enclosed, comma-separated list of initialization parameters. The base class constructor is called before the body of the derived class constructor executes. The compiler chooses that base class constructor whose signature matches the parameters supplied in the base class initializer.

The following code is the header to the definition of the constructor C_NamedBox (char* , int, int, int, int). First in the list are the arguments to initialize the derived class data members followed by the arguments to initialize the base class data members. The base class initialization list follows. The initializer list consists of a colon followed by the base class name and its initializers separated by commas.

```
C_NamedBox::C_NamedBox (char* psBoxname, int nLeft, int nTop,  
                        int nWidth, int nHeight) : C_Box( nLeft, nTop, nWidth
```


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

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

[Table of Contents](#)[BROWSE
BY TOPIC](#)

Appendix D The Microsoft Visual C++ 4 Compiler

Overview of Compiler Tools

To use compiler tools, you must first set up a project which contains all of your source code. In this book, two types of projects are covered. These are the Application and the AppWizard projects.

The Application project is what we encountered in Chapter One of this book. It is, basically, a project in which you are responsible for the generation of all of your source code files. You will type in each of your source code files, or import them from other directories.

AppWizard, which we work with beginning in Chapter Ten, will automatically generate a project for you and it will also generate starter source code. The AppWizard starter application is a four-class application; it will have an application class, a mainframe class, a document class, and a view class.

Once a project has been generated, you can use AppStudio or ClassWizard to help you generate your code. AppStudio generates resources, which are placed into the resource file (the .rc file), and it also generates the resource.h file. ClassWizard adds handler functions to your code, and it also creates new classes, which is a capability that we use when our application has dialog boxes.

Once you have generated your project and all your source files are in place, the method of generating your project (i.e., an Application or an AppWizard project) is irrelevant to the AppStudio and ClassWizard tools. AppStudio and ClassWizard are simply performing their function on source code; how that source code was generated does not matter to these tools. It is true, however,

☒ Brief ☐ Full

[Advanced Search](#)

[Search Tips](#)

that the source code that they are called to operate on must meet certain specifications.

If you are using an AppWizard-generated starter application, you do not have to worry about your source code meeting these specification because everything is placed there automatically.

If you are generating your own code, you must be aware of what these specifications are and provide for them. If you generate your own source code and then call on the services of ClassWizard to add handler functions for you, you must have specific comment lines in your source code—this is covered in Chapter Two of this book. AppStudio, which generates resources, is easier to satisfy; it only requires that you start generating your resource files with AppStudio and continue to do so, not removing any of the comments that it places in the resource files, which allow it to re-enter these files and continue its work.

The AppStudio and ClassWizard discussions given in this appendix apply to any established project. That is, the AppStudio discussion can be used for both an Application project, in which you generated all your own source code, and also for an AppWizard application, in which the starter source code was generated for you. Likewise, the ClassWizard discussion can be used for either an Application project or for an AppWizard-generated project.

Creating an Application Project

To create an Application project:

1. Choose “File | New” from the Microsoft Developer Studio menu.
 - a. The “New” query box shown in Figure D-1 opens up. Choose “Project Workspace.”

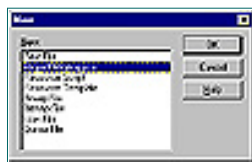


Figure D-1: The “New” Query Box

2. The “New Project Workspace” query box shown in Figure D-2 opens up. In it choose the location (directory and path) that you wish your project to be installed in and enter the project’s name.

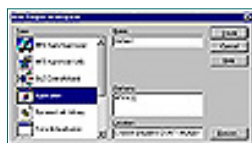


Figure D-2: The New Project Workspace Query Box

- a. A directory of the same name as the project will be created and your project will be located in that directory. Then choose “Create.”

The project workspace has been created as shown in Figure D-3, but the project has no source files. We will have to type them in (or import them from

other directories), and we need to change the project settings to suit our situation.



Figure D-3: The Project Workspace

First, we will change the project settings:

1. In the drop-down list in the toolbar of the Project Workspace, select “Release.”
2. From the “Project Workspace” menu, choose “Build | Setting” and the “Project Settings” query box shown in Figure D-4 opens, in which you further tailor the compiler settings that you want.

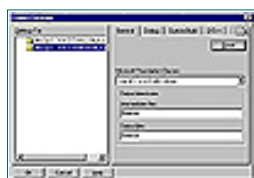


Figure D-4: “Project Settings” Query Box

- Make the choices as shown on the “General” tab (the rest of the default compiler settings are OK).
- Click OK to get these settings.

Next, we will type in the source code files and add them to the project:

1. Choose “File | New” from the “Project Workspace” menu bar.
2. Choose “Text File” on the “New” query box as shown in Figure D-5.



Figure D-5: Opening a Text File

- c. A window opens up in which you type one of your source code files. We use the first example in the book, which is “MyApp1” from Chapter One and we will type in the source code files. This is shown in Figure D-6.



Figure D-6: Entering a Source Code File

- d.** Choose “File | Save As” and save your file as MyApp.h in the directory <MyApp1> as shown in Figure D-7.



Figure D-7: Saving the Source Code File

- e. Save the file and then close it.
3. Continue typing in source files.
 - a. Again choose “File | New | Text File” to enter the next file.
 - b. Repeat this process until you have typed in and saved all of the source files, which in this example are: MyApp.h, MyApp.cpp, Mainfrm.h, and Mainfrm.cpp.
4. Add all of your .cpp files to the project.
 - a. From the “Project Workspace” menu choose “Insert | Files into Project,” and you get the “Insert Files into Project” display shown in Figure D-8.



Figure D-8: Adding Files to the Project

- b. Select source files of type .cpp so only the .cpp files appear and then highlight (select) the files that are shown as .cpp files. The highlighted files will appear in the “File name” edit box.
 - c. Click on “Add” and they will be added to your project.

Note: All the .cpp files as well as your .rc file must be added to your project. We do not have a .rc file in this example, so we only add the .cpp files. A project can have only .cpp files, as the application MyApp1 of Chapter One does. When we have a project for which we must generate a .rc file, we must then remember to add the .rc file to the project. If an application has a .rc file, it must be added to the project.

Now we will look at the files in our project, which are shown in Figure D-9. We see that the .cpp files have been added and the dependencies (their .h files) also are shown on our project files list.



Figure D-9: The Project Files List

Adding Resources

We do not have a resource script file yet in our project, so we will look at how to add a resource for the first time. For purposes of illustration in this

appendix, we will add a resource to the “MyApp1” application, although in the “MyApp1” application in Chapter One, we did not use a resource.

1. From the “Project Workspace” menu, select “Insert | Resource” and the “Insert Resource” query box shown in Figure D-10 opens up.



Figure D-10: Creating a New Resource

Note: Anytime you are opening a new type of resource you will use this same procedure.

- a. Select “Icon,” and the editor opens up on which you will create an icon.
- b. When you are finished select “File | Save.” (It will be saved as Script1.rc unless you choose a different file name.)
- c. Save the file and close the editor by choosing “File | Close” from the “Project Workspace” menu.

The Script1.rc file we just created does not show on our project files list. This is because when a .rc file is first created, it needs to be added to our project.

2. From the “Project Workspace” menu, choose “Insert | Files” into “Project” and add Script1.rc to the project. This is shown in Figure D-11. A resource.h file has also been generated, but this file does not show in the project list.

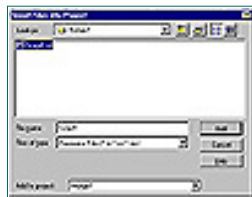


Figure D-11: Adding the Resource Script File to the Project

- a. If you want to open and edit the resource.h file, you must open it by choosing “File | Open” from the “Project Workspace” menu.
- b. As discussed in Chapters One and Two, we sometimes need to open the resource.h file and edit it. If you need to open the Script1.rc file for editing, go to the DOS prompt, and open and edit it from there.

The project files list now appears as shown in Figure D-12 with the .rc file listed, and it also shows the file name of the icon that we just generated.



Figure D-12: The Project Files

You can open the .rc file to edit its resources by double-clicking on the Script1.rc file name. The opened resource file looks as shown in Figure D-13. If we were to double-click on the name of the icon, IDI_ICON1, it would open the editor with that icon in it so that we could edit it.



Figure D-13: The Opened Resource Script1 File

Adding ClassWizard

To use ClassWizard for the first time, you must add special comment lines into your code. These are explained in Chapter Two. Now we will add the special comment lines into the mainframe class in our “MyApp1” example. In the Mainfrm.h file, the lines that ClassWizard needs to see are given below and are within the C_MainFrame declaration block:

```
//{{AFX_MSG(C_MainFrame)
//}}AFX_MSG
DECLARE_MESSAGE_MAP( )
```

In the file Mainfrm.cpp, the lines that ClassWizard needs to see are given below and are outside of any block of code:

```
BEGIN_MESSAGE_MAP(C_MainFrame, CFrameWnd)
//{{AFX_MSG_MAP(C_MainFrame)
//}}AFX_MSG_MAP
END_MESSAGE_MAP( )
```

1. To open ClassWizard, select “View | ClassWizard” from the “Project Workspace” menu.

Note: If you are building your own source files as we have done here and this is the first time you have asked for ClassWizard, it needs to build its file, known as .clw, so you get the query box shown in Figure D-14. This query box appears whenever you do not have a .clw file, which could also happen if you have deleted the file to save space on your disk. You can delete the .clw file and rebuild it whenever you need to use it.



Figure D-14: The ClassWizard Database Query

- a. Choose Yes and the “Select Source Files” query box shown in Figure D-15 opens up.



Figure D-15: The ClassWizard Source Files Query Box

- b. Simply choose OK on this query box, it will be filled in OK.

The .clw file is then created, and the next thing that happens is that the ClassWizard box as shown in Figure D-16 opens up. (Refer to the sections of this appendix on ClassWizard for how to use this tool.)

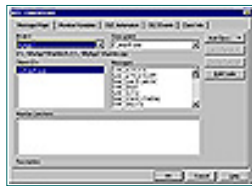


Figure D-16: ClassWizard

Creating an AppWizard Starter Application

The AppWizard will generate a four-class starter application for you. The four classes are the application class, the main frame class, the document class, and the view class. The AppWizard setup allows you to make choices on the type of application that you want, e.g., SDI vs MDI, etc. Once these choices are made, AppWizard generates your starter code and creates your project. The starter application code can be compiled by simply choosing “Build | Build Elip.exe” from the “Project Workspace” menu. Everything else has been done for you.

Note: If you discover that you have made the wrong setup choices after your starter application has been generated, the best thing to do is to start all over. AppWizard does not provide for changes and deletions in the code. If you are a very experienced programmer you may be able to enter the code and fix it, but this approach is not recommended for the novice.

This section shows the steps to generate the AppWizard starter application for the first example in the book, the “Elip” program in Chapter Ten.

1. Beginning from the Microsoft Developer Studio, choose “File | New” and the “New” query box shown in Figure D-17 opens up.



Figure D-17: Creating a New Project

2. Choose “Project Workspace” and you will then get the “New Project Workspace” query box.

3. In the “New Project Workspace,” choose the type MFC AppWizard(exe) as shown in Figure D-18.

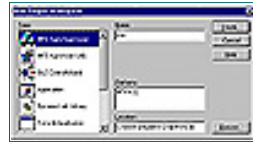


Figure D-18: Creating a New AppWizard Project

4. Fill in the project’s name and the location of your project. AppWizard will create a subdirectory at your chosen location. The subdirectory will have the same name as your project and all your files will be placed in that subdirectory.

5. When you have done this, choose “Create” and you will get the first in a series of six query boxes in which you will make choices about your application.

a. Step 1 gives the choices, as shown in Figure D-19. You note that the default choice is “Multiple documents,” which is not what we want. We want “Single document,” so change the query box to look as shown in Figure D-19. Then choose “Next” to go to the next step.

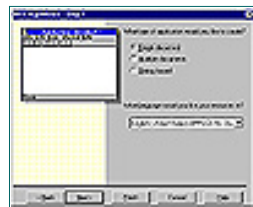


Figure D-19:

b. Figure D-20 shows the Step 2 query box defaults, which are OK as is.

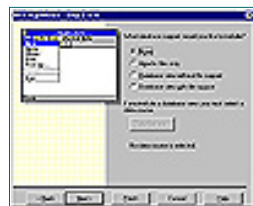


Figure D-20: Selections for Step 2 of the AppWizard Project

c. Figure D-21 gives the Step 3 query box, for which the defaults are OK as is.

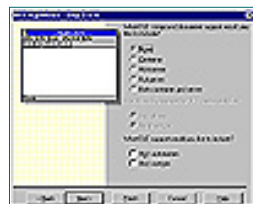


Figure D-21: Selections for Step 3 for the AppWizard Project

d. The Step 4 query box allows us to choose the application’s features that we want. You only want printing and print preview. Figure D-22 shows how Step 4 should look when you are finished.

