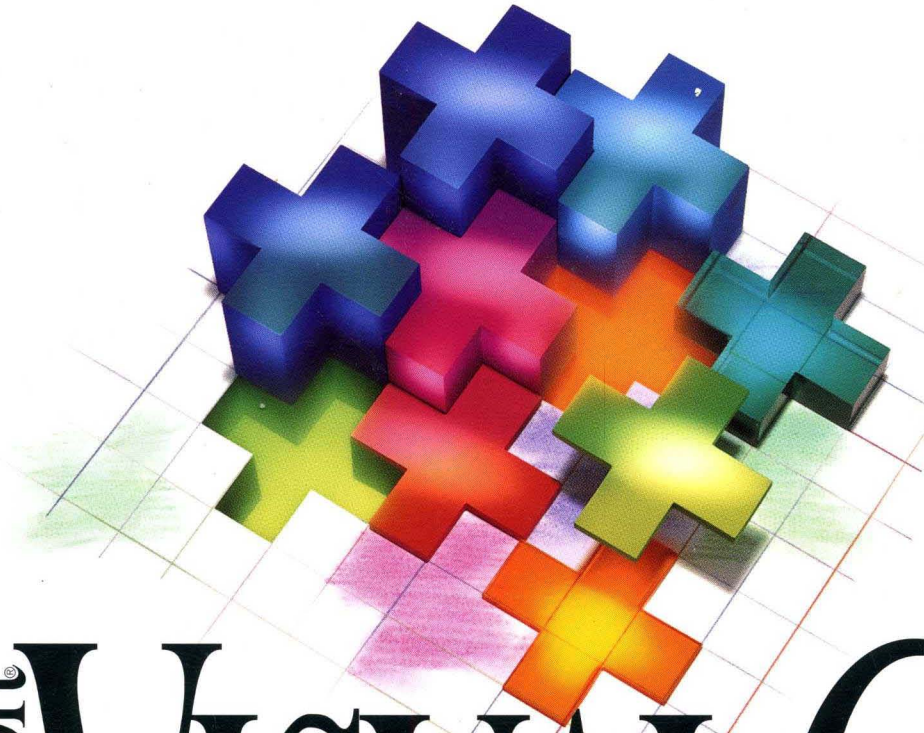
Programmer's Guides

C++ Tutorial

Class Library User's Guide

Programming Techniques



Microsoft® VISUAL C++™

Development System for Windows™

Programmer's Guides

Microsoft® Visual C++™

Development System for Windows™
Version 1.0

This volume contains three separate books:
C++ Tutorial
Class Library User's Guide (for the Microsoft Foundation Class Library)
Programming Techniques

Microsoft Corporation

Document No. DB29682-0193
Printed in the United States of America.

C++ Tutorial

Contents

Part 2 Classes

Part 3 Object-Oriented Design

Figures and Tables

Figures

Tables

Introduction

The *C++ Tutorial* provides an introduction to the C++ language and object-oriented programming. This book assumes you are familiar with C, and therefore doesn't cover the parts of the C++ language that are also found in C. In some places, this book compares C++ with C in order to demonstrate how the same problem might be solved in each language.

This book is not an exhaustive description of the C++ language. It introduces the major features of C++ and gives examples of how they can be used. For more detailed information on C++, see the *C++ Language Reference*.

About This Book

The following list summarizes the book's contents:

- Part 1, "Introduction to C++," describes some of the simple enhancements that C++ has made to C. These features are not object-oriented, but they provide conveniences that C programmers can readily appreciate.
- Part 2, "Classes," covers the most important elements of C++: classes, inheritance, and polymorphism. These features are what make C++ an object-oriented language.
- Part 3, "Object-Oriented Design," covers the conceptual aspects of object-oriented programming. This section describes how to design an object-oriented program and provides an in-depth example.

You should read the chapters in order, because each one assumes that you know the material covered in previous chapters.

Document Conventions

This book uses the following document conventions.

Example	Description
STDIO.H	Uppercase letters indicate filenames, segment names, registers, and terms used at the MS-DOS command level.
printf	Boldface letters indicate C or C++ keywords, operators, language-specific characters, and library functions. Within discussions of syntax, bold type indicates that the text must be entered exactly as shown.
expression	Words in italic indicate placeholders for information you must supply, such as a filename. Italic type is also occasionally used for emphasis in the text.

`void main()`
`{`
`}`
	This font is used for example programs, program fragments, and the names of user-defined functions and variables. It also indicates user input and screen output.

`while()`
`{`
.
.
.
`}`
	A vertical ellipsis tells you that part of the example program has been intentionally omitted.

"inheritance"	Quotation marks enclose a new term the first time it is defined in text.
American National Standards Institute (ANSI)	The first time an acronym is used, it is often spelled out.

PART 1

Introduction to C++

CHAPTER 1

A First Look at C++

The C++ language is derived from C. With few exceptions, it is a superset of C, meaning that everything available in C is also available in C++. C++ adds some simple enhancements to C's own features and some major new features that don't exist in C.

This chapter covers some of the differences in conventions between C and C++. It begins with a new way of handling input and output, which you'll need to know for later programs that print results on the screen. This chapter also covers C++ comments and function prototypes.

Using Streams for Input and Output

Here is HELLO.CPP, a very simple C++ program:

```
#include <iostream.h>
void main()
{
    cout << "Hello, world\n";
}
```

This program is the C++ version of the C program HELLO.C. However, instead of including STDIO.H, the program includes IOSTREAM.H, and instead of a **printf** call, it uses an unfamiliar syntax with an undefined variable named **cout**, the bitwise left shift operator (<<), and a string.

The Standard Output Stream

In C++, there are facilities for performing input and output known as "streams." The example programs throughout this book use streams to read and display information. The name **cout** represents the standard output stream. You can use **cout** to display information:

```
cout << "Hello, world\n";
```

The string "`Hello, world\n`" is sent to the standard output device, which is the screen. The << operator is called the "insertion" operator. It points from what is being sent (the string) to where it is going (the screen).

Suppose you want to print an integer instead of a string. In C, you would use **printf** with a format string that describes the parameters:

```
printf( "%d", amount );
```

In C++, you don't need the format string:

```
#include <iostream.h>

void main()
{
   int amount = 123;
   cout << amount;
}
```

The program prints 123.

You can send any built-in data types to the output stream without a format string. The **cout** stream is aware of the different data types and interprets them correctly.

The following example shows how to send a string, an integer, and a character constant to the output stream using one statement.

```
#include <iostream.h>

void main()
{
   int amount = 123;
   cout << "The value of amount is " << amount << '.';
}
```

This program sends three different data types to **cout**: a string literal, the integer `amount` variable, and a character constant `'.'` to add a period to the end of the sentence. The program prints this message:

```
The value of amount is 123.
```

Notice how multiple values are displayed using a single statement: The << operator is repeated for each value.

Formatted Output

So far, the examples haven't sent formatted output to **cout**. Suppose you want to display an integer using hexadecimal instead of decimal notation. The **printf** function handles this well. How does **cout** do it?

Note Whenever you wonder how to get C++ to do something that C does, remember that the entire C language is part of C++. In the absence of a better way, you can revert to C.

C++ associates a set of "manipulators" with the output stream. They change the default format for integer arguments. You insert the manipulators into the stream to make the change. The manipulators' names are **dec**, **oct**, and **hex**.

The next example shows how you can display an integer value in its three possible configurations.

```
#include <iostream.h>

main()
{
   int amount = 123;
   cout << dec << amount << ' '
        << oct << amount << ' '
        << hex << amount;
}
```

The example inserts each of the manipulators (**dec**, **oct**, and **hex**) to convert the value in amount into different representations.

This program prints this:

```
123 173 7b
```

Each of the values shown is a different representation of the decimal value 123 from the `amount` variable.

The Standard Error Stream

To send output to the standard error device, use **cerr** instead of **cout**. You can use this technique to send messages to the screen from programs that have their standard output redirected to another file or device.

The Standard Input Stream

In addition to printing messages, you may want to read data from the keyboard. C++ includes its own version of standard input in the form of **cin**. The next example shows you how to use **cin** to read an integer from the keyboard.

```
#include <iostream.h>

void main()
{
   int amount;
   cout << "Enter an amount...\n";
   cin >> amount;
   cout << "The amount you entered was " << amount;
}
```

This example prompts you to enter an amount. Then **cin** sends the value that you enter to the variable `amount`. The next statement displays the amount using **cout** to demonstrate that the **cin** operation worked.

You can use **cin** to read other data types as well. The next example shows how to read a string from the keyboard.

```
#include <iostream.h>

void main()
{
   char name[20];
   cout << "Enter a name...\n";
   cin >> name;
   cout << "The name you entered was " << name;
}
```

The approach shown in this example has a serious flaw. The character array is only 20 characters long. If you type too many characters, the stack overflows and peculiar things happen. The **get** function solves this problem. It is explained in the *iostream Class Library Reference*. For now, the examples assume that you will not type more characters than a string can accept.

Note Recall that **printf** and **scanf** are not part of the C language proper but are functions defined in the run-time library. Similarly, the **cin** and **cout** streams are not part of the C++ language. Instead, they are defined in the stream library, which is why you must include IOSTREAM.H in order to use them. Furthermore, the meaning of the << and >> operators depends on the context in which they are used. They can display or read data only when used with **cout** and **cin**.

C++ Comments

C++ supports the C comment format where /* begins a comment and */ ends it. But C++ has another comment format, which is preferred by many programmers. The C++ comment token is the double-slash (//) sequence. Wherever this sequence appears (unless it is inside a string), everything to the end of the current line is a comment.

The next example adds comments to the previous program.

```
// C++ comments
#include <iostream.h>

void main()
{
   char name[20];                // Declare a string
   cout << "Enter a name...\n";  // Request a name
   cin >> name;                  // Read the name
   cout << "The name you entered was " << name;
}
```

Function Prototypes

In standard C, you can declare a function before you define it. The declaration describes the function's name, its return value, and the number and types of its parameters. This feature, called a "function prototype," allows the compiler to compare the function calls to the prototype and to enforce type checking.

C does not require a prototype for each function. If you omit it, at worst you get a warning. C++, on the other hand, requires every function to have a prototype.

The next example uses a function named `display` to print "Hello, world."

```
// A program without function prototypes
// Note: This will not compile.
#include <iostream.h>

void main()
{
   display( "Hello, world" );
}

void display( char *s )
{
   cout << s;
}
```

Because the `display` function has no prototype, this program does not survive the syntax-checking phase of the C++ compiler.

The next example adds a function prototype to the previous program. Now the program compiles without errors.

```
// A program with a function prototype
#include <iostream.h>

void display( char *s );

void main()
{
   display( "Hello, world" );
}

void display( char *s )
{
   cout << s;
}
```

In some C programs, the function definitions declare the types of the parameters between the parameter list and the function body. C++ requires that function definitions declare the types of all parameters within the parameter list. For example:

```
void display( char *s )  // New style required in C++
{
   cout << s;
}

void display( s )        // Error: old style doesn't work
char *s
{
   cout << s;
}
```

If you define a function before you call it, you don't need a separate prototype; the function definition acts as the prototype. However, if you don't define the function until after you call it, or if the function is defined in another file, you must provide a prototype.

Keep in mind that the prototype requirement is an exception to the general rule that a C++ compiler can handle a C program. If your C programs do not have function prototypes and new-style function-declaration blocks, then you must add them before compiling the programs in C++.

Note If you need to generate new-style function prototypes for existing C programs, use the CL.EXE program with the /Zg option. See Chapter 1, "CL Command Reference," in *Command-Line Utilities User's Guide* for more details.

CHAPTER 2

C++ Enhancements to C

This chapter introduces some simple enhancements and improvements that C++ offers the C programmer. New features covered in this chapter include the following:

- Default function arguments
- More flexible placement of variable declarations
- The scope resolution operator
- Inline functions
- The **const** keyword
- Enumerations
- Function overloading

This chapter also describes how to link C and C++ modules together.

Default Function Arguments

A C++ function prototype can list default values for some of the parameters. If you omit the corresponding arguments when you call the function, the compiler automatically uses the default values. If you provide your own arguments, the compiler uses them instead of the defaults. The following prototype illustrates this feature:

```
void myfunc( int i = 5, double d = 1.23 );
```

Here, the numbers 5 and 1.23 are the default values for the parameters. You could call the function in several ways:

```
myfunc( 12, 3.45 );  // Overrides both defaults
myfunc( 3 );         // Same as myfunc( 3, 1.23 )
myfunc();            // Same as myfunc( 5, 1.23 )
```

If you omit the first argument, you must omit all arguments that follow. You can omit the second argument, however, and override the default for the first. This rule applies to any number of arguments. You cannot omit an argument unless you omit all the arguments to its right. For example, the following function call is illegal:

```
myfunc( , 3.5 );   // Error: cannot omit only first argument
```

A syntax like this is error-prone and makes reading and writing function calls more difficult.

The following example uses default arguments in the `show` function.

```
// DEFARG.CPP: A program with default arguments in a function prototype
#include <iostream.h>

void show( int = 1, float = 2.3, long = 4 );

void main()
{
   show();                 // All three arguments default
   show( 5 );              // Provide 1st argument
   show( 6, 7.8 );         // Provide 1st and 2nd
   show( 9, 10.11, 12L ); // Provide all three arguments
}

void show( int first, float second, long third )
{
   cout << "\nfirst = "  << first;
   cout << ", second = " << second;
   cout << ", third = "  << third;
}
```

When you run this example, it prints

```
first = 1, second = 2.3, third = 4
first = 5, second = 2.3, third = 4
first = 6, second = 7.8, third = 4
first = 9, second = 10.11, third = 12
```

Default values provide a lot of flexibility. For example, if you usually call a function using the same argument values, you can put them in the prototype and later call the function without supplying the arguments.

Placement of Variable Declarations

C requires you to declare variables at the beginning of a block. In C++, you can declare a variable anywhere in the code, as long as you declare it before you reference it. Using this feature, you can place the declaration of a variable closer to the code that uses it, making the program more readable.

The following example shows how you can position the declaration of a variable near its first reference.

```
// Declaring a variable near its first reference
#include <iostream.h>

void main()
{
   cout << "Enter a number: ";
   int n;
   cin >> n;
   cout << "The number is: " << n;
}
```

The freedom to declare a variable anywhere in a block makes expressions such as the following one possible:

```
for( int ctr = 0; ctr < MAXCTR; ctr++ )
```

However, you cannot have expressions such as the following:

```
if( int i == 0 )        // Error
    ;
while( int j == 0 )     // Error
    ;
```

Such expressions are meaningless, because there is no need to test the value of a variable the moment it is declared.

The following example declares a variable in a block.

```
// VARDECL.CPP: Variable declaration placement
#include <iostream.h>

void main()
{
   for( int lineno = 0; lineno < 3; lineno++ )
   {
       int temp = 22;
       cout << "\nThis is line number " << lineno
            << " and temp is " << temp;
   }
   if( lineno == 4 )   // lineno still accessible
       cout << "\nOops";
   // Cannot access temp
}
```

This example produces the following output:

```
This is line number 0 and temp is 22
This is line number 1 and temp is 22
This is line number 2 and temp is 22
```

Note that the two variables `lineno` and `temp` have different scopes. The `lineno` variable is in scope for the current block (in this case, until **main** ends) and all blocks subordinate to the current one. Its scope, however, begins where the declaration appears. C++ statements that appear before a variable's declaration cannot refer to the variable even though they appear in the same block. The `temp` variable, however, goes out of scope when the **for** loop ends. It is accessible only from within the loop.

You should exercise care when declaring variables in places other than the beginning of a block. If you scatter declarations haphazardly throughout your program, a person reading your program may have difficulty finding where a variable is declared.

The Scope Resolution Operator

In C, a local variable takes precedence over a global variable with the same name. For example, if both a local variable and a global variable are called `count`, all occurrences of `count` while the local variable is in scope refer to the local variable. It's as if the global variable becomes invisible.

In C++, you can tell the compiler to use the global variable rather than the local one by prefixing the variable with ::, the scope resolution operator. For example:

```
// SCOPERES.CPP: Scope resolution operator
#include <iostream.h>

int amount = 123;    // A global variable

void main()
{
   int amount = 456; // A local variable

   cout << ::amount; // Print the global variable
   cout << '\n';
   cout << amount;   // Print the local variable
}
```

The example has two variables named `amount`. The first is global and contains the value 123. The second is local to the **main** function. The two colons tell the compiler to use the global `amount` instead of the local one. The program prints this on the screen:

```
123
456
```

Note that if you have nested local scopes, the scope resolution operator doesn't provide access to variables in the next outermost scope. It provides access to only the global variables.

The scope resolution operator gives you more freedom in naming your variables by letting you distinguish between variables with the same name. However, you shouldn't overuse this feature; if two variables have different purposes, their names should reflect that difference.

Inline Functions

C++ provides the **inline** keyword as a function qualifier. This keyword causes a new copy of the function to be inserted in each place it is called. If you call an inline function from 20 places in your program, the compiler inserts 20 copies of that function into your .EXE file.

Inserting individual copies of functions eliminates the overhead of calling a function (such as loading parameters onto the stack) so your program runs faster. However, having multiple copies of a function can make your program larger.

You should use the **inline** function qualifier only when the inserted function is very small or is called from few places.

Inline functions are similar to macros declared with the **#define** directive; however, inline functions are recognized by the compiler, while macros are implemented by a simple text substitution. One advantage of this is that the compiler performs type checking on the parameters of an inline function. Another advantage is that an inline function behaves just as an ordinary function does, without any of the side effects that macros have. For example:

```
// INLINE.CPP: A macro vs. an inline function
#include <iostream.h>

#define MAX( A, B ) ((A) > (B) ? (A) : (B))

inline int max( int a, int b )
{
    if ( a > b ) return a;
    return b;
}

void main()
{
   int i, x, y;

   x = 23; y = 45;
   i = MAX( x++, y++ );  // Side-effect:
                         //     larger value incremented twice
   cout << "x = " << x << " y = " << y << '\n';

   x = 23; y = 45;
   i = max( x++, y++ );  // Works as expected
   cout << "x = " << x << " y = " << y << '\n';
}
```

This example prints the following:

```
x = 24 y = 47
x = 24 y = 46
```

If you want a function like `max` to accept arguments of any type, the way a macro does, you can use overloaded functions. These are described in the section "Overloaded Functions" on page 20.

To be safe, always declare inline functions before you make any calls to them. If an inline function is to be called by code in several source files, put its declaration

in a header file. Any modifications to the body of an inline function require recompilation of all the files that call that function.

The **inline** keyword is only a suggestion to the compiler. Functions larger than a few lines are not expanded inline even if they are declared with the **inline** keyword.

Note The Microsoft C/C++ compiler supports the **__inline** keyword for C, which has the same meaning as **inline** does in C++.

The const Qualifier

C++, like C, supports the **const** qualifier, which turns variables into constants. In C, the **const** qualifier specifies that a variable is read-only, except during its one-time initialization. Only through initialization can a program specify a **const** variable's value. C++ goes a step further and treats such variables as if they are true constant expressions (such as 123). Wherever you can use a constant expression, you can use a **const** variable. For example:

```
// CONST.CPP: The const qualifier
#include <iostream.h>

void main()
{
   const int SIZE = 5;
   char cs[SIZE];

   cout << "The size of cs is " << sizeof cs;
}
```

This program is illegal in C, because C does not let you use a **const** variable to specify the size of an array. However, even in C++ you cannot initialize a **const** variable with anything other than a constant expression. For example, even though `SIZE` is declared within a function, you cannot initialize it with a parameter of the function. This means you cannot use **const** to declare an array whose size is determined at run time. To dynamically allocate an array in C++, see Chapter 5, "Classes and Dynamic Memory Allocation."

You can use **const** declarations as a replacement for constants defined with the **#define** directive. C++ lets you place **const** declarations in header files, which is illegal in C. (If you try doing this in C, the linker generates error messages if the header file is included by more than one module in a program.) Constants

declared with **const** have an advantage over those defined by **#define** in that they are accessible to a symbolic debugger, making debugging easier.

You can also use **const** in pointer declarations. In such declarations, the placement of **const** is significant. For example:

```
char *const ptr = mybuf;  // const pointer
*ptr = 'a';       // Change char that p points to; legal
ptr = yourbuf;    // Change pointer; error
```

This declares `ptr` as a constant pointer to a string. You can modify the string that `ptr` points to, but you cannot modify `ptr` itself by making it point to another string.

However, the following declaration has a different meaning:

```
const char *ptr = mybuf;  // Pointer to const
ptr = yourbuf;    // Change pointer; okay
*ptr = 'a';       // Change char that p points to; error
```

This declares `ptr` as a pointer to a constant string. You can modify `ptr` itself so that it points to another string, but you cannot modify the string that `ptr` points to. In effect, this makes `ptr` a "read-only" pointer.

You can use **const** when declaring a function to prevent the function from modifying one of its parameters. Consider the following prototype:

```
// Node is a large structure
int readonly( const struct Node *nodeptr );
```

This prototype declares that the `readonly` function cannot modify the `Node` structure that its parameter points to. Even if an ordinary pointer is declared inside the function, the parameter is still safeguarded, because you cannot assign a read-only pointer to an ordinary pointer. For example:

```
int readonly( const struct Node *nodeptr )
{
    struct Node *writeptr;     // Ordinary pointer

    writeptr = nodeptr;        // Error - illegal assignment
}
```

If such an assignment were legal, the `Node` structure could be modified through `writeptr`.

Enumerations

An enumeration is a user-defined data type whose values consist of a set of named constants. In C++, you declare an enumeration with the **enum** keyword, just as in C. In C, declarations of instances of an enumeration must include the **enum** keyword. In C++, an enumeration becomes a data type when you define it. Once defined, it is known by its identifier alone (the same as with any other type) and declarations can use the identifier name alone, without including the **enum** qualifier.

The following example demonstrates how a C++ program can reference an enumeration by using the identifier without the **enum** keyword.

```
// enum as a data type
#include <iostream.h>

enum color { red, orange, yellow, green, blue, violet };

void main()
{
    color myFavorite;

    myFavorite = blue;
}
```

Notice that the declaration of `myFavorite` uses only the identifier `color`; the **enum** keyword is unnecessary. Once `color` is defined as an enumeration, it becomes a new data type. (In later chapters, you'll see that classes have a similar property. When a class is defined, it becomes a new data type.)

Each element of an enumeration has an integer value, which, by default, is one greater than the value of the previous element. The first element has the value 0, unless you specify another value. You can also specify values for any subsequent element, and you can repeat values. For example:

```
enum color { red, orange, yellow, green, blue, violet );
// Values: 0, 1, 2, 3, 4, 5

enum day { sunday = 1, monday,
           tuesday, wednesday = 24,
           thursday, friday, saturday };
// Values: 1, 2, 3, 24, 25, 26, 27

enum direction { north = 1, south,
                 east = 1, west };
// Values: 1, 2, 1, 2
```

You can convert an enumeration into an integer. However, you cannot perform the reverse conversion unless you use a cast. For example:

```
// ENUM.CPP
enum color { red, orange, yellow, green, blue, violet };

void main()
{
    color myFavorite, yourFavorite;
    int i;

    myFavorite = blue;
    i = myFavorite;          // Legal; i = 4

 // yourFavorite = 5;        // Error: cannot convert
                             //     from int to color
    myFavorite = (color)4;   // Legal
}
```

Explicitly casting an integer into an enumeration is generally not safe. If the integer is outside the range of the enumeration or if the enumeration contains duplicate values, the result of the cast is undefined.

Overloaded Functions

Function overloading is a C++ feature that can make your programs more readable. For example, suppose you write one square root function that operates on integers, another square root function for floating-point variables, and yet another for doubles. In C, you have to give them three different names, even though they all perform essentially the same task. But in C++, you can name them all `square_root`. By doing so, you "overload" the name `square_root`; that is, you give it more than one meaning.

When you declare multiple functions with the same name, the compiler distinguishes them by comparing the number and type of their parameters. The following example overloads the `display_time` function to accept either a **tm** structure or a **time_t** value.

```
// OVERLOAD.CPP: Overloaded functions for different data formats
#include <iostream.h>
#include <time.h>

void display_time( const struct tm *tim )
{
   cout << "1. It is now " << asctime( tim );
}
void display_time( const time_t *tim )
{
   cout << "2. It is now " << ctime( tim );
}

void main()
{
   time_t tim = time( NULL );
   struct tm *ltim = localtime( &tim );

   display_time( ltim );
   display_time( &tim );
}
```

The example gets the current date and time by calling the **time** and **localtime** functions. Then it calls its own overloaded `display_time` function once for each of the formats. The compiler uses the type of the argument to choose the appropriate function for each call.

Depending on what time it is, the previous example prints something like this:

```
1. It is now Wed Jan 31 12:05:20 1992
2. It is now Wed Jan 31 12:05:20 1992
```

The different functions described by an overloaded name can have different return types. This makes it possible to have a `max` function that compares two integers and returns an integer, a `max` function that compares two floats and returns a float, and so on. However, the functions must also have different parameter lists. You cannot declare two functions that differ only in their return type. For example:

```
int search( char *key );
char *search( char *name );   // Error: has same parameter list
```

The compiler considers only the parameter lists when distinguishing functions with the same name.

You can also overload a name to describe functions that take different numbers of parameters but perform similar tasks. For example, consider the C run-time library functions for copying strings. The **strcpy** function copies a string from the

source to the destination. The **strncpy** function copies a string but stops copying when the source string terminates or after it copies a specified number of characters.

The following example replaces **strcpy** and **strncpy** with the single function name **string_copy**.

```
// An overloaded function
#include <iostream.h>
#include <string.h>

inline void string_copy( char *dest, const char *src )
{
    strcpy( dest, src );
}

inline void string_copy( char *dest, const char *src, int len )
{
    strncpy( dest, src, len );
}

static char stringa[20], stringb[20];

void main()
{
    string_copy( stringa, "That" );
    string_copy( stringb, "This is a string", 4 );
    cout << stringb << " and " << stringa;
}
```

This program has two functions named `string_copy`, which are distinguished by their different parameter lists. The first function takes two pointers to characters. The second function takes two pointers and an integer. The C++ compiler tells the two functions apart by examining their different parameter lists.

Default arguments can make one function's parameter list look like another's. Consider what happens if you give the second `string_copy` function a default value for the `len` parameter, as follows:

```
string_copy( char *dest, const char *src, int len = 10 );
```

In this case, the following function call is ambiguous:

```
string_copy( stringa, "That" );   // Error
```

This function call matches both the `string_copy` that takes two parameters and the one that takes three parameters with a default argument supplied. The compiler cannot tell which function should be called and gives an error.

You shouldn't overload a function name to describe completely unrelated functions. For example, consider the following pair:

```
void home();                 // Move screen cursor to ( 0, 0 )
char *home( char *name );    // Look up person's home address
                             // and return it as a string
```

Because these functions perform totally different operations, they should have different names.

Linkage Specifications

This next feature is not so much a C++ extension to C as a way to let the two languages coexist. A "linkage specification" makes C functions accessible to a C++ program. Because there are differences in the way the two languages work, if you call functions originally compiled in C you must inform the C++ compiler of that fact.

The following example uses a linkage specification to tell the C++ compiler that the functions in MYLIB.H were compiled by a C compiler.

```
// Linkage specifications
#include <iostream.h>

extern "C"
{                    // The linkage specification
#include "mylib.h"   // tells C++ that mylib functions
}                    // were compiled with C

void main()
{
   cout << myfunc();
}
```

The **extern "C"** statement says that everything in the scope of the braces is compiled by a C compiler. If you do not use the braces, the linkage specification applies only to the declaration that follows the **extern** statement on the same line.

You can also put the linkage specification in the header file that contains the prototypes for the C functions. You don't need to use the **extern "C"** statement when you're calling standard library functions because Microsoft includes the linkage specification in the standard C header files.

Sometimes, however, you need to use linkage specifications for other C header files. If you have a large library of custom C functions to include in your C++ program and you do not want to port them to C++, you must use a linkage

specification. For example, perhaps you have libraries, but not the original source code.

Occasionally, you need to tell the C++ compiler to compile a function with C linkages. You would do this if the function was to be called from another function that was itself compiled with C linkage.

The following example illustrates a function that is to be compiled with C linkage because it is called from a C function.

```
// LINKAGE.CPP: Linkage specifications
#include <iostream.h>
#include <stdlib.h>
#include <string.h>

// ------ Prototype for a C function
extern "C" int comp( const void *a, const void *b );

void main()
{
   // --------- Array of string pointers to be sorted
   static char *brothers[] = {
      "Frederick William",
      "Joseph Jensen",
      "Harry Alan",
      "Walter Elsworth",
      "Julian Paul"
   };
   // ---------- Sort the strings in alphabetical order
   qsort( brothers, 5, sizeof(char *), comp );
   // ---------- Display the brothers in sorted order
   for( int i = 0; i < 5; i++ )
      cout << '\n' << brothers[i];
}

// ---------- A function compiled with C linkage
extern "C"
{
   int comp( const void *a, const void *b )
   {
      return strcmp( *(char **)a, *(char **)b );
   }
}
```

This program calls the C **qsort** function to sort an array of character pointers. The **qsort** function expects you to provide a function that compares two items. But **qsort** is a C function, so you must provide a C-compatible `comp` function. Because this program is a C++ program, you must tell the C++ compiler to use C linkage for this function alone. Both the prototype and the function definition have the **extern "C"** linkage specification.

CHAPTER 3

References

This chapter explains how to use references, a new type of variable that C++ provides. References are primarily used to pass parameters to functions and return values back from functions. However, before you see how references are useful in those situations, you need to understand the properties of references. The first four sections of this chapter describe some characteristics of references, and the later sections explain the use of references with functions.

References as Aliases

You can think of a C++ reference as an alias for a variable—that is, an alternate name for that variable. When you initialize a reference, you associate it with a variable. The reference is permanently associated with that variable; you cannot change the reference to be an alias for a different variable later on.

The unary **&** operator identifies a reference, as illustrated below:

```
int actualint;
int &otherint = actualint;   // Reference declaration
```

These statements declare an integer named `actualint` and tell the compiler that `actualint` has another name, `otherint`. Now all operations on either name have the same result.

The following example shows how you can use a variable and a reference to that variable interchangeably:

```
// REFDEMO.CPP: The reference
#include <iostream.h>

void main()
{
   int actualint = 123;
   int &otherint = actualint;

   cout << '\n' << actualint;
   cout << '\n' << otherint;
   otherint++;
   cout << '\n' << actualint;
   cout << '\n' << otherint;
   actualint++;
   cout << '\n' << actualint;
   cout << '\n' << otherint;
}
```

The example shows that operations on `otherint` act upon `actualint`. The program displays the following output, showing that `otherint` and `actualint` are simply two names for the same item:

```
123
123
124
124
125
125
```

A reference is not a copy of the variable to which it refers. Instead, it is the same variable under a different name.

The following example displays the address of a variable and a reference to that variable.

```
// REFADDR.CPP: Addresses of references
#include <iostream.h>

void main()
{
   int actualint = 123;
   int &otherint = actualint;

   cout << &actualint << ' ' << &otherint;
}
```

When you run the program, it prints the same address for both identifiers, the value of which depends on the configuration of your system.

Note that the unary operator **&** is used in two different ways in the example above. In the declaration of `otherint`, the **&** is part of the variable's type. The variable `otherint` has the type **int &**, or "reference to an **int**." This usage is unique to C++. In the **cout** statement, the **&** takes the address of the variable it is applied to. This usage is common to both C and C++.

Initializing a Reference

A reference cannot exist without a variable to refer to, and it cannot be manipulated as an independent entity. Therefore, you usually initialize a reference, explicitly giving it something to refer to, when you declare it.

There are some exceptions to this rule. You need not initialize a reference in the following situations:

- It is declared with **extern**, which means it has been initialized elsewhere.
- It is a member of a class, which means it is initialized in the class's constructor function. (For more information on class constructor functions, see Chapter 4, "Introduction to Classes.")
- It is declared as a parameter in a function declaration or definition, which means its value is established when the function is called.
- It is declared as the return type of a function, which means its value is established when the function returns something.