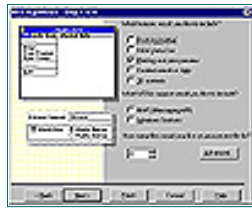


Figure D-22: Selections for Step 4 of the AppWizard Project

- (1) Choose the “Advanced” option if you want to make choices on the file extension and splitter windows.

When you choose the “Advanced” option in Step 4 you get the following query box shown in Figure D-23 with the tabs “Document Template Strings” and “Window Styles.” When you select the “Document Template Strings” tab, you can make the choice of the file extension that you want your files to be saved with. The rest of the string is filled in for you.



Figure D-23: Document Template String Options for the AppWizard Project

When you choose the “Window Styles” tab of the “Advanced Options” query box, you can choose to have the code for the splitter window automatically added and you can choose the style of your main frame window. The default options are shown in Figure D-24.



Figure D-24: Window Style Options for the AppWizard Project

- e. For the Step 5 choices in this example, we do not want source comments. Choose the statically linked library. Figure D-25 shows the Step 5 selections.



Figure D-25: Selections for Step 5 for the AppWizard Project

f. The last step shows you the names of the classes and their files that will be created. Change all of the derived class names to begin with C_ rather than C, so that you can distinguish them. Do not change any of the file names.

- (1) You have a choice as to what base class you wish to derive your view class from. When you put the focus on the view class as shown in Figure D-26, a drop-down box becomes available, and you can choose which base class to derive your view class from.
- (2) Choose Finish from this query box.

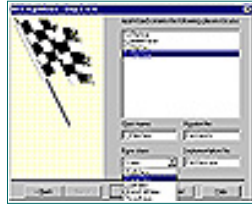


Figure D-26: Step 6 Selections for the AppWizard Project

6. Now you get a final display of “New Project Information” shown in Figure D-27. Choose OK and your starter application source code files will be generated.



Figure D-27: Display of the AppWizard Starter Application Information

7. At this point you can compile and run and see what your starter application looks like.

Note: Before you compile, you need to make sure that the project settings are what you want. Be sure to choose the “Release” setting from the drop-down box on the toolbar of the “Project Workspace.”

a. Choose “Project | Settings” from the “Project Workspace” menu and select the items on the display given in Figure D-28. We choose “Release” to speed the build. The remaining default project settings are OK.

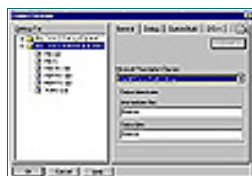


Figure D-28: Selecting Project Settings

Once you have compiled your AppWizard starter application, take a look at

the files that you have in your project. These are shown in Figure D-29. AppWizard has generated an initial resource script file for you. It has also generated icons to be used for the main window (Elip.ico) and for MDI documents (ElipDoc.ico). Even though this is an SDI application, we still get the MDI document icon in our files. Also, an .rc2 file has been generated where you may keep resources that you do not want AppStudio to look at.

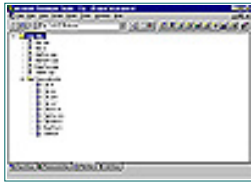


Figure D-29: List of Project Files for the AppWizard Starter Project

The program is also ready to use ClassWizard; a ClassWizard database has been generated and placed in a file named Elip.clw. This file does not show on the project list of files, shown in the previous figure. To view the Elip.clw file, you need to go to the Windows Explorer.

At this point, your starter application is ready to tailor to your needs. This includes adding code to the four classes, which is covered in Chapters Ten through Fourteen. You will also be adding resources to your application. Your starter resource script file has been included with your basic application. As you have noted, at this point, you simply have a project established and all the other tools operate as before on an established project. The other tools are ClassWizard and AppStudio.

When you operate ClassWizard, it does not care how you generated your project and your source code; it operates the same on any established project. The same is true of AppStudio. It does not care how you generated your project and your source code; it only cares about whether your resource files were generated by AppStudio, so that it can enter and continue the process. To work with ClassWizard and AppStudio, refer to the sections of this appendix that cover these tools.

Creating Resources Using AppStudio

In Visual C++4, AppStudio has been integrated into the Project Workspace. The .rc file appears as a project file. When you open it up by double-clicking on it, it gives you a list of all the resources in the file. As explained in Chapter One, there can be a lot of different types of resources, all of which can be in your resource file. You use AppStudio to add new resources and to edit an existing resource.

1. To add a new resource not currently in your resource file, select “Insert | Resource” from the “Project Workspace” menu. The “Insert Resource” query box appears (refer to Figure D-10).
 - a. Refer to each of the following chapters for information on how to use the resources.
 - (1) In Chapter Two adding accelerators is looked at.
 - (2) The cursors, bitmap, and icon resources all come with an image editor that allows you to draw these resources and

then save them. We discuss how to use these image resources in Chapters One and Four.

(3) Adding menus is covered in Chapter Two and in Chapter Ten.

(4) Using String Tables is covered in Chapter Seven and in Chapter Eleven.

(5) The Dialog Editor which is used to generate your dialog boxes is more complex, and we have covered it in Chapters Six and Seven.

(6) The Toolbar resource is covered in Chapter Thirteen.

AppStudio is also used to edit existing resources. Figure D-30 shows the resource files for the Chapter Ten “Elip” program, most of which were generated by AppWizard. This figure serves to illustrate that there can be a large number of resources in a resource file, and that your program does not need to use them all. It also illustrates that there are many different kinds of resources in the resource file and that you may have multiple resources of a given kind; for example we have two dialog boxes.



Figure D-30: AppWizard Generated Resources for the Elip Program

2. To edit an existing resource, simply double-click on that resource, and the AppStudio Editor will open up.

a. Do your editing, save the file, and close AppStudio.

Note: AppStudio does not care whether you originally generated your source code as an Application project or an AppWizard project. It operates the same for any source code.

Using ClassWizard

ClassWizard is used to perform two basic tasks. The first is to add handler functions to your code. This is covered in Chapter Two for an Application project. However, ClassWizard operates exactly the same when used on an application that was started as an AppWizard project. The Chapter Two discussion applies to all developed projects.

The second basic task that ClassWizard is used for is to add a new class. This is important when we are designing and using dialog boxes. Chapters Six and Seven cover how to generate a dialog box resource using AppStudio and then how to wrap it with a new class generated by ClassWizard. Refer to these chapters. Like AppStudio, the operation of ClassWizard is the same for any established project.

Table of Contents

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

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

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



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

Go!

[Table of Contents](#)

BROWSE
BY TOPIC

Appendix E The Microsoft Visual C++ 1.5 Compiler

Overview of Compiler Tools

In this appendix, compiler tools that are used for the examples in this book are discussed. The first task is to set up the project. This setup of your project is shown manually (without using AppWizard), which is how the examples in Chapters One through Nine are done. Then setting up your project using AppWizard is shown; AppWizard not only sets up your project but provides you with automatically generated starter source code, which has four classes: the application class, the mainframe class, the document class, and the view class. The examples in Chapters Ten through Fourteen are initially started using AppWizard.

In addition to setting up the project, there are other tools available. AppStudio manages the resource files. ClassWizard adds handler functions and new classes. These tools, AppStudio and ClassWizard, are used after your project is established and you are developing your code. These tools, AppStudio and ClassWizard, do not care how you set up your project, whether you did it manually or used AppWizard. They simply operate on the source code files.

In this appendix, setting up your project is first discussed. Then a discussion of the AppStudio and ClassWizard tools is presented.

Creating a Project for Your Own Code

To create a new project *without* using AppWizard:

1. Create the subdirectory <myapp1> in your chosen directory.
2. From the Visual Workbench select “File | New” and a window opens

up labeled “UNTITLED,” in which you type your source code.

- a. Type in your file myapp.h. Figure E-1 shows the Visual Workbench with the file myapp.h entered as a text file.



Figure E-1: Entering Your Project Files

3. Then from the Visual Workbench choose “File | Save As” and save the file as myapp.h in your <myapp1> directory as shown in Figure E-2.

Note: In Win3.1, your file names will consist of lowercase letters and can have no more than eight letters.

4. Close the window.
5. Repeat these steps, typing in and saving the remaining project files myapp.cpp, mainfrm.h, and mainfrm.cpp. (You could import files into your directory <myapp1> if you already have them in another directory.)



Figure E-2: Saving Your Source Code File

6. Go to the Visual Workbench and select “Project | New” and the “New Project” query box appears.
 - a. Type in the full path name and the project name myapp1.mak as shown in Figure E-3.
 - b. Select OK.



Figure E-3: Creating a New Project

- c. The project’s “Edit” query box shown in Figure E-4 will come up next. Add your .cpp files to your project.
 - d. Click “Close.”



Figure E-4: Adding Your Files to Your Project

7. To add/delete files to your project, choose “Project | Edit” from the Visual Workbench menu which opens the project’s “Edit” query box.
 - a. Add the following files to your project: all your .cpp files, the .def file, and the .rc file.

Note: These files must be added. In this example, there is no .rc file, but if you have one it must be added to your project. The .def file will be automatically added to your project by the compiler before a build can begin or you can create your own .def file if you wish.

8. Customize the project settings.

Note: Before you compile you need to look at the project settings to see if they are what you want. In order to minimize the number of files that the compiler will automatically generate for you, you need to customize your project settings.

- a. From the Visual Workbench, choose “Options | Project” and you get the “Project Options” query box shown in Figure E-5.
- b. Select the “Release” build mode since we will not need to debug this small example. (If you compile in the debug mode the compiler generates a lot of auxiliary files.)
- c. Choose the “Compiler” button to further customize the build options.



Figure E-5: Project Options Query

- d. When the “C/C++ Compiler Options” query box shown in Figure E-6 opens up, select the category “Listing Files” and uncheck the “Browser Information” checkbox. (If you want browser information the compiler generates additional auxiliary files.) Also, make sure that the “Precompiled Headers” | “Automatic Use of Precompiled Headers” checkbox is unchecked. (If you generate a precompiler header file, it takes up a lot of memory.)
- e. Once you have chosen the settings you want, click on OK.



Figure E-6: Customizing the Compiler Settings

9. When you are done customizing, compile your program.

- a. Your project is open and you are ready to compile your program; from the Visual Workbench, select “Project | Build MYAPP1.EXE.” Figure E-7 shows the Visual Workbench with the myapp1.mak project open and with “Project | Build MYAPP1.EXE” selected.



Figure E-7: Building the Project

- b.** The compiler will at this point generate a .def file for you if you have not generated your own, so next you will get the display shown in Figure E-8. Choose Yes on this query.



Figure E-8: Query for .def File

- c.** Next a window containing your .def file opens which gives you the opportunity to edit the .def file. Close the window—you do not need to edit the .def file, but you may do so if you choose. The compiler automatically adds the .def file to your project.

Your project is now compiled. To view the files in your project file at this point, click the leftmost toolbar button and a small window with the list of files in your project opens up, which is shown in Figure E-9. If you double-click on any file in this project list, that file will open up for editing.



Figure E-9: List of Project MYAPP1 Files

After your project is compiled, use Program Manager and view the files in your directory <myapp1>, which will be

```
mainfrm.cpp
mainfrm.h
mainfrm.obj
myapp.cpp
myapp.h
myapp.obj
myapp1.def
myapp1.exe
myapp1.mak
```

myapp1.vcw	a project status file; here it is in its entirety: [MSVC Status File] Version=1.00 ProjectType=0 External=0 BrkptCount=0 WatchCount=0
myapp1.wsp	workspace file; stores the windows and the positions of the windows inside the Visual Workbench, fonts selected, etc. (Up to three workspace files may be maintained for a project.)

Although there are a lot of files generated by the compiler for these choices, there are additional files that the compiler generates for other compiler choices.

If you choose the option “Automatic Use of Precompiled Headers,” the compiler will generate a .pch file for your application (if we had chosen this option for the above case, we would get a myapp1.pch file). On each subsequent build of your application, the .pch file will be used in lieu of recompiling the header file each build. This saves a lot of compile time and is very useful to speed up the debug phase of your application’s development. The header files are very large (the source code for the windows.h file alone is over 30 pages in length). The .pch file consumes at least 1 megabyte of memory. When you are doing many small programs, you will generally not want to use the memory required for the .pch files.

If you choose the option “Browser Information,” the compiler will generate more files. An .sbr (source browser file) will be generated for each .cpp file. The .sbr file is used by the browser in the Visual Workbench. The compiler will also generate a .bsc file, a project-wide browser database which is produced from the individual .sbr files.

If you compile in the Debug mode, the compiler generates a .pdb file (project database); information about the debug options is centralized here. In previous compiler versions, debug type information was attached to the individual object files.

If you have resource data, you have a .rc file (resource script) which is input to the resource compiler. It contains the menus, dialog boxes, accelerators, icons, bitmaps, etc., of your application. The compiler will generate a .res file for you; the information in this file is bound to the .exe file during the final phase of a project build. When you use AppStudio to generate resources, the compiler generates an .aps file (AppStudio data file). The resource data files, such as .bmp, .ico, and .cur, are stored in the current working directory.