As you work through the examples in this and later chapters, note that a reference is initialized every time it is used, unless it meets one of these criteria.

References and Pointers: Similarities and Differences

You can also view references as pointers that you can use without the usual dereferencing notation. In the first example in this chapter, the reference `otherint` can be replaced with a constant pointer, as follows:

```
int actualint = 123;
int *const intptr = &actualint;  // Constant pointer
                                 //     points to actualint
```

A declaration like this makes `*intptr` another way of referring to `actualint`. Consider the similarities between this and a reference declaration. Any assignments you make to `*intptr` affect `actualint`, and vice versa. As described earlier, a reference also has this property, but without requiring the *, or indirection operator. And because `*intptr` is a constant pointer, you cannot make it point to another integer once it's been initialized to `actualint`. Again, the same is true for a reference.

However, references cannot be manipulated as pointers can. With a pointer, you can distinguish between the pointer itself and the variable it points to by using the * operator. For example, `intptr` describes the pointer, while `*intptr` describes the integer being pointed to. Because you don't use a * with a reference, you can manipulate only the variable being referred to, not the reference itself.

As a result, there are a number of things that you cannot do to references themselves:

- Point to them
- Take the address of one
- Compare them
- Assign to them
- Do arithmetic with them
- Modify them

If you try to perform any of these operations on a reference, you instead act on the variable that the reference is associated with. For example, if you increment a reference, you actually increment what it refers to. If you take the address of a reference, you actually take the address of what it refers to.

Recall that with pointers, you can use the **const** keyword to declare constant pointers and pointers to constants. Similarly, you can declare a reference to a constant. For example:

```
int actualint = 123;
const int &otherint = actualint;   // Reference to constant int
```

This declaration makes `otherint` a read-only alias for `actualint`. You cannot make any modifications to `otherint`, only to `actualint`. The similarity to pointers does not go any further, however, because you cannot declare a constant reference:

```
int &const otherint = actualint;  // Error
```

This declaration is meaningless because all references are constant by definition.

As mentioned earlier, the first sections of this chapter are intended to demonstrate the properties of references, but not their purpose. The previous examples notwithstanding, references should *not* be used merely to provide another name for a variable. The most common use of references is as function parameters.

References as Function Parameters

In C, there are two ways to pass a variable as a parameter to a function:

- Passing the variable itself. In this case, the function gets its own copy of the variable to work on. Creating a new copy of the variable on the stack can be time-consuming if, for example, the variable is a large structure.
- Passing a pointer to the variable. In this case, the function gets only the address of a variable, which it uses to access the caller's copy of the variable. This technique is much faster for large structures.

In C++, you have a third option: passing a reference to the variable. In this case, the function receives an alias to the caller's copy of the variable.

The following example illustrates all three techniques:

```
// REFPARM.CPP: Reference parameters for reducing
//   overhead and eliminating pointer notation
#include <iostream.h>

// ---------- A big structure
struct bigone
{
   int serno;
   char text[1000];   // A lot of chars
} bo = { 123, "This is a BIG structure" };

// -- Three functions that have the structure as a parameter
void valfunc( bigone v1 );          // Call by value
void ptrfunc( const bigone *p1 );   // Call by pointer
void reffunc( const bigone &r1 );   // Call by reference

void main()
{
   valfunc( bo );   // Passing the variable itself
   ptrfunc( &bo );  // Passing the address of the variable
   reffunc( bo );   // Passing a reference to the variable
}

// ---- Pass by value
void valfunc( bigone v1 )
{
   cout << '\n' << v1.serno;
   cout << '\n' << v1.text;
}
// ---- Pass by pointer
void ptrfunc( const bigone *p1 )
{
   cout << '\n' << p1->serno;     // Pointer notation
   cout << '\n' << p1->text;
}

// ---- Pass by reference
void reffunc( const bigone &r1 )
{
   cout << '\n' << r1.serno;       // Reference notation
   cout << '\n' << r1.text;
}
```

The parameter `rl` is a reference that is initialized with the variable `bo` when `reffunc` is called. Inside `reffunc`, the name `rl` is an alias for `bo`. Unlike the previous examples of references, this reference has a different scope from that of the variable it refers to.

When you pass a reference as a parameter, the compiler actually passes the address of the caller's copy of the variable. Passing a reference is therefore just as efficient as passing a pointer, and, when passing large structures, far more efficient than passing by value. Also, the syntax for passing a reference to a variable is identical to that for passing the variable itself. No & is needed in the function call statement, and no -> is needed when using the parameter within the function. Passing a reference thus combines the efficiency of passing a pointer and the syntactical cleanliness of passing by value.

When you pass a reference as a parameter, any modifications to the parameter are actually modifications to the caller's copy of the variable. This is significant because, unlike the syntax of passing a pointer, the syntax of passing a reference doesn't give any indication that such a modification is possible. For example:

```
valfunc( bo );    // Function can't modify bo
ptrfunc( &bo );   // & implies that function can modify bo
reffunc( bo );    // Same syntax as valfunc;
                  //     implies that function can't modify bo
```

The syntax for calling `reffunc` could make you think that the function cannot modify the variable you pass. In the case of `reffunc`, this assumption is correct. Because `reffunc`'s parameter is a reference to a constant, its parameter is a read-only alias for the caller's copy of the variable. The `reffunc` function cannot modify the `bo` variable.

But you can also use an ordinary reference as a parameter instead of a reference to a constant. This allows the function to modify the caller's copy of the

parameter, even though the function's calling syntax implies that it can't. For example:

```
// BAD TECHNIQUE: modifying a parameter through a reference

void print( int &parm )
{
    cout << parm;
    parm = 0;
};

void main()
{
    int a = 5;

    print( a );  // Parameter is modified;
                 //     unexpected side effect
}
```

Using references this way could be very confusing to someone reading your program.

For precisely this reason, you should use caution when passing references as function parameters. Don't assume that a reader of your program can tell whether a function modifies its parameters or not just by looking at the function's name. Without looking at the function's prototype, it is impossible to tell whether a function takes a reference or the variable itself. The function's calling syntax provides no clues.

To prevent such confusion, you should use the following guidelines when writing functions that take parameters too large to pass by value:

- If the function modifies the parameter, use a pointer.
- If the function doesn't modify the parameter, use a reference to a constant.

These rules are consistent with a common C-programming convention: When you explicitly take the address of a variable in order to pass it to a function, the function can modify the parameter. By following this convention, you make your C++ program more readable to C programmers. This is strictly a coding convention and cannot be enforced by the compiler. These rules do not make your programs correct or more efficient, but they do make them easier to read and understand.

Note that references are only needed when the parameter to be passed is a large, user-defined type. Parameters of built-in types, such as characters, integers, or floats, can be efficiently passed by value.

References as Return Values

Besides passing parameters to a function, references can also be used to return values from a function. For example:

```
int mynum = 0;     // Global variable

int &num()
{
    return mynum;
}

void main()
{
    int i;

    i = num();
    num() = 5;    // mynum set to 5
}
```

In this example, the return value of the function `num` is a reference initialized with the global variable `mynum`. As a result, the expression `num()` acts as an alias for `mynum`. This means that a function call can appear on the receiving end of an assignment statement, as in the last line of the example.

You'll learn some more practical applications of this technique in Chapter 5, "Classes and Dynamic Memory Allocation," and Chapter 8, "Operator Overloading and Conversion Functions."

Passing a reference to a function and returning a reference from a function are the only two operations that you should perform on references themselves. Perform other operations on the object a reference refers to.

Summary

You will use references extensively when you build C++ classes, the subject of Part 2. As you do so, remember the following points about references:

- A reference is an alias for an actual variable.
- A reference must be initialized and cannot be changed.
- References are most useful when passing user-defined data types to a function and when returning values from a function.

Reference declarations are sometimes confused with the operation of taking the address of a variable, because both have the form *&identifier*. To distinguish between these two uses of **&**, remember the following rules:

- When *&identifier* is preceded by the name of a type, such as **int** or **char**, the **&** means "reference to" the type. This form of **&** occurs only in declarations, such as declaring the type of a reference variable, the type of a parameter, or a function's return type.
- When *&identifier* is not preceded by the name of a type, the **&** means "address of" the variable. This form of **&** occurs most commonly when passing an argument to a function or when assigning a value to a pointer.

Note that there is no difference between *type &identifier* and *type& identifier*. Both syntax variations declare references.

PART 2

Classes

CHAPTER 4

Introduction to Classes

The most important feature of C++ is its support for user-defined types, through a mechanism called "classes." Classes are far more powerful than the user-defined types you can create in C. While an instance of a built-in type is called a variable, an instance of a class is called an "object," hence the phrase "object-oriented programming." Part 2 of this book describes classes, and Part 3 describes object-oriented programming.

This chapter covers the following topics:

- Declaring a class
- Using objects of a class
- Data members and member functions
- Constructors and destructors
- **const** objects and member functions
- Member objects
- Header and source files

Before explaining how to define a class in C++, let's consider one way you can create a new data type in C.

Creating a New Data Type in C

Suppose you're writing a C program that frequently manipulates dates. You might create a new data type to represent dates, using the following structure:

```
struct date
{
   int month;
   int day;
   int year;
};
```

This structure contains members for the month, day, and year.

To store a particular date, you can set the members of a `date` structure to the appropriate values:

```
struct date my_date;

my_date.month = 1;
my_date.day = 23;
my_date.year = 1985;
```

You cannot print a date by passing a `date` structure to **printf**. You must either print each member of the structure individually or write your own function to print the structure as a whole, as follows:

```
void display_date( struct date *dt )
{
    static char *name[] =
    {
        "zero", "January", "February", "March", "April", "May",
        "June", "July", "August", "September", "October",
        "November", "December"
    };

    printf( "%s %d, %d", name[dt->month], dt->day, dt->year );
}
```

This function prints the contents of a `date` structure, printing the month in string form, then the day and the year.

To perform other operations on dates, such as comparing two of them, you can compare the structure members individually, or you can write a function that accepts `date` structures as parameters and does the comparison for you.

When you define a structure type in C, you are defining a new data type. When you write functions to operate on those structures, you define the operations permitted on that data type.

This technique for implementing dates has some drawbacks:

- It does not guarantee that a `date` structure contains a valid date. You could accidentally set the members of a structure to represent a date like February 31, 1985, or you might have an uninitialized structure whose members represent the one-thousand-and-fifty-eighth day of the eighteenth month of a certain year. Any function that blindly uses such a variable generates nonsense results.
- Once you've used the `date` data type in your programs, you cannot easily change its implementation. Suppose later you become concerned about the amount of space that your `date` variables are taking up. You might decide to store both the month and day using a single integer, either by using bit fields or by saving only the day of the year (as a number from 1 to 365). Such a change would save two bytes per instance. To make this change, every program that uses the `date` data type must be rewritten. Every expression that accesses the month or day as separate integer members must be rewritten.

You could avoid these problems with more programming effort. For example, instead of setting the members of a `date` structure directly, you could use a function that tests the specified values for validity. And instead of reading the members of the structure directly, you could call functions that returned the value of a structure's members. Unfortunately, many programmers don't follow such practices when using a new data type in C. They find it more convenient to access the members of a `date` structure directly. As a result, their programs are more difficult to maintain.

Unlike C, C++ was designed to support the creation of user-defined data types. As a result, you don't have to expend as much programming effort to create a data type that is safe to use.

Creating a New Data Type in C++

With C++, you define both the data type and its operations at once, by declaring a "class." A class consists of data and functions that operate on that data.

Declaring the Class

A class declaration looks similar to a structure declaration, except that it has both functions and data as members, instead of just data. The following is a preliminary version of a class that describes a date.

```
// The Date class
#include <iostream.h>

// ------------ a Date class
class Date
{
public:
   Date( int mn, int dy, int yr );  // Constructor
   void display();                  // Function to print date
   ~Date();                         // Destructor
private:
   int month, day, year;            // Private data members
};
```

This class declaration is roughly equivalent to the combination of an ordinary structure declaration plus a set of function prototypes. It declares the following:

- The contents of each instance of `Date`: the integers `month`, `day`, and `year`. These are the class's "data members."
- The prototypes of three functions that you can use with `Date`: `Date`, `~Date`, and `display`. These are the class's "member functions."

You supply the definitions of the member functions after the class declaration. Here are the definitions of `Date`'s member functions:

```
// Some useful functions
inline int max( int a, int b )
{
    if( a > b ) return a;
    return b;
}

inline int min( int a, int b )
{
    if( a < b ) return a;
    return b;
}
```

```
// ---------- The constructor
Date::Date( int mn, int dy, int yr )
{
    static int length[] = { 0, 31, 28, 31, 30, 31, 30,
                                31, 31, 30, 31, 30, 31 };
    // Ignore leap years for simplicity
    month = max( 1, mn );
    month = min( month, 12 );

    day = max( 1, dy );
    day = min( day, length[month] );

    year = max( 1, yr );
}

// -------- Member function to print date
void Date::display()
{
    static char *name[] =
    {
        "zero", "January", "February", "March", "April", "May",
        "June", "July", "August", "September", "October",
        "November","December"
    };

    cout << name[month] << ' ' << day << ", " << year;
}
// ---------- The destructor
Date::~Date()
{
   // Do nothing
}
```

The `display` function looks familiar, but the `Date` and `~Date` functions are new. They are called the "constructor" and "destructor," respectively, and they are used to create and destroy objects, or instances of a class. They are described later in this chapter.

These are not all the member functions that a `Date` class needs, but they are sufficient to demonstrate the basic syntax for writing a class. Later in this chapter, we'll add more functionality to the class.

Here's a program that uses the rudimentary `Date` class:

```
// Program that demonstrates the Date class
void main()
{
    Date myDate( 3, 12, 1985 );      // Declare a Date
    Date yourDate( 23, 259, 1966 );  // Declare an invalid Date

    myDate.display();
    cout << '\n';
    yourDate.display();
    cout << '\n';
}
```

Because MS-DOS has a DATE command, specify the path explicitly to execute your own DATE.EXE program.

Using the Class

Once you've defined a class, you can declare one or more instances of that type, just as you do with built-in types like integers. As mentioned before, an instance of a class is called an "object" rather than a variable.

In the previous example, the **main** function declares two instances of the `Date` class called `myDate` and `yourDate`:

```
Date myDate( 3, 12, 1985 );      // Declare a Date
Date yourDate( 23, 259, 1966 );  // Declare an invalid Date
```

These are objects, and each one contains month, day, and year values.

The declaration of an object can contain a list of initializers in parentheses. The declarations of `myDate` and `yourDate` each contain three integer values as their initializers. These values are passed to the class's constructor, described on page 48.

Note the syntax for displaying the contents of `Date` objects. In C, you would pass each structure as an argument to a function, as follows:

```
// Displaying dates in C
display_date( &myDate );
display_date( &yourDate );
```

In C++, you invoke the member function for each object, using a syntax similar to that for accessing a structure's data member:

```
// Displaying dates in C++
myDate.display();
yourDate.display();
```

This syntax emphasizes the close relationship between the data type and the functions that act on it. It makes you think of the `display` operation as being part of the `Date` class.

However, this joining of the functions and the data appears only in the syntax. Each `Date` object does not contain its own copy of the `display` function's code. Each object contains only the data members.

Class Members

Now consider how the class declaration differs from a structure declaration in C:

```
class Date
{
public:
   Date( int mn, int dy, int yr );  // Constructor
   void display();                  // Function to print date
   ~Date();                         // Destructor
private:
   int month, day, year;            // Private data members
};
```

Like a structure declaration, it declares three data members: the integers `month`, `day`, and `year`. However, the class declaration differs from a structure declaration in several ways:

- It has the keywords **public** and **private**.
- It declares functions, such as `display`.
- It includes the constructor `Date` and the destructor `~Date`.

Let's examine these differences one by one.

Class Member Visibility

The **private** and **public** labels in the class definition specify the visibility of the members that follow the labels. The mode invoked by a label continues until another label occurs or the class definition ends.

Private members can be accessed only by member functions. (They can also be accessed by friend classes and functions; for more information on friends, see Chapter 6, "More Features of Classes.") The private members define the internal workings of the class. They make up the class's "implementation."

Public members can be accessed by member functions, and by any other functions in the program as long as an instance of the class is in scope. The public members determine how the class appears to the rest of the program. They make up the class's "interface."

The `Date` class declares its three integer data members as private, which makes them visible only to functions within the class. If another function attempts to access one of these private data members, the compiler generates an error. For example, suppose you try to access the private data members of a `Date` object:

```
void main()
{
    int i;
    Date myDate( 3, 12, 1985 );

    i = myDate.month;     // Error: can't read private member
    myDate.day = 1;       // Error: can't modify private member
}
```

By contrast, the `display` function is public, which makes it visible to outside functions.

You can use the **private** and **public** labels as often as you want in a class definition, but most programmers group the private members together and the public members together. All class definitions begin with the private label as the default mode, but it improves readability to explicitly label all sections.

The `Date` class demonstrates a common C++ convention: Its public interface consists entirely of functions. You can view or modify a private data value only by calling a public member function designed for that purpose. This convention is discussed further in the section "Accessing Data Members" on page 52.

Member Functions

The `Date` class has a member function named `display`. This function corresponds to the `display_date` function in C, which prints the contents of a `date` structure. However, notice the following differences.

First, consider the way the function is declared and defined. The function's prototype appears inside the declaration of `Date`, and when the function is defined, it is called `Date::display()`. This indicates that it is a member of the class and that its name has "class scope." You could declare another function named `display` outside the class, or in another class, without any conflict. The class name (combined with `::`, the scope resolution operator) prevents any confusion between the definitions.

You can also overload a member function, just as you can any other function in C++, as long as each version is distinguishable by its parameter list. All you have to do is declare each member function's prototype in the class declaration and prefix its name with the class name and `::` when defining it.

Now compare the implementation of the `display` member function with that of the corresponding function in C, `display_date`. The C function refers to `dt.month`, `dt.day`, and `dt.year`. In contrast, the C++ member function refers to `month`, `day`, and `year`; no object is specified. Those data members belong to the object that the function was called for. For example:

```
myDate.display();
yourDate.display();
```

The first time `display` is called, it uses the data members of `myDate`. The second time it's called, it uses the members of `yourDate`. A member function automatically uses the data members of the "current" object, the object to which it belongs.

You can also call a member function through a pointer to an object, using the `->` operator. For example:

```
Date myDate( 3, 12, 1985 );
Date *datePtr = &myDate;

datePtr->display();
```

This code declares a pointer to a `Date` object and calls `display` through that pointer.

You can even call a member function through a reference to an object. For example:

```
Date myDate( 3, 12, 1985 );
Date &otherDate = myDate;

otherDate.display();
```

This code calls `display` through the reference variable `otherDate`. Because `otherDate` is an alias for `myDate`, the contents of `myDate` are displayed.

These techniques for calling a member function work only if the function is declared **public**. If a member function is declared **private**, only other member functions within the same class can call it. For example:

```
class Date
{
public:
    void display();     // Public member function
    // ...
private:
    int daysSoFar();    // Private member function
    // ...
};

// --------- Display date in form "DDD YYYY"
void Date::display()
{

    cout << daysSoFar()         // Call private member function
         << " " << year;
}

 // -------- Compute number of days elapsed
int Date::daysSoFar()
{
    int total = 0;
    static int length[] = { 0, 31, 28, 31, 30, 31, 30,
                                31, 31, 30, 31, 30, 31 };

    for( int i = 1; i <= month; i++ )
        total += length[i];

    total += day;
    return total;
}
```

Notice that `display` calls `daysSoFar` directly, without preceding it with an object name. A member function can use data members and other member functions without specifying an object. In either case, the "current" object is used implicitly.

The Constructor

Remember that the `date` structure in C had the drawback of not guaranteeing that it contained valid values. In C++, one way to ensure that objects always contain valid values is to write a constructor. A constructor is a special initialization function that is called automatically whenever an instance of your class is

declared. This function prevents errors resulting from the use of uninitialized objects.

The constructor must have the same name as the class itself. For example, the constructor for the `Date` class is named `Date`.

Look at the implementation of the `Date` constructor:

```
Date::Date( int mn, int dy, int yr )
{
    static int length[] = { 0, 31, 28, 31, 30, 31, 30,
                            31, 31, 30, 31, 30, 31 };
    // Ignore leap years for simplicity
    month = max( 1, mn );
    month = min( month, 12 );

    day = max( 1, dy );
    day = min( day, length[month] );

    year = max( 1, year );
}
```

Not only does this function initialize the object's data members, it also checks that the specified values are valid; if a value is out of range, it substitutes the closest legal value. This is another way that a constructor can ensure that objects contain meaningful values.

Whenever an instance of a class comes into scope, the constructor is executed. Observe the declaration of `myDate` in the **main** function:

```
Date myDate( 3, 12, 1985 );
```

The syntax for declaring an object is similar to that for declaring an integer variable. You give the data type, in this case `Date`, and then the name of the object, `myDate`.

However, this object's declaration also contains an argument list in parentheses. These arguments are passed to the constructor function and are used to initialize the object. When you declare an integer variable, the program merely allocates enough memory to store the integer; it doesn't initialize that memory. When you declare an object, your constructor function initializes its data members.

You cannot specify a return type when declaring a constructor, not even **void**. Consequently, a constructor cannot contain a **return** statement. A constructor doesn't return a value; it creates an object.

You can declare more than one constructor for a class if each constructor has a different parameter list; that is, you can overload the constructors. This is useful

if you want to initialize your objects in more than one way. This is demonstrated in the section "Accessing Data Members" on page 52.

You aren't required to define any constructors when you define a class, but it is a good idea to do so. If you don't define any, the compiler automatically generates a do-nothing constructor that takes no parameters, just so you can declare instances of the class. However, this compiler-generated constructor doesn't initialize any data members, so any objects you declare are not any safer than C structures.

The Destructor

The destructor is the counterpart of the constructor. It is a member function that is called automatically when a class object goes out of scope. Its purpose is to perform any cleanup work necessary before an object is destroyed. The destructor's name is the class name with a tilde (~) as a prefix.

The `Date` class doesn't really need a destructor. One is included in this example simply to show its format.

Destructors are required for more complicated classes, where they're used to release dynamically allocated resources. For more information on such classes, see Chapter 5, "Classes and Dynamic Memory Allocation."

There is only one destructor for a class; you cannot overload it. A destructor takes no parameters and has no return value.

The Creation and Destruction of Objects

The following example defines a constructor and destructor that print messages, so you can see exactly when these functions are called.

```
// DEMO.CPP
#include <iostream.h>
#include <string.h>

class Demo
{
public:
    Demo( const char *nm );
    ~Demo();
private:
    char name[20];
};

Demo::Demo( const char *nm )
{
    strncpy( name, nm, 20 );
    cout << "Constructor called for " << name << '\n';
}
Demo::~Demo()
{
    cout << "Destructor called for " << name << '\n';
}

void func()
{
    Demo localFuncObject( "localFuncObject" );
    static Demo staticObject( "staticObject" );

    cout << "Inside func" << endl;
}

Demo globalObject( "globalObject" );

void main()
{
    Demo localMainObject( "localMainObject" );

    cout << "In main, before calling func\n";
    func();
    cout << "In main, after calling func\n";
}
```

The program prints the following:

```
Constructor called for globalObject
Constructor called for localMainObject
In main, before calling func
Constructor called for localFuncObject
Constructor called for staticObject
Inside func
Destructor called for localFuncObject
In main, after calling func
Destructor called for localMainObject
Destructor called for staticObject
Destructor called for globalObject
```

For local objects, the constructor is called when the object is declared and the destructor is called when the program exits the block in which the object is declared.

For global objects, the constructor is called when the program begins and the destructor is called when the program ends. For static objects, the constructor is called before the first entry to the function in which the static objects are declared and the destructor is called when the program ends.

Accessing Data Members

As it is currently defined, the `Date` class does not permit any access to its individual month, day, and year components. For example, you cannot read or modify the month value of a `Date` object. To remedy this, you can revise the `Date` class as follows:

```
class Date
{
public:
   Date( int mn, int dy, int yr );  // Constructor
                                    // Member functions:
   int getMonth();                  //   Get month
   int getDay();                    //   Get day
   int getYear();                   //   Get year
   void setMonth( int mn );         //   Set month
   void setDay( int dy );           //   Set day
   void setYear( int yr );          //   Set year
   void display();                  //   Print date
   ~Date();                         // Destructor
private:
   int month, day, year;            // Private data members
};
```

This version of `Date` includes member functions to read and modify the month, day, and year members. The function definitions are as follows:

```
inline int Date::getMonth()
{
    return month;
}

inline int Date::getDay()
{
    return day;
}

inline int Date::getYear()
{
    return year;
}

void Date::setMonth( int mn )
{
    month = max( 1, mn );
    month = min( month, 12 );
}

void Date::setDay( int dy )
{
    static int length[] = { 0, 31, 28, 31, 30, 31, 30,
                                31, 31, 30, 31, 30, 31 };
    day = max( 1, dy );
    day = min( day, length[month] );
}

void Date::setYear( int yr )
{
    year = max( 1, yr );
}
```

The various `get` functions simply return the value of the appropriate data member. However, the `set` functions do not simply assign a new value to a data member. These functions also check the validity of the specified value before assigning it. This is another way to ensure that `Date` objects contain valid values.

The following example uses these new member functions:

```
void main()
{
    int i;
    Date deadline( 3, 10, 1980 );

    i = deadline.getMonth();     // Read month value
    deadline.setMonth( 4 );      // Modify month value
    deadline.setMonth( deadline.getMonth() + 1 ); // Increment
}
```

Notice that the `get` functions are declared **inline** because they're so short. Because those functions have no function call overhead, calling them is as efficient as directly accessing public data members.

Member functions can also be declared inline without using the **inline** keyword. Instead, you can place the body of the function inside the class declaration, as follows:

```
class Date
{
public:
    Date( int mn, int dy, int yr );
    int getMonth() { return month; }   // Inline member functions
    int getDay() { return day; }
    int getYear() { return year; }
// etc....
};
```

This style of declaration has precisely the same effect as using the **inline** keyword with separate function definitions. You can use whichever style you find more readable.

Now that the class has member functions to set its values, you can change the way a `Date` object is constructed. You can overload constructors in the same way

you overload other functions. The following example defines two versions of Date's constructor, one that takes parameters and one that doesn't:

```
class Date
{
public:
    Date();     // Constructor with no parameters
    Date( int mn, int dy, int yr );  // Constructor with parameters
// etc....
};

Date::Date()
{
    month = day = year = 1;    // Initialize data members
}

Date::Date( int mn, int dy, int yr )
{
    setMonth( mn );
    setDay( dy );
    setYear( yr );
}

void main()
{
    Date myDate;       // Declare a date without arguments
    Date yourDate( 12, 25, 1990 );

    myDate.setMonth( 3 );    // Set values for myDate
    myDate.setDay( 12 );
    myDate.setYear( 1985 );
}
```

The declaration of myDate doesn't specify any initial values. As a result, the first constructor is used to create myDate and initialize it with the default value "January 1, 1." The values for myDate are specified later with the set functions. In contrast, the declaration of yourDate specifies three arguments. The second constructor is used to create yourDate, and this constructor calls the member functions to set the data members to the specified values. It is legal for a constructor to call member functions, as long as those functions don't read any uninitialized data members.

The first constructor in the example above is known as a "default constructor," because it can be called without arguments. If you define a default constructor,

the compiler calls it automatically in certain situations; for more information, see page 60, "Member Objects," and page 99, "Arrays of Class Objects," in Chapter 6.

Access Functions vs. Public Data Members

Writing access functions might seem like a lot of needless work. You may argue that it's much simpler to declare the data members as public and manipulate them directly. After all, why call a `setMonth` and `getMonth` function when you could simply access the `month` member itself?

The advantages of access functions become apparent when we recall the example of the `date` structure defined in C. Access functions ensure that your objects never contain invalid values. You can always be sure that you can display the contents of a `Date` object without printing out nonsense.

More importantly, access functions let you change the implementation of your class easily. For example, remember the scenario in which you decide to encode the month and day within the bits of a single integer in order to save space. In C, you have to modify every program that uses `date` structures. This could involve thousands of lines of code.

In C++, however, all you have to rewrite are the class's member functions, which constitute far fewer lines. This change has no effect on any programs that use your `Date` class. They can still call `getMonth` and `setMonth`, just as they did before. The use of access functions instead of public members saves you a huge amount of rewriting.

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. Encapsulation is discussed in more detail in Chapter 9, "Fundamentals of Object-Oriented Design."

Returning a Reference

Occasionally, you may see a C++ program that declares member functions that act like public data members. Such functions return references to private data members. For example:

```
// BAD TECHNIQUE: Member function that returns a reference
class Date
{
public:
   Date( int mn, int dy, int yr );  // Constructor
   int &month();                    // Set/get month
   ~Date();                         // Destructor
private:
   int month_member,                //  (_member appended to
       day_member,                  //   distinguish names from
       year_member;                 //   member functions)
};

int &Date::month()
{
    month_member = max( 1, month_member );
    month_member = min( month_member, 12 );
    return month_member;
}

// ...
```

The `month` member function returns a reference to the data member. This means that the function call expression `month()` can be treated as an alias for the private data member. For example:

```
// BAD TECHNIQUE: using member function that returns a reference
void main()
{
    int i;
    Date deadline( 3, 10, 1980 );

    i = deadline.month();    // Read month value
    deadline.month() = 4;    // Modify month value
    deadline.month()++;      // Increment
}
```

The member function behaves just like a data member. Consequently, the function call `deadline.month()` can appear on the left side of an assignment, in the same way that `deadline.month_member` could if the data member were public. You can even increment its value with the ++ operator.

You can assign an illegal value to `month_member` this way, but the `month` function performs range-checking to correct any such illegal values the next time it is called. As long as all of `Date`'s other member functions don't access the data member directly, but always use the `month` function instead, the `Date` class works correctly.

You should not use this technique for a variety of reasons. First, the syntax can be very confusing to people reading your program. Second, range checking is performed every time a data member is read, which is inefficient. Finally, and most importantly, this technique essentially makes the data member public. With this design, you cannot change the implementation of a private data member without rewriting all the programs that use the class. If you wanted to encode the month and day values within a single integer, you would have to change the member functions and rewrite all the programs that used `Date`.

To retain the benefits that member functions offer, you should always give your classes separate member functions to read and modify private data members.

const Objects and Member Functions

Just as you can use the **const** keyword when declaring a variable, you can also use it when declaring an object. Such a declaration means that the object is a constant, and none of its data members can be modified. For example:

```
const Date birthday( 7, 4, 1776 );
```

This declaration means that the value of `birthday` cannot be changed.

When you declare a variable as a constant, the compiler can usually identify operations that would modify it (such as assignment), and it can generate appropriate errors when it detects them. However, this is not true when you declare an object as a constant. The compiler can't tell whether a given member function might modify one of an object's data members, so it plays it safe and prevents you from calling any member functions for a **const** object.

However, some member functions don't modify any of an object's data members, so you should be able to call them for a **const** object. If you place the **const** keyword after a member function's parameter list, you declare the member function as a read-only function that doesn't modify its object. The following example declares some of the `Date` class's member functions as **const**.

```
class Date
{
public:
   Date( int mn, int dy, int yr );  // Constructor
                                    // Member functions:
   int getMonth() const;            //   Get month - read-only
   int getDay() const;              //   Get day - read-only
   int getYear() const;             //   Get year - read-only
   void setMonth( int mn );         //   Set month
   void setDay( int dy );           //   Set day
   void setYear( int yr );          //   Set year
   void display() const;            //   Print date - read-only
   ~Date();                         // Destructor
private:
   int month, day, year;            // Private data members
};

inline int Date::getMonth() const
{
    return month;
}
// etc...
```

The various `get` functions and the `display` function are all read-only functions. Note that the **const** keyword is used in both the declaration and the definition of each member function. These functions can be safely called for a constant object.