If you use ClassWizard, the compiler generates a .clw file (ClassWizard data file). This file can always be rebuilt by calling ClassWizard from within AppStudio.

Creating a New Resource

1. From the Visual Workbench, choose “Tools | AppStudio” and AppStudio opens up. As shown in Figure E-10, there are no resources yet in our project, so everything is blank.



2. Select the “New” button and we get the “New Resource” query box shown in Figure E-11.



- a.** Choose “Cursor.”
- b.** Draw your cursor when an image editor opens up.
- c.** Save it as the file name suggested by AppStudio, which is app1.rc.
- d.** Close the image editor and close AppStudio.

3. If you wish to edit the .rc file, double-click on app1.rc. If you wish to edit the file cursor1.cur, double-click on it; the image editor will open up with cursor1 in it ready for editing.



Adding ClassWizard

To use ClassWizard for the first time, you must add special comment lines into your code. These are explained in Chapter Two. Let's now add the special comment lines into the mainframe class in our "Myapp1" example. In the mainfrm.h file, the lines that ClassWizard needs to see are given in the following code and are within the C_MainFrame class declaration:

```
//{{AFX_MSG(C_MainFrame)
//}}AFX_MSG
DECLARE_MESSAGE_MAP( )
```

In the file mainfrm.cpp, the lines that ClassWizard needs to see are given below and are outside of any block of code:

```
BEGIN_MESSAGE_MAP(C_MainFrame, CFrameWnd)
//{{AFX_MSG_MAP(C_MainFrame)
//}}AFX_MSG_MAP
END_MESSAGE_MAP( )
```

To open ClassWizard for the first time:

1. Open AppStudio by selecting "Tools | AppStudio."
2. Open ClassWizard from within AppStudio by selecting "Resource | ClassWizard" from the AppStudio menu.

Note: If you are building your own source files, as we have done here, and this is the first time you have asked for ClassWizard, it needs to build its file, known as .clw, so you get the query box shown in Figure E-13. This query box appears whenever you do not have a .clw file, which could also happen if you have deleted the file to save space on your disk. You can delete the .clw file and rebuild it whenever you need to use it.



Figure E-13: The ClassWizard Database Query.

3. Select "Yes" and the "Select Source Files" query box shown in Figure E-14 opens up. This query box is filled in fine, so simply select "Yes" again.

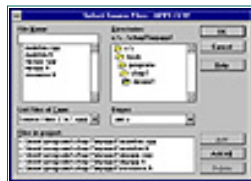


Figure E-14: The ClassWizard Select Source Files Query Box

4. The .clw file is created, and the next thing that happens is that ClassWizard opens up, as shown in Figure E-15; select the class

C_MainFrame and the list of messages appears.

Note: Once the .clw file is available, the query boxes shown in Figures E-13 and E-14 will not appear; ClassWizard will open up directly when you select “Resource | ClassWizard” from the AppStudio menu. Another way to open up an established ClassWizard (the .clw file exists) is to choose “Browse | ClassWizard” from the Visual Workbench.



Figure E-15: ClassWizard

Creating an AppWizard Starter Application

AppWizard will generate a four-class starter application for you. The four classes are: the application class, the mainframe class, the document class, and the view class. The AppWizard setup allows you to make choices on the type of application that you want, e.g., SDI vs MDI, etc. Once these choices are made, AppWizard generates your starter code and creates your project. The starter application code can be compiled by simply choosing “Build | Build ELLIPSE.EXE” from the Visual Workbench menu. Everything else has been done for you.

Note: If you discover that you have made the wrong setup choices after your starter application has been generated, the best thing to do is to start all over. AppWizard does not provide for changes and deletions in the code. If you are a very experienced programmer you may be able to enter the code and fix it, but this approach is not recommended for the novice.

This section shows the steps to generate the AppWizard starter application for the first AppWizard example in the book which is from Chapter Ten. We will name our program “Ellipse.”

1. From the Visual Workbench, choose “Project | AppWizard” and the AppWizard query box shown in Figure E-16 opens up.



Figure E-16: Generating an AppWizard Project

2. Click on “Options” and you get the “Options” query box shown in Figure E-17.



Figure E-17: AppWizard Options

3. In the “Options” query box, select the application features that you want.
 - a. You want a Single Document Interface (SDI), so uncheck the “Multiple Document Interface” checkbox. (Multiple Document Interface is the default, if you make no choice.)
 - b. Turn off all the options except “Printing” and “Print Preview.”
 - c. Select OK and you will be returned to the AppWizard query box shown in Figure E-16.
4. You have returned to the AppWizard query box shown in Figure E-16. From this query box click on the “Classes” button and the “Classes” query box shown in Figure E-18 appears.

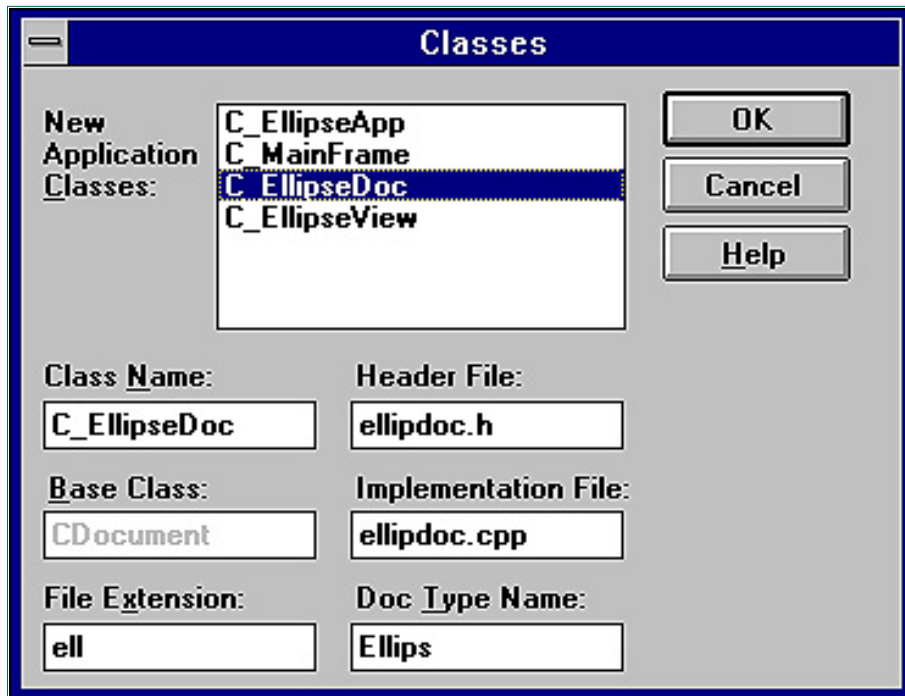


Figure E-18: The ClassWizard Classes Query

5. Focus on the CellipseDoc class and enter “ell” in the “File Extension” edit box.

Note: The non-grayed entries are active edit boxes which you can focus on and change the name that appears there. Focus on the Class Name edit box and change the class name to “C_EllipseDoc.” Focus on the other classes: CellipseApp, CMainFrame, and CellipseView, which also have active edit boxes in which you can change the entry and change the class names to C_EllipseApp, C_MainFrame, and C_EllipseView. When you focus on CellipseView you will see that you can change the base class from which it is to be derived. In this example, we did not change any of the AppWizard-suggested file names, but you can do so if you wish to.

6. To continue, click on OK and you will return to the AppWizard query box given in Figure E-16.

- a. You have been returned to the AppWizard query box. Click on OK in the AppWizard query box and you will get the “New Application Information” box shown in Figure E-19.
- b. Click on the “Create Button” in the “New Application Information” query box, and the starter application will be generated by AppWizard.



Figure E-19: The New Application Information Display

Your starter application has been generated for you. The project has been created and is open. You can click on the leftmost toolbar button and view the files that have been entered into your project. These files are shown in Figure E-20; there are many of them. So many, in fact, that scroll bars appear on the window and you will find that only half of them are visible without scrolling. AppWizard has generated an initial resource script for you. It has also generated an icon to be used for the main window (ellipse.ico). An .rc2 file has been generated which you can use for resources that you do not want AppStudio to see.

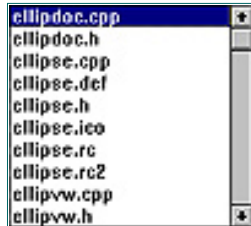


Figure E-20: AppWizard Project Files

7. To view all the resources that AppWizard has added for you, open AppStudio. Figure E-21 shows the resources that are now available in your .rc file.

- a. To open each one, focus on it and its name appears, then double-click on its name to open it for editing.



Figure E-21: Resources in the AppWizard Starter Application

The project is now ready to compile, but first you must select the compiler options that you prefer.

1. Choose “Options | Project” from the Visual Workbench.
 - a. From the “Project Options” query box that appears (refer to Figure E-5), choose “Release” and customize the compiler build

options as you previously did. (Refer to Figure E-6.)

The program is also ready to use ClassWizard. This means a ClassWizard database has been generated and placed in a file named `ellipse.clw`. This file does not show on the project list of files. To view the `ellipse.clw` file, you need to use the Program Manager.

At this point, your starter application is ready to tailor to your needs. This includes adding code to the four classes, which is covered in Chapters Ten through Fourteen of this book. You will also be adding resources to your application. Your starter resource script file has been included with your basic application. At this point, you have a project established and the other tools (AppStudio and ClassWizard) operate as before on the established project.

When you operate ClassWizard, it does not care how you generated your project and your source code; it operates the same on any established project. The same is true of AppStudio; it does not care how you generated your project and your source code. It only cares about whether your resource files were generated by AppStudio, so that it can enter and continue the process. AppStudio requires that you originally generate your resource files using AppStudio and that you do not remove or alter any of the comment lines of code that AppStudio inserts. It needs these lines to re-enter the files and to continue its work. To work with ClassWizard and AppStudio, refer to the sections of this appendix that cover these tools.

Creating Resources Using AppStudio

As shown in Chapter One, there can be a lot of different types of resources, all of which can be in your resource file. You use AppStudio to add new resources and to edit existing resources. To add a new resource not currently in your resource file:

1. Select “Tools | AppStudio | New” from the Visual Workbench menu. The New Resource query box appears (refer to Figure E-11).
2. In the New Resource query box, select the new resource that you desire, complete the resource, and save the resource file; the new resource will be included in your existing `.rc` file.

Working with each of these various resources is covered in the examples of this book. We look at adding accelerators in Chapter Two. The cursors, bitmap, and icon resources all come with an image editor that allows you to draw these resources and then save them. We discuss how to use these image resources in Chapters One and Four. Adding menus is covered in Chapter Two and in Chapter Ten. Using String Tables is covered in Chapters Seven and Eleven. The Dialog Editor which is used to generate your dialog boxes is more complex, and we have covered it in Chapters Six and Seven. The Toolbar resource is covered in Chapter Thirteen.

AppStudio is also used to edit existing resources. Refer to Figure E-20, which shows the resources for the Chapter Ten “Ellipse” program, which were generated by AppWizard. This figure serves to illustrate that there can be a large number of resources in a resource file, and that your program does not need to use them all. It also illustrates that there are many different kinds of

resources in the resource file. You may have multiple resources of a given kind; for example, we have two dialog boxes in the finished Chapter Ten example.

To edit an existing resource:

1. Double-click on that resource and the AppStudio Editor will open up.
2. Do your editing, save the file, and close AppStudio.

Note: AppStudio does not care whether you originally generated your source code manually or using AppWizard to get your starter code. It operates the same for any source code. It only cares that you continuously use AppStudio for your resource files.

Using ClassWizard

You can navigate to ClassWizard in two ways if you already have the .clw file:

1. From the Visual Workbench, choose “Browse | ClassWizard” and ClassWizard opens if the .clw file exists;
Or
2. From the Visual Workbench, choose “Tools | AppStudio” to open AppStudio.
 - a. From the AppStudio menu, choose “Resource | ClassWizard” and ClassWizard opens up.

Note: Navigate through the “Tools | AppStudio” route if you do not have a .clw file (a .clw file will be built for you).

ClassWizard is used to perform two basic tasks. The first is to add handler functions to your code. This is covered in Chapter Two for a manually generated project. However, ClassWizard operates exactly the same when used on an application that was started as an AppWizard project. AppWizard will automatically insert the comment code needed by ClassWizard. Once the appropriate comment code is in the source code, ClassWizard operates the same for any project.

The second basic task that ClassWizard is used for is to add a new class. ClassWizard is used for generating the CDialog-derived class when we design and use dialog boxes. Chapters Six and Seven cover how to generate a dialog box resource using AppStudio and then how to wrap it with a new class generated by ClassWizard. Refer to these chapters. As with AppStudio, the operation of ClassWizard is the same for any established project. The visual displays associated with Visual C++ 4 have been shown in the examples presented in this book. However, the visual differences between the Visual C++ 1.5 and Visual C++ 4 displays are minor.

Table of Contents

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

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

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

Keyword

☒ Brief ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

[Table of Contents](#)

Appendix F

The Symantec C++ 7 Compiler

Overview of Compiler Tools

In this appendix, the compiler tools that you need to use to do the examples given in this book are discussed. We start with how to set up your project, because this is the first thing that you need to do. In this book, we cover two project types; one is when you set up a project to use your own code and the second is when you use the tool AppExpress to help you set up a project with starter code generated by AppExpress. In the book, we first cover projects using your own code; so in this appendix setting up a project to use your own code is first discussed. The examples in Chapter One through Nine require projects set up manually.

Next in this appendix, the steps required to set up a project using AppExpress are shown. AppExpress not only sets up your project for you, but also automatically generates starter source code with four classes: the application class, the mainframe class, the document class, and the view class. The examples in Chapter Ten through Fourteen are started using AppExpress.

Once you have set up the project, there are tools available to help you develop your code. These tools are ResourceStudio and ClassExpress. ResourceStudio creates and manages your resource files. ClassExpress adds handler functions and classes. These tools, ResourceStudio and ClassExpress, do not care how you set up your project, whether you did it manually or used AppExpress. They simply operate on the source code files of an established project.

In this appendix, setting up the project is first discussed. Then ResourceStudio and ClassExpress are discussed.

Creating a Project For Your Own Code

1. Create the directory in which you want your project and the project's files to be located.
 - a. From the Integrated Development and Debugging Environment (IDDE) menu shown in Figure F-1, select “File | New” and a window named “Untitled-1” appears for typing your source code.



Figure F-1: The IDDE Menu

- b. The first example given in this book is used here. Type in your first file, MyApp.h, as shown in Figure F-2.

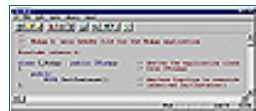


Figure F-2: Entering a Source Code File

2. Save the file in the directory that you have created for your project.
 - a. Choose “File | Save As” from the “Untitled-1” window.
 - b. Enter the directory and the file's name, MyApp.h, as shown below.

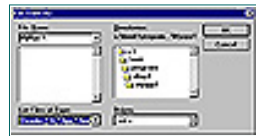


Figure F-3: Saving the Source Code File

- c. Save the file by selecting OK.
3. Close the file by choosing “File | Close” from the text editing window.
4. Continue typing in source files.
 - a. Again choose “File | New” from the IDDE menu to enter the next file.
 - b. Repeat this process until you have typed in and saved (in the MyApp1 directory) all of the source files, which in this example are: MyApp.h, MyApp.cpp, Mainfrm.h, and Mainfrm.cpp.
5. Go to the IDDE and select “Project | New” and the first of a series of ProjectExpress pages appears.
 - a. Choose the name “MyApp1.prj” for your project and select the directory that it is to be installed in. When you are finished it should look like Figure F-4.



Figure F-4: Creating a Project

6. Then choose “Next >” and the second of the series of pages appears as shown in Figure F-5.

- a. Make your selections and when you are finished choose “Next >.”



Figure F-5: Choosing the Project Type

- b. When the “Add Files to Project” page appears next, select all of your .cpp files and add them to your project. When you are finished the page should look like Figure F-6.



Figure F-6: Adding the Source Files to Your Project

7. Choose ” Next >”; the “Initial Settings” page appears. When you are finished with this page it should look like Figure F-7.



Figure F-7: Initial Settings for Your Project

Note: You do not need to fill in the include directories for an MFC project; it has all been handled automatically.

- a. Choose “Finish”; your project will be created and automatically installed in the IDDE. The IDDE should now look like Figure F-8.



Figure F-8: The IDDE with “MyApp1” Project Open

8. The project is ready to build, so choose “Project | Build” from the IDDE menu.
9. The project builds and when you have successfully eliminated all compiler errors, run the executable from the IDDE by choosing “Project | Execute Program.”

Adding Resource Files

At this point, we have an executing program, but it is one that does not use any resource files. We create resources by using the IDDE’s tools. To add our resources with ResourceStudio, we next create and add two files to the project.

1. Add a resource.h file that we leave completely blank, as shown in

Figure F-9, and save it as resource.h in the “MyApp1” directory.

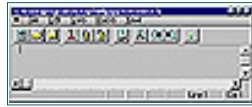


Figure F-9: Adding an Initial Resource.h File

2. We also create a MyApp.rc file that has the appropriate include files, but has no more in it at this point, as shown in Figure F-10. Save this file in the MyApp1 directory.

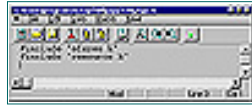


Figure F-10: The Initial MyApp.rc file

- a. This .rc file must be added to our project. Choose “Project | Edit” from the IDDE and the “Edit Project” query box appears. (1) In the “Edit Project” query box, you will find the MyApp.rc file listed. Add it to your project. When you are finished the “Edit Project” query box should look like Figure F-11.



Figure F-11: Adding the .rc File to Your Project

3. Having a .rc file in your project will activate the ResourceStudio. Now from the IDDE menu choose Resource and you will find its submenu activated at this point.

- a. Choose “Resource | New” and the “Create Resource” query box shown in Figure F-12 opens up.



Figure F-12: Creating Resources With ResourceStudio

4. The appropriate editor will open up when you click on the resource that you want to create.

- a. Create whatever resource you need.
- b. Save the file.
- c. Close the file.
- d. Close ResourceStudio.

Note: ResourceStudio creates image files (if cursors, icons, or bitmaps have been created) and it automatically adds all necessary code to your .rc file and the resource.h file.

- b. Click on “Next >” and you will get the “Select Directory” page shown in Figure F-15, in which you will select the directory that you want your new project to be installed in.

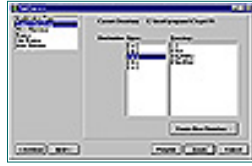


Figure F-15: Selecting the New Project's Directory

- (1) Select your directory, then click on “Create New Directory.”
- (2) You will get a query box as shown in Figure F-16. Fill in the name. A directory of that name will be created and your new project will be installed in that directory. (The project will bear the name that you will select on the next Miscellaneous page.)



Figure F-16: Creating the Directory for the New Project

- (3) Choose the name of your project and click on “Create” and you will notice that the directory has been installed in the “Select Directory” page which you are still on.
- (4) Now choose “Next >” to move on to the next page.

The “Miscellaneous” page shown on Figure F-17 appears. The project will bear the name selected on this page. The name “Ellipse” was chosen, which will give the project the same name as the directory in which it will be installed. The other data will appear in the “About” dialog box that is generated and included in the application class of your starter code.



Figure F-17: Selecting Miscellaneous Names

3. Click on “Next >” to go to the “Names” page where you can customize the names of all the classes that AppExpress will generate automatically.
 - a. From the name drop-down list, select each class name and customize it. I customized the class names by adding an underbar in each, e.g., I changed C_EllipseApp to C_EllipseApp, etc. You can also change the header and source file names if you choose to. The page for customizing class and file names is shown in Figure F-18.



Figure F-18: Customizing the Class and File Names

- b.** The next page is the “File Names” page, shown in Figure F-19. Do not make any changes on this page; this page shows the files and resources that will be generated by AppExpress.

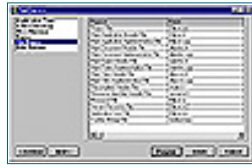


Figure F-19: Your New Project Files

4. At this point, click on “Finish”; you do not need to go to the “Help Options” page. AppExpress now generates your starter code and the new project is loaded into the IDDE automatically.

Your starter application has been generated for you. The project has been created and is open. AppExpress has generated an initial resource script file and its companion resource.h file. It has generated an icon, ellipse.ico, to be used for the main window. An .rc2 file has been generated, which you can use for resources that you do not want ResourceStudio to see. Your application contains a toolbar and a status bar and you also have print and print preview capability. These features will be generated for every AppExpress starter application; they are not optional. If you do not want them, you need to remove them from the code.

The program is also ready to use ClassExpress; the ClassExpress database has been generated and placed in a file named `Ellipse.cle`. To view the `Ellipse.cle` file, you need to use Windows Explorer. ClassExpress inserts a message map suited to ClassExpress's needs in all four classes—`C_EllipseApp`, `C_MainFrame`, `C_EllipseDoc`, and `C_EllipseView`.

At this point, the project is ready to compile, but we need to first customize the project settings.

1. Choose “Project | Settings” from the IDDE menu.
 - a. On the Project Settings “Target” tab you may wish to use the settings shown in Figure F-20.



Figure F-20: Customizing the Project Settings

- b.** On the Project Settings “Build” tab, make sure that precompiled headers are selected since you will add code and rebuild a lot of times to develop your application.
- 2.** After you have customized your project settings, compile your project by choosing “Project | Build” from the IDDE menu.
- 3.** To run the compiled program, choose “Project | Execute Program” from the IDDE menu. The compiled program is shown in Figure F-21. Note that it has a toolbar and a status bar.



Figure F-21: AppExpress Starter Application for Project Ellipse

At this point, your starter application is ready to tailor to your needs. This includes adding code to the four classes to include additional capabilities that you need for your application, which is covered in Chapters Ten through Fourteen of this book. You have an established project and you can operate ResourceStudio and ClassExpress on your project. These tools, ResourceStudio and ClassExpress, operate the same on any established project; they do not care whether you generated your project manually or as an AppExpress starter application.

ResourceStudio

In Chapter One you learned that there can be many kinds of resources, any or all of which can be in your resource file. You use ResourceStudio to add new resources and to edit existing ones.

1. To add a new resource not currently in your resource file, select “Resource | New” from the IDDE menu.
 - a. The “Create Resource” query box opens up (refer to Figure F-12). Select the new resource that you desire and the appropriate editor opens up for you to complete the resource. There are image editors for bitmaps, cursors, and icons; a dialog editor for dialog boxes; a string editor for string tables; etc.
 - b. Complete and save the resource; it will be automatically added to your .rc file.

There are only minor variations between Microsoft’s AppStudio and Symantec’s ResourceStudio. In this book, the image resources (cursors, icons, bitmaps) are covered in Chapters One through Four. Chapter Two covers menus and accelerators. Dialog resources are covered in Chapters Six and Seven. String Table resources are covered in Chapters Seven and Eleven.

ResourceStudio is also used to edit existing resources that have already been added to your resource file.

1. To edit resources, open them for editing by choosing “Resource | Edit” from the IDDE menu.
 - a. Resource Studio opens up; highlight the resource type that you wish to edit, and the resources of that type appear in the lower left list box.
 - (1) To open a specific resource for editing, double-click on the specific resource that you wish to edit and it appears in the right-hand window. Figure F-22 shows the resources for the Chapter Ten “Ellipse” program, which were generated by AppExpress. There can be many types of resources in your resource file.

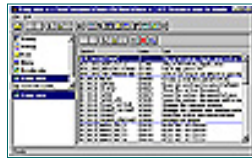


Figure F-22: Editing Resources Using ResourceStudio

- b.** Do your editing, save the file, and close ResourceStudio.

ClassExpress

ClassExpress is used like ClassWizard, which has been described in Chapters Two, Six, and Seven. Chapter Two describes how ClassWizard adds handler functions. ClassExpress operates in the same way; the differences between the two are cosmetic. Chapters Six and Seven describe how to add the dialog box's member variables and to create a CDialog-derived class using ClassWizard. ClassExpress operates the same way with only cosmetic differences.

- 1.** Navigate to ClassExpress by choosing “Tools | ClassExpress” from the IDDE menu. ClassExpress opens (refer to Figure F-13).
 - a.** Focus on Message Maps in the list box in the upper left-hand corner and you can then add message handler functions to your application.

You also use ClassExpress when you are creating a dialog box.

- 1.** Create a new dialog resource.
- 2.** Using ClassExpress create a CDialog-derived class for the dialog resource you just created.
- 3.** Add member variables to your new CDialog-derived class by choosing “Data Transfer” from the list box in the upper left corner of the ClassExpress dialog box.

This process is the same as that described in Chapters Six and Seven for ClassWizard. Your ClassExpress dialog boxes look different, but they operate in the same fashion.

[Table of Contents](#)

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

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

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

☒ Brief
 ☐ Full

[Advanced Search](#)

[Search Tips](#)

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

[Bookmark It](#)

Search this book:

[Table of Contents](#)

Appendix G

The Borland C++ 5 Compiler

This appendix covers how to generate an application project in which you type in your own MFC code. It also shows you how to use the Resource Workshop with your MFC application. You will be able to do the examples in this book with the Borland compiler. The code for each example has been given in this book, so that you can type it in. The exceptions to this are the examples in Chapters Eleven, Thirteen, and Fourteen which do not list all the code in the book; however, the source code for all examples is included on the enclosed diskettes.