With the `Date` class modified in this way, the compiler can ensure that `birthday` is not modified:

```
int i;
const Date birthday( 7, 4, 1776 );

i = birthday.getYear();      // Legal
birthday.setYear( 1492 );    // Error: setYear not const
```

The compiler lets you call the **const** member function `getYear` for the `birthday` object, but not the function `setYear`, which is a non-**const** function.

A member function that is declared with **const** cannot modify any data members of the object, nor can it call any non-**const** member functions. If you declare any of the `set` functions as **const**, the compiler generates an error.

You should declare your member functions as **const** whenever possible. This allows people using your class to declare constant objects.

Member Objects

You can write a class that contains objects as members. This is known as "composition," the act of making a new class by using other classes as components. Suppose that you want a `PersonInfo` class that stores a person's name, address, and birthday. You can give that class a `Date` as a member, as follows:

```
class PersonInfo
{
public:
    // Public member functions...
private:
    char name[30];
    char address[60];
    Date birthday;            // Member object
};
```

This declaration specifies a private member named `birthday`, which is a `Date` object. Note that no arguments are specified in the declaration of `birthday`. However, this does not mean that the default constructor is called. The `birthday` object is not constructed until a `PersonInfo` object is constructed.

To call the constructor for a member object, you must specify a "member initializer." Place a colon after the parameter list of the containing class's constructor, and follow it with the name of the member and a list of arguments. For example, the constructor for `PersonInfo` is written as follows:

```
class PersonInfo
{
public:
    PersonInfo( char *nm, char *addr, int mn, int dy, int yr );
    // ...
private:
    // ...
};

PersonInfo::PersonInfo( char *nm, char *addr,
                        int mn, int dy, int yr )
    : birthday( mn, dy, yr ) // Member initializer
{
    strncpy( name, nm, 30 );
    strncpy( address, addr, 60 );
}
```

This syntax causes the `Date` class constructor to be invoked for the `birthday` member object, using the three arguments specified. The `Date` constructor is called first, so the `birthday` member is initialized before the `PersonInfo` constructor begins executing. If your class has more than one member object, you can specify a list of member initializers, separating them with commas.

If you don't use the member initializer syntax, the compiler implicitly calls the default constructor for the member object before constructing the containing object. You can then assign values to the member object using its access functions. For example, because `Date` has a default constructor, you could write the `PersonInfo` constructor as follows:

```
PersonInfo::PersonInfo( char *nm, char *addr, int mn, int dy, int yr )
// Default constructor sets birthday to January 1, 1
{
    strncpy( name, nm, 30 );
    strncpy( address, addr, 60 );
    birthday.setMonth( mn );
    birthday.setDay( dy );
    birthday.setYear( yr );
}
```

If the member object's class doesn't define a default constructor, the compiler generates an error.

However, this is an inefficient technique because the value of `birthday` is set twice. First it is initialized to January 1, 1, by the default constructor, and then it is assigned the value specified by the member functions. In general, you should use member initializers to initialize your member objects, unless the default constructor performs all the initialization you need.

A member initializer is required when you have a constant member object. Because a person's birthday never changes, you can declare `birthday` with the

const keyword. In this case, omitting the member initializer syntax is fatal. For example:

```
class PersonInfo
{
public:
   // ...
private:
   char name[30];
   char address[60];
   const Date birthday;     // Constant member object
};
PersonInfo::PersonInfo( char *nm, char *addr, int mn, int dy, int yr )
// Default constructor sets birthday to January 1, 1
{
    strncpy( name, nm, 30 );
    strncpy( address, addr, 60 );
    birthday.setMonth( mn );    // Error
    birthday.setDay( dy );      // Error
    birthday.setYear( yr );     // Error
}
```

Because `birthday` is a **const** object, you can't call any of its `set` member functions, because those are non-**const** functions. Thus, you have no way to change the value of `birthday` from the value that the default constructor initialized it to.

The same is true of any member declared **const**, even if it's a variable of a built-in type, like an integer. A **const** integer member cannot be assigned a value in the constructor; you must use a member initializer. For example:

```
class Count
{
public:
    Count( int i );    // Constructor
private:
    const int cnt;     // Constant integer member
};

Count( int i )
   : cnt( i )   // Member initializer for integer
{
}
```

Use a member initializer to initialize any **const** member, whether or not it's an object.

Using Header and Source Files

In C++, it's common practice to divide your source code into header and source files. You place the class declarations in the header files and place the definitions of the member functions in the source files. Header files usually have the filename extension .H, and source files have the filename extension .CPP. For example, here's a partial header file for the `Date` class:

```
// DATE.H
#if !defined( _DATE_H_ )

#define _DATE_H_

class Date
{
   Date();
   int getMonth() const;
   // ...
};

inline Date::getMonth() const
{
    return month;
}
// etc...

#endif  // _DATE_H_
```

Notice that this header file contains the definitions for inline member functions. The compiler must have access to the source code of an inline function in order to insert the code each time the function is called.

Also note that the header file uses the **#if** preprocessor directive and the **defined** preprocessor operator for conditional compilation. This prevents multiple inclusion of header files in a multimodule program.

Here's the beginning of a source file for the `Date` class:

```
// DATE.CPP
#include "date.h"

Date::Date()
{
    // ...
}
// etc...
```

Note that the source file includes its corresponding header file. See the DATE.H and DATE.CPP example files.

In general, you should use one header file and one source file for each class unless you are writing very small classes, or classes that are very closely related and should always be used together.

Roughly speaking, a header file describes a class's interface and a source file describes its implementation. This distinction is important when your classes may be used by other programmers. To use the `Date` class, for example, other programmers would simply include the header file DATE.H in their source files. Those programmers don't need to see how the member functions are implemented; all they need to see are the prototypes of the member functions. As long as they can link with DATE.OBJ when linking their program, they don't need to see DATE.CPP. If you rewrite DATE.CPP, you can simply recompile it to produce a new DATE.OBJ file; the other programmers don't need to change their code.

Unfortunately, it's necessary that some aspects of a class's implementation be revealed in the header file. The private members of a class are visible in the header file, even though they aren't accessible. Furthermore, if your class has inline member functions, their implementation is also visible. If you change the private members or inline functions of your class, those changes are reflected in the header file, and all the programmers who use that class must recompile their code with the new header file. However, they still don't have to rewrite any of their code, as long as the class's interface hasn't changed—that is, as long as you haven't changed the prototypes of the public member functions.

You should also consider whether your **#include** statements need to be in your header file or your source file. For example, if one of your class's member functions takes a **time_t** structure as a parameter, you have to place `#include "time.h"` in the header file. On the other hand, if the **time_t** structure is used only in the internal computations of a member function, and is not visible to

someone calling the function, then you should place `#include "time.h"` in the source file instead. In the first case, the interface requires TIME.H, and in the second case, the implementation requires it. Don't place **#include** statements in your header files if placing them in the source file suffices.

By separating a class's interface and implementation, you make your classes as self-contained as possible, so they don't depend on each other's implementation details. This practice follows the principle of encapsulation, which is discussed in more detail in Chapter 9, "Fundamentals of Object-Oriented Design."

CHAPTER 5

Classes and Dynamic Memory Allocation

C++ supports dynamic allocation and deallocation of objects from a pool of memory called the "free store." This chapter discusses the way objects are created, destroyed, copied, and converted to objects of other types.

Topics discussed include:

- The free store
- The assignment operator
- The **this** pointer
- The copy constructor
- Passing and returning objects
- Passing and returning references

Before continuing the discussion of classes, let's consider how you perform dynamic memory allocation in C++.

The Free Store

In C, the region of memory that is available at run time is called the heap. In C++, the region of available memory is known as the free store. The difference between the two lies in the functions you use to access this memory.

To request memory from the heap in C, you use the **malloc** function. For instance, you can dynamically allocate a `date` structure as follows:

```
struct date *dateptr;

dateptr = (struct date *)malloc( sizeof( struct date ) );
```

The **malloc** function allocates a block of memory large enough to hold a `date` structure and returns a pointer to it. The **malloc** function returns a void pointer,

which you must cast to the appropriate type when you assign it to `dateptr`. You can now treat that block of memory as a `date` structure.

In C++, however, **malloc** is not appropriate for dynamically allocating a new instance of the `Date` class, because `Date`'s constructor is supposed to be called whenever a new object is created. If you used **malloc** to create a new `Date` object, you would have a pointer to an uninitialized block of memory. You could then call member functions for an improperly constructed object, which would probably produce erroneous results. For example:

```
Date *datePtr;
int i;
datePtr = (Date *)malloc( sizeof( Date ) );
i = datePtr->getMonth();      // Returns undefined month value
```

If you use **malloc** to allocate objects, you lose the safety benefits that constructors provide. A better technique is to use the **new** operator.

The new Operator

As an alternative to **malloc**, C++ provides the **new** operator for allocating memory from the free store. The **malloc** function knows nothing about the type of the variable being allocated; it takes a size as a parameter and returns a void pointer. In contrast, the **new** operator knows the class of the object you're allocating, and it automatically calls the class's constructor to initialize the memory it allocates. Compare the previous example with the following:

```
Date *firstPtr, *secondPtr;
int i;
firstPtr = new Date;          // Default constructor called
i = firstPtr->getMonth();     // Returns 1 (default value)

secondPtr = new Date( 3, 15, 1985 );   // Constructor called
i = secondPtr->getMonth();             // Returns 3
```

The **new** operator calls the appropriate `Date` constructor, depending on whether you specify arguments or not. This ensures that any objects you allocate are properly constructed. You also don't have to use the **sizeof** operator to find the size of a `Date` object, because **new** can tell what size it is.

The **new** operator returns a pointer, but you don't have to cast it to a different type when you assign it to a pointer variable. The compiler checks that the type of the pointer matches that of the object being allocated and generates an error if they don't match. For example:

```
void *ptr;

ptr = new Date;   // Error; type mismatch
```

If **new** cannot allocate the memory requested, it returns 0. In C++, a null pointer has the value 0 instead of the value **NULL**.

The delete Operator

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 for subsequent allocations.

The syntax for **delete** is simple:

```
Date *firstPtr;
int i;
firstPtr = new Date( 3, 15, 1985 ); // Constructor called
i = firstPtr->getMonth();           // Returns 3

delete firstPtr;              // Destructor called, memory freed
```

The **delete** operator automatically calls the destructor for the object before it deallocates the memory. Because the `Date` class's destructor doesn't do anything, this feature is not demonstrated in this example.

You can only apply **delete** to pointers that were returned by **new**, and you can only delete them once. Deleting a pointer not obtained from **new** or deleting a pointer twice causes your program to behave strangely, and possibly to crash. It is your responsibility to guard against these errors; the compiler cannot detect them. You can, however, delete a null pointer (a pointer with value 0) without any adverse effects.

The Free Store and Built-in Types

The **new** and **delete** operators can be used not only with classes that you've defined, but also with built-in types such as integers and characters. For example:

```
int *ip;

ip = new int;   // Allocate an integer
// use ip
delete ip;
```

You can also allocate arrays whose size is determined at run time:

```
int length;
char *cp;

// Assign value to length, depending on user input
cp = new char[length];        // Allocate an array of chars
// Use cp
delete [] cp;
```

Notice the syntax for declaring an array: You place the array size within brackets after the name of the type. Also note the syntax for deleting an array: You place an empty pair of brackets before the name of the pointer. The compiler ignores any number you place inside the brackets.

You can even allocate multidimensional arrays with **new**, as long as all of the array dimensions except the first are constants. For example:

```
int (*matrix)[10];
int size;

// Assign value to size, depending on user input
matrix = new int[size][10];    // Allocate a 2-D array
// Use matrix
delete [] matrix;
```

Dynamic allocation of arrays of objects, as opposed to arrays of built-in types, is discussed in Chapter 6, "More Features of Classes."

Classes with Pointer Members

You can use the **new** and **delete** operators from within the member functions of a class. Suppose you wanted to write a `String` class, where each object contains a character string. It's inappropriate to store the strings as arrays, because you don't know how long they'll be. Instead, you can give each object a character pointer as a member and dynamically allocate an appropriate amount of memory for each object. For example:

```
// See STRNG.H and STRNG.CPP for final versions of this example
#include <iostream.h>
#include <string.h>

// ------- A string class
class String
{
public:
   String();
   String( const char *s );
   String( char c, int n );
   void set( int index, char newchar );
   char get( int index ) const;
   int getLength() const { return length; }
   void display() const { cout << buf; }
   ~String();
private:
   int length;
   char *buf;
};

// Default constructor
String::String()
{
    buf = 0;
    length = 0;
}
// ---------- Constructor that takes a const char *
String::String( const char *s )
{
   length = strlen( s );
   buf = new char[length + 1];
   strcpy( buf, s );
}

// ---------- Constructor that takes a char and an int
String::String( char c, int n )
{
   length = n;
   buf = new char[length + 1];
   memset( buf, c, length );
   buf[length] = '\0'; }

// ---------- Set a character in a String
void String::set( int index, char newchar )
{
    if( (index > 0) && (index <= length) )
    buf[index - 1] = newchar;
}
```

```
// ---------- Get a character in a String
char String::get( int index ) const
{
    if( (index > 0) && (index <= length) )
    return buf[index - 1];
    else
    return 0;
}
// ---------- Destructor for a String
String::~String()
{
    delete [] buf;   // Works even for empty String; delete 0 is safe
}

main()
{
   String myString( "here's my string" );
   myString.set( 1, 'H' );
}
```

The `String` constructor that takes a character pointer uses the **new** operator to allocate a buffer to contain the string. It then copies the contents of the string into the buffer. As a result, a `String` object is not a contiguous block of memory the way a structure variable is. Each `String` object consists of two blocks of memory, one that contains `length` and `buf`, and another that stores the characters themselves.

If you call **sizeof** to find the size of a `String` object, you get only the size of the block containing the integer and the pointer. However, different `String` objects may have character buffers of different lengths.

In fact, you can write a member function that changes the length of a `String` object's character buffer. For example:

```
void String::append( const char *addition )
{
    char *temp;

    length += strlen( addition );
    temp = new char[length + 1];   // Allocate new buffer
    strcpy( temp, buf );           // Copy contents of old buffer
    strcat( temp, addition );      // Append new string
    delete [] buf;                 // Deallocate old buffer
    buf = temp;
}
```

This function appends a new string to the contents of an existing `String` object. For example:

```
String myString( "here's my string" );

myString.append( " and here's more of it" );
// myString now holds "here's my string and here's more of it"
```

The `String` object defined above is thus dynamically resizable. All the details of the resizing are handled by the member functions.

The `String` class is an example of a class that requires a destructor. When a `String` object goes out of scope, the block of memory containing `length` and `buf` is deallocated automatically. However, the character buffer was allocated with **new**, so it must be deallocated explicitly. As a result, the `String` class defines a destructor that uses the **delete** operator to deallocate the character buffer. If the class didn't have a destructor, the character buffers would never be deallocated and the program might eventually run out of memory.

The `String` class has potential problems, however. Suppose you added the following code to the **main** function in this example:

```
String yourString( "here's your string" );
yourString = myString;
```

The program constructs a `String` object named `yourString` and then assigns the contents of `myString` to it. This looks harmless enough, but it actually causes problems.

When you assign one object to another, the compiler performs a memberwise assignment; that is, it does the equivalent of the following:

```
// Hypothetical equivalent of yourString = myString
yourString.length = myString.length;
yourString.buf = myString.buf;
```

The assignment of the `length` member is no problem. However, the `buf` member is a pointer. The result of the pointer assignment is that `yourString.buf`

and `myString.buf` point to the same location in memory. The two objects share the same character buffer. This is illustrated in Figure 5.1.

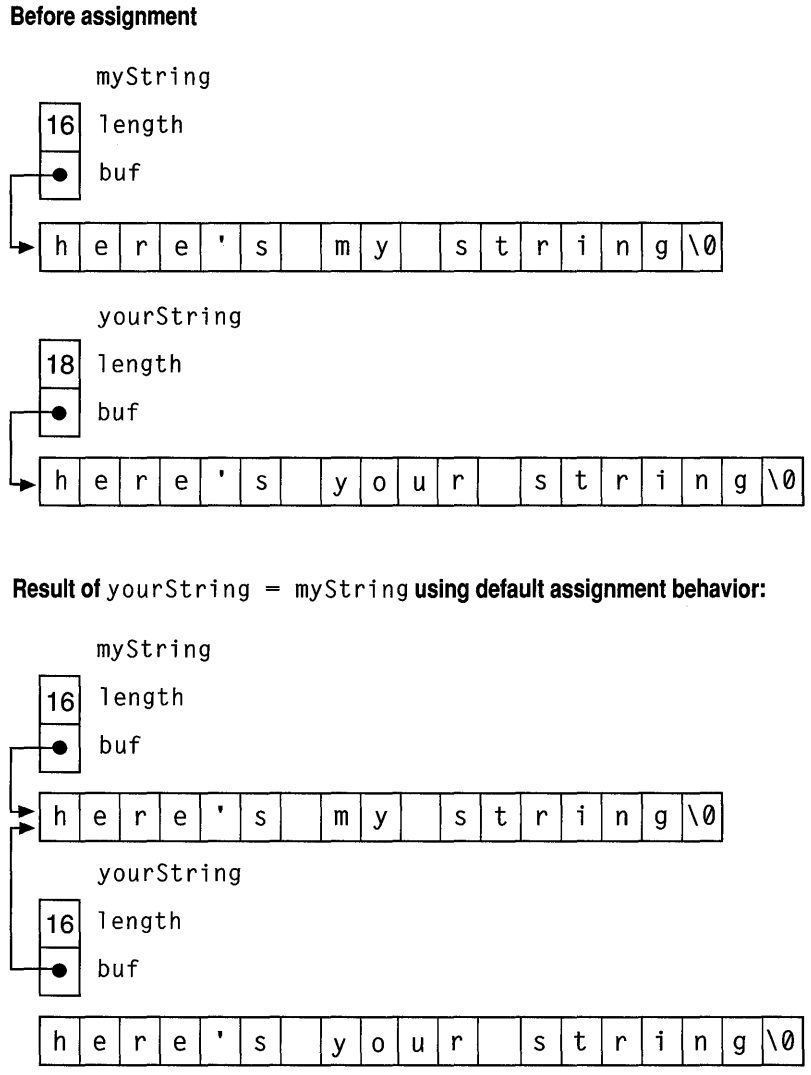


Figure 5.1 Default Assignment Behavior

This means that any modifications to one of the `String` objects affects both of them. If you call `myString.set()`, you modify `yourString` as well. This behavior probably isn't what you desired.

More serious problems arise when the objects go out of scope. When the `String` class's destructor is called for `myString`, it deletes the object's `buf` pointer, deallocating the memory that it points to. Then the destructor is called again for `yourString`, and it deletes that object's `buf` pointer. But both `buf` members have the same value, which means the pointer is deleted twice. This can cause unpredictable results. Furthermore, the original buffer in `yourString`, containing "here's your string," is lost. That block of memory is never deleted.

These problems occur for any class that has pointer members and allocates memory from the free store. The compiler's default behavior for assigning one object to another is unsatisfactory for such classes. The solution is to replace the compiler's default behavior by writing a special function to perform the assignment, called the "assignment operator."

The Assignment Operator

As described in Chapter 2, "C++ Enhancements to C," in C++ you can overload a function name so that it applies to more than one function. Similarly, you can overload the assignment operator (the = sign) to have more than one meaning; you can specify what happens when it is applied to instances of a particular class. This is known as "operator overloading." Chapter 8, "Operator Overloading and Conversion Functions," explains operator overloading in greater detail.

To redefine the meaning of the assignment operator for a class, you write a member function with the name **operator=**. If your class defines such a function, the compiler calls it whenever one object is assigned to another. The compiler interprets an assignment statement like this

```
yourstring = mystring;
```

as a function call that looks like this:

```
yourstring.operator=( mystring );
```

In fact, you can explicitly use the second syntax to perform assignments; however, you should use the first syntax because it is more readable.

The assignment operator for `String` can be written as follows:

```
// Class Assignment
#include <iostream.h>
#include <string.h>

class String
{
public:
   String();
   String( const char *s );
   String( char c, int n );
   void operator=( const String &other );
// etc...
};

// ---------- Assignment operator
void String::operator=( const String &other )
{
   length = other.length;
   delete [] buf;
   buf = new char[length + 1];
   strcpy( buf, other.buf );
}
```

The assignment operator takes a reference to an object as its parameter. (Note that a reference to a constant is used, indicating that the function doesn't modify the object.) To perform the assignment, the function first copies the `length` data member. Next, it deletes the receiving object's `buf` pointer, returning that block of memory to the free store (this is safe even for an uninitialized string, because deleting a 0 pointer has no effect). Then the function allocates a new buffer and copies the other buffer's contents into it. This is illustrated in Figure 5.2.

Result of `yourString = myString` after assignment operator has been defined:

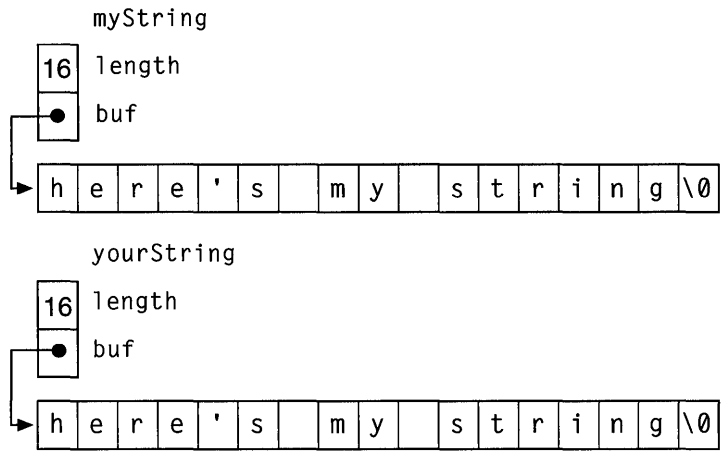


Figure 5.2 Correct Assignment Behavior

Here's a program that uses the new `String` class with its assignment operator:

```
main()
{
   String myString( "here's my string" );
   myString.display();
   cout << '\n';

   String yourString( "here's your string" );
   yourString.display();
   cout << '\n';

   yourString = myString;
   yourString.display();
   cout << '\n';
}
```

This program prints the following messages.

```
here's my string
here's your string
here's my string
```

What if a programmer using the `String` class accidentally assigns an object to itself? For instance:

```
myString = myString;  // Self-assignment
```

Few people would write such a statement, but self-assignment can take other forms. For instance:

```
String *stringPtr = &myString;

// Later...
myString = *stringPtr;     // Inconspicuous self-assignment
```

What happens during such an assignment? The **operator=** defined above first deletes `myString`'s buffer and allocates a new buffer. Then it copies the contents of `myString`'s newly allocated buffer into itself. This causes unpredictable behavior in your program.

In order for the **operator=** function to work safely in all cases, it must check against self-assignment. This requires the use of the **this** pointer.

The this Pointer

The **this** pointer is a special pointer that is accessible to member functions. The **this** pointer points to the object for which the member function is called. (There is no **this** pointer accessible to static member functions. Static member functions are described in Chapter 6, "More Features of Classes.")

When you call a member function for an object, the compiler assigns the address of the object to the **this** pointer and then calls the function. Every time a member function accesses one of the class's data members, it is implicitly using the **this** pointer.

For example, consider the following C++ code fragment, describing a member function definition and function call:

```
void Date::setMonth( int mn )
{
    month = mn;
}

...
// Member function call
myDate.setMonth( 3 );
```

This is roughly equivalent to the following C fragment:

```
// C equivalent of C++ member function
void Date_setMonth( Date *const this, int mn )
{
    this->month = mn;
}
...

// Function call
Date_setMonth( &myDate, 3 );
```

Notice that the type of **this** is `Date *` for member functions of `Date`; the type is different for member functions of other classes.

When you write a member function, it is legal to explicitly use the **this** pointer when accessing any members, though it is unnecessary. You can also use the expression ***this** to refer to the object for which the member function was called. Thus, in the following example, the three statements are equivalent:

```
void Date::month_display()
{
   cout << month;           // These three statements
   cout << this->month;     //     do the same thing
   cout << (*this).month;
 }
```

A member object can use the **this** pointer to test whether an object passed as a parameter is the same object that the member function is called for. For example, the **operator=** function for the `String` class can be rewritten as follows:

```
void String::operator=( const String &other )
{
   if( &other == this )
       return;
   delete [] buf;
   length = other.length;
   buf = new char[length + 1];
   strcpy( buf, other.buf );
}
```

The function tests whether the address of the `other` object is equal to the value of the **this** pointer. If so, a self-assignment is being attempted, so the function exits without doing anything. Otherwise, it performs the assignment as usual.

Using *this in a Return Statement

The **this** pointer can also be used in the **return** statement of a member function. In both C and C++, an assignment statement can be treated as an expression, which has the value of what was being assigned. For example, the statement

```
i = 3;
```

is an expression with the value 3.

One result of this is that you can chain together multiple assignment statements:

```
a = b = c;
```

The assignment operator is right associative, so the expression is evaluated from right to left. This means the expression is equivalent to the following:

```
a = (b = c);
```

To make your overloaded class assignments work this way, you must make the assignment function return the result of the assignment. You want the assignment operator to return the object to which it belongs. You get the address of the object from the **this** pointer.

Returning ***this** involves a simple modification to the assignment operator (in the **operator=** function):

```
String &String::operator=( const String &other )
{
   if( &other == this )
       return *this;
   delete [] buf;
   length = other.length;
   buf = new char[length + 1];
   strcpy( buf, other.buf );
   return *this;
}
```

With this version of the **operator=** function, you can chain together assignments of `String` objects:

```
herString = yourString = myString;
```

Note that the function returns a reference to a `String`. This is more efficient than returning an actual `String` object; for more information on returning objects from functions, see the section "The Copy Constructor" on page 83.

The practice of returning ***this** also explains how the chained `cout` statements used in previous examples work. You have seen many statements similar to the following:

```
cout << a << b << c;
```

The left-shift operator is left-associative, so this expression is evaluated from left to right. The overloaded left-shift operator returns ***this**, which is the `cout` object, so each variable is printed successively.

Bad Uses of the this Pointer

The **this** pointer is a **const** pointer, so a member function cannot change the pointer's value to make it point to something else. In early versions of C++, the **this** pointer was not a **const** pointer. This made it possible for a programmer to make assignments to the **this** pointer in order to perform customized memory allocation. For example:

```
// BAD TECHNIQUE: assignment to this
class foo
{
public:
    foo() { this = my_alloc( sizeof( foo ) ); }
   ~foo() { my_dealloc( this ); this = 0; }
};
```

This type of special processing is not allowed in the current version of C++. If you need customized memory allocation, you can write your own versions of **new** and **delete**. For more information, see the section "Class-Specific new and delete Operators" on page 107 in Chapter 6.

Early versions of C++ also let you examine the **this** pointer to distinguish between objects allocated on the stack and those allocated with the free store. On entry to a constructor, the **this** pointer had a value of 0 if the constructor was being called for an object allocated with **new** and had a nonzero value otherwise. This made it possible for you to perform different processing for dynamically allocated objects. This behavior is not supported in the current version of C++.

Assignment vs. Initialization

Consider the following two code fragments:

```
int i;

i = 3;
```

and

```
int i = 3;
```

In C, these two fragments have the same effect and can be regarded as the same. In C++, however, they are very different. In the first example, the integer `i` is *assigned* a value. In the second example, it is *initialized* with a value.

The differences are as follows:

- An assignment occurs when the value of an existing object is changed; an object can be assigned new values many times.
- An initialization occurs when an object is given an initial value when it is first declared; an object can be initialized only once.

One way to illustrate the difference is to consider variables declared as **const**. A constant variable can only be initialized; it cannot be assigned a new value. (Similarly, references are initialized with a variable, but they cannot be assigned a new variable.)

This distinction becomes important when using objects. Consider the previous examples with the integers replaced by `String` objects. Here's an assignment:

```
String myString( "this is my string" );
String yourString;

yourString = myString;  // Assign one String the value of another
```

Here's an initialization:

```
String myString( "this is my string" );
String yourString = myString;  // Initialize one String with another
```

As previously described, the assignment statement causes the compiler to invoke the **operator=** function defined for the class. However, the initialization does not invoke the same function. The **operator=** function can only be called for an object that has already been constructed. In the above example, `yourString` is being constructed at the same time that it receives the value of another object. To

construct an object in this way, the compiler invokes a special constructor called the "copy constructor."

The Copy Constructor

A copy constructor is a constructor that takes an object of the same type as an argument. It is invoked whenever you initialize an object with the value of another. It can be invoked with the = sign, as in the example above, or with function-call syntax. For example, the initialization in the example above could be rewritten with the following syntax:

```
String yourString( myString );
```

This follows the traditional syntax for calling a constructor.

The way the `String` class is currently written, the compiler executes the previous statement by initializing each member of `yourString` with the values of the members of `myString`. Just as with the default behavior during assignment, this is generally undesirable when the class contains pointers as members. The result of the previous initialization is to give `yourString` and `myString` the same character buffer, which can cause errors when the destructor destroys the objects.

The solution is to write your own copy constructor. The copy constructor for the `String` class can be written as follows:

```
#include <iostream.h>
#include <string.h>

// ------- string class
class String
{
public:
   String();
   String( const char *s );
   String( char c, int n );
   String( const String &other );   // Copy constructor
// etc...
};

// ----------- Copy constructor
String::String( const String &other )
{
   length = other.length;
   buf = new char[length + 1];
   strcpy( buf, other.buf );
}
```

The implementation of the copy constructor is similar to that of the assignment operator in that it allocates a new character buffer for the object being created. Note that the copy constructor actually takes a reference to an object, instead of an object itself, as a parameter.

In general, there are only a few differences between copy constructors and assignment operators:

- An assignment operator acts on an existing object, while a copy constructor creates a new one. As a result, an assignment operator may have to delete the memory originally allocated for the receiving object.
- An assignment operator must check against self-assignment. The copy constructor doesn't have to, because self-initialization is impossible.
- To permit chained assignments, an assignment operator must return ***this**. Because it is a constructor, a copy constructor has no return value.

Passing and Returning Objects

There are two other situations besides ordinary declarations in which the copy constructor may be called:

- When a function takes an object as a parameter.
- When a function returns an object.

The following example shows a function that takes an object as a parameter:

```
// Function that takes a String parameter
void consume( String parm )
{
    // Use the parm object
}

void main()
{
    String myString( "here's my string" );

    consume( myString );
}
```

The function `consume` takes a `String` object passed by value. That means that the function gets its own private copy of the object.

The function's parameter is initialized with the object that is passed as an argument. The compiler implicitly calls the copy constructor to perform this initialization. It does the equivalent of the following:

```
// Hypothetical initialization of parameter
String parm( myString );   // Call copy constructor
```

Consider what happens if you don't define a copy constructor to handle initialization. As a result of the compiler's default initialization, the function's copy of the object has the same character buffer as the caller's copy; any operations on `parm`'s buffer also modify `myString`'s buffer. More importantly, the parameter has local scope, so the destructor is called to destroy it when the function finishes executing. That means that `myString` has a pointer to deleted memory, which makes it unsafe to use after the function is done.

The following example shows a function that returns an object:

```
// Function that returns a String
String emit()
{
    String retValue( "here's a return value" );

    return retValue;
}

void main()
{
    String yourString;

    yourString = emit();
}
```

The function `emit` returns a `String` object. The compiler calls the copy constructor to initialize a hidden temporary object in the caller's scope, using the object specified in the function's **return** statement. This hidden temporary object is then used as the right-hand side of the assignment statement. That is, the compiler performs the equivalent of the following:

```
// Hypothetical initialization of return value
String temp( retValue ); // Call copy constructor
yourString = temp;       // Assignment of temp object
```

Once again, a copy constructor is needed. Otherwise, the temporary object shares the same character buffer as `retValue`, which is deleted when the function finishes executing, and the subsequent assignment to `yourString` is not guaranteed to work.

As a rule, you should always define both a copy constructor and an assignment operator whenever you write a class that contains pointer members and allocates memory from the free store.