Overview of Compiler Tools

Note: Borland has recently licensed MFC and is now distributing MFC for Win32 operating systems (i.e., MFC 3.2 or 4.0 and above) with its newest versions of Borland C++ 5 (5.01 and above). As of this writing, the Borland compiler tool support for MFC is not as well developed as the MFC tool support that is provided with the other major compilers. This may change in future versions.

The Borland C++ compiler has long hosted the *ObjectWindows Class Library*, commonly referred to as *OWL*. The AppExpert tool provided with the Borland compiler lets you create a starter application based on the ObjectWindows Library. These AppExpert applications have features such a printing, document-view, toolbars, etc. AppExpert works with the Resource Workshop and ClassExpert. ClassExpert lets you add entries to the message response table, create new classes, and associate a *TDialog* class (an OWL class) with a dialog resource. The Resource Workshop provides editors for creating resources such as icons, cursors, accelerator tables, menus, dialogs, and string tables.

Note: AppExpert and ClassExpert work for ObjectWindows-based applications.

Currently Borland does not provide equivalent compiler tools that work with MFC-based applications. The Resource Workshop can be used with MFC-based applications.

Creating an Application Project

1. Create a new project
 - a. From the Borland C++ Integrated Development Environment (IDE), which is shown in Figure G-1, choose “File | New | Project.”

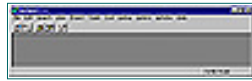


Figure G-1: The Borland C++ IDE

The “New Target” query box appears, as shown in Figure G-2. Here you can create and name the new project. The skeleton project that will be created inherits the option settings of the project that was last opened. If this is the first project in your current session, the new project is given default settings. After you create the initial target for your project, you can modify the settings to fit your needs.



Figure G-2: Target Expert with Initial Settings

When the “New Target” query box first opens up, it has the OWL framework checked, with the MFC framework grayed as shown in Figure G-2. You need to change the options in the “New Target” query box, as discussed in the following steps.

- b. Enter the location of where your project will be stored in the “Project Path and Name:” input box.

Note: If the directory does not exist, the IDE will create it for you.

- c. Enter the name of your project in the “Target Name:” input box.
 - d. Uncheck the OWL framework and check the MFC framework.
 - e. Select “Static” Library.
 - f. Click the “Advanced” button and the “Advanced Options” query box appears:

- (1) Uncheck both the “.rc” and “.def” options. Unchecking these settings tells the Project Manager that this project does not use definition and resource files.

- (2) When you are finished, the “Advanced Options” query box should look as shown in Figure G-3. Click “OK” and you will return to the “New Target” query box.



Figure G-3: The “Advanced Options” Query Box

- g. Make sure that in the “Target Type:” list box, the selection “Application[.exe]” has been made, since this is the type of program that you plan on creating. (It is the default setting.)
- h. When you are finished with the “New Target” query box, check to make sure that it looks like Figure G-4.



Figure G-4: Target Expert with Settings for an MFC Program

- i. Click “OK” and your project is created for you and is displayed in the Project Manager window; the IDE will now look like Figure G-5.



Figure G-5: Borland C++ Project Manager

2. Enter your source code files.
 - a. In the MyApp1 example, we have two .cpp files, so add a source node. We want to add a node on the same level as the existing node, “myapp.cpp[.cpp],” so we select the node “myapp.exe[.exe]” under which we want the new node to appear.
 - b. Press Insert or right-click the “myapp.exe[.exe]” node to open the Project window’s menu.
 - (1) Choose “Add Node.”
 - (2) Add the node and name it mainfrm.cpp. The Project Manager now looks as shown in Figure G-6.

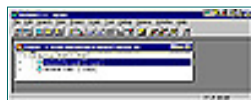


Figure G-6: Adding Nodes to the Project

Note: To delete nodes, select the node and press Delete or right-click the node to open the Project window’s menu and then choose “Delete Node.” The Project Manager asks if you want to delete the node before it proceeds.

- c. Next you will enter your .cpp files.
 - (1) In the Project window, double-click on the node to open an empty file in the Edit window.
 - (2) Type in the file MyApp.cpp, as shown in Figure G-7. Be sure to

save the file by choosing “File | Save” from the IDE menu; then close the file.

(3) Repeat this step, typing in all your .cpp files.



Figure G-7: Entering .cpp Files

d. Enter the .h files.

(1) Choose “File | New | Text Edit” from the IDE menu.

(2) Type the MyApp.h file into the edit window that appears.

(3) Choose “File | Save as” from the IDE menu. The “Save File As” query box shown in Figure G-8 appears; save the file as MyApp.h in your project’s directory.

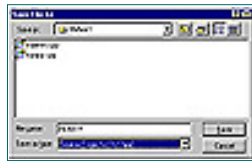


Figure G-8: Saving Files In Your Project

(4) Repeat this step for all your .h files.

3. Set the project options by choosing “Options | Project” from the IDE menu. When you are finished, the “Project Options” query box should look like Figure G-9.



Figure G-9: Customizing Your Project Settings

Note: You must provide the paths and names of the include directories. You must include the path to where the MFC library is. The output directories will put your project into the specified directory, you only need to specify a directory if you want it to go somewhere other than into your project directory.

4. Compile and run the project.

a. Choose “Project | Build All” to compile your program. Warnings will be generated, but it is safe to ignore them. Choose OK after the compilation is successful.

b. Run the program from the IDE menu, by choosing “Debug | Run.” You can also run the program by double-clicking the “myapp.exe[.exe]” node in the Project Manager.

Adding Resources

1. Attach a .rc file to your project by either:

- a. Creating a project selecting the “.rc” option in the “Advanced Options” query box, or
 - b. Adding a .rc node to an existing project.
2. Use Resource Workshop to create your resources.
 - a. Start the Resource Workshop by double-clicking the existing .rc node in the Project window, and the Resource Workshop opens up as shown in Figure G-10.

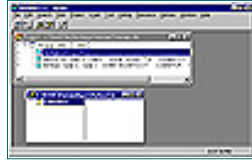


Figure G-10: Opening the Resource Workshop

- b. Right-click on the string “Identifiers” in the Resource Workshop and choose “New Resource” from the menu that appears. The “New Resource” query box will appear as shown in Figure G-11.

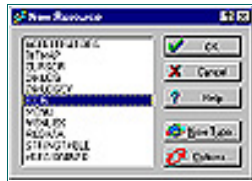


Figure G-11: The “New Resource” Query Box

Note: There is an editor for each of the resources that are discussed in this book: icons, cursors, accelerators, menus, dialogs, and string tables.

- c. Choose the resource that you wish to create; in this case, an icon has been chosen. Double-click on the resource that you wish to create, and its editor opens up.

In this example, an icon and also a cursor resource were created. The Resource Workshop creates your resources for you. They are not put into a separate file, but are contained within the .rc file. The Resource Workshop will give your resources an ID and #define those IDs as UINTs; all this is done in the .rc file. You must separate out the #define statements and put them into a resource.h file, as discussed in the following steps.
- d. Modify the .rc file, removing the #define statements and insert the proper #include statements. Add the following statements to the .rc file:

```
#include "resource.h"
#include "afxres.h"
```

- e. Place the #define statements in a separate resource.h file. For this example, the resource.h file will look as follows:

```
//
//  resource.h
//
#define IDI_ICON1
#define IDC_CURSOR1
```

130
131

f. Modify the mainfrm.cpp by adding the following statement to the #include's:

```
#include "resource.h"
```

g. To reopen the .rc file to modify a resource, open the Resource Workshop by double-clicking on the string “myapp.rc[.rc].”

(1) Expand the nodes in the Resource Workshop window, then double-click on the resource that you wish to modify. This is shown in Figure G-12.

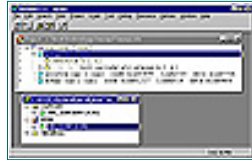


Figure G-12: Opening an Existing Resource for Editing

h. You can use editors to create accelerator tables, menus, dialogs, and string tables. However, you can also type these resources (e.g., small menus or dialogs) directly into the .rc file if it is more convenient to do so. Use the DOS editor for this. Enter a #define statement for every ID into the resource.h file.

[Table of Contents](#)

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

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

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



HOME



ACCOUNT INFO



SUBSCRIBE



LOGIN



SEARCH



MY ITKNOWLEDGE



FAQ



SITEMAP



CONTACT US

SEARCH
ITKNOWLEDGE

To access the contents, click the chapter and section titles.

Learn the MFC C++ Classes

(Publisher: Wordware Publishing, Inc.)

Author(s): Shirley Wodtke

ISBN: 1556225121

Publication Date: 01/01/97

Bookmark It

Search this book:

Go!

[Table of Contents](#)

BROWSE
BY TOPIC

Index

Special Characters

:: (scope resolution operator), 549

>> (extraction operator), 359-360

<< (insertion operator), 359-360

& (keyboard accelerator operator), 49

& (reference operator), 575

16 color (4-bit) mode, 97

24-bit mode, 151

256 color (8-bit) mode, 150

A

accelerator table, 49-50

See also accelerator, resource file

accelerators,

See also keyboard alternatives

ID value, 51

introduced, 42

resource file, 49-50

See also accelerator table

use of, 49-51

access functions, 568

active window, 8

adding a class,

- CFormView, 428
- dialog, 214-215
- splitter window, 476
- adding message map entries, 56-61
- AFX_IDI_STD_FRAME, 31, 34
- <afxcmn.h>, 494
- <afxdlgs.h>, 181, 185, 509
- AfxGetApp(), 29
- AfxRegisterWndClass(),
 - parameters, 26
 - use of, 28-29, 113-114, 166
- <afxwin.h>,
 - contents, 16
 - defined, 15
- Alt key, 49
- ampersand (&), 49
- Animate application,
 - animate.rc file, 147
 - mainfrm.cpp file, 146-147
 - mainfrm.h file, 145-146
 - message handler discussion, 149-150
 - overview, 143-145
 - resource.h file, 148
- animation, 143
- API functions,
 - ::CreateSolidBrush(), 30
 - ::DefDlgProc(), 12
 - ::DefMDIChildProc(), 12;
 - ::DefWindowProc(), 12, 20
 - ::DeleteObject(), 30
 - ::LoadResource(), 155
 - ::MessageBox(), 52
 - ::SetCursorPos(), 71
 - ::SetFocus(), 9
- API, *See* Application Programming Interface
- AppExpert, 635
 - See also* Borland C++ compiler tools
- AppExpress,
 - See also* Symantec C++ compiler tools
 - introduced, 5, 619

- use of, 627-632
- application class,
 - with document-view structure, 313, 329
 - with two-class structure, 15
- application framework, 1
- Application Programming Interface,
 - defined, 11-13
 - function notation, 9
 - introduced, 1
- applications,
 - Animate application, 143-150
 - BackingStore application, 132-137
 - Bars application, 455-482
 - Buttons application, 204-225
 - ChildWin application, 159-165
 - ColorDlg application, 183-185
 - CustCtrl application, 487-493
 - Custom application, 315-327
 - Elip application, 336-366
 - ElipMdl application, 379-383
 - ElipsMin, 367-378
 - FastDraw application, 123-130
 - FixedChd application, 173-176
 - Form application, 421-442
 - ForMin application, 442-450
 - Graphics application, 98-107
 - Hello application, 85-87
 - HelloB application, 88-89
 - MenuA application, 42-45
 - MenuB application, 61-69
 - ModalCom application, 278-288
 - Modeless application, 292-306
 - MyApp1 application, 16-18
 - MyApp2 application, 23-24
 - MyApp3 application, 27-28
 - MyApp4 application, 34-35
 - MyApp5 application, 36-37
 - NewCmnCtrls application, 494-507
 - OwnerDC application, 115-117
 - PopUp application, 172-173

- PropertySheet application, 508-516
- Tac application, 388-418
- UsrInput application, 240-262
- UsrStrng application, 270-273
- Apply button, property sheet, 517
- AppStudio,
 - See also* Microsoft Visual C++ compiler tools
 - editing string tables, 389-390
 - editing toolbar bitmaps, 469-475
- AppWizard,
 - See also* Microsoft Visual C++ compiler tools
 - adding splitter windows, 364
 - introduced, 4, 335, 577, 599
 - project, 587-596, 610-615
 - starter application, 337, 389, 429-430, 457-458, 498-499
- archiving files, *See* filing
- arrays, 555, 560-563
- ASSERT_VALID() macro, 384
- automatic storage class, 546

B

- background color, dialog windows, 225
- background color, setting, 82, 88
- background mode, setting, 82, 98
- backing store,
 - introduced, 123
 - steps, 139-142
 - use of, 131-132
- BackingStore application,
 - mainfrm.cpp file, 134-137
 - mainfrm.h file, 133-134
 - overview, 132
- Bars application,
 - adding document template, 475-482
 - document class, 459-461
 - message handler functions, 466-468
 - overview, 455-457
 - starter application, 457-458
 - scroll bars, 463-464
 - splitter window, 475-474
 - status bar, 464, 465

- toolbar, 469-475
- tooltips, 470
- view class, 461-464
- base class,
 - See also* class, superclass
 - defined, 563
 - initializer, 571
- BEGIN_MESSAGE_MAP() macro, 45-46
- BITMAPINFOHEADER structure, 150-151
- bitmaps,
 - defined, 137
 - for animation, 143
 - introduced, 123
 - loading, 148
 - with memory device context, 138-139
- BitBlt-ing bitmaps, 150
- bounding rectangles, 86
- Borland C++ compiler tools,
 - AppExpert, 635
 - ClassExpert, 635
 - Resource Workshop, 5, 635, 642 thru 644
- brushes,
 - deleting, 109
 - logbrush, 94
 - overview, 92
 - selecting, 81, 82, 108
 - stock brushes, 83
 - styles of, 93
- buddy edit box, 494
- button controls,
 - button styles, 192-193
 - check box styles, 196-197
 - group box styles, 197-198
 - messages, 203-204
 - owner draw style, 198
 - placed on window, 201
 - push button styles, 193-194
 - radio button group, 212, 280
- Buttons application,
 - application class files, 220-221

- BtnDialog.cpp file, 223
- BtnDialog.h file, 222
- creating dialog class, 214-218
- creating DIALOG resource, 210-213
- mainfrm.cpp file, 221-222
- mainfrm.h file, 221
- overview, 204-206
- resource.h file, 224
- script1.rc file, 224-225

C

- CAnimateCtrl class, 493
- capturing mouse, 111
- CArchive class,
 - data types supported, 359
 - introduced, 330
- CArchive::IsStoring(), 359
- CBitmap class, 81
- CBitmap::CreateCompatibleBitmap(), 139, 142
- CBitmap::LoadBitmap(), 148, 149, 155
- CBitmapButton class, 198, 485, 486
- CBrush,
 - construction, 93-94
 - overview, 80-81
- CBrush::CreateBrushIndirect(), 94
- CBrush::CreateHatchBrush(), 94
- CBrush::CreateSolidBrush(), 93
- CButton class,
 - Create() function, 201, 202, 205
 - introduced, 190
 - member functions, 202
- CByteArray class,
 - introduced, 397
 - member functions, 402
 - use of, 398-399
- CCheckListBox class, 486
- CClientDC class,
 - constructor, 88
 - introduced, 85
 - use of, 130, 133
- CCmdTarget class, 314

- CCmdUI class,
 - introduced, 277
 - member functions, 281
 - use of, 280-282
- CColorDialog class,
 - introduced, 183
 - use of, 183-185
- CComboBox class,
 - introduced, 190,
 - overview, 231
- CCreateContext class, 363, 477
- CCtrlView class, 332
- CDataExchange class, 265
- CDC class,
 - drawing functions, 84
 - introduced, 84-85
 - use of, 86
- CDC::BitBlt(), 139, 141, 142, 148
 - parameters, 142
- CDC::CreateCompatibleDC(), 138, 148, 149
- CDC::GetTextMetrics(), 408
- CDC::IsPrinting(), 353
- CDC::RealizePalette(), 154
- CDC::SelectObject(), 82, 109, 139, 148, 149
- CDC::SelectPalette(), 154
- CDC::SetBkColor(), 113
- CDC::SetBkMode(), 113
- CDC::SetROP2(), 95
- CDC::SetViewportExt(), 354-355
- CDC::SetViewportOrg(), 88, 354-355
- CDC::SetWindowExt(), 354-355
- CDC::SetWindowOrg(), 88
- CDC::TextOut(), 403
- CDialog class,
 - encapsulating DIALOG resource, 186, 214, 240, 244, 346
 - introduced, 179-180
- CDialog::Create(), 295, 296, 382
- CDialog::DoDataExchange(), 264
- CDialog::DoModal(), 289
- CDialog::OnCancel(), 291, 307

- CDialog::OnInitDialog(), 252, 489, 502
- CDialog::OnOK(), 290, 291, 307, 382
- CDocument class, 311, 329, 335, 342, 432
- CDocument::DeleteContents(), 422, 434
- CDocument::DoSave(), 379
- CDocument::OnCloseDocument(), 422, 434
- CDocument::OnNewDocument(), 361, 399, 422, 426, 433
- CDocument::OnOpenDocument(), 399, 434, 422
- CDocument::OnSaveDocument(), 422, 435
- CDocument::Serialize(), 342, 358, 359, 360, 399
- CDocument::SetModifiedFlag(), 418, 438
- CDocument::UpdateAllViews(), 366, 437, 439
- CDragListBox class, 486
- CDWordArray class, 397
- CEdit class, 190, 231, 239, 332
- CEditView class, 332
- CFileDialog class, 181
- CFindReplaceDialog class, 183
- CFont class, 80, 81, 406, 414
- CFont::CreateFont(),
 - introduced, 406
 - parameters, 407
 - use of, 408-410
- CFontDialog class, 183
- CFormView class, 332, 421, 427-428, 432, 437
- CFormView::OnInitialUpdate(), 426, 437, 438, 422
- CFormView::OnUpdate(), 422
- CFrameWnd class, 14-15, 18, 20, 316, 329, 335, 363, 394, 423
- CFrameWnd::Create(),
 - input parameters, 22
 - use of, 21, 24, 47
- CFrameWnd::LoadAccelTable(), 50
- CFrameWnd::OnCreateClient(), 363, 394
- CFrameWnd::PreCreateWindow(), 318
- CGdiObject class, 80
- CGdiObject::DeleteObject(), 109
- CGdiObject::GetSafeHandle(), 82
- checkbox controls, 196-197
- child window,
 - clipping, 160

- construction, 161, 166
- controls, 201
- defined, 159
- fixed, 159, 173
- introduced, 12
- MDI, 421
- message maps, 169
- styles, 168
- ChildWin application,
 - application class files, 162
 - child.cpp file, 164-165
 - child.h file, 164
 - ChildWin.rc file, 165
 - overview, 159-161
 - mainfrm.cpp file, 163-164
 - mainfrm.h file, 162
 - resource.h file, 165
- CHeaderCtrl class, 494
- CHotKeyCtrl class, 494
- CImageList class, 485, 504
- CImageList::Create(), 504
- class definition block, 540
- class hierarchy, 520
- class libraries, 1, 635
 - See also* Microsoft Foundation Class Library; ObjectWindows Class Library
- class styles, 28
- class,
 - base, 563
 - constructor, 546
 - container, 573
 - declaration, 541
 - definition, 539-540
 - derived, 563-565
 - destructor, 547
 - forward declaration, 541
 - inheritance, 563-564
 - subclass, 563
 - superclass, 563
- ClassExpert, 635

- See also* Borland C++ compiler tools
- ClassExpress,
 - See also* Symantec C++ compiler tools
 - adding message map entries, 56-60
 - introduced, 5, 619
 - use of, 626
- ClassWizard,
 - adding a class, 427-428, 476-277
 - adding handler functions, 347, 428, 437
 - adding member variables, 216, 250-251, 350, 428
 - creating a dialog class, 214-215, 249, 349
 - data map, 263
 - introduced, 4
 - message map entries, 56-60, 215-216, 249, 349
 - update handler functions, 277, 281-282, 347
 - virtual map, 263
- client area, 6
- clipping, child windows, 160
- CListBox class, 190, 231, 239, 486
- CListCtrl class, 485
- CListView class, 333
- CMapPtrToPtr class, 398
- CMapPtrToWord class, 398
- CMapStringToOb class, 398
- CMapStringToPtr class, 398
- CMapStringToString class, 398
- CMapWordToOb class, 398
- CMapWordToPtr class, 398
- CMDIChildWnd class, 423, 432, 475
- CMDIFrameWnd class, 423
- CMenu class,
 - introduced, 41
 - member functions, 71, 72
 - use of, 65, 70-71
- CMetafileDC class, 85
- CMultiDocTemplate class, 328, 432
- CObArray class, 397
- CObject class, 344, 383, 384, 432
- CObject::AssertValid(), 383
- CObject::Dump(), 383

- CObject::IsKindOf(), 330
- COBList class, 397
- collection classes,
 - array collections, 397
 - introduced, 387
 - list collections, 397
 - map collections, 397
 - shape, 397
- color palettes,
 - defined, 150-151
 - introduced, 123
- ColorDlg application, 183-185
- combo box controls, 232, 237-239
- command messages, 45-46, 507
 - See also* WM_COMMAND message; WM_NOTIFY message
- command routing, 314, 321, 423
- common controls,
 - button control styles, 192-198
 - See also* button controls
 - combo box control styles, 237-239
 - control classes, 191
 - edit control styles, 232-234
 - list box control styles, 235-236
 - new common controls, 485, 493, 507
 - overview, 190
 - static control styles, 198-200
 - Win 3.1 common controls, 190-191, 231, 239
 - Win95 common controls, 191
- Common Dialogs, 179, 181-185
- compiling resources, 32-33
- concatenated keypresses, 8
 - See also* keyboard alternatives; accelerators
- constructors, derived classes, 571
- container class, 573
- control functions, for member objects, 265
- control notification messages,
 - button control, 192
 - combo box control, 239
 - edit control, 239
 - introduced, 179

- list box control, 239
- list view messages, 507
- new common controls, 507
- slider messages, 507
- spinner messages, 507
- tree view messages, 507
- control resource line, 494
- controls, *See* button controls; common controls; custom controls
- CPageSetupDialog class, 183
- CPaintDC class,
 - introduced, 85
 - use of, 90
 - with document-view, 320-352
- CPalette class, introduced, 81
- CPalette::CreatePalette(), 154
- CPalette::GetPaletteEntries(), 154
- CPen class,
 - construction, 91-92
 - overview, 80-81
- CPen::CreatePen(), 91-21
- CPen::CreatePenIndirect(), 91-92
- CPoint class, 113
- CPreviewDC class, 352
- CPrintDialog class, 182
- CPrintInfo class, 353, 356
- CProgressCtrl class, 485, 494, 503
- CPropertyPage class, 485, 510
- CPropertyPage::OnApply(), 517
- CPropertySheet class, 485
- CPropertySheet::AddPage(), 510
- CPropertySheet::Create(), 509
- CPropertySheet::DoModal(), 509
- CPropertySheet::SetTitle(), 510
- CPtrArray class, 397
- CPtrList class, 397
- CreateSolidBrush(),
 - API function, 30
 - CBrush function, 93
- CREATESTRUCT members, 317
- CRect class,

- archiving, 359
- defined, 24
- use of, 113
- CRichEditView class, 332
- CRichEditCtrl class, 494
- CRgn class, 80
- CRunTimeClass class, 330
- CRuntimeClass::CreateObject(), 330
- CS_CLASSDC style attribute, 28, 114
- CS_OWNDC style attribute, 28, 114, 319
- CS_PARENTDC style attribute, 28, 114
- CScrollBar class, 190
- CScrollView class, 331, 332, 427, 455, 462
- CScrollView::SetScrollSizes(), 462, 463
- CSingleDocTemplate class, 327, 328
- CSize class, 113
- CSliderCtrl class, 485, 503
- CSpinButtonCtrl class, 485, 504
- CSplitterWnd class, 362, 363, 394, 475, 478
- CSplitterWnd::CreateView(), 394
- CSplitterWnd::OnCreateClient(), 477
- CStatic class,
 - Create() function, 203
 - introduced, 190
 - member functions, 202
 - use of, 201
- CStatusBar class, 465
- CStatusBar::SetPaneText(), 465
- CStatusBarCtrl class, 493
- CString class,
 - archiving, 359
 - construction, 268-269
 - functions, 269
 - introduced, 231
 - operators, 269
- CString::LoadString(), 271
- CStringArray class, 397
- CStringList class, 397
- CTabCtrl class, 493
- CToolBar class, 469

- CToolBarCtrl class, 493
- CToolTipCtrl class, 493
- CTreeCtrl class, 485
- CTreeView class, 333
- CUIIntArray class, 397
- cursors,
 - changing, 62-65
 - customized, 33, 36, 37-38
- CustCtrl application,
 - code excerpts, 490-491
 - CustCtrl.rc excerpts, 492
 - overview, 487-489
 - resource.h file, 492-493
- custom controls,
 - dynamic subclassing, 487
 - introduced, 485
 - owner-draw, 486
 - self-draw, 486
- Custom application,
 - CustDoc.cpp file, 326
 - CustDoc.h file, 326
 - custom.cpp file, 323
 - custom.h file, 323
 - custom.rc file, 327
 - customizing the mainframe window, 318-319
 - customizing the view window, 319-320
 - CustView.cpp file, 325-326
 - CustView.h file, 325
 - mainfrm.cpp file, 324
 - mainfrm.h file, 324
 - message response functions, 321-322
 - overview, 315-316
 - resource.h file, 327
- CView class, 311, 316, 329, 331, 332, 335, 355
 - member functions for printing, 356
- CView::OnDraw(), 320, 344, 352, 353, 414, 461
- CView::OnFilePrint(), 379
- CView::OnFilePrintPreview(), 379
- CView::OnInitialUpdate(), 412, 414, 418
- CView::OnPaint(), 353