Passing and Returning References to Objects

There is some overhead involved in calling the copy constructor every time an object is passed by value to a function. However, you can duplicate the effect of passing the parameter by value, while avoiding the expense of the constructor call, by passing a reference to a constant object. For example:

```
void consume( const String &parm )
{
    // Use the parm object
}

void main()
{
    String myString( "here's my string" );

    consume( myString );
}
```

The copy constructor is not called when a parameter is passed this way, because a new object is not being constructed. Instead, a reference is initialized with the object being passed. The compiler performs the equivalent of the following:

```
// Hypothetical initialization of reference parameter
const String &parm = myString;   // Initialize reference
```

As a result, the function uses the same object as the caller.

Notice that the **const** keyword is used. Because a reference to a constant is passed, the function cannot modify the parameter, so the caller is guaranteed that the object remains safe. Only **const** member functions (that is, read-only member functions) can be invoked on the object.

Note that the copy constructor itself takes a reference to an object, rather than an object, as its parameter. If the copy constructor took an object itself as a parameter, it would have to call itself in order to initialize the parameter. This would cause an infinite recursion.

Returning a reference from a function can also be more efficient than returning an object. Recall the example of the **operator=** function:

```
String &String::operator=( const String &other )
{
   //...
   return *this;
}

void main()
{
    String myString( "here's my string" );
    String yourString, herString;

    herString = yourString = myString;
}
```

The copy constructor is not called when the function returns, because a temporary object is not being created: Only a temporary reference is created. When the `herString` object receives the value of the `yourString = myString` assignment statement, the compiler performs the equivalent of the following:

```
// Hypothetical initialization of reference return value
String &tempRef = yourString;   // Initialize reference
                                // NOTE: yourString == *this
herString = tempRef;     // Assignment of temp reference
                         // Equivalent to herString = yourString
```

However, you must use caution when returning a reference to objects or variables other than ***this**. The rules for returning references are similar to those for returning pointers. You cannot return a pointer to an automatic variable. For example:

```
// BAD TECHNIQUE: returning pointer to automatic variable
int *emitPtr()
{
    int i;

    return &i;
}
```

The integer `i` is an automatic variable, so it goes out of scope at the end of the function. That means the function returns a pointer to an integer that no longer exists. It is unsafe for the calling program to use such a pointer.

The same restriction applies to references:

```
// BAD TECHNIQUE: returning reference to automatic variable
int &emitRef()
{
    int i;

    return i;
}
```

It is safe, however, to return a reference or a pointer to a variable that has been dynamically allocated. Dynamically allocated objects remain in existence until they are deallocated, so references and pointers to them remain valid even after the function has exited. You can also safely return references or pointers to static or global variables.

CHAPTER 6

More Features of Classes

This chapter describes the following additional features of classes:

- Static members
- Friend classes and functions
- Creating arrays of objects
- The **_set_new_handler** function
- Writing your own **new** and **delete** operators

Static Members

Suppose you write a class `SavingsAccount` to represent savings accounts at a bank. Each object represents a particular customer's account and has data members storing the customer's name and the account's current balance. The class also has a member function to increase an account's balance by the interest earned in one day.

In such a class, how would you represent the daily interest rate? The interest rate may change, so it has to be a variable instead of a constant. You could make it a member of the class, but then each object would have its own copy. This is not only a waste of space, but it also requires you to update every single object each time the interest rate changes, which is inefficient and could lead to inconsistencies.

You could make the interest rate a global variable, but then every function would be able to modify its value. What you want is a kind of global variable for an individual class. For such situations, C++ lets you declare a member of a class to be **static**.

Static Data Members

When a data member is declared **static**, only one copy of it is allocated, no matter how many instances of the class are declared. However, it can be treated like an ordinary data member by the class's member functions. If it is declared **private**, only the member functions can access it. For example, here's a declaration of a `SavingsAccount` class that contains a static member called `currentRate`:

```
class SavingsAccount
{
public:
    SavingsAccount();
    void earnInterest() { total += currentRate * total; }
    //...
private:
    char name[30];
    float total;
    static float currentRate;
    //...
};
```

Only one copy of `currentRate` exists, and it is accessible to all `SavingsAccount` objects. Whenever the `earnInterest` member is called for any `SavingsAccount` object, the same value of `currentRate` is used. This is illustrated in Figure 6.1.

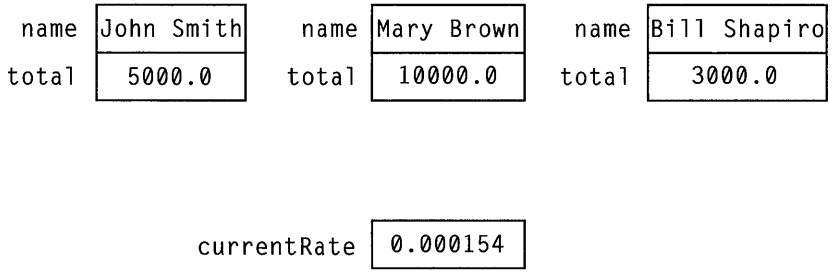


Figure 6.1 A Static Data Member

You can declare a static member **public**, making it visible to the rest of the program. You can then access it as if it were an ordinary data member of an object.

For example, if `currentRate` were a public member, you could access it as follows:

```
// If currentRate were a public member
void main()
{
    SavingsAccount myAccount;

    myAccount.currentRate = 0.000154;
}
```

However, this syntax is misleading, because it implies that only the interest rate of `myAccount` is being modified, when in fact the interest rate of all `SavingsAccount` objects is being modified. A better way of referring to a static member is to prefix its name with the class name and the scope resolution operator. For example:

```
// If currentRate were a public member
void main()
{
    SavingsAccount::currentRate = 0.000154;
}
```

This syntax reflects the fact that the value being modified applies to the class as a whole, rather than an individual object. You can use this syntax even if you haven't declared any `SavingsAccount` objects; a static data member exists even if no instances of the class are declared.

You cannot initialize a static data member from within a constructor of the class, because the constructor may be called many times and a variable can be initialized only once. A static data member must be initialized at file scope, as if it were a global variable. The access specifier for a static data member is not in effect during initialization; private static members are initialized in the same way as public ones. For example:

```
#include "savings.h"

// Initialize private static member at file scope
float SavingsAccount::currentRate = 0.0001;

    SavingsAccount::SavingsAccount()
{
    // ...
}
// etc....
```

Notice that the initialization is not placed in a header file, because that file may be included more than once in a program. The initialization is placed in the source module that contains the definitions of the class's member functions. Also note that the type of the static member is specified, because it is an initialization

rather than an assignment. The static member is being declared at that point, not inside the class.

Static Member Functions

If you have a member function that accesses only the static data members of a class, you can declare the function **static** as well. For example:

```
// See SAVINGS.H and SAVINGS.CPP for final versions
//      of this example
class SavingsAccount
{
public:
    SavingsAccount();
    void earnInterest() { total += currentRate * total; }
    static void setInterest( float newValue )
        { currentRate = newValue; }
    //...
private:
    char name[30];
    float total;
    static float currentRate;
    //...
};
```

Static member functions can be called using the same syntax as that used for accessing static data members. That is:

```
// Calling a static member function
void main()
{
    SavingsAccount myAccount;

    myAccount.setInterest( 0.000154 );
    SavingsAccount::setInterest( 0.000154 );
}
```

Because a static member function doesn't act on any particular instance of the class, it has no **this** pointer. Consequently, a static member function cannot access any of the class's nonstatic data members or call any nonstatic member functions, as doing so would mean implicitly using the **this** pointer. For example, the function `setInterest` cannot access the `total` data member; if it could, which object's value of `total` would it use?

Static members are useful for implementing common resources that all the objects need, or maintaining state information about the objects. One use of static members is to count how many instances of a class exist at any particular moment. This is done by incrementing a static member each time an object is created and decrementing it each time an object is destroyed. For example:

```
class Airplane
{
public:
    Airplane() { count++; }
    static int howMany() { return count; }
   ~Airplane() { count--; }
private:
    static int count;
};

// Initialize static member at file scope
int Airplane::count = 0;
```

By calling `howMany`, you can get the number of `Airplane` objects that exist at any particular time.

Friends

As mentioned in Chapter 4, "Introduction to Classes," you should declare your class's data members **private**, so that they're inaccessible to functions outside of the class. This lets you change the implementation of a class without affecting the programs that use the class.

Sometimes, however, you may find that two or more classes must work together very closely—so closely that it's inefficient for them to use each other's access functions. You may want one class to have direct access to another class's private data. You can permit this by using the **friend** keyword.

Friend Classes

In the following example, the class `YourClass` declares that the `YourOtherClass` class is a friend. This permits member functions of `YourOtherClass` to directly read or modify the private data of `YourClass`:

```
class YourClass
{
friend class YourOtherClass;
private:
    int topSecret;
};

class YourOtherClass
{
public:
    void change( YourClass yc )
};

void YourOtherClass::change( YourClass yc )
{
    yc.topSecret++;    // Can access private data
}
```

The **friend** declaration is not affected by the **public** or **private** keywords; you can place it anywhere in the class's declaration.

Notice that the **friend** declaration appears in `YourClass`. When you write `YourClass`, you specify those classes that you wish to have access to `YourClass`'s private data. Another programmer cannot write a class called `HisClass` and declare it to be a friend in order to gain access. For example:

```
class HisClass
{
// Cannot declare itself to be a friend of YourClass
public:
    void change( YourClass yc )
};

void HisClass::change( YourClass yc )
{
    yc.topSecret++;    // Error: can't access private data
}
```

Thus, you control who has access to the classes you write.

Notice that the **friend** keyword provides access in one direction only. While `YourOtherClass` is a friend of `YourClass`, the reverse is not true. Friendship is not mutual unless explicitly specified as such.

A list class demonstrates the usefulness of friend classes more realistically. Suppose you want to maintain a list of names and phone numbers, and you want to be able to specify someone's name and find his or her phone number. You could write a class like the following:

```
#include <string.h>

struct Record
{
    char name[30];
    char number[10];
};
const int MAXLENGTH = 100;

class PhoneList
{
friend class PhoneIter;
public:
    PhoneList();
    int add( const Record &newRec );
    Record *search( char *searchKey );
private:
    Record aray[MAXLENGTH];
    int firstEmpty;          // First unused element
};

PhoneList::PhoneList()
{
    firstEmpty = 0;
}

int PhoneList::add( const Record &newRec )
{
    if( firstEmpty < MAXLENGTH - 1 )
    {
        aray[firstEmpty++] = newRec;
        return 1;   // Indicate success
    }
    else return 0;
}
```

```
Record *PhoneList::search( char *searchKey )
{
    for( int i = 0; i < firstEmpty; i++ )
    if( !strcmp( aray[i].name, searchKey ) )
        return &aray[i];

    return 0;
}
```

Each `PhoneList` object contains an array of `Record` structures. You can add new entries and search through the existing entries by specifying a name. You can create as many `PhoneList` objects as you need for storing separate lists of names.

Now suppose you want to examine each of the entries stored in a `PhoneList` object, one by one; that is, you want to "iterate" through all the entries. One way to do this is to write an iterator class that is a friend of the `PhoneList` class.

Here's the friend class `PhoneIter`:

```
class PhoneIter
{
public:
    PhoneIter( PhoneList &m );
    Record *getFirst();
    Record *getLast();
    Record *getNext();
    Record *getPrev();
private:
    PhoneList *const mine;      // Pointer to a PhoneList object
    int currIndex;
};

PhoneIter::PhoneIter( const PhoneList &m )
    : mine( &m )          // Initialize the constant member
{
    currIndex = 0;
}

Record *PhoneIter::getFirst()
{
    currIndex = 0;
    return &(mine->aray[currIndex]);
}
```

```
Record *PhoneIter::getLast()
{
    currIndex = mine->firstEmpty - 1;
    return &(mine->aray[currIndex]);
}

Record *PhoneIter::getNext()
{
    if( currIndex < mine->firstEmpty - 1 )
    {
        currIndex++;
        return &(mine->aray[currIndex]);
    }
    else return 0;
}

Record *PhoneIter::getPrev()
{
    if( currIndex > 0 )
    {
        currIndex--;
        return &(mine->aray[currIndex]);
    }
    else return 0;
}
```

When you declare a `PhoneIter` object, you initialize it with a `PhoneList` object. The `PhoneIter` object stores your current position within the list. Here's a function that demonstrates the use of a `PhoneIter` object:

```
void printList( PhoneList aList )
{
    Record *each;
    PhoneIter anIter( aList );

    each = anIter.getFirst();
    cout << each->name << ' ' << each->number << '\n';
    while( each = anIter.getNext() )
    {
        cout << each->name << ' ' << each->number << '\n';
    }
}
```

By calling the `getNext` and `getPrev` member functions, you can move the current position forward or back, reading the elements in the list at the same time. With the `getFirst` and `getLast` functions, you can start at either end of the list.

The `PhoneIter` class is useful because you can declare several iterator objects for a particular `PhoneList` class. Thus, you can maintain several current positions within the list, like bookmarks, and you can move each one back and forth independently. This type of functionality is cumbersome to implement using only member functions.

An important characteristic of the `PhoneList` class is that users of the class don't know that it's implemented with an array. You could replace the array with a doubly linked list without affecting the class's interface. You would have to rewrite the `add` function to append a new node to the linked list and rewrite the `search` function to traverse the list, but the prototypes of those functions would remain the same as they are now. Programs that call the `add` and `search` functions don't have to be modified at all.

If you were to rewrite the `PhoneList` class in this way, you would also have to rewrite the `PhoneIter` class. Instead of containing the index of the current element, each `PhoneIter` object would contain a pointer to the current node. However, the available operations would not change; the class's interface would remain the same. (Together, the `PhoneList` and `PhoneIter` classes form an "abstract" phone list, which is defined only by its operations, not by its internal workings. Abstraction is discussed in Chapter 9, "Fundamentals of Object-Oriented Design.")

When you use the friend mechanism in C++, you are no longer writing a class that stands alone; you are writing two or more classes that are always used together. If you rewrite one class, you must also rewrite the other(s). You should therefore use the friend mechanism very sparingly; otherwise, you may have to rewrite large amounts of code whenever you change one class.

Friend Functions

You can also declare a single function with the **friend** keyword, instead of an entire class. For example:

```
class YourClass
{
friend void YourFunction( YourClass yc );
private:
    int topSecret;
};

void YourFunction( YourClass yc )
{
    yc.topSecret++;     // Modify private data
}
```

Friend functions are often used for operator overloading. For more information, see Chapter 8, "Operator Overloading and Conversion Functions."

Arrays of Class Objects

You can declare arrays of objects in the same way that you can declare arrays of any other data type. For example:

```
Date birthdays[10];
```

When you declare an array of objects, the constructor is called for each element in the array. If you want to be able to declare arrays without initializing them, the class must have a default constructor (that is, one that can be called without arguments). In the above example, the default `Date` constructor is called, initializing each element in the array to January 1, 1.

You can also provide initializers for each element in the array by explicitly calling the constructor with arguments. If you don't provide enough initializers for the entire array, the default constructor is called for the remaining elements. For example:

```
Date birthdays[10] = { Date( 2, 10, 1950 ),
                       Date( 9, 16, 1960 ),
                       Date( 7, 31, 1953 ),
                       Date( 1, 3, 1970 ),
                       Date( 12, 2, 1963 ) };
```

The previous example calls the `Date` constructor that takes three parameters for the first five elements of the array, and the default constructor for the remaining five elements.

Notice the syntax for calling a constructor explicitly. Unlike the usual syntax, which declares an object and initializes it, this syntax creates an object with a particular value directly. This is analogous to specifying the integer constant 123 instead of declaring an integer variable and initializing it.

If the class has a constructor that takes only one argument, you can specify just the argument as the initializer for an element. You can also mix different styles of initializer. For example:

```
String message[10] = { "First line of message\n",
                       "Second line of message\n",
                       String( "Third line of message\n"),
                       String( '-', 25 ),
                       String() };
```

In the previous example, the single-parameter constructor is called for the first three elements of the array, implicitly for the first two elements and explicitly for the third. The two-parameter constructor is called explicitly for the fourth element. The default constructor is called explicitly for the fifth element and implicitly for the remaining five elements.

The Free Store and Class Arrays

You can also use the **new** operator to dynamically allocate arrays of objects. For example:

```
String *text;
text = new String[5];
```

There is no way to provide initializers for the elements of an array allocated with **new**. The default constructor is called for each element in the array.

When you deallocate an array of objects with the **delete** operator, you must specify a pair of empty brackets to indicate that an array is being deleted. The consequences of using the wrong syntax are serious. For example:

```
delete text;     // Incorrect syntax for deleting array
```

When the previous statement is executed, the compiler treats `text` as a pointer to a `String`, so it calls the destructor for the object `*text`, and then it deallocates the space pointed to by `text`. However, `text` points to an entire array, not just a single object. The destructor is called only for `text[0]`, not for `text[1]` through `text[4]`. As a result, the character buffers allocated for those four `String` objects are never deallocated. This is illustrated in Figure 6.2.

If you use the correct syntax for deleting arrays, the destructor is called properly. For example:

```
delete [] text;
```

This syntax tells the compiler that `text` points to an array. The compiler looks up the size of the array, which was stored when the array was first allocated with **new**. Then the compiler calls the destructor for all the elements in the array, from `text[0]` to `text[4]`. The destructor deallocates the buffer for each of the objects in turn, and then the compiler deallocates the space pointed to by `text`. This is illustrated in Figure 6.3.

In earlier versions of C++, you had to specify the array size within the brackets when you called **delete**, and errors resulted if you specified a different size than used in the **new** call. In the latest version of C++, the compiler stores the sizes of all arrays allocated with **new** and ignores numbers specified when calling **delete**.

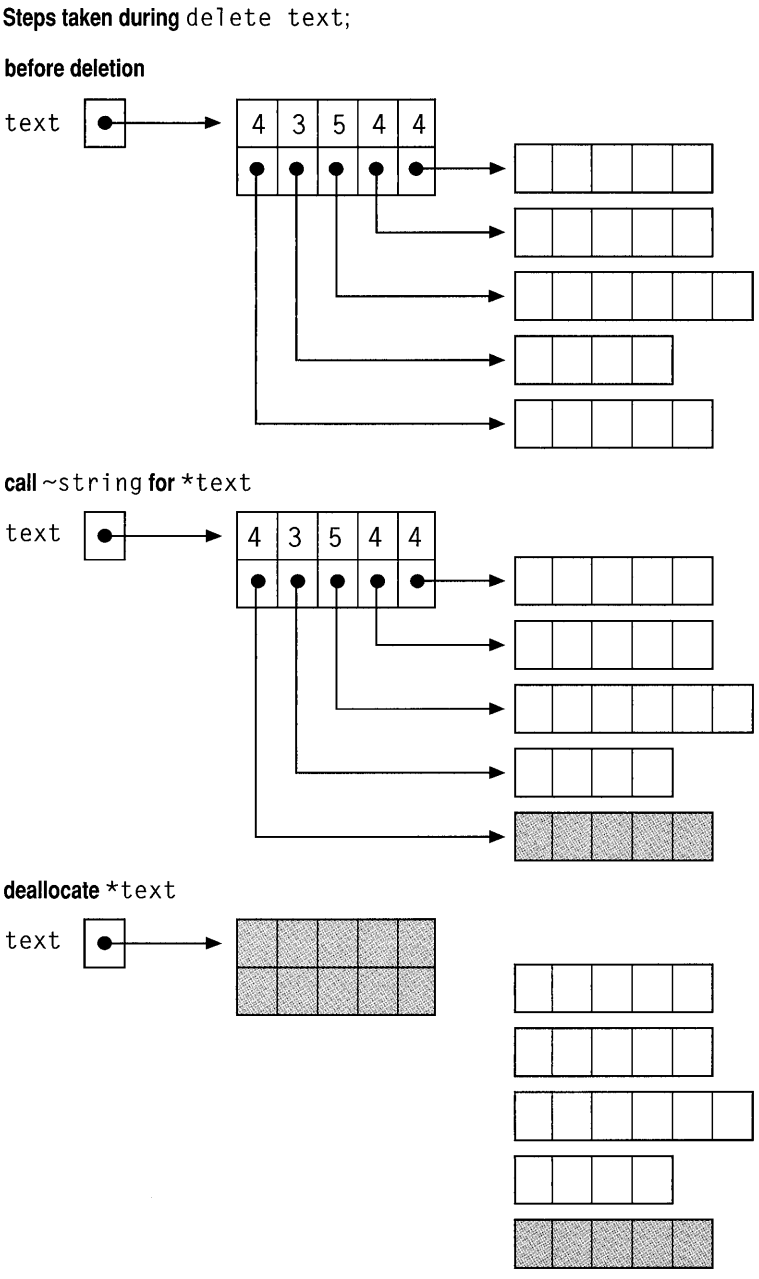


Figure 6.2 Incorrect Behavior for Deleting an Array

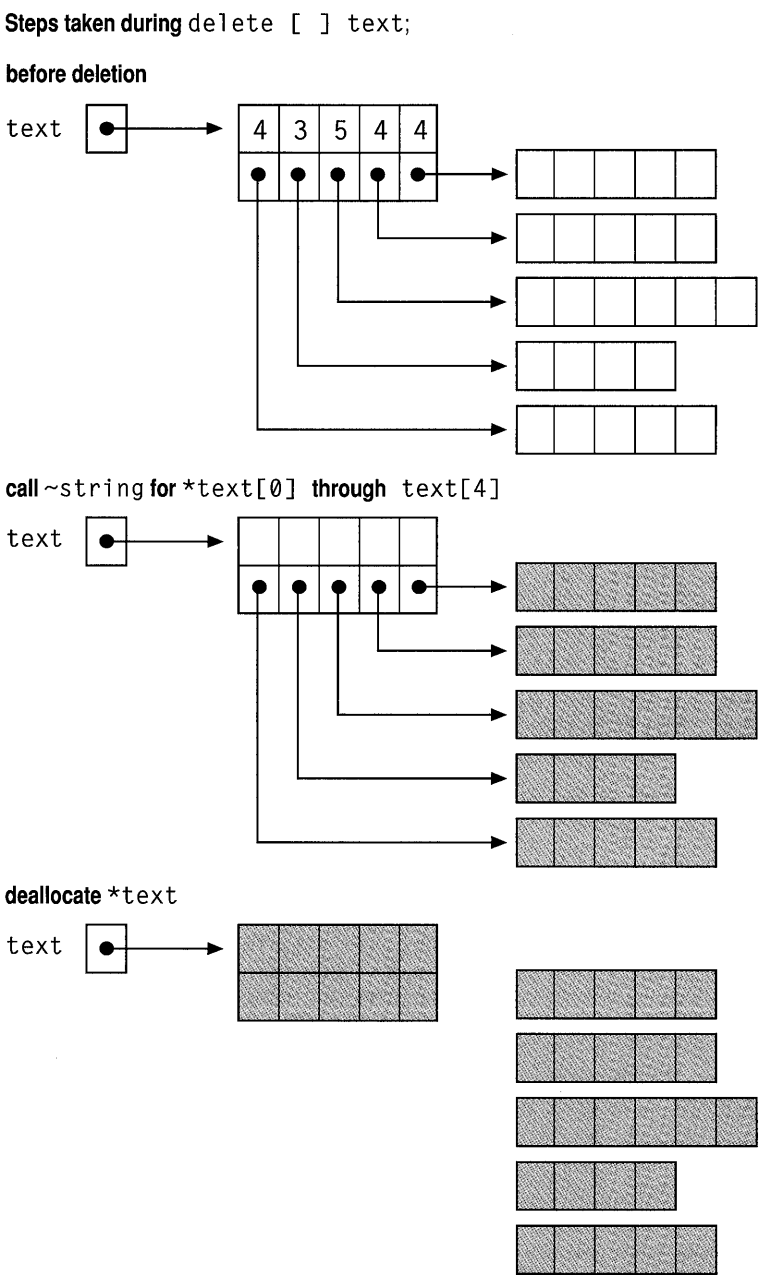


Figure 6.3 Correct Behavior for Deleting an Array

If you're using a class that has no destructor, it is possible, although inadvisable, to delete an array of objects without specifying the [ ]. For example, because the `Date` class has no destructor, the following example works:

```
// BAD TECHNIQUE: deleting array of objects without []
//     for a class that has no destructor
Date *appointments;

appointments = new Date[10];
// Use the array
delete appointments;   // Same as delete [] appointments;
```

In this case, the compiler notices that the `Date` class doesn't have a destructor, so it immediately deallocates the space pointed to by `appointments`. Because the `Date` objects have no buffers attached to them, no problems result from the lack of the destructor calls.

However, you should always use the [ ] syntax when deleting arrays, even for classes that have no destructors. The reason is that a class may be reimplemented later on, and the new implementation could perform dynamic memory allocation and require a destructor. (For example, you might implement a class using an array and then switch to a linked list later. The first version doesn't require a destructor, but the second version does.) If your programs assume that the class doesn't have a destructor, they might have to be modified later on. By consistently using the [ ] syntax whenever you delete arrays, you ensure that your programs work properly no matter how the class is implemented.

Advanced Free Store Techniques

C++ gives you much more control over dynamic allocation of memory than C does. The following sections describe ways you can customize the memory allocation in your program.

The _set_new_handler Function

The C function **malloc** returns **NULL** when it cannot allocate the requested amount of memory. When programming in C, it is good practice to check for a **NULL** return value every time you call **malloc**. This way, your program can exit gracefully instead of crashing as a result of trying to dereference a **NULL** pointer.

Similarly, the **new** operator returns 0 when it cannot allocate the requested amount of memory. Just as in C, you can check for a 0 return value every time you call **new**. However, C++ provides a more convenient alternative in the **_set_new_handler** function (declared in the include file NEW.H).

The **_set_new_handler** function takes a function pointer as an argument. This pointer must point to an error-handling function that you write. By calling **_set_new_handler**, you install this function as the error handler for the free store. When **new** cannot allocate the memory requested, it checks to see if an error handler has been installed. If no error handler is installed (which is the default), **new** returns 0. If you have installed an error handler, **new** calls it.

You can write a simple error-handling function that prints an error message and exits the program. For example:

```
// Free store exhaustion and the _set_new_handler function
#include <iostream.h>
#include <stdlib.h>
#include <new.h>

int all_gone( size_t size )
{
   cerr << "\n\aThe free store is empty\n";
   exit( 1 );
   return 0;
}

void main()
{
   _set_new_handler( all_gone );
   long total = 0;
   while( 1 )
   {
      char *gobble = new char[10000];
      total += 10000;
      cout << "Got 10000 for a total of " << total << '\n';
   }
}
```

This example executes a loop that consumes memory and displays the total amount of memory currently allocated. When **new** cannot allocate any more memory, it calls the `all_gone` function, which prints an error message and exits. Note that the `all_gone` function takes a parameter of type **size_t**, which represents the size of the block requested when **new** failed, and that it returns an integer. Any error-handling function you write must have this parameter and return type.

The previous example might print the following messages, depending on how much memory is available:

```
Got 10000 for a total of 10000
Got 10000 for a total of 20000
Got 10000 for a total of 30000
Got 10000 for a total of 40000
Got 10000 for a total of 50000
The free store is empty
```

An error-handling function like this removes the need for you to check the return value of **new** every time you call it. You can write code to handle the possibility of memory exhaustion in just one place, rather than throughout your program.

Overloading the new and delete Operators

C++ lets you redefine the behavior of the **new** and **delete** operators if you want to perform customized memory management. For example, suppose you want **new** to initialize the contents of a memory block to zero before returning the allocated memory. You can implement this by writing special functions named **operator new** and **operator delete**. For example:

```
// Customized new and delete
#include <iostream.h>
#include <stdlib.h>
#include <stddef.h>

// ------------- Overloaded new operator
void *operator new( size_t size )
{
    void *rtn = calloc( 1, size );
    return rtn;
}

// ----------- Overloaded delete operator
void operator delete( void *ptr )
{
    free( ptr );
}
```

```
void main()
{
    // Allocate a zero-filled array
    int *ip = new int[10];
    // Display the array
    for( int i = 0; i < 10; i++ )
        cout << " " << ip[i];
    // Release the memory
    delete [] ip;
}
```

Note that the **new** operator takes an parameter of type **size_t**. This parameter holds the size of the object being allocated, and the compiler automatically sets its value whenever you use **new**. Also note that the **new** operator returns a **void** pointer. Any **new** operator you write must have this parameter and return type.

In this particular example, **new** calls the standard C function **calloc** to allocate memory and initialize it to zero.

The **delete** operator takes a **void** pointer as a parameter. This parameter points to the block to be deallocated. Also note that the **delete** operator has a **void** return type. Any **delete** operator you write must have this parameter and return type.

In this example, **delete** simply calls the standard C function **free** to deallocate the memory.

Redefining **new** to initialize memory this way does not eliminate the call to a class's constructor when you dynamically allocate an object. Thus, if you allocate a `Date` object using your version of **new**, the `Date` constructor is still called to initialize the object after the **new** operator returns the block of memory.

You can also redefine **new** to take additional parameters. The following example defines a **new** operator that fills memory with the character specified when you allocate memory.

```
// new and delete with character fill
#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <stddef.h>

// ------------ Overloaded new operator
void *operator new( size_t size, int filler )
{
    void *rtn;
    if( (rtn = malloc( size )) != NULL )
        memset( rtn, filler, size );
    return rtn;
}

// ----------- Overloaded delete operator
void operator delete( void *ptr )
{
    free( ptr );
}

void main()
{
    // Allocate an asterisk-filled array
    char *cp = new( '*' ) char[10];
    // Display the array
    for( int i = 0; i < 10; i++ )
        cout << " " << cp[i];
    // Release the memory
    delete [] cp;
}
```

Notice that when you call this version of **new**, you specify the additional argument in parentheses.

For information about the behavior of **new**, **delete**, and **set_new_handler** in mixed-model programs, see Chapter 3, "Managing Memory for 16-Bit C++ Programs," in *Programming Techniques*.

Class-Specific new and delete Operators

You can also write versions of the **new** and **delete** operators that are specific to a particular class. This lets you perform memory management that is customized for a class's individual characteristics.

For example, you might know that there will never be more than a certain small number of instances of a class at any one time, but they'll be allocated and

deallocated frequently. You can use this information to write class-specific versions of **new** and **delete** that work faster than the global versions. You can declare an array large enough to hold all the instances of the class and then have **new** and **delete** manage the array.

To write class-specific **new** and **delete** operators, you declare member functions named **operator new** and **operator delete**. These operators take precedence over the global **new** and **delete** operators, in the same way that any member function takes precedence over a global function with the same name. These operators are called whenever you dynamically allocate objects of that class. For example:

```
// Class-specific new and delete operators
#include <iostream.h>
#include <string.h>
#include <stddef.h>

const int MAXNAMES = 100;

class Name
{
public:
   Name( const char *s ) { strncpy( name, s, 25 ); }
   void display() const { cout << '\n' << name; }
   void *operator new( size_t size );
   void operator delete( void *ptr );
   ~Name() {};    // do-nothing destructor
private:
   char name[25];
};
// -------- Simple memory pool to handle fixed number of Names
char pool[MAXNAMES] [sizeof( Name )];
int inuse[MAXNAMES];

// -------- Overloaded new operator for the Name class
void *Name::operator new( size_t size )
{
    for( int p = 0; p < MAXNAMES; p++ )
    if( !inuse[p] )
    {
        inuse[p] = 1;
        return pool + p;
    }
    return 0;
}
```

```
// --------- Overloaded delete operator for the Names class
void Name::operator delete( void *ptr )
{
    inuse[((char *)ptr - pool[0]) / sizeof( Name )] = 0;
}

void main()
{
    Name *directory[MAXNAMES];
    char name[25];

    for( int i = 0; i < MAXNAMES; i++ )
    {
        cout << "\nEnter name # " << i+1 << ": ";
        cin >> name;
        directory[i] = new Name( name );
    }
    for( i = 0; i < MAXNAMES; i++ )
    {
        directory[i]->display();
        delete directory[i];
    }
}
```

This program declares a global array called `pool` that can store all the `Name` objects expected. There is also an associated integer array called `inuse`, which contains true/false flags that indicate whether the corresponding entry in the pool is in use.