- CView::OnPrepareDC(), 353
- CView::OnPreparePrinting(), 356
- CView::OnPrint(), 352-353
- CView::OnUpdate(), 437, 461
- CView::PreCreateWindow(), 319, 461
- CWinApp class, 15, 18, 19, 20, 335
- CWinApp::AddDocTemplate(), 478
- CWinApp::GetDescendantWindow(), 465
- CWinApp::InitInstance(), 19, 327
- CWinApp::LoadCursor(), 37
- CWinApp::LoadIcon(), 37
- CWinApp::LoadStandardCursor(), 29-30
- CWinApp::m_nCmdShow, 19
- CWinApp::OnAppExit(), 379
- CWinApp::OnFileNew(), 378
- CWinApp::OnFileOpen(), 378
- CWinApp::OnFilePrintSetup(), 378
- CWinApp::SetDialogBkColor(), 225
- CWindowDC class, 85
- CWnd::ClientToScreen(), 71
- CWnd::Create(),
 - introduced, 161
 - defined, 166
 - parameters, 167
- CWnd::DragAcceptFiles(), 451
- CWnd::EnableWindow(), 489
- CWnd::GetClientRect(), 344, 71
- CWnd::GetDlgItem(), 205
- CWnd::GetDlgItemInt(), 204
- CWnd::GetDlgItemText(), 204
- CWnd::Invalidate(), 110, 112, 90
- CWnd::InvalidateRect(), 90, 111, 112, 124
- CWnd::InvalidateRgn(), 90
- CWnd::MessageBox(), 53
- CWnd::OnCompareItem(), 486
- CWnd::OnCreate(), 74, 149, 320
- CWnd::OnDeleteItem(), 486
- CWnd::OnDrawItem(), 486
- CWnd::OnLButtonDown(), 73, 110
- CWnd::OnLButtonUp(), 73, 111, 130

- CWnd::OnMeasureItem(), 486
- CWnd::OnMouseMove(), 73, 110, 124, 130, 139, 140
- CWnd::OnPaint(), 75, 90, 111-112, 115, 118, 124
- CWnd::OnSetCursor(), 74
- CWnd::OnSize(), 75
- CWnd::OnTimer(), 52, 53, 74, 150
- CWnd::PreCreateWindow(), 316-317
- CWnd::SendDlgItemMessage(), 204, 205, 298
- CWnd::SetCapture(), 110
- CWnd::SetDlgItemInt(), 204
- CWnd::SetDlgItemText(), 204
- CWnd::SetFocus(), 9
- CWnd::SetTimer(), 53
- CWnd::SetWindowText(), 290, 322
- CWnd::ShowWindow(), 18, 19, 296-97
- CWnd::SizeToContent(), 489
- CWnd::SubclassDlgItem(), 487, 489
- CWnd::SubclassWindow(), 487
- CWnd::UpdateData(), 253, 264, 289, 290, 296, 306, 437
- CWordArray class, 397

D

- DACs, *See* digital-to-analog converters
- data encapsulation, 549
- data map, 263
- data transfer, 264-265, 288-291, 438
- data types, 22, 522
- DDB, *See* device dependent bitmaps
- DDV, *See* DoDataValidation()
- DDX, *See* DoDataExchange()
- DECLARE_DYNAMIC() macro, 330-331
- DECLARE_DYNCREATE() macro, 330-331
- DECLARE_MESSAGE_MAP() macro, 45
- DECLARE_SERIAL(), macro, 330-331
- default palette, 155
- ::DefDlgProc() API function, 12
- ::DefMDIChildProc() API function, 12
- ::DefWindowProc() API function, 12
- delete* operator, 547
- DeleteObject(),
 - API function, 30

- CGdiObject function, 109
- derived classes, 17
- device context,
 - drawing attributes, 78-79
 - drawing objects, 80, 83
 - for printing, 352-353
 - introduced, 78
 - mapping modes, 353-355
 - private, 114-115
 - setting attributes, 81-82
- device coordinates, 352
- device dependent bitmaps, 142, 148, 155
- device drivers, 12
- device independent bitmaps, 123, 142, 150, 155
- device units, 212, 353
- devices, 12
- dialog boxes,
 - common dialogs, 181-185
 - introduced, 5
 - modal, 54
 - modal vs. modeless, 180
 - modeless example, 292
 - styles, 188-190
- Dialog Editor,
 - button control property pages, 193-199
 - combo box property page, 237
 - dialog box property page, 187
 - edit control property page, 233
 - list box property page, 235
 - introduced, 179
 - setting tab order, 248
 - use of, 211-213, 349-350
- dialog resource, 180, 186
- DIALOG resource script,
 - creating, 210, 245
 - examples of, 224-225, 501-502
- dialog units, 212
- DIB, *See* device independent bitmaps
- dictionaries, 397
- digital-to-analog converters, 152

- display pixels, 352
- dithering, 95
- documents,
 - filing and retrieving, 358-359, 398-399
 - introduced, 311-314
 - multiple views, 362-365, 421, 424
- document template string, 339-340
- document templates,
 - adding, 475, 478
 - defined, 327-328, 432
 - resources, 480
 - RUNTIME_CLASS() macro, 330
 - selecting, 455, 479, 482
 - using multiple, 455
- document-view architecture,
 - creation, 328-329
 - customizing window features, 315
 - designing document data, 340, 398, 459
 - designing user interface, 340, 344, 403, 411, 461
 - introduced, 311
 - structure, MDI, 423-424
 - structure, SDI, 313
 - traversing between classes, MDI, 425
 - traversing between classes, SDI, 328-329
- DoDataExchange(),
 - data transfer, 264-265, 288-291, 438
 - functions, 266
 - introduced, 205, 231
 - use of, 253, 263, 280, 438, 510
 - variable types, 267
 - with CFormView, 428, 437
- DoDataValidation(),
 - functions, 267-268
 - introduced, 231
 - validation limits, 352
 - with CFormView, 437
- drag-drop capability, 451
- drag-drop files, 361
- drawing attributes, 78
- drawing objects, deleting, 109

- dynamic allocation, 559
- dynamic data, 31
- dynamic instantiation, 559
- dynamic object creation, 330-331
- dynamic splitter windows, 362-365, 475-477
- dynamically allocated objects, 546

E

- edit controls, 232-234

- Elip application,

- adding dynamic splitter, 362-364
 - adding filing, 358-360
 - adding printing, 352-355
 - changing document template string, 339-340
 - coding handler functions, 347-348
 - creating dialog class, 349-350
 - designing document class, 341-343
 - designing menu, 346-347
 - designing user interface, 344-346
 - overview, 336-337
 - starter application, 337-338
 - synchronizing multiple views, 365-366

- ElipMdl application,

- making dialog modeless, 379-383
 - modified DIALOG resource, 383

- ElipsMin application,

- Elip.cpp file, 368-369
 - Elip.h file, 368
 - Elip.rc file, 377-378
 - ElipDoc.cpp file, 374-376
 - ElipDoc.h file, 373-374
 - ElipView.cpp file, 370-373
 - ElipView.h file, 369-370
 - MainFrm.cpp file, 369
 - MainFrm.h file, 369
 - overview, 367
 - SizeDlg.cpp file, 376-377
 - SizeDlg.h file, 376
 - resource.h file, 377

- encapsulation, 568

- end menu item, 41

- END_MESSAGE_MAP() macro, 45-56
- erasing the client area, 93
- event, defined, 10
- example programs, *See* applications
- exclusive-nor,
 - drawing, 123-124
 - limitations, 131
- exclusive-or,
 - drawing, 123-124
 - limitations, 131
- extended window styles, 189-190

F

- FastDraw application,
 - mainfrm.cpp file, 127-130
 - mainfrm.h file, 125-127
 - overview, 123-125
- filing,
 - introduced, 335
 - storing to file, 341, 359-360, 433
 - retrieving from file, 341, 359-360, 433
 - using collection classes, 387, 398-399
- fixed child windows, 173
- FixedChd application,
 - mainfrm.cpp file, 175-176
 - mainfrm.h file, 175
 - overview, 173-175
- focus, 8
- font,
 - customized, 387
 - logical, 403, 406
 - values defining, 406
 - use of, 409-410
- Form application,
 - adding code to document class, 435-437
 - adding code to view class, 439-440
 - adding controls, 431
 - data transfer, 438
 - document class functions, 433-435
 - output, 440-442

- overview, 421-423, 427
- starter application, 429-430
- synchronizing views, 439
- view class functions, 437
- ForMin application,
 - ChildFrm.cpp file, 445
 - ChildFrm.h file, 445
 - Form.cpp file, 443-444
 - Form.h file, 443
 - Form.rc file, 449-450
 - FormDoc.cpp file, 448-449
 - FormDoc.h file, 447-448
 - FormView.cpp file, 446-447
 - FormView.h file, 445-446
 - Mainfrm.cpp file, 445
 - Mainfrm.h file, 444
 - overview, 442
 - resource.h file, 449
- frame window, 328
- free store, 547, 559

G

- GDI, *See* graphics device interface
- GDI objects, 80
- global functions,
 - AfxGetApp(), 29, 465-476
 - AfxRegisterWndClass(), 26, 28-29, 113-114, 166
 - defined, 26
- graphical user interface, 5
- Graphics application,
 - graphics.rc file, 106-107
 - mainfrm.cpp file, 100-106
 - mainfrm.h file, 99-100
 - overview, 98
 - resource.h file, 107
- graphics device interface, 11-13, 77
- graphics drawing,
 - example, 85-87
 - using private device context, 115-119, 315-316
- graphics output process, 108-109
- grayed state, 197

group box controls, 197-198
GUI, *See* graphical user interface

H

handler functions,
 compiler tool setup, 56-61
 CWnd functions, 72-75
 defined, 10-13, 52
 for opening files, 378
 for print preview, 379
 for file/print setup, 378
 for printing, 379
 introduced, 41
heap, 118
Hello application,
 hello.rc file, 87
 mainfrm.cpp file, 87
 mainfrm.h file, 86
 overview, 85-86
 resource.h file, 87
HelloB application,
 mainfrm.cpp file excerpts, 89
 overview, 88-89
hourglass cursor,
 identifier, 35
 use of, 61-69
Hungarian Notation, 22, 65, 523

I

icon,
 customizing, 34
 loading, 37
 mainframe, 319
IDCANCEL, 289, 291, 307, 510
IDE, *See* Integrated Development Environment
IDDE, *See* Integrated Development and Debugging Environment
IDOK, 289, 307, 510
IDR_MAINFRAME,
 icon, 319
 menu, 319, 346
 string, 339, 389, 404

- toolbar bitmap, 469
- use in document template, 328, 432
- image list, 504
- IMPLEMENT_DYNAMIC() macro, 330-331
- IMPLEMENT_DYNCREATE() macro, 330-331
- IMPLEMENT_SERIAL() macro, 330-331
- indirect storage, 551
- Integrated Development Environment, 620
- Integrated Development and Debugging Environment, 636
- initialization, 571
- interface elements, 5
- internationalization, 270

K

- Kernel, 10
- keyboard alternatives, 48
 - See also* concatenated keypresses

L

- list box control,
 - introduced, 232
 - styles, 235-236
 - use of, 240-241
- list control, 332-333
- list view control,
 - introduced, 493
 - styles, 497
 - use of, 494-495, 505
- ::LoadResource() API function, 155
- logbrush,
 - defined, 93-94
 - use of, 99-110
- logical palette, 154-155
- logical units, 354, 406
- logpens,
 - defined, 91-92
 - use of, 99-110
- LPCREATESTRUCT structure, 364

M

- main window, 6, 12

- See also* window; mainframe window
- mainframe class, 14, 34
- mainframe window,
 - See also* window; main window
 - customizing, 316-319
 - resources, 318-319
- mainframe-application structure, 313
- MAKEINTRESOURCE, use of, 47
- making a class serializable, 344
- mapping mode,
 - introduced, 88, 353
 - FixedScale, 354
 - MM_TEXT, 88, 354
 - setting, 355
 - VariableScale, 354
- MDI child frame, 423-424
- MDI frame window, 423
- MDI, *See* Multiple Document Interface
- member access level,
 - private, 541
 - protected, 541
 - public, 541-542
- member initializers, 574
- member variable initialization, 263
- memory device context,
 - introduced, 77
 - procedure for using, 138-139
 - use of, 143-148
- menu bars, 6, 41
- menu item,
 - adding, 346
 - end menu, 41, 48
 - identifier, 46
 - introduced, 41
 - submenus, 41
 - top-level, 41
- MenuA application,
 - mainfrm.cpp file, 43-44
 - mainfrm.h file, 43
 - overview, 42