When the statement `directory[i] = new Name( name )` is executed, the compiler calls the class's **new** operator. The **new** operator finds an unused entry in `pool`, marks it as used, and returns its address. Then the compiler calls `Name`'s constructor, which uses that memory and initializes it with a character string. Finally, a pointer to the resulting object is assigned to an entry in `directory`.

When the statement `delete directory[i]` is executed, the compiler calls `Name`'s destructor. In this example, the destructor does nothing; it is defined only as a placeholder. Then the compiler calls the class's **delete** operator. The **delete** operator finds the specified object's location in the array and marks it as unused, so the space is available for subsequent allocations.

Note that **new** is called before the constructor, and that **delete** is called after the destructor. The following example illustrates this more clearly by printing messages when each function is called:

```
// Class-specific new and delete operators with constructor, destructor
#include <iostream.h>
#include <malloc.h>

class Name
{
public:
    Name()  { cout << "\nName's constructor running"; }
    void *operator new( size_t size );
    void operator delete( void *ptr );
    ~Name() { cout << "\nName's destructor running"; }
private:
    char name[25];
};

// -------- Simple memory pool to handle one Name
char pool[sizeof( Name )];

// -------- Overloaded new operator for the Name class
void *Name::operator new( size_t )
{
    cout << "\nName's new running";
    return pool;
}

// --------- Overloaded delete operator for the Name class
void Name::operator delete( void *p )
{
    cout << "\nName's delete running";
}

void main()
{
    cout << "\nExecuting: nm = new Name";
    Name *nm = new Name;
    cout << "\nExecuting: delete nm";
    delete nm;
}
```

The previous example does nothing with the class except display the following messages as the various functions execute:

```
Executing: nm = new Name
Name's new running
Name's constructor running
Executing: delete nm
Name's destructor running
Name's delete running
```

One consequence of the order in which **new** and **delete** are called is that they are static member functions, even if they are not declared with the **static** keyword. This is because the **new** operator is called before the class's constructor is called; the object does not exist yet, so it would be meaningless for **new** to access any of its members. Similarly, the **delete** operator is called after the destructor is called and the object no longer exists. To prevent **new** and **delete** from accessing any nonstatic members, the operators are always considered static member functions.

The class-specific **new** and **delete** operators are not called when you allocate or deallocate an array of objects; instead the global **new** and **delete** are called for array allocations. You can explicitly call the global versions of the operators when you allocate a single object by using the scope resolution operator (**::**). For example:

```
Name *nm = ::new Name;     // Use global new
```

If you have also redefined the global **new** operator, this syntax calls your version of the operator. The same syntax works for **delete**.

CHAPTER 7

Inheritance and Polymorphism

Besides making it easy for you to define new data types, C++ also lets you express relationships between those types. This is done with two of C++'s features: The first is "inheritance," which lets you define one type to be a subcategory of another. The second is "polymorphism," which lets you use related types together.

This chapter describes the mechanics of inheritance and polymorphism. In Part 3, "Object-Oriented Design," you'll see how these features play a role when you design a program.

This chapter covers the following topics:

- Base and derived classes
- Redefining members of a base class
- Conversions between base and derived classes
- Virtual functions and late binding
- Abstract classes
- The **protected** keyword

Before describing in detail C++'s features for handling related types, let's consider how you might handle them in C.

Handling Related Types in C

Suppose you need a program that maintains a database of all the employees in a company. The company has several different types of employee: regular employees, salespersons, managers, temporary employees, and so on, and your program must be able to handle all of them.

If you're writing this program in C, you could define a structure type called `employee` that has fields for the name, birth date, social security number, and other characteristics. However, each type of employee requires slightly different

information. For example, a regular employee's salary is based on an hourly wage and the number of hours worked, while a salesperson's salary also includes a commission on the number of sales made, and a manager's salary is a fixed amount per week.

It's difficult to find a way to represent the information about each employee. You could define a different structure type for each type of employee, but then you couldn't write a function that worked on all kinds of employees; you couldn't pass a `manager` structure to a function expecting an `employee` structure. Another possibility is to include all the possible fields in the `employee` structure type, but that would be a waste of space, because several fields would be empty for any given employee.

One solution in C is to define a structure that contains a union. For example:

```
/* Example of implementing related types in C */

struct wage_pay
{
    float wage;
    float hrs;
};

struct sales_pay
{
    float wage;
    float hrs;
    float commission;
    float sales_made;
};

struct mgr_pay
{
    float weekly_salary;
};

enum { WAGE_EMPLOYEE, SALESPERSON, MANAGER } EMPLOYEE_TYPE;
```

```
struct employee
{
    char name[30];
    EMPLOYEE_TYPE type;
    union
    {
        struct wage_pay worker;
        struct sales_pay seller;
        struct mgr_pay mgr;
    };                          // Anonymous union
};
```

The `employee` structure contains a union of the various salary structures. The program uses the `type` field to indicate the type of employee and to keep track of which form of salary is stored in the union.

Now consider how you would compute the salary of an employee. You might write a function that looks like this:

```
/* Example of type-specific processing in C */

float compute_pay( struct employee *emp )
{
    switch( emp->type )
    {
    case WAGE_EMPLOYEE:
        return emp->worker.hrs * emp->worker.wage;
        break;
    case SALESPERSON:
        return emp->seller.hrs * emp->seller.wage +
               emp->seller.commissions * emp->seller.sales_made;
        break;
    case MANAGER:
        return emp->mgr.weekly_salary;
        break;
    // ...
    };
}
```

This function uses the value of the `type` field to determine how it accesses the contents of the union. This way, the function can perform a different salary computation for each type of employee.

Salary computation is only one example of a task that is different for each type of employee. The employee-database program might use unions and **switch** statements for a wide variety of tasks, such as health plan management or vacation computation.

These **switch** statements have a couple of disadvantages:

- They can be difficult to read, especially if there is common processing for two or more types. It's also difficult to isolate the code that describes a particular type; for example, the code to handle the `SalesPerson` class is spread throughout the program.
- They are difficult to maintain. If you add a new type of employee, you have to add a new **case** statement that handles that type to every **switch** statement in the program. This makes updating the program error-prone, because it's possible to overlook a **switch** statement somewhere. In addition, every time you modify the code that handles one type, you must recompile the code that handles all the other types. This can be time-consuming when you're testing code for a new type of employee.

There are other ways to write this program in C, but they require much more programming effort. C doesn't provide an easy and maintainable way to express relations among multiple user-defined types. One of the goals in designing C++ was to remedy C's weakness in this area.

Handling Related Types in C++

Suppose you're writing the employee-database program in C++. First, define a class called `Employee` that describes the common characteristics of all employees. For example:

```
class Employee
{
public:
    Employee();
    Employee( const char *nm );
    char *getName() const;
private:
    char name[30];
};
```

For simplicity, this `Employee` class stores only a name, though it could store many other characteristics as well, such as a birth date, a social security number, and an address.

Next, you can define a `WageEmployee` class that describes a particular type of employee: those who are paid by the hour. These employees have the characteristics common to all employees, plus some additional ones.

There are two ways you can use `Employee` when you define the `WageEmployee` class. One way is to give `WageEmployee` an `Employee` object as a data member. However, that doesn't properly describe the relationship between the two types. A wage-earning employee doesn't contain a generic employee; rather, a wage-earning employee is a special type of employee.

The second possibility is inheritance, which makes one class a special type of another. You can make `WageEmployee` inherit from `Employee` with the following syntax:

```
class WageEmployee : public Employee
{
public:
    WageEmployee( const char *nm );
    void setWage( float wg );
    void setHours( float hrs );
private:
    float wage;
    float hours;
};
```

`WageEmployee` is a "derived class," and `Employee` is its "base class." To declare a derived class, you follow its name with a colon and the keyword **public**, followed by the name of its base class (you can also use the keyword **private**; this is described in the section "Public and Private Base Classes" on page 138). In the declaration of the derived class, you declare the members that are specific to it; that is, you describe the additional qualities that distinguish it from the base class.

Each instance of `WageEmployee` contains all of `Employee`'s data members, in addition to its own. You can call any of `Employee`'s or `WageEmployee`'s member functions for a `WageEmployee` object. For example:

```
WageEmployee aWorker( "Bill Shapiro" );
char *str;

aWorker.setHours( 40.0 );    // call WageEmployee::setHours
str = aWorker.getName();     // call Employee::getname
```

Figure 7.1 illustrates the members contained in `Employee` and `WageEmployee`.

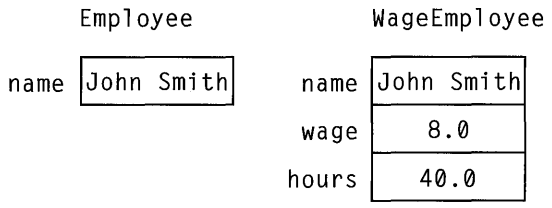


Figure 7.1 Data Members in Base and Derived Classes

The member functions of a derived class do not have access to the private members of its base class. Therefore, the member functions of `WageEmployee` cannot access the private members of its base class `Employee`. For example, suppose you write the following function:

```
void WageEmployee::printName() const
{
    cout << "Worker's name: "
         << name << '\n';        // Error: name is private
                                 //     member of Employee
}
```

Because `name` is one of the private members of the base class, it is inaccessible to any member function of `WageEmployee`.

This restriction may seem surprising. After all, if a `WageEmployee` is a kind of `Employee`, why shouldn't it have access to its own `Employee` characteristics? This restriction is designed to enforce encapsulation. If a derived class had access to its base class's private data, then anyone could access the private data of a class by simply deriving a new class from it. The point of making data private is to prevent programmers who use your class from writing code that depends on its implementation details, and this includes programmers who write derived classes. If the original class's implementation were changed, every class that derived from it would have to be rewritten as well.

Consequently, a derived class must use the base class's public interface, just as any other user of the class must. You could rewrite the previous example as follows:

```
void WageEmployee::printName() const
{
    cout << "Worker's name: "
         << getName() << '\n'; // Call Employee::getName
}
```

This function uses `Employee`'s public interface to get the information it needs.

To make this C++ example more like the employee example in C, you can also define classes that describe salespersons and managers. Because salespersons are a kind of wage-earning employee, you can derive the `SalesPerson` class from the `WageEmployee` class.

```
class SalesPerson : public WageEmployee
{
public:
    SalesPerson( const char *nm );
    void setCommission( float comm );
    void setSales( float sales );
private:
    float commission;
    float salesMade;
};
```

A `SalesPerson` object contains all the data members defined by `Employee` and `WageEmployee`, as well as the ones defined by `SalesPerson`. Similarly, you can call any of the member functions defined in these three classes for a `SalesPerson` object. (The `Employee` class is considered an "indirect" base class of `SalesPerson`, while the `WageEmployee` class is a "direct" base class of `SalesPerson`.)

Notice that this declaration means that `WageEmployee` is both a derived class and a base class. It derives from the `Employee` class and serves as the base for the `SalesPerson` class. You can define as many levels of inheritance as you want.

Managers are a type of employee that receives a fixed salary. Accordingly, you can derive the `Manager` class from `Employee`, as follows:

```
class Manager : public Employee
{
public:
    Manager( const char *nm );
    void setSalary( float salary );
private:
    float weeklySalary;
};
```

The inheritance relationships among all of these classes are shown in Figure 7.2. This figure illustrates a "class hierarchy," or a group of user-defined types organized according to their relationship to one another. The class at the top represents the most general type, and the classes at the bottom represent the more specialized types. As you'll learn in Part 3, "Object-Oriented Design," designing an appropriate class hierarchy is one of the most important steps in writing an object-oriented program.

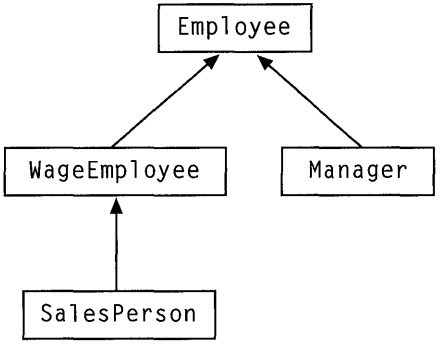


Figure 7.2 Employee Class Hierarchy

Notice that `Employee` acts as a base class for more than one class (`WageEmployee` and `Manager`). Any number of derived classes can inherit from a given base class.

Also notice that the `Manager` class shares members only with `Employee`. It doesn't have any of the members defined by `WageEmployee` or `SalesPerson`.

Redefining Members of the Base Class

Now consider how to compute the weekly pay of the various types of employees. You can define a member function for `WageEmployee` called `computePay`. For example:

```
float WageEmployee::computePay() const
{
    return wage * hours;
}
```

You can also give the `SalesPerson` class a `computePay` member function, just as with its base class. As mentioned above, this function cannot access any private members of `WageEmployee`, so the following function generates an error:

```
float SalesPerson::computePay() const
{
    return hours * wage +           // Error: hours and
           commission * salesMade;  //    wages are private
}
```

You must call a public member function of the base class. The following implementation calls such a function, but it does not work either:

```
float SalesPerson::computePay() const
{
    return computePay() +           // Bad recursive call
           commission * salesMade;
}
```

The compiler assumes that `computePay` refers to `SalesPerson`'s version of the function. This results in infinite recursion. You must use the scope resolution operator (`::`) to specify the base class's version of the function. For example:

```
float SalesPerson::computePay() const
{
    // Call base class's version of computePay
    return WageEmployee::computePay() +
           commission * salesMade;
}
```

This technique is commonly used when redefining a member function in a derived class. The derived class's version calls the base class's version and then performs any additional operations needed.

When you call a redefined member function for an object of a derived class, the derived class's version of the function is used. For example, when using a `SalesPerson` object, any call to `computePay` invokes `SalesPerson`'s version of the function. For example:

```
SalesPerson aSeller( "John Smith" );

aSeller.setHours( 40.0 );
aSeller.setWage( 6.0 );
aSeller.setCommission( 0.05 );
aSeller.setSales( 2000.0 );
//   Call SalesPerson::computePay
cout << "Seller salary: "
     << aSeller.computePay() << '\n';
```

Within this class, the `computePay` function defaults to the definition in the `SalesPerson` class. Again, to call the base class's version of the function, you must use the scope resolution operator. For example:

```
cout << "Seller base salary: "
     << aSeller.WageEmployee::computePay() << '\n';
```

You can also give the `Manager` class a `computePay` member function:

```
float Manager::computePay() const
{
    return weeklySalary;
}
```

This function involves no redefining of the similarly named functions in `WageEmployee` or `SalesPerson`, because neither of those classes are derived or base classes of `Manager`.

Derived Class Constructors

An instance of a derived class contains all the members of the base class, and all of those members must be initialized during construction. Consequently, the base class's constructor has to be called by the derived class's constructor. When you write the constructor for a derived class, you must specify a "base initializer," using syntax similar to that of the member initializer list for constructing member objects. Place a colon after the argument list of the derived class's constructor, and follow it with the name of the base class and an argument list. For example:

```
// Constructor function for WageEmployee
WageEmployee::WageEmployee( const char *nm )
    : Employee( nm )
{
    wage = 0.0;
    hours = 0.0;
}

// Constructor function for SalesPerson
SalesPerson::SalesPerson( const char *nm )
    : WageEmployee( nm )
{
    commission = 0.0;
    salesMade = 0.0;
}
```

```
// Constructor function for Manager
Manager::Manager( const char *nm )
    : Employee( nm )
{
    weeklySalary = 0.0;
}
```

When you declare an object of a derived class, the compiler executes the constructor for the base class first and then executes the constructor for the derived class. (If the derived class contains member objects, their constructors are executed after the base class's constructor, but before the derived class's constructor.)

You can omit the base initializer if the base class has a default constructor. As with member objects, however, you should use the base initializer syntax rather than perform redundant initialization.

If you're defining a derived class that also has member objects, you can specify both a member initializer list and a base initializer. However, you cannot define member initializers for member objects defined in the base class, because that would permit multiple initializations.

Conversions Between Base and Derived Classes

Because a salesperson is a kind of wage-earning employee, it makes sense to be able to use a `SalesPerson` object whenever an `WageEmployee` object is needed. To support this relationship, C++ lets you implicitly convert an instance of a derived class into an instance of a base class. For example:

```
WageEmployee aWorker;
SalesPerson aSeller( "John Smith" );

aWorker = aSeller;   // Convert SalesPerson to WageEmployee
                     //     derived => base
```

All the members of the `SalesPerson` object receive the values of the corresponding members in the `WageEmployee` object. However, the reverse assignment is not legal:

```
aSeller = aWorker;    // Error; cannot convert
```

Because `SalesPerson` has members that `WageEmployee` doesn't, their values would be undefined after such an assignment. This restriction follows the conceptual relationship between the types of employee: A worker is not necessarily a salesperson.

You can also implicitly convert a pointer to a derived class object into a pointer to a base class object. For example:

```
Employee *empPtr;
WageEmployee aWorker( "Bill Shapiro" );
SalesPerson aSeller( "John Smith" );
Manager aBoss( "Mary Brown" );

empPtr = &aWorker;    // Convert WageEmployee * to Employee *
empPtr = &aSeller;    // Convert SalesPerson * to Employee *
empPtr = &aBoss;      // Convert Manager * to Employee *
```

You can use a pointer to an `Employee` to point to a `WageEmployee` object, a `SalesPerson` object, or a `Manager` object.

When you refer to an object through a pointer, the type of the pointer determines which member functions you can call. If you refer to a derived class object with a base class pointer, you can call only the functions defined by the base class. For example:

```
SalesPerson aSeller( "John Smith" );
SalesPerson *salePtr;
WageEmployee *wagePtr;

salePtr = &aSeller;
wagePtr = &aSeller;

wagePtr->setHours( 40.0 );      // Call WageEmployee::setHours
salePtr->setWage( 6.0 );        // Call WageEmployee::setWage
wagePtr->setSales( 1000.0 );    // Error;
                                //     no WageEmployee::setSales
salePtr->setSales( 1000.0 );    // Call SalesPerson::setSales
salePtr->setCommission( 0.05 ); // Call SalesPerson::setCommission
```

Both `wagePtr` and `salePtr` point to a single `SalesPerson` object. You cannot call `setSales` through `wagePtr`, because `WageEmployee` doesn't define that member function. You have to use `salePtr` to call the member functions that `SalesPerson` defines.

If you call a member function that is defined by both the base class and the derived class, the function that is called depends on the type of the pointer. For example:

```
float base, total;

base = wagePtr->computePay();   // Call WageEmployee::computePay
total = salePtr->computePay();  // Call SalesPerson::computePay
```

When you use `wagePtr`, you call the version defined by `WageEmployee`. When you use `salePtr`, you call the version defined by `SalesPerson`.

To perform the reverse conversion (that is, from a pointer to a base class to a pointer to a derived class), you must use an explicit cast.

```
WageEmployee *wagePtr = &aSeller;
SalesPerson *salePtr;

salePtr = (SalesPerson *)wagePtr;   // Explicit cast required
                                    //     base => derived
```

This conversion is dangerous, because you can't be sure what type of object the base class pointer points to. Suppose `empPtr` points to something other than a `SalesPerson` object:

```
Employee *empPtr = &aWorker;
SalesPerson *salePtr;

salePtr = (SalesPerson *)empPtr;  // Legal, but incorrect
salePtr->setCommission( 0.05 );   // Error: aWorker has no
                                  //    setCommission member
```

This can cause your program to crash. Accordingly, you should be extremely careful when converting a base class pointer to a derived class pointer.

Collections Using Base Class Pointers

The conversion from a derived class pointer to a base class pointer is very useful. For example, if you have a function that expects a pointer to an `Employee` as a parameter, you can pass this function a pointer to any type of employee.

One application of this is to maintain a collection of employees. You could write an `EmployeeList` class that maintains a linked list, each node holding a pointer to an `Employee` object. For example:

```
class EmployeeList
{
public:
    EmployeeList();
    add( Employee *newEmp );
    // ...
private:
    // ...
};
```

Using the `add` function, you can insert any type of employee into an `EmployeeList` object:

```
EmployeeList myDept;
WageEmployee *wagePtr;
SalesPerson *salePtr;
Manager *mgrPtr;

// Allocate new objects
wagePtr = new WageEmployee( "Bill Shapiro" );
salePtr = new SalesPerson( "John Smith" );
mgrPtr = new Manager( "Mary Brown" );
// Add them to the list
myDept.add( wagePtr );
myDept.add( salePtr );
myDept.add( mgrPtr );
```

Once you have a list of employees, you can manipulate its contents using the `Employee` class's interface, even though the list contains all different types of employees. For example, you can define an iterator class called `EmpIter` (like the one described in Chapter 6, "More Features of Classes"), which can return each element of an `EmployeeList`. Then you can print a list of all the employees' names as follows:

```
void printNames( EmployeeList &dept )
{
    int count = 0;
    Employee *person;
    EmpIter anIter( dept );    // Iterator object
    person = anIter.getNext();
    count++;
    cout << count << ' ' << person->getName() << '\n';

    while( person = anIter.getNext() )
    {
        count++;
        cout << count << ' '
             << person->getName() << '\n';
     }
}
```

This function iterates through all the elements in the `EmployeeList` object passed as a parameter. For each employee in the list, no matter what type it is, the iterator returns an `Employee` pointer. Using this pointer, the function prints out the employee's name.

The problem with this technique is that you cannot treat an object as anything more than a generic `Employee`. For instance, how could you compute the weekly salary of each employee in the list? If you were to give the `Employee` class a `computePay` function, calling that function wouldn't invoke the `computePay` functions defined in the derived classes. As mentioned earlier, the function that is called is determined by the type of the pointer. Accordingly, calling `computePay` using only `Employee` pointers would perform the same computation for every type of employee, which is clearly unsatisfactory.

What you need is a way to call each class's individual version of `computePay` while still using generic `Employee` pointers. C++ provides a way to do this using virtual functions.

Virtual Functions

A "virtual function" is a member function that you expect to be redefined in derived classes. When you call a virtual function through a pointer to a base class, the derived class's version of the function is executed. This is precisely the opposite behavior of ordinary member functions.

A virtual function is declared by placing the keyword **virtual** before the declaration of the member function in the base class. Global functions and static members cannot be virtual functions. The **virtual** keyword is not necessary in the declarations in the derived classes; all subsequent versions of a virtual function

are implicitly declared **virtual**. For example, here is a revised version of the employee class hierarchy that has a virtual `computePay` function:

```
class Employee
{
public:
    Employee( const char *nm );
    char *getName() const;
    virtual float computePay() const;
    virtual ~Employee() {}
private:
    char name[30];
};

class WageEmployee : public Employee
{
public:
    WageEmployee( const char *nm );
    void setWage( float wg );
    void setHours( float hrs );
    float computePay() const;     // Implicitly virtual
private:
    float wage;
    float hours;
};

class SalesPerson : public WageEmployee
{
public:
    SalesPerson( const char *nm);
    void setCommission( float comm );
    void setSales( float sales );
    float computePay() const;     // Implicitly virtual
private:
    float commission;
    float salesMade;
};

class Manager : public Employee
{
public:
    Manager( const char *nm );
    void setSalary( float salary );
    float computePay() const;     // Implicitly virtual
private:
    float weeklySalary;
};
```

The definitions of each class's version of `computePay` do not have to be modified. However, because `computePay` has been added to the base class, a definition for that version of the function is needed:

```
float Employee::computePay() const
{
    cout << "No salary computation defined\n";
    return 0.0;
}
```

This function is needed primarily as a placeholder. It would be called if a plain `Employee` object were used, or if one of the derived classes did not provide its own definition of `computePay`.

Now consider what happens when `computePay` is called through an `Employee` pointer:

```
Employee *empPtr;
float salary;

empPtr = &aWorker;
salary = empPtr->computePay();  // Call WageEmployee::computePay
empPtr = &aSeller;
salary = empPtr->computePay();  // Call SalesPerson::computePay
empPtr = &aBoss;
salary = empPtr->computePay();  // Call Manager::computePay
```

If `computePay` hadn't been declared **virtual**, each statement would call `Employee::computePay`, which would return 0.0. However, because `computePay` is a virtual function, the function executed is different for each call, even though the calls are exactly the same. The function called is the one appropriate for the actual object that `empPtr` points to. (You can also use the scope resolution operator to explicitly specify a different version of the function if you want.)

To calculate the weekly payroll for a department, you can write a function like the following:

```
float computePayroll( EmployeeList &dept )
{
    float payroll = 0;
    Employee *person;
    EmpIter anIter( dept );

    person = anIter.getFirst();
    payroll += person->computePay();
    while( person = anIter.getNext() )
    {
        // Call appropriate function
        //     for each type of employee
        payroll += person->computePay();
    }

    return payroll;
}
```

The statement `person->computePay` executes the appropriate function, no matter what type of employee `person` points to.

Polymorphism

The ability to call member functions for an object without specifying the object's exact type is known as "polymorphism." The word "polymorphism" means "the ability to assume many forms," referring to the ability to have a single statement invoke many different functions. In the above example, the pointer `person` can point to any type of employee and the name `computePay` can refer to any of the salary computation functions.

Compare this with the implementation in C provided earlier in this chapter. In C, if all you have is a pointer to an employee, you have to call the `compute_pay` function shown earlier, which must execute a **switch** statement to find the exact type of employee. In C++, the statement `person->computePay()` calls the appropriate function automatically, without requiring you to examine the type of object that `person` points to. (There is only a tiny amount of overhead, as described in the section "How Virtual Functions are Implemented" on page 132.) No **switch** statement is needed.

Computing salaries is just one example of a task that differs depending on the type of employee. A more realistic `Employee` class would have several virtual functions, one for each type-dependent operation. An employee-database

program would have many functions like `computePayroll`, all of which manipulate employees using `Employee` pointers and virtual functions.

In such a program, all the information about any particular type of employee is localized in a single class. You don't have to look at every employee-database function to see how salespersons are handled. All the specialized salesperson processing is contained in the `SalesPerson` class. It's also easy to add a new type of employee, due to a property known as "dynamic binding."

Dynamic Binding

At compile time, the compiler cannot identify the function that is called by the statement `person->computePay()`, because it could be any of several different functions. The compiler must evaluate the statement at run time, when it can tell what type of object `person` points to. This is known as "late binding" or "dynamic binding." This behavior is very different from function calls in C, or nonvirtual function calls in C++. In both these cases, the function call statement is translated at compile time into a call to a fixed function address. This is known as "early binding" or "static binding."

Dynamic binding makes it possible for you to modify the behavior of code that has already been compiled. You can make an existing module handle new types without having to modify the source and recompile it.

For example, suppose that the function `computePayroll` and all the other employee-database functions have been compiled into a module called EMPUTIL.OBJ. Now suppose that you want to define a new type of employee called a `Consultant` and use it with all the existing employee-database functions.

You don't have to modify the source code for `computePayroll` or any other functions in EMPUTIL.OBJ. You simply derive `Consultant` from the `Employee` class, define its member functions in a new source file CONSULT.CPP, and compile it into CONSULT.OBJ. Then you modify your program's user interface to allow users to enter information on consultants. After you've recompiled that portion of the program, you can link it with CONSULT.OBJ and EMPUTIL.OBJ to produce a new executable file.

You can then add `Consultant` objects to the `EmployeeList` object that the program already maintains. When you compute the payroll, the `computePayroll` function works just as it did before. If there are any `Consultant` objects in the list, the statement `person->computePay()` automatically calls `Consultant::computePay`, even though that function didn't exist when the statement was first compiled.

Dynamic binding makes it possible for you to provide a library of classes that other programmers can extend even if they don't have your source code. All you need to distribute are the header files (the .H files) and the compiled object code (.OBJ or .LIB files) for the hierarchy of classes you've written and for the functions that use those classes. Other programmers can derive new classes from yours and redefine the virtual functions you declared. Then the functions that use your classes can handle the classes they've defined.

How Virtual Functions Are Implemented

An obvious question about dynamic binding is "how much overhead is involved?" Is the added convenience gained at the expense of execution speed? Fortunately, virtual functions are very efficient, so calling one takes only slightly longer than calling a normal function.

In some situations, a virtual function call can be implemented as a normal function call—that is, using static binding. For example:

```
SalesPerson aSeller( "John Smith" );
SalesPerson *salePtr;
float salary;

salePtr = &aSeller;
salary = aSeller.computePay();   // Static binding possible
salary = salePtr->computePay();  // Static binding possible
```

In this example, `SalesPerson::computePay` can be called directly because the type of `aSeller` is known. Similarly, the type of the object that `salePtr` points to is known, and again the function can be called directly. In situations where the compiler cannot use static binding, such as the statement `person->computePay()` in the earlier example, the compiler uses dynamic binding.

Dynamic binding is implemented in C++ through the use of a virtual function table, or a "v-table." This is an array of function pointers that the compiler constructs for every class that uses virtual functions. For example, `WageEmployee`, `SalesPerson`, and `Manager` each have their own v-table.

The v-table contains one function pointer for each virtual function in the class. All of the employee classes have only one virtual function, so all of their v-tables contain just one pointer. Each pointer points to the version of the function that is appropriate to that class. Thus, the v-table for `SalesPerson` has a pointer to `SalesPerson::computePay`, and the v-table for `Manager` has a pointer to `Manager::computePay`. This is illustrated in Figure 7.3.

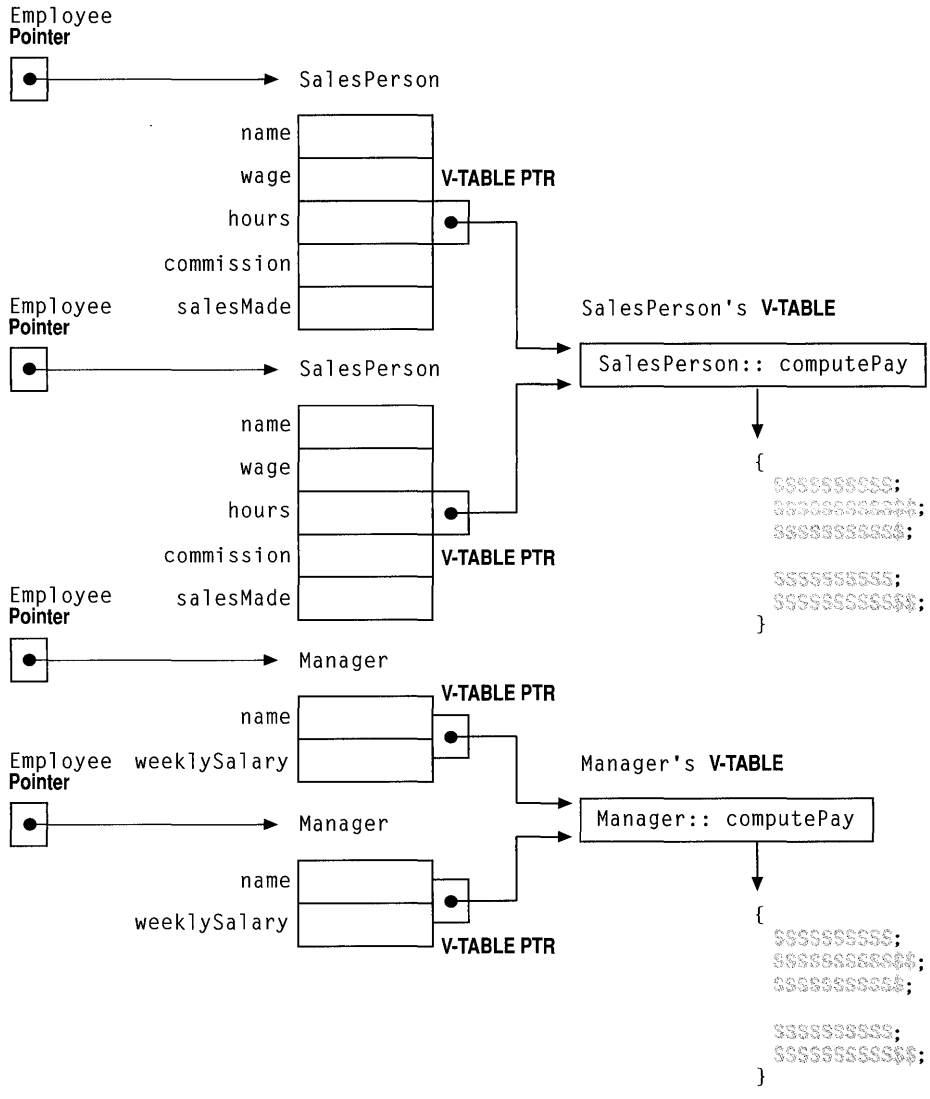


Figure 7.3 How Virtual Functions Are Implemented

Note that it is not required for a derived class to redefine a virtual function declared in its base class. For example, suppose `SalesPerson` did not define a `computePay` function. Then `SalesPerson`'s v-table would contain a pointer to `WageEmployee::computePay`. If `WageEmployee` in turn did not define `computePay`, both classes' v-tables would have pointers to `Employee::computePay`.