- script.rc file, 44-45
- resource.h file, 45
- MenuB application,
 - mainfrm.cpp file, 66-68
 - mainfrm.h file, 65-66
 - menub.rc file, 69
 - overview, 61-65
 - resource.h file, 69
- menus,
 - changing, 61-65
 - descriptive ID, 48
 - end menu items, 48
 - introduced, 41
 - popup, 57-58
 - resource scripts, 47-48
 - sub-menus, 41
- message box,
 - button style list, 55
 - example, 42
 - function definition, 54-56
 - icon specifier list, 55
 - modifier list, 55
 - return value, 55
- message handler function, *See* handler function
- message loop, 15
- message map entries,
 - adding, 56
 - for button controls, 239
 - for combo box controls, 239
 - for edit controls, 239
 - for list box controls, 239
 - for control notification messages, 203
 - for user messages, 170, 298, 380
 - introduced, 41
 - ON_COMMAND() macro, 46
 - ON_MESSAGE() macro, 170, 298
 - ON_NOTIFY() macro, 507
 - ON_NOTIFY_RANGE() macro, 507
 - ON_UPDATE_COMMAND_UI() macro, 281
- message map table, 46

- message maps,
 - defined, 46
 - introduced, 11, 45
 - ClassExpress, 56-60
 - ClassWizard, 56-60
- message response functions, *See* handler functions
- messages,
 - See also* command messages; ordinary messages
 - data members, 10
 - defined, 10
 - command routing, 314, 321, 423
- MessageBox(),
 - API function, 52
 - CWnd function, 53
- MFC, *See* Microsoft Foundation Class Library
- Microsoft Foundation Class Library,
 - class hierarchy, 520-521
 - class notation, 14
 - definition, 1
 - derived classes, 17
- Microsoft Visual C++ compiler tools,
 - AppStudio, 4, 577, 583-585, 596-598, 599, 607-608, 615-616
 - AppWizard, 4, 355, 577, 587-596, 610-615
 - ClassWizard, 4, 577, 586-587, 598, 599, 608-610, 616-617
- Microsoft Windows operating system, 1
 - See also* Windows operating system
- modal dialog,
 - construction, 235, 240, 289
 - introduced, 54
 - system modal, 180
 - task modal, 180
- ModalCom application,
 - mainfrm.cpp file, 283-284
 - mainfrm.h file, 283-284
 - ModalDlg.cpp file, 287
 - ModalDlg.h file, 286-287
 - MyApp.cpp file, 283
 - MyApp.h file, 283
 - overview, 278-280
 - resource.h file, 287

- script1.rc file, 287-288
- modeless dialog boxes,
 - creation, 295
 - defined, 172, 180, 277
 - use of, 292, 379-380
- Modeless application,
 - mainfrm.cpp file, 299-302
 - mainfrm.h file, 299
 - Modeless.cpp file, 303-305
 - Modeless.h file, 302-303
 - overview, 292-294
 - script1.rc file, 306
- Most Recently Used file list, 361, 379
- MRU file list, *See* Most Recently Used file list
- Multiple Document Interface,
 - accelerators, 425, 427
 - child window, 421
 - class structure, 424
 - creating new views, 425
 - features, 421, 425
 - introduced, 12, 328
 - keyboard accelerators, 427
 - menus, 426
 - minimized child window, 422
 - multiple documents, 456
 - multiple views, 456
 - use of, 421-422, 455-456
- multiple view classes,
 - creating, 392-394
 - using, 391
- multiple views,
 - of different classes, 387-388
 - of single document, 335, 362-365
 - synchronizing, 366
- multithreading, 3
- MyApp1 application,
 - mainfrm.cpp file, 18
 - mainfrm.h file, 18
 - MyApp.cpp file, 17-18
 - MyApp.h file, 17

- overview, 16-17
- MyApp2 application,
 - mainfrm.cpp file, 24
 - overview, 23
- MyApp3 application,
 - mainfrm.cpp file, 27-28
 - overview, 27
- MyApp4 application,
 - mainfrm.cpp file, 35
 - MyApp4.rc file, 35
 - overview, 34-35
- MyApp5 application,
 - mainfrm.cpp file, 36-37
 - MyApp5.rc file, 37
 - overview, 36
 - resource.h file, 37

N

- new document, data initialization, 361
- new* operator, use of, 19
- NewCmnCtrls application, 494-507
- NOINVERT keyword, 51
- non-client area, 6
- notification, *See* control notification messages
- NT, *See* Windows NT operating system

O

- object,
 - dynamic allocation, 559
 - dynamic deinstantiation, 561
 - dynamic-instantiaition, 559-563
 - dynamic-instantiation of arrays, 561
 - instance, 540
 - instantiation, 542, 553-557
- ObjectWindows Class Library, 635
- ON_COMMAND() macro, 46
- ON_MESSAGE() macro, 170, 298
- ON_NOTIFY() macro, 507
- ON_NOTIFY_RANGE() macro, 507
- ON_UPDATE_COMMAND_UI() macro, 281
- optional code, 312

- ordinary messages,
 - defined, 46
 - message map entry for, 46
- overloaded constructors, 37
- overloaded functions,
 - defined, 547
 - signature, 547
- overriding, 565
- OWL, *See* ObjectWindows Class Library
- owner-draw controls, 198, 485
- OwnerDC application,
 - mainfrm.cpp file, 116-117
 - mainfrm.h file, 116
 - overview, 115
 - OwnerDC.rc file, 117
 - resource.h file, 117

P

- PALETTERGB, 155
- pane-splitting controls, 362
- parent window, 159
- pens,
 - deleting, 109
 - logpen, 91-92
 - overview, 90-91
 - selecting, 81, 82, 108
 - stock pens, 83
 - styles, 91
- peripheral hardware, 12
- permissive task switching, 3
- popup menus, 57-58, 282
- PopUp application,
 - mainfrm.cpp file excerpts, 173
 - overview, 171-173
- popup window,
 - behavior, 171
 - defined, 159
- print preview, 357
- print setup, 357-358
- printer device context, 320
- printing, 335, 352, 355

- private inheritance, 563
- progress control, 494-495
- Project Workspace, 578
- project,
 - AppExpress, 627-632
 - Application, 578-583, 600-606, 636 thru 642
 - AppWizard, 587-596, 610-615
- property page, *See* Dialog Editor
- property sheets,
 - creating, 509-510
 - introduced, 508
 - modal, 509
 - modeless, 509
- PropertySheet application,
 - creating property sheet, 509-512
 - mainfrm.cpp file, 513
 - overview, 508-509
 - Property.cpp file, 515-516
 - Property.h file, 513-515
 - resource.h file, 516
 - script1.rc file, 516
- pure virtual function, 108

R

- radio button controls, 193-194
- radio button group, 212, 280
- raster drawing mode, 95-97
- reference, 575-576
- registering a new window class, 14, 25, 27, 319
- ::ReleaseCapture() API function, 111
- resource compiler, 32-33
- resource data,
 - defined, 31
 - list of, 32
- resource files, 34-37
- resource identifier argument, 47
- resource script file, 31-32
- resource scripts for controls, 227
- Resource Workshop, 635, 642 thru 644
 - See also* Borland C++ compiler tools

- ResourceStudio,
 - See also* Symantec compiler tools
 - introduced, 5, 619
 - use of, 624-626, 632-633
- retrieving from file, 341
- RGB macro, 94-95
- RGB color values, 97
- RGBQUAD structure, 150-151
- ROP2 Codes, 96
- run-time class information, 330-331
- RUNTIME_CLASS macro, 330

S

- sample programs, *See* applications
- scope-resolution notation, 549
- scroll bars, 6, 468
- SDI, *See* Single Document Interface
- serialization,
 - conditions, 342
 - data types supported, 359
 - defined, 330-331
- ::SetCursorPos() API function, 71
- ::SetFocus() API function, 9
- Single Document Interface,
 - class structure, 313-314
 - command message routing, 314
 - document template, 327
- slider control, 495
- slider control styles, 496
- spin button control, 495
- spin button control styles, 496
- splitter bars, 362, 394
- splitter box, 362
- splitter intersection, 362
- splitter windows,
 - adding, 475
 - creation, 394
 - dynamic, 362, 364, 387, 425
 - static, 362, 387, 392, 394
- static controls, 198-200
- static data, 31

- static objects, 557
- static storage class, 546
- status bars,
 - customizing, 466-468
 - message line panes, 464-465
 - standard setup, 464
 - status indicator panes, 464-465
- stock cursor shapes, 30
- stock icon shapes, 31
- string table, 271-273
- String Table, Resource Editor, 270
- strings, adding, 403
- submenus, 41
- Symantec C++ compiler tools,
 - AppExpress, 5, 627-632
 - ClassExpress, 5, 56-60, 619, 626
 - ResourceStudio, 5, 619, 624-626, 632-633
- system palette,
 - default palette, 153-154
 - loading, 154-155

T

- Tac application,
 - adding static splitter, 394-396
 - creating multiple view classes, 391-394
 - designing the document, 398-402
 - designing the views, 403-405, 409-418
 - overview, 388
 - starter application, 389-390
- TEXTMETRIC structure, 408
- this* keyword, 544
- timers,
 - animation, 143, 149-150
 - introduced, 42
 - setting, 53-54
- toolbar,
 - customizing, 455, 469
 - editing, 469-471, 473
 - resource, 472
- tooltips, 470

- top-level menu items, 41
- TRACE macro,
 - defined, 432
 - introduced, 384
 - output, 441-442
 - use of, 422, 437-438, 440
- tree control, 332-333
- tree view control, 493-496
- tree view control styles, 498

U

- uparrow cursor,
 - identifier, 35
 - use of, 27-28, 61-19
- update command messages, 281-282
- update handler functions, 281-282
- user input, 8-9
- user messages,
 - defined, 169
 - handler function, 170-171, 298, 380-381
 - introduced, 159
 - message map entry, 170, 298, 380
- User, 10
- user-defined message, *See* user messages
- UsrInput application,
 - adding dialog class, 249-255
 - adding message map entries, 249-250
 - creating DIALOG resource, 244-248
 - mainfrm.cpp file, 257-258
 - mainfrm.h file, 257
 - main window code, 241-244
 - Myapp.cpp file, 257
 - Myapp.h file, 256-257
 - overview, 240-241
 - resource.h file, 261
 - script1.rc file, 261-262
 - UsrDlg.cpp file, 259-261
 - UsrDlg.h file, 258-259
- UsrStrng application, 270-273
 - creating string table, 270-271
 - excerpts from UsrDlg.cpp file, 272

- excerpts from script1.rc file, 272
- overview, 270
- resource.h file, 273

V

- value functions, for member objects, 265
- view class, 311
- view window, 173, 328
- views,
 - customizing, 319
 - synchronizing, 439
- viewport extent, 354-355
- viewport origin, 88-90
- virtual functions, 566
- virtual key code, 50-51
- virtual keyword, 565-567
- virtual map, 263
- Visual C++, 4, 577, 599

W

- Win3.1x, *See* Windows 3.1x operating system
- Win32, 2
- Win95, *See* Windows 95 operation system
- window class,
 - defined, 26
 - registering, 25
 - styles, 28-29
- window messages, *See* ordinary messages
- window registration class, 21
- window style attributes, 14
- window styles,
 - child, 168
 - extended, 189-190
 - for dialogs, 186
 - for Win3.1x common controls, 191-192
 - list of, 25
 - use of, 24
- window,
 - client area, 6
 - defined, 5-6
 - non-client area, 6

- non-client elements, 7
- optional elements, 6
- styles, 25
 - extended styles, 189-190
- Windows 3.1x operating system, 2-4
- Windows 95 operating system, 2-4
- Windows NT operating system, 2-4
- Windows OS,
 - See also* Windows
 - introduced, 1-2
 - major components, 11-12
 - Windows 3.1x, 2-4
 - Windows 95, 2-4
 - Windows NT, 2-4
- Windows, 1
 - See also* Windows OS
- WM_COMMAND message, 45, 49, 52, 314, 321, 423
- WM_CREATE message, 74, 149, 320
- WM_DESTROY message, 169, 175, 297
- WM_INITDIALOG message, 249, 252, 502
- WM_LBUTTONDOWN message, 73, 416
- WM_LBUTTONUP message, 73, 110
- WM_MOUSEMOVE message, 73, 110
- WM_NOTIFY message, 507
- WM_PAINT message, 75, 90, 110-111, 115, 118, 169, 290, 320, 437
- WM_SETCURSOR message, 74
- WM_SIZE message, 75, 175
- WM_TIMER message, 42, 52-53, 74, 149
- WM_USER message, 169-170
- WS_BORDER style, 232
- WS_CAPTION style, 188
- WS_CHILD style, 167, 172, 232
- WS_DISABLED style, 232
- WS_GROUP style, 167
- WS_HSCROLL style, 232
- WS_POPUP style, 167, 171-173, 188
- WS_SYSMENU style, 188
- WS_TABSTOP style, 167, 232
- WS_VISIBLE style, 188, 232
- WS_VSCROLL style, 232

[Table of Contents](#)

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

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

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