Each instance of a class contains a hidden pointer to the class's v-table. When a statement like `person->computePay()` is executed, the compiler dereferences the v-table pointer in the object pointed to by `person`. The compiler then calls the `computePay` function pointed to by the pointer in the v-table. In this way, the compiler calls a different function for each type of object.

The only difference between a normal function call and a virtual function call is the extra pointer dereferencing. A virtual function call usually executes as fast as or faster than the **switch** statement that would otherwise be used.

Pure Virtual Functions

In the example of the employee-database program, the `Employee` class defines a do-nothing `computePay` function. This solution is somewhat inelegant, because the function is intended never to be called.

A better solution is to declare `computePay` as a "pure virtual function." This is done by specifying an equal sign and a zero after the member function's prototype, as follows:

```
class Employee
{
public:
    // ...
    virtual float computePay() const = 0;  // Pure virtual
};
```

A pure virtual function requires no definition; you don't have to write the body of `Employee::computePay`. It is intended to be redefined in all derived classes. In the base class, the function serves no purpose except to provide a polymorphic interface for the derived classes.

You cannot declare any instances of a class in which a function is declared as pure virtual. For example, because `computePay` is now a pure virtual function, you cannot declare any objects of type `Employee`.

This restriction is necessary to prevent anyone from calling a pure virtual function for an object. If you could declare a generic `Employee` object, you could call `computePay` for it, which would be meaningless. You can only declare objects of the derived classes that provide a definition for `computePay`.

A class that defines pure virtual functions is known as an "abstract class," because you cannot declare any instances of it. (Classes that you can declare instances of are sometimes called "concrete classes.") It is legal, however, to declare pointers to an abstract class. For example, you can declare `Employee` pointers and use them for manipulating objects of derived classes. This is the way `computePayroll` and the other employee-database functions work.

If a derived class does not provide a definition for a pure virtual function, the function is inherited as pure virtual, and the derived class becomes an abstract class too. This does not happen with ordinary virtual functions, because when a derived class omits a definition of an ordinary virtual function, it uses the base class's version. With pure virtual functions, the derived class cannot use the base class's version because the base class doesn't have a version. Thus, if `WageEmployee` did not define a version of `computePay`, it would be an abstract class too.

It's common to write a class hierarchy consisting of one or more abstract classes at the top that act as base classes for the concrete classes at the bottom. You cannot derive an abstract class from a concrete class.

Sometimes it's useful to write an abstract class that has few or no data members or code, consisting primarily of pure virtual functions. Most of the data and the code for the functions is defined when a new class is derived from such a base class. This is desirable when the base class's interface embodies a set of properties or operations that you'd like many other classes to have, but the implementations of those properties or operations differ for each class.

For example, consider a `SortedList` class that can store objects of any class.

```
class SortedList
{
public:
    SortedList();
    void addItem( const SortableObject &newItem );
    // ...
private:
    // ...
};
```

A `SortedList` object stores pointers to objects of class `SortableObject`. This is an abstract class that has pure virtual functions named `isEqual` and `isLessThan`:

```
class SortableObject
{
public:
    virtual int isEqual( const SortableObject &other ) const = 0;
    virtual int isLessThan( const SortableObject &other ) const = 0;
};
```

If you want to store names in a SortedList, you can derive a class called SortableName from SortableObject. You can then implement isEqual and isLessThan to perform string comparisons. For example:

```
class SortableName : public SortableObject
{
public:
    int isEqual( const SortableObject &other ) const;
    int isLessThan( const SortableObject &other ) const;
private:
    char name[30];
};

int SortableName::isEqual( const SortableObject &other ) const
{
    return (strncmp( name, other.name, 30 ) == 0);
};

// Similar implementation for isLessThan
```

Similarly, if you want to store ZIP Codes, you can derive a class SortableZIP from SortableObject and implement the member functions to compare numbers. SortableObject thus provides a template for you to use when writing your own classes. By itself, SortableObject isn't a useful class, because it contains no code or data. You supply those when you derive a class from it.

Destructors in Base and Derived Classes

If destructors are defined for a base and a derived class, they are executed in the reverse order that the constructors are executed. When an object of a derived class goes out of scope, the destructor for the derived class is called and then the destructor for the base class is called.

When destroying dynamically created objects with the **delete** operator, a problem can arise. If **delete** is applied to a base class pointer, the compiler calls the base class destructor, even if the pointer points to an instance of a derived class.

The solution is to declare the base class's destructor as **virtual**. This causes the destructors of all derived classes to be virtual, even though they don't share the same name as the base class's destructor. Then if **delete** is applied to a base class pointer, the appropriate destructor is called, no matter what type of object the pointer is pointing to.

Notice that Employee has a virtual destructor, even though the destructor does nothing. Whenever you write a class that has virtual functions, you always should

give it a virtual destructor, even if the class doesn't need one. The reason is that a derived class might require a destructor. For example, suppose you derive a class from `Employee` called `Consultant`, and that derived class defines a destructor. By defining a virtual destructor in the base class, you ensures that the derived class's destructor is called when needed.

Note that while destructor functions can be virtual, constructor functions cannot.

Protected Members

Besides the **public** and **private** keywords, C++ provides a third keyword controlling the visibility of a class's members: the **protected** keyword. Protected members are just like private members except that they are accessible to the member functions of derived classes.

As noted earlier, derived classes have no special privileges when it comes to accessing a class's private members. If you want to permit access by only the derived classes, and not by anyone else, you can declare some of your data members as **protected**. For example:

```
class Base
{
public:
protected:
    int secret;
private:
    int topSecret;
};

class Derived : public Base
{
public:
    void func();
};

void Derived::func()
{
    secret = 1;         // Can access protected member
    topSecret = 1;      // Error: Can't access private member
}
```

```
void main()
{
    Base aBase;
    Derived aDerived;

    aBase.secret = 1;      // Error: Can't access protected member
    aBase.topSecret = 1;   // Error: Can't access private member
    aDerived.secret = 1;   // Error: Can't access protected member
                           //        in derived class either
}
```

In this example, the private member `topSecret` is inaccessible to the derived class's member functions, but the protected member `secret` is accessible. However, the protected member is never accessible to outside functions.

Protected members in the base class are protected members in the derived class too. Functions using the derived class cannot access its protected members.

You should use the **protected** keyword with care. If the protected portion of a base class is rewritten, all the derived classes that used those protected members must be rewritten as well.

Public and Private Base Classes

The derived classes in this chapter all specify the keyword **public** in front of the base class's name. This specifies public derivation, which means that the public members of the base class are public members of the derived class, and protected members of the base class are protected members of the derived class.

You can also specify the keyword **private** in front of the base class's name. This specifies private derivation, which means that both the public and protected members of the base class are private members of the derived class. Those members are accessible to the derived class's member functions, but not to anyone using the derived class.

Private derivation is rarely used. Someone using the class cannot implicitly convert a pointer to a derived class object into a pointer to a base class object, or use polymorphism. (However, within the member functions of the derived class, you can perform such conversions and use polymorphism). Private derivation is very similar to defining a member object of another class; it's a method of using another class, but it's not appropriate for indicating that one class is a subtype of another.

Multiple Inheritance

The previous examples in this chapter demonstrate "single inheritance," where a class is derived from a single base class. C++ also supports "multiple inheritance," where a class can be derived from more than one base class.

For example, suppose you wanted to declare a class `SalesManager` to describe employees who have characteristics of both the `SalesPerson` and `Manager` classes:

```
class SalesManager : public SalesPerson, public Manager
{
// ...
};
```

The `SalesManager` class inherits all the data members and member functions of `SalesPerson` and `Manager`.

You cannot specify a class as a direct base class more than once (for example, you cannot enter `Manager` twice in the list of base classes). However, a class can be an indirect base class more than once. For example, `SalesManager` has `Employee` as an indirect base class twice: once through `SalesPerson` and once through `Manager`. As a result, each `SalesManager` object contains two copies of `Employee`'s data members.

If a class acts as an indirect base class more than once, it is more complicated to call member functions defined by that base class. For example:

```
SalesManager aSellerBoss;
char *str;

str = aSellerBoss.getName();     // Error: ambiguous
```

The problem is that the compiler can't tell which copy of `Employee` should be used; because each copy's data members might contain different values, the value returned by `getName` depends on which copy is used. You must specify which copy you want, using the scope resolution operator:

```
str = aSellerBoss.Manager::getName();
```

This statement uses the name stored in the `Manager`'s copy of `Employee`'s data members.

The same ambiguity problem can arise even if an indirect base class is not repeated. If a base class (either direct or indirect) defines a member with the

same name as a member defined in another base class, you must again use the scope resolution operator to specify whose member you want.

If a class acts as an indirect base more than once, there are also possible ambiguities when performing conversions between base and derived classes. For example, suppose you want to convert a pointer to a `SalesManager` into a pointer to an `Employee`:

```
Employee *empPtr;
SalesManager *salemgrPtr;

empPtr = salemgrPtr;   // Error: ambiguous
```

Once again, the compiler can't tell which copy of `Employee` it should use for `empPtr`. To disambiguate, you must use a cast:

```
empPtr = (Manager *)salemgrPtr;
```

This statement converts `salemgrPtr` into a pointer to a `Manager` and then converts that into a pointer to an `Employee`. As a result, `empPtr` points to `Manager`'s copy of `Employee`'s data members.

Because sales managers don't have two names and two social security numbers, the `SalesManager` class shouldn't contain two copies of `Employee`. To avoid this duplication, you can make `Employee` a "virtual base class."

To do this, the classes that specify `Employee` as a direct base class must use the **virtual** keyword:

```
class WageEmployee : public virtual Employee
{
    // ...
};

class Manager : public virtual Employee
{
    // ...
};
```

Note that only `WageEmployee` and `Manager` need to use the **virtual** keyword. `SalesPerson` and `SalesManager` do not, because `Employee` is an indirect base class for them.

By making `Employee` a virtual base class, each instance of `SalesManager` now has only one copy of `Employee`'s data members; there is no duplication. Any references to members defined by `Employee` are unambiguous, and so are conversions from a `SalesManager` pointer to a `Employee` pointer.

For a class like `SalesManager`, virtual base classes save space and allow a more accurate representation. However, virtual base classes impose a greater processing overhead. Consequently, you should use virtual base classes sparingly.

For information on design issues surrounding multiple inheritance, see Chapter 9, "Fundamentals of Object-Oriented Design." For more information on single inheritance, virtual functions, multiple inheritance, and virtual base classes, see the *C++ Language Reference*.

CHAPTER 8

Operator Overloading and Conversion Functions

Classes are useful for representing numeric data types that are not built into the language. This chapter covers two features of C++ that can make these classes behave more like the built-in types: operator overloading, which makes it possible for you to use operators with your classes, and conversion functions, which make it possible for you to convert between classes.

This chapter covers the following topics:

- Overloading operators as member functions
- Overloading operators as friend functions
- Constructors that perform conversions
- Conversion operators

Operator Overloading

Chapter 5, "Classes and Dynamic Memory Allocation," described how you can redefine the meaning of the assignment operator (=) when it is used to assign objects of a class you write. That was an example of operator overloading, and the assignment operator is the operator most commonly overloaded when writing classes.

You can overload other operators to make your code more readable. For example, suppose you needed a function that compares `Date` objects, to see if one comes before another. You can write a function called `lessThan` and use it as follows:

```
if( lessThan( myDate, yourDate ) )
    // ...
```

As an alternative, you can overload the less-than operator (<) to compare two `Date` objects. This would allow you to write an expression like the following:

```
if( myDate < yourDate )
    // ...
```

This format is more intuitive and convenient to use than the previous one.

You have already seen overloaded operators in many examples in the previous chapters. All of the example programs printed their output with the << operator, which is overloaded in the **iostream** class library.

Operator overloading is most useful when writing classes that represent numeric types. For example, scientific programs often use complex numbers—that is, numbers with a real and an imaginary component. You could write a class `Complex` to represent these numbers. To perform tasks such as adding and multiplying complex numbers, you could write functions with names like `add` and `mult`, but this often results in lengthy statements that are hard to understand. For example:

```
a = mult( mult( add( b, c ), add( d, e ) ), f );
```

Typing an equation in this format is tedious and error-prone, and reading an unfamiliar equation in this format is even more difficult.

A better alternative is to overload the + and * operators to work on `Complex` objects. This results in statements like this:

```
a = (b + c) * (d + e) * f;
```

This format is easier for both the programmer writing the equation and the programmer who reads it later.

Rules of Operator Overloading

You can overload any of the following operators:

Table 8.1 Overloadable Operators

+	–	*	/	%	^	&	|
~	!	,	=	<	>	<=	>=
++	--	<<	>>	==	!=	&&	||
+=	-=	*=	/=	%=	^=	&=	|=
<<=	>>=	[ ]	( )	->	->*	**new**	**delete**

The last two operators, **new** and **delete**, are the free store operators, which were described in Chapter 5, "Classes and Dynamic Memory Allocation." The last operator before those (->*) is the pointer-to-member operator, which is described in Chapter 4 of the *C++ Language Reference*.

Certain operators can be used as either binary or unary operators. For example, the – operator means subtraction when used as a binary operator and negation when used as a unary operator. You can overload such operators to have different meanings in their different usages.

You can use overloaded operators in ways that, if they were not overloaded, would be meaningless. Consider the following expression:

```
cout << "Hello";
```

If the << operator were not overloaded, this expression would left-shift **cout** a number of bits equal to the value of the pointer to the string. The compiler would generate a run-time error when trying to execute this statement. But the statement is syntactically legal, so you can write an overloaded operator function that executes when this statement appears. (For information on overloading the << and >> operators to make your classes work with streams, see the *iostream Class Library Reference*.)

There are a number of restrictions on operator overloading:

- You cannot extend the language by inventing new operators. For example, you cannot create your own "exponentiation" operator using the characters **. Those characters do not form a legal operator in C or C++, and you cannot make them one. You must limit yourself to existing operators.
- You cannot change an operator's "arity," that is, the number of operands that it takes. For example, the logical-NOT operator (~) is a unary operator, meaning that it takes one operand. You cannot use it as a binary operator for built-in types, so you cannot overload it to act as a binary operator for your class. For example:

```
a = ~b;          // Legal
a = b ~ c;       // Error
```

- You cannot change an operator's precedence. For example, the multiplication operator has a higher precedence than the addition operator, so the multiplication is performed first when the following expression is evaluated:

```
a = b + c * d;      // Same as a = b + (c * d);
```

  You cannot overload the * and + operators in such a way that the addition is performed first. You must use parentheses to alter the order of evaluation. For example:

```
a = (b + c) * d;    // Parentheses control evaluation
```

  One consequence of this is that the operator you choose may not have the precedence appropriate for the meaning you give it. For example, the ^ operator might seem an appropriate choice to perform exponentiation, but its precedence is lower than that of addition, which could confuse people using it.
- You cannot change an operator's associativity. When an operand is between two operators that have the same precedence, it is grouped with one or the other depending on its associativity. For example, the addition and subtraction operators are both left-associative, so the following expressions is evaluated from left to right:

```
a = b + c - d;      // Same as a = (b + c) - d;
```

  You cannot overload the + and – operators in such a way that the subtraction is performed first. You must use parentheses to alter the order of evaluation. For example:

```
a = b + (c - d);    // Parentheses control evaluation
```

- You cannot change the way an operator works with built-in data types. For example, you cannot change the meaning of the + operator for integers.
- You cannot overload the following operators:

Operator	Definition
.	Class member operator
.*	Pointer-to-member operator
::	Scope resolution operator
?:	Conditional expression operator

C++ lets you overload any of the other operators. However, just because you can overload an operator doesn't necessarily mean it's a good idea.

When Not to Overload Operators

You should overload operators only when the meaning of the operator is clear and unambiguous. The arithmetic operators, for example + and *, are meaningful when applied to numeric classes, such as complex numbers, but not to everything. For example, consider overloading the + operator for `Date` objects:

```
laterDate = myDate + yourDate;   // Meaning?
```

It's anyone's guess what this statement means.

Many programmers overload the + operator for a `String` class to perform concatenation of two `String` objects. However, overloading relational operators for a `String` class might not be appropriate:

```
String myString( "John Smith" ),
       yourString( "JOHN SMITH" );

if( myString == yourString )    // True or false?
    // ...
```

A person reading this statement cannot tell whether the comparison being performed is case sensitive or not. You could define a separate function to control case sensitivity, but the combination might not be as clear as using named member functions.

Sometimes, an overloaded operator clearly suggests a particular meaning to one programmer but suggests a different meaning to another programmer. For example, consider having a class `Set`, where each object represents a collection of objects. It would be useful to be able to calculate the "union" of two sets, that

is, the set that contains the contents of both without duplications. Someone might pick the **&&** operator for this purpose. For example:

```
ourSet = mySet && yourSet;    // Clearly union
```

The programmer who wrote this statement might think it clearly indicates that `ourSet` contains everything in `mySet` combined with everything in `yourSet`. But another programmer might use the operator in the following way:

```
// Intersection or union?
targetSet = wealthySet && unmarriedSet;
```

Does `targetSet` contain everyone who is *both* wealthy and unmarried? Or does it contains everyone who is *either* wealthy or unmarried?

Too many overloaded operators, or even a few badly chosen operators, can make your programs exceedingly difficult to read. Don't use overloaded operators simply to make it easier for you to type in your programs. Other programmers may have to maintain your programs later on, and it's much more important that they be able to understand your code. Accordingly, choose your overloaded operators with great care, and use them sparingly.

Because numeric classes are usually the best candidates for operator overloading, let's consider how to overload arithmetic operators for such a class.

Overloading Operators for a Numeric Class

As an example of a numeric class, consider a class called `Fraction`, which stores a number as the ratio of two long integers. This is useful because many numbers cannot be expressed exactly in floating-point notation. For example, the quantity 1/3 is represented as 0.33333. The expression 1/3 + 1/3 + 1/3 should add up to 1, but represented in floating-point notation it adds up to 0.99999. Over the course of a lengthy calculation, this type of error is cumulative and can become quite significant. A `Fraction` class removes this type of error.

To add two `Fraction` objects, you can overload the + operator as follows:

```
// Overloading the + operator
#include <stdlib.h>
#include <math.h>
#include <iostream.h>
```

```
class Fraction
{
public:
    Fraction();
    Fraction( long num, long den );
    void display() const;
    Fraction operator+( const Fraction &second ) const;
private:
    static long gcf( long first, long second );
    long numerator,
         denominator;
};

// ----------- Default constructor
Fraction::Fraction()
{
    numerator = 0;
    denominator = 1;
}

// ----------- Constructor
Fraction::Fraction( long num, long den )
{
    int factor;

    if( den == 0 )
        den = 1;
    numerator = num;
    denominator = den;
    if( den < 0 )
    {
        numerator = -numerator;
        denominator = -denominator;
    }
    factor = gcf( num, den );
    if( factor > 1 )
    {
        numerator /= factor;
        denominator /= factor;
    }
}

// ----------- Function to print a Fraction
void Fraction::display() const
{
    cout << numerator << '/' << denominator;
}
```

```
// ---------- Overloaded + operator
Fraction Fraction::operator+( const Fraction &second ) const
{
    long factor,
         mult1,
         mult2;

    factor = gcf( denominator, second.denominator );
    mult1 = denominator / factor;
    mult2 = second.denominator / factor;

    return Fraction( numerator * mult2 + second.numerator * mult1,
                     denominator * mult2 );
}

// ---------- Greatest common factor
// computed using iterative version of Euclid's algorithm
long Fraction::gcf( long first, long second )
{
    int temp;

    first = labs( first );
    second = labs( second );

    while( second > 0 )
    {
        temp = first % second;
        first = second;
        second = temp;
    }

    return first;
}
```

A `Fraction` object is declared with two integers, the numerator and the denominator. The constructor checks that the denominator is nonzero and nonnegative and simplifies the fraction if possible. The class defines a private static function named `gcf` to calculate the greatest common factor of two numbers.

The following program uses the `Fraction` class and demonstrates the use of the overloaded + operator.

```
void main()
{
    Fraction a,
             b( 23, 11 ),
             c( 2, 3 );

    a = b + c;

    a.display();
    cout << '\n';
}
```

The expression `b + c` is interpreted as `b.operator+( c )`. The **operator+** function is called for the `b` object, using `c` as a parameter.

An overloaded operator doesn't have to have objects for both operands. You can add a `Fraction` and an integer as well by writing another function:

```
Fraction Fraction::operator+( long second ) const
{
    return Fraction( numerator + second * denominator,
                     denominator );
}
```

This permits statements like the following:

```
void main()
{
    Fraction a,
             b( 2, 3 );

    a = b + 1234;
}
```

However, you cannot write a statement like this:

```
a = 1234 + b;  // Error
```

Because **operator+** is defined as a member function, the previous statement is interpreted as follows:

```
a = (1234).operator+( b );  // Error
```

This statement is clearly an error, because an integer doesn't have a member function that can be invoked to perform the addition.

To allow expressions where a variable of a built-in type is the first operand, you must use friend functions (described in Chapter 6, "More Features of Classes").

Defining Operators as Friend Functions

To overload an operator using a nonmember function, you define a function named **operator+** that takes two arguments, as follows:

```
class Fraction
{
public:
    Fraction( long num, long den );
    Fraction operator+( const Fraction &second ) const;
    Fraction operator+( long second ) const;
    friend Fraction operator+( long first,
                               const Fraction &second );
// ...
};

// ...

Fraction operator+( long first, const Fraction &second )
{
    return Fraction( second.numerator + first * second.denominator,
                     second.denominator );
}
```

With a function like this, an expression like this

```
a = 1234 + b;     // Friend function called
```

is interpreted as follows:

```
a = operator+( 1234, b );     // Friend function called
```

Notice that the friend function requires two parameters while the member function requires only one. The + operator requires two operands. When **operator+** is defined as a member function, the first operand is the object for which it is called, and the second the parameter. In contrast, when **operator+** is defined as a friend function, both operands are parameters to the function. You cannot define a friend and a member function that define the same operator.

You can also use either a member function or a friend function to implement a unary operator. For example, suppose you want to implement the negation (–) operator. You could do it as a member function that takes no parameters:

```
inline Fraction Fraction::operator-() const
{
    return Fraction( -numerator, denominator );
}
```

You could also implement it as a friend function that takes one parameter:

```
inline Fraction operator-( Fraction &one )
{
    return Fraction( -one.numerator, one.denominator );
}
```

When you overload an operator using a friend function, you must make at least one of the function's parameters an object. That is, you cannot write a binary **operator+** function that takes two integers as parameters. This prevents you from redefining the meaning of operators for built-in types.

Notice that you have to define three separate functions to handle the addition of `Fraction` objects and long integers. If you overload other arithmetic operators, such as * or /, you must also provide three functions for each operator. A technique for avoiding multiple versions of each operator is described in the section "Class Conversions" on page 157.

Tips for Overloading Arithmetic Operators

Overloading the + operator does not mean that the += operator is overloaded. You must overload that operator separately. If you do, make sure that the normal identity relationships are maintained, that is, `a += b` has the same effect as `a = a + b`.

If you're overloading operators for a class whose objects are relatively large, you should pass parameters as references rather than by value. Also, be sure to pass references to constants, which allows constant objects to be operands.

The return type of an overloaded operator depends on the specific operator. Overloaded + or * operators for the `Fraction` class must return `Fraction` objects. Operators such as += and *=, on the other hand, can return references to `Fraction` objects for efficiency. This is because + and * create temporary objects containing new values, and they cannot return references to objects created within the function. In contrast, += and *= modify an existing object, ***this**, so they can safely return references to that object. (Recall that the overloaded = operator, described in Chapter 5, "Classes and Dynamic Memory Allocation," also returns a reference to an object.)

Overloading Operators for an Array Class

The array mechanism that is built into C is very primitive; it is essentially an alternate syntax for using pointers. An array doesn't store its size, and there is no way to keep someone from accidentally indexing past the end of the array. In C++, you can define a much safer and more powerful array type using a class. To make such a class look more like an array, you can overload the subscript operator ([ ]).

The following example defines the `IntArray` class, where each object contains an array of integers. This class overloads the [ ] operator to perform range checking.

```
// Overloaded [] operator
#include <iostream.h>
#include <string.h>

class IntArray
{
public:
     IntArray( int len );
     int getLength() const;
     int &operator[]( int index );
     ~IntArray();
private:
     int length;
     int *aray;
};

// ------------ Constructor
IntArray::IntArray( int len )
{
    if( len > 0 )
    {
        length = len;
        aray = new int[len];
        // initialize contents of array to zero
        memset( aray, 0, sizeof( int ) * len );
    }
    else
    {
        length = 0;
        aray = 0;
    }
}
```

```
// ------------ Function to return length
inline int IntArray::getLength() const
{
    return length;
}

// ------------ Overloaded subscript operator
// Returns a reference
int &IntArray::operator[]( int index )
{
    static int dummy = 0;

    if( (index >= 0) &&
        (index < length) )
        return aray[index];
    else
    {
        cout << "Error: index out of range.\n";
        return dummy;
    }
}

// ------------ Destructor
IntArray::~IntArray()
{
    delete aray;
}

void main()
{
    IntArray numbers( 10 );
    int i;

    for( i = 0; i < 10; i++ )
        numbers[i] = i;          // Use numbers[i] as lvalue

    for( i = 0; i < 10; i++ )
        cout << numbers[i] << '\n';
}
```

This program first declares an `IntArray` object that can hold 10 integers. Then it assigns a value to each element in the array. Notice that the array expression appears on the left side of the assignment. This is legal because the **operator[ ]** function returns a reference to an integer. Because the expression `numbers[i]` acts as an alias for an element in the private array, it can be the recipient of an

assignment statement. In this situation, returning a reference is not simply more efficient—it is necessary.

The **operator[ ]** function checks whether the specified index value is within range or not. If it is, the function returns a reference to the corresponding element in the private array. If it isn't, the function prints out an error message and returns a reference to a static integer. This prevents out-of-range array references from overwriting other regions of memory, though it will probably cause unexpected program behavior. As an alternative, you could have the **operator[ ]** function exit the program when it receives an out-of-range index value.

As it is currently implemented, the index values for an `IntArray` object of size *n* range from 0 to *n*–1, but that is not a requirement. You can use any value you want for the bounds of the array or even have the bounds specified when an object is declared.

The `IntArray` class has a number of advantages over conventional arrays in C. The size of an `IntArray` doesn't have to be a constant; you can determine the size at run time without having to use the **new** and **delete** operators. An `IntArray` object also stores its size, so you can pass one to a function without having to pass the size separately. One possible enhancement is to make `IntArrays` resizable, so that you could expand one if it became full.

You can also overload **operator[ ]** for classes that aren't implemented as arrays. For example, you could write a linked list class and allow users to use array notation to access the nodes in the list. You can even use values other than integers for indexing. For example, consider the following prototype:

```
int &operator[]( const char *key );
```

This **operator[ ]** function takes a string as an index. This permits expressions like the following:

```
salary["John Smith"] = 25000;
```

You could use the string as a key for searching through a data structure, which could be an array or a linked list or something else. Because it would be difficult to iterate through such a class using a **for** loop, this class would probably benefit from having an iterator implemented with a friend class, as mentioned in Chapter 6, "More Features of Classes."

The **operator[ ]** function takes only one parameter. You cannot give it multiple parameters to simulate a multidimensional array. For example:

```
int &SquareArray::operator[]( int row, int col );  // Error
```

You can, however, overload the ( ) operator, which can take an arbitrary number of parameters. For example:

```
int &SquareArray::operator()( int row, int col );
```

This allows statements like the following:

```
SquareArray myArray;

myArray( 3, 4 ) = 1;
```

Note that this is not standard array notation in C, so it may be confusing for other programmers reading your code.

The **operator[ ]** function cannot be defined as a friend function. It must be a non-static member function.

Class Conversions

Both C and C++ have a set of rules for converting one type to another. These rules are used in the following situations:

- When assigning a value. For example, if you assign an integer to an variable of type **long**, the compiler converts the integer to a long.
- When performing an arithmetic operation. For example, if you add an integer and a floating-point value, the compiler converts the integer to a float before it performs the addition.
- When passing an argument to a function—for example, if you pass an integer to a function that expects a long.
- When returning a value from a function—for example, if you return a float from a function that has **double** as its return type.

In all of these situations, the compiler performs the conversion implicitly. You can make the conversion explicit by using a cast expression.

When you define a class in C++, you can specify the conversions that the compiler can apply when you use instances of that class. You can define conversions between classes, or between a class and a built-in type.

Conversion by Constructor

Chapter 4, "Introduction to Classes," described constructors, the functions that create objects. A constructor that takes only one parameter is considered a conversion function; it specifies a conversion from the type of the parameter to the type of the class. For example, suppose you specify a default value for the `denominator` parameter of the `Fraction` constructor, as follows:

```
class Fraction
{
public:
    Fraction( long num, long den = 1 );
    // ...
};
```

This constructor not only lets you initialize a `Fraction` object using only one number, it also lets you assign integers to a `Fraction` object directly. For example:

```
Fraction a( 2 );   // Equivalent to Fraction a( 2 , 1 );

a = 7;          // Equivalent to a = Fraction( 7 );
                //               a = Fraction( 7, 1 );
```

In the above statement, the compiler uses the single-argument constructor to implicitly convert an integer into a temporary `Fraction` object and then uses the object for the assignment. Similarly, if you pass an integer to a function that expects a `Fraction` object, the integer is converted before the function is called.

To be more precise, when you pass the `Fraction` constructor an integer, the compiler actually performs a standard conversion and a user-defined conversion at once. For example:

```
a = 7;   // int -> long -> Fraction
```

The compiler first performs a standard conversion to make the integer into a long integer. Then it converts the long integer into a `Fraction` and performs the assignment.

One result of defining an implicit conversion is that one operator function can handle several different types of expression. Suppose you define just one **operator+** function for the `Fraction` class:

```
class Fraction
{
public:
    Fraction( long num, long den = 1 );
    friend Fraction operator+( const Fraction &first,
                               const Fraction &second );
    // ...
};
```

The combination of that constructor and that operator function accepts all of these expressions:

```
Fraction a,
         b( 2, 3 ),
         c( 4, 5 );

a = b + c;           // Okay as is
a = b + 1234;        // a = b + Fraction( 1234 );
a = 1234 + b;        // a = Fraction( 1234 ) + b;
a = 1234 + 5678;     // a = Fraction( 6912 );
```

When the compiler evaluates each expression, it looks for an **operator+** function that fits. If it can't find one, it tries to convert one or more of the operands so that they match the **operator+** function that does exist. As a result, the compiler converts the integers into `Fraction` objects and performs the addition on them.

Notice that in the last assignment statement the compiler does not convert the integers into `Fraction` objects. It is able to add the integers directly. The compiler then converts the resulting sum into a temporary `Fraction` object to perform the assignment.

A single-argument constructor defines an implicit conversion that turns instances of other types into objects of your class, so that your class is the *target* of the conversion. You can also define an implicit conversion that turns objects of your class into instances of other types, making your class the *source* of the conversion. To do this, you define a "conversion operator."

Conversion Operators

Suppose you want to be able to pass a `Fraction` object to a function that expects a **float**, that is, you want to convert `Fraction` objects into floating-point values. The following example defines a conversion operator to do just that:

```
// Conversion member function
#include <iostream.h>

class Fraction
{
public:
    Fraction( long num, long den = 1 );
    operator float() const;
    // ...
};

Fraction::operator float() const
{
    return (float)numerator / (float)denominator;
}
```

The function `operator float` converts a `Fraction` object to a floating-point value. Notice that the operator function has no return type and takes no parameters. A conversion operator must be a nonstatic member function; you cannot define it as a friend function.

You can call the conversion operator using one of several syntax variations:

```
Fraction a;
float f;

f = a.operator float();  // Convert using explicit call
f = float( a );          // Convert using constructor syntax
f = (float)a;            // Convert using cast syntax
f = a;                   // Convert implicitly
```

The compiler can perform a standard conversion and a user-defined conversion at once. For example:

```
Fraction a( 123, 12 );
int i;

i = a;  // Fraction -> float -> integer
```

The compiler first converts the `Fraction` object into a floating-point number. Then it performs a standard conversion, making the floating-point number into an integer, and performs the assignment.

A conversion operator doesn't have to convert from a class to a built-in type. You can also use a conversion operator that converts one class into another. For example, suppose you had defined the numeric class `FixedPoint` to store fixed-point numbers. You could define a conversion operator as follows:

```
class Fraction
{
public:
    operator FixedPoint() const;
};
```

This operator would permit implicit conversions of a `Fraction` object into a `FixedPoint` object.

Conversion operators are useful for defining an implicit conversion from your class to a class whose source code you don't have access to. For example, if you want a conversion from your class to a class that resides within a library, you cannot define a single-argument constructor for that class. Instead, you must use a conversion operator.

Ambiguities with Conversions and Operators

The inclusion of the `operator float` conversion operator creates problems for the `Fraction` class. Consider the following statement:

```
a = b + 1234;   // Error: ambiguous
                //     a = (float)b + 1234;
                //     a = b + Fraction( 1234 );
```

The compiler could either convert `b` to a floating-point number and then add that together with the integer, or it could convert 1234 to a `Fraction` and then add the two `Fraction` objects. That is, the compiler could add the two values as built-in types, or it could add them as objects. The compiler has no basis for choosing one conversion over the other, so it generates an error.

There are a few ways to resolve this ambiguity. One is to use an ordinary member function to perform addition, instead of overloading the + operator. For example:

```
class Fraction
{
    friend Fraction add( const Fraction &first,
                         const Fraction &second );
};
```

Because this function does not look like the + operator, there is no confusion between adding two values as `Fraction` objects or adding them as built-in types.

Another solution is to remove an implicit conversion. You could remove the implicit conversion from an integer to a `Fraction` by getting rid of the single-argument constructor. This requires you to rewrite the previous statement as:

```
a = b + Fraction( 1234, 1 );
```

If you wanted to add the values as built-in types, you'd write the following:

```
a = Fraction( b + 1234, 1 );
```

Or you could remove the implicit conversion from a `Fraction` to a floating-point number, by changing the conversion operator into an ordinary member function. For example:

```
class Fraction
{
public:
    float cvtToFloat() const;
    // ...
};
```

This leaves only one interpretation for the following statement:

```
a = b + 1234;    // a = b + Fraction( 1234 );
```

If you wanted to add the two values as built-in types, you'd write the following:

```
a = b.cvtToFloat() + 1234;
```

If you wish to retain the convenience of both of these implicit conversions, as well as use operator overloading, you must explicitly define all three versions of the **operator+** function:

```
class Fraction
{
    friend Fraction operator+( const Fraction &first,
                               const Fraction &second );
    friend Fraction operator+( long first,
                               const Fraction &second );
    friend Fraction operator+( const Fraction &first,
                               long second );
    // ...
};
```

If all three functions are defined, the compiler doesn't have to perform any conversions to resolve expressions that mix `Fraction` objects and integers. The compiler simply calls the function that matches each possibility. This solution requires more work when writing the class, but it makes the class more usable.

As this example illustrates, you must use care if you define both overloaded operators and implicit conversions.

Ambiguities Between Conversions

An ambiguity can arise when two classes define the same conversion. For example:

```
class FixedPoint;

class Fraction
{
public:
    Fraction( FixedPoint value );   // FixedPoint -> Fraction
};

class FixedPoint
{
public:
    operator Fraction();            // FixedPoint -> Fraction
};
void main()
{
    Fraction a;
    FixedPoint b;

    a = b;          // Error; ambiguous
                    //     a = Fraction( b );
                    //     a = b.operator Fraction();
}
```

The compiler cannot choose between the constructor and the conversion operator. You can explicitly specify the conversion operator, but not the constructor:

```
a = b.operator Fraction();          // Call conversion operator
a = Fraction( b );                  // Error: still ambiguous
a = (Fraction)b;                    // Error: still ambiguous
```

This type of ambiguity is easy to avoid, because it occurs only when the classes know of each other, which means that they were written by the same programmer(s). If you simply remove one of the conversion functions, the problem does not arise.

Ambiguities can also arise when multiple classes define similar implicit conversions. For example, suppose you have the `Fraction` class and some associated functions that use `Fraction` objects, as follows:

```
class Fraction
{
public:
    Fraction( float value );        // float -> Fraction
};

void calc( Fraction parm );
```

You might also have a `FixedPoint` class that includes a similar set of associated functions:

```
class FixedPoint
{
public:
    FixedPoint( float value );      // float -> FixedPoint
};

void calc( FixedPoint parm );
```

Now consider what happens if you try to use both the `Fraction` and the `FixedPoint` classes in the same program:

```
void main()
{
    calc( 12.34 );  // Error: ambiguous
                    //     calc( Fraction( 12.34 ) );
                    //     calc( FixedPoint( 12.34 ) );
}
```

The compiler cannot choose which conversion to apply when calling the `calc` function. This type of ambiguity is difficult to avoid, because it can occur even if `Fraction` and `FixedPoint` are written by different programmers. Neither programmer would have noticed the problem because it doesn't appear when either class is used by itself; the ambiguity arises only when they are used together. This problem can be solved if the user of the classes explicitly specifies a conversion by using the constructor for the class desired.

It is difficult to anticipate all possible ambiguities that may involve your class. When you write a class, you might define only a small number of conversions. However, when other programmers write their classes, they can also define conversions involving your class. They can define constructors that convert an object of your class into an object of one of theirs, or they can define conversion operators that turn an object of one of their classes into an object of one of your classes.

To reduce the likelihood of ambiguities, you should define implicit conversions for your classes only when there is a clear need for them. You can always perform conversions explicitly by using constructors that require more than one argument, or by using ordinary member functions to perform conversions (for example, `cvtToOtherType`).

See the *C++ Language Reference* for a complete description of the rules governing conversions.

PART 3

Object-Oriented Design

CHAPTER 9

Fundamentals of Object-Oriented Design

The preceding chapters covered the basic syntax of the C++ language. This chapter and the next one discuss object-oriented programming, the style of programming that C++ is designed to support. This chapter describes the key principles of object-oriented programming and how to apply them. The next chapter describes an example program written in C++ using those principles.

The first section of this chapter discusses the major concepts of object-oriented programming. This chapter then outlines the process of designing an object-oriented program.

Features of Object-Oriented Programming

In the traditional, procedure-oriented view of programming, a program describes a series of steps to be performed—that is, an algorithm. In the object-oriented view of programming, a program describes a system of objects interacting. It's possible to use C++ as a strictly procedural language. An object-oriented approach, however, lets you take full advantage of C++'s features.

Object-oriented programming involves a few key concepts. The most basic of these is abstraction, which makes writing large programs simpler. Another is encapsulation, which makes it easier to change and maintain a program. Finally, there is the concept of class hierarchies, a powerful classification tool that can make a program easily extensible. While you can apply these concepts using any language, object-oriented languages have been specifically designed to support them explicitly.

Abstraction

"Abstraction" is the process of ignoring details in order to concentrate on essential characteristics. A programming language is traditionally considered "high-level" if it supports a high degree of abstraction. For example, consider two programs that perform the same task, one written in assembly language, one in C.

The assembly-language program contains a very detailed description of what the computer does to perform the task, but programmers usually aren't concerned with what happens at that level. The C program gives a much more abstract description of what the computer does, and that abstraction makes the program clearer and easier to understand.

While traditional languages support abstraction, object-oriented languages provide much more powerful abstraction mechanisms. To understand how, consider the different types of abstraction.

Procedural Abstraction

The most common form of abstraction is "procedural abstraction," which lets you ignore details about processes.

There are many levels of procedural abstraction. For example, it's possible to describe what a program does in even greater detail than assembly language does, by listing the individual steps that the CPU performs when executing each assembly language instruction. On the other hand, a program written in the macro language of an application program can describe a given task on a much higher level than C does.

When you write a program in a given language, you aren't restricted to using the level of abstraction that the language itself provides. Most languages allow you to write programs at a higher level of procedural abstraction, by supporting user-defined functions (also known as procedures or subroutines). By writing your own functions, you define new terms to express what your program does.

As a simple example of procedural abstraction, consider a program that frequently has to check whether two strings are the same, ignoring case:

```
while (*s != '\0')
{
    if ( (*s == *t) ||
         ((*s >= 'A') && (*s <= 'Z') && ((*s + 32) == *t)) ||
         ((*t >= 'A') && (*t <= 'Z') && ((*t + 32) == *s))   )
    {
         s++; t++;
    }
    else break;
}
if ( *s == '\0' )
    printf("equal \n");
else
    printf("not equal \n");
```

By writing a program this way, you are constantly reminded of the comparisons that the program performs to check whether two strings are equal. An alternate way to write this program is to place the string comparison in a function:

```
if ( !_stricmp( s, t ) )
    printf("equal \n");
else
    printf("not equal \n");
```

The use of **_stricmp** does more than save you a lot of typing. It also makes the program easier to understand, because it hides details that can distract you. The precise steps performed by the function aren't important. What's important is that a case-insensitive string comparison is being performed.

Functions make large programs easier to design by letting you think in terms of logical operations, rather than in specific statements of the programming language.

Data Abstraction

Another type of abstraction is "data abstraction," which lets you ignore details of how a data type is represented.

For example, all computer data can be viewed as hexadecimal or binary numbers. However, because most programmers prefer to think in terms of decimal numbers, most languages support integer and floating-point data types. You can simply type "3.1416" rather than some hexadecimal bytes. Similarly, Basic provides a string data type, which lets you perform operations on strings intuitively, ignoring the details of how they're represented. On the other hand, C does not support the abstraction of strings, because the language requires you to manipulate strings as series of characters occupying consecutive memory locations.

Data abstraction always involves some degree of procedural abstraction as well. When you perform operations on variables of a given data type, you don't know the format of the data, so you can ignore the details of how operations are performed on those data types. How floating-point arithmetic is performed in binary is, thankfully, something C programmers don't have to worry about.

Compared to their capacity for procedural abstraction, most languages have very limited support for creating new levels of data abstraction. C supports user-defined data types through structures and **typedef**s. Most programmers use structures as no more than aggregates of variables. For example:

```
struct PersonInfo
{
    char name[30];
    long phone;
    char address1[30];
    char address2[30];
};
```

Such a user-defined type is convenient because it lets you manipulate several pieces of information as a unit instead of individually. However, this type doesn't provide any conceptual advantage. There's no point in thinking about the structure without thinking about the three pieces of information it contains.

A better example of data abstraction is the **FILE** type defined in STDIO.H:

```
typedef struct _iobuf
{
   char __far *_ptr;
   int _cnt;
   char __far *_base;
   char _flag;
   char _file;
} FILE;
```

A **FILE** structure is conceptually much more than the fields contained within it. You can think about **FILE**s without knowing how they're represented. You simply use a **FILE** pointer with various library functions, and let them handle the details.

Notice that it's possible to declare a structure without declaring the functions needed to use the structure. The C language lets you view data abstraction and procedural abstraction as two distinct techniques, when in fact they're integrally linked.

Classes

This is where object-oriented programming comes in. Object-oriented languages combine procedural and data abstraction, in the form of classes. When you define a class, you describe everything about a high-level entity at once. When using an object of that class, you can ignore the built-in types contained in the class and the procedures used to manipulate them.

Consider a simple class: polygonal shapes. You might think of a polygon as a series of points, which can be stored as a series of paired numbers. However, a polygon is conceptually much more than the sum of its vertices. A polygon has a perimeter, an area, and a characteristic shape. You might want to move one, rotate it, or reflect it. Given two polygons, you might want to find their intersection or their union or see if their shapes are identical. All of these properties and operations are perfectly meaningful without reference to any low-level entities that might make up a polygon. You can think about polygons without thinking about the numbers that might be stored in a polygon object, and without thinking about the algorithms for manipulating them.

With support for combined data abstraction and procedural abstraction, object-oriented languages make it easy for you to create an additional layer of separation between your program and the computer. The high-level entities you define have the same advantage that floating-point numbers and **printf** statements have when compared to bytes and **MOV** instructions: They make it easier to write long and complex applications.

Classes can also represent entities that you usually wouldn't consider data types. For example, a class can represent a binary tree. Each object is not simply a node in a tree, the way a C structure is; each object is a tree in itself. It's just as easy to create multiple binary trees as it is to create one. More importantly, you can ignore all the nonessential details of a binary tree. What features of a binary tree are you really interested in? The ability to quickly search for an item, to add or delete items, and to enumerate all the items in sorted order. It really doesn't matter what data structure you use, as long as you can perform the same set of operations on it. It might be a tree implemented with nodes and pointers, or a tree implemented with an array, or some data structure you've never heard of.

Such a class shouldn't be called `BinaryTree`, because that name implies a particular implementation. Based on the operations that can be performed on it, the class should be called `SortedList` or something similar.

By designing your program around abstract entities that have their own set of operations, rather than using data structures made of built-in types, you make your program more independent from implementation details. This leads to another advantage of object-oriented programming: encapsulation.

Encapsulation

"Encapsulation," which was mentioned in Chapter 4, "Introduction to Classes," is the process of hiding the internal workings of a class to support or enforce abstraction. This requires drawing a sharp distinction between a class's "interface," which has public visibility, and its "implementation," which has private visibility. A class's interface describes what a class can do, while its implementation describes how it does it. This distinction supports abstraction by exposing only the relevant properties of a class; a user views an object in terms of the operations it can perform, not in terms of its data structure.

Sometimes encapsulation is defined as the act of combining functions and data, but this is slightly misleading. You can join functions and data together in a class and make all the members public, but that is not an example of encapsulation. A truly encapsulated class "surrounds" or hides its data with its functions, so that you can access the data only by calling the functions. This is illustrated in Figure 9.1.

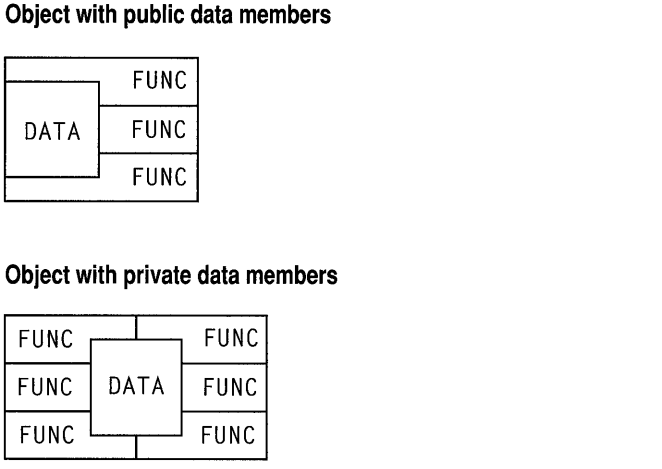


Figure 9.1 Hiding Data with Functions

Encapsulation is not unique to object-oriented programming. The principle of "data hiding" in traditional structured programming is the same idea applied to modules rather than classes. It is common practice to divide a large program into modules, each of which has a clearly defined interface of functions that the other modules can use. The aim of data hiding is to make each module as independent of one another as possible. Ideally, a module has no knowledge of the data structures used by other modules, and it refers to those modules only through their interfaces. The use of global variables or data structures is kept to a minimum to limit the opportunity for modules to affect one another.

For example, suppose a program needs to maintain a table of information. All the functions acting on the table could be defined in one module, the file TABLE.C, and their prototypes could be declared in a file called TABLE.H:

```
/* TABLE.H */
#include "record.h"   /* Get definition of RECORD data type */

void add_item( RECORD *new_item );
RECORD *search_item( char *key );
.
.
.
```

If any function in the program needs to use the table, it calls one of the functions defined in TABLE.H. The TABLE.C module might implement the table as an array, but the other modules don't know about it. If that array is declared **static**, it is actually inaccessible outside of TABLE.C. Only the interface is visible then, while the implementation is completely hidden.

Data hiding provides a number of benefits. One of them is abstraction, which was described previously; you can use a module without having to think about how it works. Another is "locality," which means that changes to one part of the program don't require changes to the other parts. A program with poor locality is very fragile; modifying one section causes other sections to break, because they all depend on one another. A program with good locality is stable and easier to maintain; the effects of a change are confined to a small portion of the program. If you change the array in TABLE.C to a linked list or some other data structure, you don't have to rewrite any module that uses the table.

Hiding data within a module has its limitations. In the example mentioned above, the TABLE module does not permit you to have more than one table of information in your program, nor does it let you declare a table that is local to a particular function. You can gain these capabilities by using structures and pointers. For example, you could use pointers as handles to tables and write functions that take a table pointer as a parameter:

```
/* TABLE.H */
#include "record.h"

/* define TABLE with a typedef */

TABLE *create_table();
void add_item( TABLE *handle, RECORD *new_item );
RECORD *search_item( TABLE *handle, char *key );
void *destroy_table( TABLE *handle );
```

This technique is considerably more powerful than that used in the previous example. It lets you use multiple tables at once and have separate tables for

different functions. However, the TABLE type provided by this module cannot be used as easily as built-in data types. For example, local tables are not automatically destroyed upon exit from a function. Like dynamically allocated variables, these tables require extra programming effort to be used properly.

Now consider the corresponding implementation in C++:

```
// TABLE.H
#include "record.h"

class Table
{
public:
   Table();
   void addItem( Record *newItem );
   Record *searchItem( char *key );
   ~Table();
private:
   //...
};
.
.
.

// PROG.CPP
#include "table.h"

void func()
{
   Table first, second;

    //...
}
```

This class has two advantages over the technique of using table handles in C. The first one, as mentioned earlier, is ease of use. You can declare instances of Table the same way you declare integers or floating-point numbers, and the same scoping rules apply to all of them.

Second, and more important, the class enforces encapsulation. In the technique using table pointers, it is only a matter of convention that programmers do not access what's behind the table handle. Many programmers may choose to circumvent the interface of functions and manipulate a table directly. If the implementation of a table changes, it's very time consuming to locate every place in the source code where the programmer's assumptions about the data structure are now invalid. Such errors might not be detected by the compiler and might remain undetected until run time, when (for example) a null pointer is dereferenced and the program fails. Even minor changes to the implementation can create such

problems. Sometimes the changes are intended to correct bugs but instead cause new ones because other functions depend on the specifics of an implementation.

In contrast, by declaring `Table` as a class, you can use the access rules of C++ to hide the implementation. You don't have to rely on the self-restraint of programmers who use your class. Any program that attempts to access the private data of a `Table` object won't compile. This makes it much more likely that locality will be maintained.

A common reason programmers break convention and access a data structure directly is that they can easily perform an operation that is cumbersome to do using only the functions in the interface. A well-designed class interface can minimize this problem if it reflects the important properties of the class. While no interface can make all possible operations convenient, it's usually best to forbid access to a class's internal data structure, even if it means an occasional piece of inefficient code. The minor loss in convenience is far outweighed by the increased maintainability of the program that encapsulation provides. By eliminating the need to modify most of the modules in a large program whenever a change is made, object-oriented languages can dramatically reduce the time and effort needed to develop new systems or update existing ones.

Even if the class interface changes in the future, it is still a good idea to use an encapsulated class rather than accessible data structures. In most cases, the changes to the interface can be formulated solely as additions to the existing interface, providing for upward compatibility. Any code that uses the old interface still works correctly. The code has to be recompiled, but that involves only computer time, not programmer time.

Note that, in C++, encapsulation does not provide a guarantee of safety. A programmer who is intent on using a class's private data can always use the **&** and * operators to gain access to them. Encapsulation simply protects against *casual* use of a class's internal representation.

Class Hierarchies

One feature of object-oriented programming that is not found at all in procedural programming is the ability to define a hierarchy of types. In C++, you can define one class as a subtype, or special category, of another class by deriving it from that class. You can also express similarities between classes, or define them as subcategories of a single broad category, by deriving them all from one base class. By contrast, the C language treats all types as completely independent of one another.

Identifying a common base class for several classes is a form of abstraction. A base class is a high-level way to view those classes. It specifies what the derived classes have in common, so you can concentrate on those shared traits and ignore their individual characteristics. This abstraction technique lets you view entities

in terms of a small number of categories instead of a large number. This technique is often used in everyday thinking; for example, it's easier to think "mammals" instead of "lions, tigers, bears..." and "bears" rather than "grizzly bears, black bears, polar bears...."

Whereas a base class is a *generalization* of a group of classes, a derived class is a *specialization* of another class. A derived class identifies a subtype of a previously recognized type and describes it in terms of its additional characteristics. For example, lions are mammals, but they also have certain traits not found in all mammals.

There are two practical benefits of defining a class hierarchy: The derived class can share the base class's code, or it can share the base class's interface. These two benefits are not mutually exclusive, though a hierarchy designed for code reuse often has different characteristics from one designed to give a common interface.

Inheriting Code

If you are writing a class and want to incorporate the functionality of an existing class, you can simply derive your class from the existing one. This is a situation in which inheritance allows code reuse. For example, the `SalesPerson` class in Chapter 7, "Inheritance and Polymorphism," incorporated the functionality of the `WageEmployee` class.

If you're implementing several classes at once that share features, a class hierarchy can reduce redundant coding. You can describe and implement those common features just once in a base class, rather than repeatedly in each derived class.

For example, consider a program for designing data entry forms, where users fill out fields on-screen. The program allows forms to contain fields that accept names, fields that accept dates, fields that accept monetary values, and so forth. Each field accepts only the appropriate type of data. You could make each type of field a separate class, with names such as `NameField`, `DateField`, and `MoneyField`, each with its own criteria for validating input. Note that all the fields share some functionality. Each field is accompanied by a description telling the user what's requested, and the procedure for defining and displaying that description is the same for all fields. As a result, all the classes have identical implementations for their `setPrompt`, `displayPrompt`, and other functions.

You can save yourself effort as well as reduce the size of the program by defining a base class called `Field` that implements those functions. The `NameField`, `DateField`, and `MoneyField` classes can be derived from `Field` and inherit those functions. Such a class hierarchy also reduces the effort required to fix bugs or add features, because the changes only have to be made in one place.

A class hierarchy designed for code sharing has most of its code in the base classes (near the top of the hierarchy). This way, the code can be reused by many classes. The derived classes represent specialized or extended versions of the base class.

Inheriting an Interface

Another inheritance strategy is for a derived class to inherit just the names of its base class's member functions, not the code; the derived class provides its own code for those functions. The derived class thus has the same interface as the base class but performs different things with the same functions.

This strategy lets different classes use the same interface, thus reinforcing the high-level similarity in their behavior. However, the main benefit of inheriting an interface is polymorphism, which was exhibited by the `Employee` class in Chapter 7, "Inheritance and Polymorphism." All the classes derived from `Employee` shared its interface, making it possible to manipulate them as generic `Employee` objects.

In the example of the data entry forms, `Field` has a member function called `getValue`, but the function doesn't do anything useful. `NameField` inherits that member function and provides it with code to validate input as a legal name. `DateField` and `MoneyField` do the same, each providing different code for the function. Thus, individual field objects may have various types and exhibit different behaviors, but they all share the same interface and can all be treated as `Field` objects.

A data entry form can simply maintain a list of `Field` objects and ignore the distinctions between types of fields. To read values into all the fields, a form can iterate through its list of `Fields` and call `getValue` for each without even knowing what types of fields are defined. The individual fields automatically get input using their own versions of `getValue`.

The example of the data entry forms uses inheritance for both code sharing and interface sharing. However, you can also design a class strictly for interface sharing by writing an abstract base class. The `SortableObject` class in Chapter 7 is an example of a class designed for pure interface sharing. The class's interface describes the necessary properties for an object to be stored in a `SortedList` object. However, the `SortableObject` class contains no code itself.

A class hierarchy designed for interface sharing has most of its code in the derived classes (near the bottom of the hierarchy). The derived classes represent working versions of an abstract model defined by the base class.

In summary, classes provide support for abstraction, encapsulation, and hierarchies. Classes are a mechanism for defining an abstract data type along with all the accompanying operations. Classes can be encapsulated, compartmentalizing your program and increasing its locality. Lastly, classes can be organized into hierarchies, highlighting their relationships to each other while letting you minimize redundant coding.

To gain the most benefit from object-oriented programming, you must do more than simply write your program in C++. The next section describes how you actually design an object-oriented program.

Designing an Object-Oriented System

In top-down structured programming, the first design step is to specify the program's intended function. You must answer the question, "What does the program do?" First you describe the major steps that the program must perform, using high-level pseudocode or flowcharts, and then you refine that description by breaking each major step into smaller ones. This technique is known as "procedural decomposition." It treats a program as a description of a process and breaks it down into subprocesses.

Object-oriented design differs dramatically from this technique. In object-oriented design, you don't analyze a problem in terms of tasks or processes. Nor do you describe it in terms of data; you don't begin by asking "What data does the program act upon?" Instead, you analyze the problem as a system of objects interacting. The first question you ask is, "What are the objects?" or "What are the active entities in this program?"

Not only does object-oriented design begin from a different premise from procedural decomposition, it proceeds in a different manner. Procedural decomposition is a *top-down* approach, starting with an abstract view of the program and ending with a detailed view. However, object-oriented design is not a top-down technique. You do not first identify large classes and then break them down into smaller ones. Nor is it necessarily a bottom-up process, where you start with small classes and build up from them (though class libraries can be used for this kind of approach). Object-oriented design involves working at both high and low levels of abstraction at all stages.

Object-oriented design requires you to do the following:

- Identify the classes
- Assign attributes and behavior
- Find relationships between the classes
- Arrange the classes into hierarchies

While you should begin by performing these steps in the order shown, remember that object-oriented design is an *iterative* process. If you perform each step in the process just once, without regard for the later steps, it's unlikely that you'll create a useful set of classes. Each step in the process may alter the assumptions you used in a previous step, requiring you to go back and repeat that step with new information. This does not mean that you shouldn't give any thought to your initial design. A good initial design is always desirable and will speed up the development process. However, you should expect revisions to occur. You successively refine your class descriptions throughout the design process.

Identifying the Classes

The first step is to find the classes that the program needs. This is more difficult than identifying the primary function of a program. You cannot simply perform a procedural decomposition of the problem, take the resulting structure types or data structures, and make them into classes. Your classes must be the central, active entities in your program.

One technique for identifying classes is to write a description of the program's purpose, list all the nouns that appear in the description, and choose your classes from that list. This is a simplistic approach whose success depends on how well the original description is written, but it can help give you ideas if you are new to object-oriented design.

It's easiest to identify classes for a program that models physical objects. For example, if your program handles airline seating reservations, you probably need an `Airplane` class and a `Passenger` class. If your program is an operating system, a `Device` class for representing disk drives and printers is a likely candidate class.

However, many programs don't model physical entities. For these situations, you must identify the conceptual entities that the program manipulates. Sometimes these are readily identifiable: A `Rectangle` class and a `Circle` class are obvious candidates for a graphic drawing program. In other cases, these are not as easy to visualize. For example, a compiler might need a `SyntaxTree` class, and an operating system might need a `Process` class.

Less obvious candidates for classes are events (things that happen to an object) and interactions (things that happen between objects). For example, a `Transaction` class could represent things such as loans, deposits, or funds transfers in a bank program. A `Command` class could represent actions performed by the user in a program.

You may see possible hierarchies for your classes. If you've identified `BinaryFile` and `TextFile` as candidate classes, you might derive them from a base class called `File`. However, it is not always obvious when a hierarchy is appropriate. For instance, a bank program could use a single `Transaction` class,

or it could use separate `Loan`, `Deposit`, and `Transfer` classes derived from `Transaction`. As with the classes themselves, any hierarchies identified at this stage are simply candidates to be refined or discarded later in the design process.

All of the above candidate classes are meant to model elements in the problem you're trying to solve. Your program may also need another category of candidate classes: those that can be used to implement the other classes you've found. For instance, you may have identified a class that can be implemented using the `SortedList` class that you previously wrote. In that case, `SortedList` becomes a candidate class, even if your program description didn't explicitly mention sorted lists. In general, it is too early to think about how each class should be implemented, but it is appropriate to find ways to build classes using existing class libraries.

Assigning Attributes and Behavior

Once you've identified a class, the next task is to determine what responsibilities it has. A class's responsibilities fall into two categories:

- The information that an object of that class must maintain. ("What does an object of this class know?")
- The operations that an object can perform or that can be performed on it. ("What can this object do?")

Every class has "attributes," which are the properties or characteristics that describe it. For example, a `Rectangle` class could have height and width attributes, a `GraphCursor` class could have a shape (arrow, cross hairs, etc.), and a `File` class could have a name, access mode, and current position. Every instance of a class has a "state" that it must remember. An object's state consists of the current values of all its attributes. For example, a `File` object could have the name FOO.TXT, the access mode "read-only," and the position "12 bytes from the beginning of the file." Some attributes may never change value, while others may change frequently. An attribute's value can be stored as a data member, or it can be computed each time it is needed.

It is important not to confuse attributes and classes; you should not define a class to describe an attribute. A `Rectangle` class is useful, but `Height` and `Width` classes probably are not. Deciding whether to have a `Shape` class is not so easy. When a shape is used only to describe a cursor's state, it is an attribute. If a shape has attributes that can have different values, and a set of operations that can be performed on it, then it should be a class in itself. Moreover, even if a program needs a `Shape` class, other classes may still have shape as an attribute. The `Shape` objects that a program manipulates are unrelated to the shape of a `GraphCursor` object.

Every class also has "behavior," which is how an object interacts with other objects and how its state changes during those interactions. There is a wide variety of possible behaviors for classes. For example, a `Time` object can display its current state without changing it. A user can push or pop elements off a `Stack` object, which does change its internal state. One `Polygon` object can intersect with another, producing a third.

When deciding what a class should know and what it can do, you must examine it in the context of the program. What role does the class play? The program as a whole has information that makes up its state, and behavior that it performs, and all of those responsibilities must be assigned to one class or another. If there is information that no class is keeping, or operations that no class is performing, a new class may be needed. It is also important that the program's work be distributed fairly evenly among classes. If one class contains most of the program, you should probably split it up. Conversely, if a class does nothing, you should probably discard it.

The act of assigning attributes and behaviors gives you a much clearer idea of what constitutes a useful class. If a class's responsibilities are hard to identify, it probably does not represent a well-defined entity in the program. Many of the candidate classes found in the first step may be discarded after this step. If certain attributes and behavior are repeated in a number of classes, they may describe a useful abstraction not previously recognized. It may be worthwhile to create a new class containing just those characteristics, for other classes to use.

One common mistake among programmers new to object-oriented programming is to design classes that are nothing more than encapsulated processes. Instead of representing types of objects, these classes represent the functions found by a procedural decomposition. These false classes can be identified during this stage of the design by their lack of attributes. Such a class doesn't store any state information; it just has behavior. If, when describing a class's responsibilities, you say something like, "This class takes an integer, squares it, and returns the result," you have a function. Another characteristic of such classes is an interface consisting of just one member function.

Once you've identified the attributes and behavior of a class, you have some candidate member functions for the class's interface. The behavior you've identified usually implies member functions. Some attributes require member functions to query or set their state. Other attributes are only apparent in the class's behavior.

The specific member functions and their parameters and return types won't be finalized until the end of the design process. Furthermore, implementation issues play only a small role in the design process at this point. These include questions such as deciding whether attributes should be stored or computed, what type of representation should be used, and how to implement the member functions.

Finding Relationships Between Classes

The immediate extension of the previous step, deciding the features of each class, is deciding how the classes use each other's features. While some of the classes you identify can exist in isolation, many of them cannot. Classes build upon and cooperate with other classes.

Often one class depends upon another class because it cannot be used unless the other class exists. This is necessary when one class calls the member functions of the other class. For example, a `Time` class may have functions that provide conversions between it and a `String` object. Such functions must call the constructor and access functions of the `String` class.

Another way one class can depend on another is when it has the other class embedded within it, meaning that it contains objects of the other class as members. For example, a `Circle` object might have a `Point` object representing its center, as well as an integer representing its radius.

This type of relationship is called a "containing relationship." Classes that contain other classes are "aggregate" or "composite" classes. The process of containing member objects of other classes, known as "composition," is described in Chapter 4, "Introduction to Classes." Composition is sometimes confused with inheritance; the distinction between these two is discussed in the next section.

Most relationships between classes arise because one class's interface depends on another. For example, the class `Circle` may have a `getCenter` function that returns a `Point` object, so users must know about `Point` to use `Circle`'s interface. However, it is also possible for a class's implementation to depend on another class. For example, you might design `AddressBook` with a private member object of the `SortedList` class. Users of `AddressBook` don't need to know anything about `SortedList`. They need only know about the interface of `AddressBook`. This provides another layer of encapsulation, because it is possible to change the implementation of `AddressBook` without changing the interface.

When identifying relationships, you must consider how a class performs its assigned behavior. Does it need to know information that is maintained by other classes? Does it use the behavior of other classes? Conversely, do other classes need to use this class's information or behavior?

As you define the relationships between classes more fully, you'll probably modify some of the decisions you made in previous steps. Information or behavior that was previously assigned to one class may be more appropriate in another. Don't give objects too much information about their context. For example, suppose you have a `Book` class and a `Library` class for storing `Book` objects. There's no need for a `Book` object to know which `Library` holds it; the `Library` objects already store that information. By adjusting the borders between classes, you refine your original ideas of each class's purpose.

You might be tempted to use friend classes when you have one class that needs special knowledge about another class. However, the friend mechanism in C++ should be used very sparingly, because it breaks the encapsulation of a class. Modifying one class may require rewriting all its friend classes.

After identifying the relationships that one class has with others, you reach a closer approximation of the class's interface. You know which attributes require member functions to set them and which attributes require only query functions. You have a clearer idea of how best to divide the class's behavior into separate functions.

Arranging Classes into Hierarchies

Creating class hierarchies is an extension of the first step, identifying classes, but it requires information gained during the second and third steps. By assigning attributes and behavior to classes, you have a clearer idea of their similarities and differences; by identifying the relationships between classes, you see which classes need to incorporate the functionality of others.

One indication that a hierarchy might be appropriate is the use of a **switch** statement on an object's type. For example, you might design an `Account` class with a data member whose value determines whether the account is a checking or

savings account. With such a design, the class's member functions might perform different actions depending on the type of the account:

```
class Account
{
public:
   int withdraw( int amount );
   // ...
private:
   int accountType;
   // ...
};

int Account::withdraw( int amount )
{
    switch ( accountType )
    {
    case CHECKING:
        // perform checking-specific processing
        break;
    case SAVINGS:
        // perform savings-specific processing
        break;
    // ...
    };
}
```

A **switch** statement such as this usually means that a class hierarchy with polymorphism is appropriate. As described in Chapter 7, "Inheritance and Polymorphism," polymorphism lets you call member functions for an object without specifying its exact type, by using virtual functions.

In the example above, the `Account` class can be made into an abstract base class with two derived classes, `Savings` and `Checking`. The `withdraw` function can be declared virtual, and the two derived classes can each implement their own version of it. Then you can call `withdraw` for an object without examining the object's precise type.

On the other hand, a hierarchy isn't needed just because you can identify different categories of a class. For example, is it necessary to have `Sedan` and `Van` as derived classes of `Car`? If you perform the same processing for every type of car, then a hierarchy is unnecessary. In this case, a data member is appropriate for storing the type of car.

Composition vs. Inheritance

Both composition and inheritance enable a class to reuse the code of another class, but they imply different relationships. Many programmers automatically use inheritance whenever they want to borrow the functionality of an existing class, without considering whether inheritance accurately describes the relationship between their new class and the existing one. Composition should be used when one class *has* another class, while inheritance should be used when one class *is* a kind of another class. For example, a circle is not a kind of point; it *has* a point. Conversely, a numeric data field does not contain a generic data field; it *is* a data field.

Sometimes it is difficult to determine whether inheritance or composition is appropriate. For example, is a stack a kind of list with a special set of operations, or does a stack contain a list? Is a window a kind of text buffer that can display itself, or does a window contain a text buffer? In such cases, you have to examine how the class fits in with the other classes in your design.

One indication that inheritance is the appropriate relationship is when you want to use polymorphism. With inheritance, you can refer to a derived object with a pointer to its base class and call virtual functions for it. With composition, however, you cannot refer to a composite object with a pointer to one of its member classes, and you cannot call virtual functions.

If you want to borrow another class's functionality more than once, composition is probably more appropriate. For example, if you're writing a `FileStamp` class and you want each object to store a file's creation date, last modification date, and last read date, composition is clearly preferable to inheritance. Rather than use a complicated multiple inheritance design, it's much simpler to include three `Date` objects as members.

Designing Classes for Inheritance

Building class hierarchies usually involves creating new classes as well as modifying or discarding ones previously identified. Most of the classes identified during the first step are probably ones that you intend to instantiate. However, when the common features of several classes are isolated, they often don't describe a class that is useful to instantiate. As a result, the new classes you create when forming a hierarchy may be abstract classes, which are not meant to have instances.

Adding abstract classes increases the ability to reuse a class. For example, you might create one abstract class that five classes inherit from directly. However, if two of those deriving classes share features that the others don't, those features can't be placed in the base class. As a result, they would have to be implemented

in each class they appeared in. To prevent this, you can create an intermediate abstract class that is itself derived from the base but adds new features. The two classes can inherit their shared characteristics from this class. This also gives you greater flexibility when extending the hierarchy later on.

However, abstract classes should not be created indiscriminately. As an extreme example, it's possible to create a series of abstract classes, each of which inherits from another and adds only one new function. While in theory this promotes reusability, in practice it creates a very clumsy hierarchy.

It is desirable to place common features as high in a hierarchy as possible to maximize their reuse. On the other hand, you should not burden a base class with features that few derived classes use. Attributes and behavior may be shifted up and down the hierarchy as you decide which features should be placed in a base class.

Inheritance not only affects the design of class hierarchies, it can also affect the design of classes that stand alone. Any class you write might later be used as a base class by another programmer. This requires a refinement to the distinction between a class's interface and implementation. A class has two types of clients: classes or functions that use the class, and derived classes that inherit from the class. When designing a class, you must decide whether you want to define different interfaces for these two types of clients. The **protected** keyword enables you to make members visible to the derived classes but not to the users. You can thus give derived classes more information about your class than you give users.

A protected interface can reveal information about a class's implementation, which violates the principle of encapsulation. Modifying the protected portion of a class means that all derived classes must be modified too. Accordingly, the **protected** keyword should be used with care.

Multiple Inheritance

Multiple inheritance can be useful for maximizing reuse while avoiding base classes with unnecessary functionality. For example, remember the example of the abstract base class `SortableObject`, which has the interface that a class needs in order to be stored in a `SortedList`. Now consider a similar abstract base class called `PrintableObject`, which has an interface that all printable classes can use. You might have some classes that are printable, some that are sortable, and some that are both. Multiple inheritance lets you inherit the properties you need from the abstract base classes.

This scenario is difficult to model using only single inheritance. To avoid duplicating functions in different classes, you'd have to define a base class `PrintableSortableObject` and derive all your other classes from it. Certain classes would have to suppress the functions of the printable interface, while others would have to suppress the functions of the sortable interface. Such a class hierarchy is top-heavy, having too much functionality in its base class.

Virtual base classes are often used in hierarchies built around multiple inheritance. One drawback of virtual base classes, besides the processing overhead they entail, is that the need for them only becomes apparent when you design an entire hierarchy at once. Consider the example of the `SalesManager` class in Chapter 7, "Inheritance and Polymorphism." The need to make `Employee` a virtual base class doesn't arise until `SalesManager` is defined. If you didn't define `SalesManager` before the other classes were used in many programs, you must modify the existing hierarchy, causing extensive recompilation. If modifying the hierarchy isn't practical, you must use some other solution instead of virtual base classes.

Just as with single inheritance, multiple inheritance is often used inappropriately. Many situations where multiple inheritance is used are better modeled with composition or with a combination of composition and single inheritance. You should always examine one of these options when considering multiple inheritance.

Now that you've seen the major steps in designing object-oriented programs, the next chapter gives a concrete example of how this design technique can be applied.

CHAPTER 10

Design Example: A Windowing Class

Chapter 9, "Fundamentals of Object-Oriented Design," described a technique for designing an object-oriented program. This chapter illustrates that technique by applying it to a design for a simple character-based windowing package, which could be implemented in C++. The primary purpose of this chapter is not to describe windowing systems or to demonstrate actual C++ code, but rather to provide a demonstration of object-oriented design.

The complete implementation of this design is beyond the scope of this book. However, the decisions that go into the design of these classes should be helpful in guiding your design of a complete system using C++ classes.

Examining the Requirements

No matter what design technique you employ, you must define what the program is supposed to do before you can begin designing. Consider how the completed program should behave.

The windowing package does not form a stand-alone application. It is meant to be used by a client who is writing a stand-alone program that displays text in multiple, overlapping windows. The windows can scroll in both vertical and horizontal directions, so a window can contain more text than it can display at once. The windows can be manipulated with either the keyboard or the mouse. For example, the user can scroll a window by moving its cursor with the keyboard cursor keys or by clicking on its scroll bar(s) with the mouse. It's possible to create windows in which the user can enter text or execute commands. (Note that the user is not necessarily the client; the user is the person who uses the program that the client has written.)

Because a user can interact with only one window at a time, whether scrolling or entering text, you need a way to specify which window receives input. That window is the "active" window. Only one window can be active at any given time.

The active window must be fully displayed. It cannot have any portion obscured by other windows. Whenever an inactive window becomes active, it must be redisplayed over any windows that might cover it (like a sheet of paper being moved to the top of a stack of paper). The user changes the active window by clicking the mouse on an inactive window or by entering a keyboard command.

The program is illustrated in Figure 10.1.

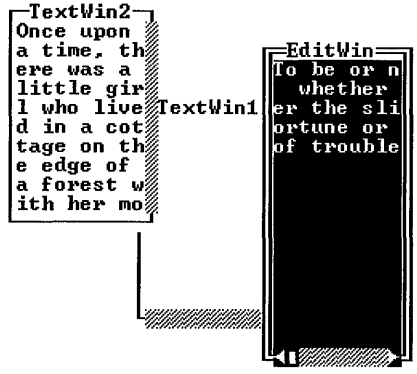


Figure 10.1 Character-Based Windows

Designing the Classes

As described in the previous chapter, the major steps in object-oriented design are:

- Identify the basic classes
- Assign attributes and behavior
- Find the relationships between the classes
- Arrange the classes into hierarchies

The following sections describe how these steps apply to the windowing package, but the sections don't have an exact one-to-one correspondence with the steps listed above. That's because the steps are all interrelated, so that more than one step is performed at the same time, and they're performed iteratively, so that some steps are performed more than once.

Identifying Candidate Classes

What classes are needed for this program? An obvious choice is a window class, where each window is an object. The first candidate class is therefore called `Win`.

The program handles user actions such as keyboard input and mouse clicks. These actions are significant, because they can change the state of a window. There should also be an `Event` class to describe such user events. For example, when a user clicks the mouse over a window, an `Event` object is passed to the appropriate `Win` object, and the `Win` object responds accordingly. This is an example of a class that describes things that happen to objects.

If there are multiple windows, should the client program be responsible for keeping track of the active window and directing events to it? That's an unreasonable burden to place on the client program. The package should provide a class that maintains information about all windows and mediates between them and the user. This candidate class is called `WinMgr`. This class doesn't model an entity mentioned in our description of the program. It is created because there is information that must be maintained by the program, but that information isn't stored by any other classes.

Because this program frequently manipulates positions, it is convenient to represent positions using a simple `Point` class, instead of a pair of integers. Similarly, because the program uses rectangular windows, a `Rect` class is a reasonable candidate. These classes are extremely simple ones, being little more than convenient structures.

Attributes and Behavior for Windows

What information does a `WinMgr` object maintain? Should it store the positions of the windows? No, because it's more appropriate for each window itself to know that. However, in keeping with the guideline that objects shouldn't know too much about their context, a window shouldn't know whether it lies above or below other windows. Instead, `WinMgr` should maintain the stacking order of the windows. Because the topmost window is always the active one, `WinMgr` always knows which window is active.

What can a `WinMgr` object do? It directs user-generated events to the active window, but it can't pass all of them to that window. For example, the user can click on an inactive window to make it active; such an event cannot be sent to the active window. `WinMgr` must respond to that event and restack the windows, so that a newly active window moves from its previous position in the stack to the top.

What information does a `Win` object maintain? It knows its size and position on the screen. It stores all the text that it must display, which, for a scrollable window, includes the text that is outside the window's bounds. The window

knows where the displayed section is located relative to the entire text, so it can position the slider on the scroll bar.

What can a `Win` object do? It can display itself on the screen. It can respond to the keyboard by moving its cursor, and, if it's an editing window, by modifying its text. It can also respond to the mouse. It can position the cursor if the mouse is clicked on the text and scroll the text if the mouse is clicked on the scroll bar.

Refining the Window Classes

Consider the behavior of the scroll bar more closely. If the user clicks on either end, the text moves one line in the appropriate direction. If the user clicks on either side of the scroll box (also called the slider), the text moves one page in the appropriate direction. (We'll ignore dragging on the scroll box itself.)

This is a fairly complex set of responses for a window to perform. Moreover, a window might have two scroll bars, one vertical and one horizontal, which requires a lot of repetition in the code for responding to a mouse click. Instead of making a window responsible for scroll-bar behavior, you can make scroll bars a separate class. A text window is thus responsible for interpreting mouse clicks on the text, while a scroll bar is responsible for interpreting mouse clicks on its surface. Objects of these two classes interact to perform text scrolling.

What information does a `ScrollBar` object maintain? It knows its size, but what about its position? Suppose it knows its absolute position on the screen. With such a design, consider how you would move a scrollable window. Not only would you update the window's position attributes, you'd also have to update the attributes of both scroll bars. This is an example of giving an object too much information about its context. By knowing its absolute position, `ScrollBar` is implicitly aware of the position of the window containing it. Instead, let `ScrollBar` store only its *relative* position, as an offset from the corner of the window that contains it. With this distribution of information, moving a window requires updating only the window's position attributes.

How is `ScrollBar` related to the classes already defined? It has a number of similarities to the `Win` class: It knows its size and relative position, and it can display itself and respond to mouse clicks. You can factor out this common behavior, place it in a base class, and make scroll bars and text windows derived classes.

Thus, the fundamental entities in our program are not text windows but something more primitive. This new class represents a screen area that interacts with the user as a logical unit, so call it `Interactor`. It should be an abstract base class, because there's no such thing as a generic interactive area. A `Win` is an `Interactor`, and so is a `ScrollBar`.

The `Interactor` class is an example of a class that wasn't among the original candidate classes but was created after examining the attributes and behavior of the candidates. The class represents an entity that wasn't immediately evident in the description of the program. The class hierarchy is shown in Figure 10.2.

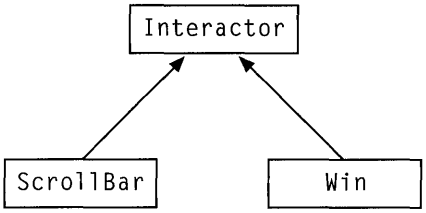


Figure 10.2 First Window Class Hierarchy

Remember that this is only the first approximation of the window class hierarchy, and that it may change as you design the classes in more detail.

Attributes and Behavior for Events

What information is stored in an `Event` object? If the event comes from the keyboard, the relevant information is the key that was pressed. If the event comes from the mouse, the relevant information is the location of the mouse cursor and the status of the mouse buttons. These are two distinct types of event, so you can make two separate classes, `KbdEvent` and `MouseEvent`.

What can a `KbdEvent` or a `MouseEvent` do? These classes don't do much, except report the information they contain. They're little more than fancy structures.

Even though these two classes don't appear to share anything in common, you can derive them both from an abstract base class `Event` as a way of indicating that both represent types of events. This provides a single means of referring to both types of events so that, if desired, objects of either type could be passed to a function. This technique is discussed in more detail in the section "The Event Hierarchy," page 203.

This event hierarchy is shown in Figure 10.3.

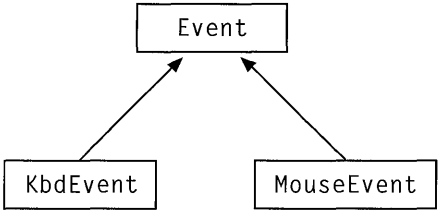


Figure 10.3 Event Hierarchy

Identifying Relationships Between Classes

What are the relationships between the classes identified thus far? A text window can have one or two scroll bars. Each scroll bar is said to be a "child" of the text window, and the text window is the "parent" of the scroll bar(s). This can be represented as a containing relationship, where `Win` contains one or two instances of `ScrollBar` as member objects. `Win` therefore uses `ScrollBar`.

The interaction between windows and scroll bars requires two-way communication, because using the scroll bar affects the window and moving the window's cursor affects the scroll bar. As a result, scroll bars must also have knowledge of text windows. This differs from the situation where, for instance, a circle contains a point to represent its center; `Circle` must know about `Point`, but the reverse is not true. `ScrollBar` must know about the interface of `Win` to communicate with it.

Events are passed to interactors, whether text windows or scroll bars, so `Interactor` must have a member function that accepts an `Event` object as a parameter. `Interactor` thus uses `Event`.

Note that `Win` must be defined before `WinMgr`, because `WinMgr` manipulates `Win` objects. Because it's possible to have an arbitrary number of windows, `Wins` cannot be member objects. Instead, you can make `WinMgr` a collection class, which allows you to insert or delete `Win` objects. `WinMgr` also uses `Events`, because it passes them to the windows.

However, because scroll bars never exist independently and are never stacked on each other, it is unnecessary for a `WinMgr` to manipulate `ScrollBar` or generic `Interactor` objects.

The relationships between the classes are shown in Figure 10.4.

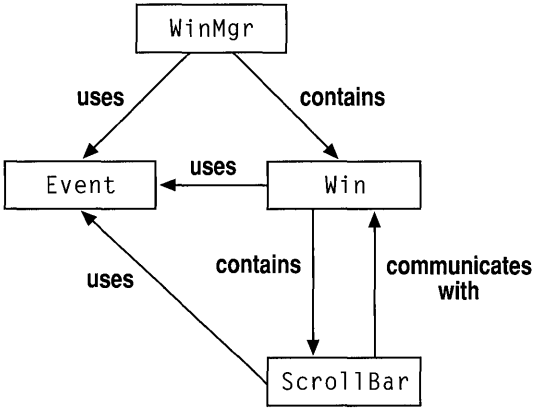


Figure 10.4 Relationships Between Classes

Defining Preliminary Class Interfaces

The hierarchies are very simple at this point, so there's no need to restructure them. You can wait until later in the design process to work on their organization. However, it is appropriate to decide which features belong in the base classes and which belong in the derived classes. This requires a closer examination of the classes' attributes and behavior, so you should make a first approximation of the classes' public interfaces.

The Window Classes

The window hierarchy consists of a base `Interactor` class, with `Win` and `ScrollBar` as derived classes. Let's consider these classes in turn, starting at the top of the hierarchy and moving down.

Interactors

What are the common characteristics of all interactors, including both text windows and scroll bars? They all have a size and a relative position, and this information must be accessible. They can check whether a given pair of coordinates falls within their borders, indicating that they should respond to a mouse click. In addition, all interactors have the ability to display themselves and to respond to events. However, because there is no way for a generic `Interactor` object to paint itself or to respond to events, those member functions should be declared as pure virtual functions.

Recall that `Point` and `Rect` classes are available for use. Using these classes, we can write a preliminary interface for `Interactor` as follows:

```
class Interactor
{
public:
   int width();
   int height();
   Point origin();
   int withinBounds( Point pos );
   virtual void paint() = 0;
   virtual void handleEvent( Event &action ) = 0;
protected:
   Rect area;
};
```

Because of its pure virtual functions, the `Interactor` class cannot be instantiated. However, the simple functions `withinBounds`, `width`, `height`, and `origin` can all be implemented here, because those functions are the same for all interactors.

Text Windows

The `Win` class is derived from the `Interactor` class. What additional information does `Win` maintain, besides that stored by `Interactor`? As mentioned earlier, a text window must store all the text it can display. An array can be used to hold the text; something dynamically resizable, such as a linked list, is more flexible, but for this example each window has a fixed amount of text.

The window might contain one or two scroll bars, or none at all, depending on the length and width of the text being displayed. Should there be separate classes for windows with two scroll bars, windows with one, and windows with none? That seems unnecessary. It's simpler to have a single, flexible window class that can

have zero, one, or two scroll bars. Because the scroll bars are optional, they shouldn't be contained as member objects, because that would require a non-scrolling window to store unnecessary objects. Instead, you can use pointers to `ScrollBars`. `Win`'s constructor can allocate `ScrollBar` objects if the text is larger than the window and set the pointers to NULL otherwise.

If the window is scrollable, the text document being displayed is larger than the window, so the document's dimensions must also be stored. Similarly, it is also necessary to store the position of the window relative to the document as a whole. A window can optionally have a title, which must be stored as well. A preliminary interface for `Win` therefore looks like this:

```
class Win : public Interactor
{
public:
   void paint();
   void handleEvent( Event &action );
   void setchar( Point pos, char newchar );
   char retchar( Point pos );
   void putstr( Point pos, char *newstr );
   int rows();
   int columns();
   void setTitle( char *newtitle );
private:
   ScrollBar *hscroller;
   ScrollBar *vscroller;
   char *textBuffer;
   int textrows, textcolumns;
   Point position;              // Position of window in text
   Point cursorPos;
   char *title;
};
```

This is a lot of information for one class to maintain. You can create a new class, `Buffer`, that stores the text and include it as a member of `Win`. For example:

```
class Buffer
{
public:
   int rows();
   int columns();
   void setchar( Point pos, char newchar );
   char retchar( Point pos );
   void putstr( Point pos, char *newstr );
private:
   int width, length;
   char *textArray;
};

class Win : public Interactor
{
public:
   void paint();
   void handleEvent( Event &action );
   void setTitle( char *newtitle );
private:
   ScrollBar *hscroller;
   ScrollBar *vscroller;
   Buffer canvas;
   Point position;            // Position of window on text
   Point cursorPos;
   char *title;
};
```

This design divides the responsibilities of the previous `Win` class into two categories: storing the text and displaying the text on screen. Each `Win` object contains a pointer to its associated `Buffer` object, which stores the text that the window displays.

Scroll Bars

What additional information does `ScrollBar` maintain, besides that stored by `Interactor`? A scroll bar knows whether it is vertical or horizontal, and that information can be represented as a flag. It knows the position of the scroll box, which can be represented as a number between 1 and 100.

A scroll bar is constrained to be one character wide if it's vertical, or one character high if it's horizontal. It cannot exist separately from a text window. A text window must exist before a scroll bar can be created. The `ScrollBar` constructor can ensure these conditions.

Is any other member function needed to use a `ScrollBar` object? If the user scrolls a text window by moving the cursor with the arrow keys, the scroll bar's scroll box should move. It doesn't seem appropriate to send keyboard events to the scroll bar; the text window itself should handle those. `ScrollBar` should therefore have a function that sets the scroll box (slider) position, which `Win` can call. This means the preliminary interface to `ScrollBar` looks like this:

```
class ScrollBar : public Interactor
{
public:
   void paint();
   void handleEvent( Event &action );
   void setSlider( int pos );
private:
   Win *parentWin;
   int sliderPos;
   int orientation;
};
```

How do a scroll bar and its parent interact? When the mouse is clicked on a scroll bar, the text in the window scrolls and the scroll box moves. A `ScrollBar` object must tell its associated `Win` object to perform a scrolling action; it can do this by sending a new type of event, a `ScrollEvent`, that contains the direction (backward or forwards) and the amount (one line or one page).

Should `ScrollBar` move its scroll box by itself? How does it know how far to move the scroll box? Consider: If the text is four times as long as the window, scrolling down one page means moving the scroll box one-fourth of the way down the scroll bar. However, if the text is 20 times as long as the window, scrolling down one page means moving the scroll box one-twentieth of the way down the scroll bar. Scrolling one line at a time is similar. Thus, to calculate the distance the scroll box should move, `ScrollBar` would have to know the length of the document and the height of the window (or the corresponding ratio for horizontal scrolling). That would be a duplication of responsibilities, because those values are already stored by `Win`. Therefore, `ScrollBar` does not move its scroll box when the mouse is clicked. It only adjusts the scroll box when `Win` calls its `setSlider` function. `Win` is responsible for computing the proper position for the scroll box.

The Window Manager

The `WinMgr` class maintains the order of the text windows and passes them events from the user. You can implement `WinMgr` as a collection class of `Win` objects, but what type of collection class should you use?

On the screen, the windows appear stacked like sheets of paper on a desktop, with the topmost window being the active one. This suggests a collection class

that implements a stack data structure. However, a stack data structure permits access to only the topmost element. `WinMgr` must have access to windows other than the active one in order to make nonactive windows active. Therefore, a stack class is inappropriate. Some kind of list class that provides access to all the elements is needed.

Consider the order in which `WinMgr` must access the windows to see which one should receive a mouse event. More than one window may occupy the position where the mouse was clicked, but only the exposed one receives the event. To find the exposed window, `WinMgr` must test the windows in order, starting from the top of the stack and going to the bottom. However, suppose `WinMgr` must refresh the screen, painting all the overlapping windows over again. This time, `WinMgr` must access the windows starting from the bottom of the stack and going to the top. Thus, `WinMgr` needs a collection class that permits iteration of its elements in both directions.

Accordingly, a `List` class that allows operations in both directions is appropriate. Such a class could be implemented with a doubly linked list or an array. The `WinMgr` class can have a member object of type `List`.

Most of the behavior of `WinMgr` is implemented by its `handleEvent` function. The interface is therefore fairly simple:

```
class WinMgr
{
public:
   void handleEvent( Event &action );
   void addWindow( Win *newWindow );
   void deleteWindow();
   void repaint();
private:
   List winlist;
};
```

Consider how `WinMgr`, `Win`, and `ScrollBar` work together in a typical scenario. (This is illustrated in Figure 10.5.) The user clicks the mouse on a scroll bar in a text window. An `Event` object, containing the location at which the click occurred, is sent to the `WinMgr` object, which queries the windows in order to see which one is exposed at that location. `WinMgr` then passes the event to the appropriate `Win` object, which in turn queries its scroll bars to see if either of them should receive it. It passes the event to the appropriate `ScrollBar`, which checks the location of the mouse click and interprets it as meaning "scroll forward one line." The `ScrollBar` object creates a new `Event` object containing this scrolling information and sends it to its parent. The text window scrolls the text it contains and updates the position of the scroll bar's scroll box.

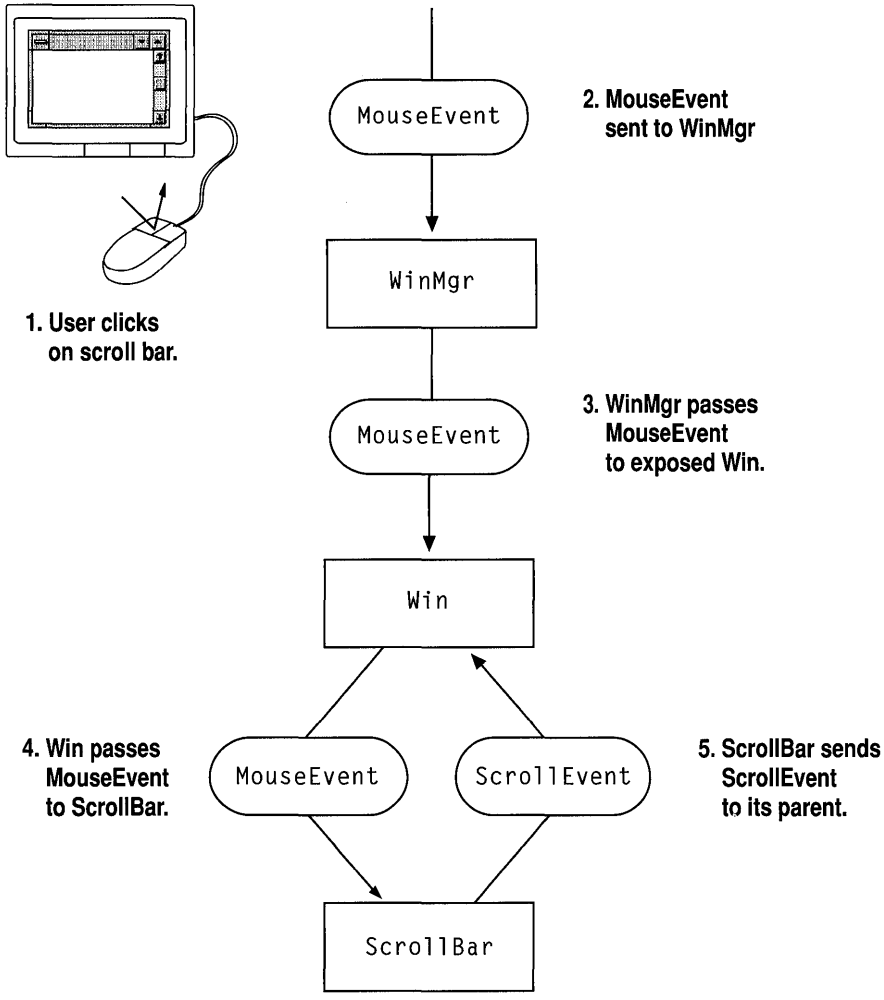


Figure 10.5 Event Passing

The Event Hierarchy

Now consider the interfaces to the various event classes. As mentioned earlier, having `Event` as a base class for all types of events allows any of them to be passed to the single `handleEvent` function in the interface to `Interactor`. However, an `Interactor` needs to know specifically which type of event it's received, because it has different responses to keyboard events than to mouse events.

For this reason, the Event class defines a virtual getType function, which returns a constant indicating the type of the event. The interfaces of Event and its derived classes look like this:

```
class Event
{
public:
   virtual EventType getType() = 0;
};

class KbdEvent : public Event
{
public:
   EventType getType() { return KBD_EVENT; }
   unsigned int val()
private:
   char ascii;
   char scancode;
};

class MouseEvent : public Event
{
public:
   EventType getType() { return MOUSE_EVENT; }
   Point getPosition()
   int getButton()
private:
   Point pos;
   int buttons;
};
```

Because there is no type associated with generic Event objects, the getType function is declared as pure virtual. Each derived class overrides getType to return a unique constant.

This design is analogous to the use of a union in C:

```
// Analogous situation in C

struct Event
{
   EventType type;
   union
   {
    struct KbdEvent keyAction;
    struct MouseEvent mouseAction;
   };
};
```

The generic `Event` interface corresponds to a structure containing a union, and the `KbdEvent` and `MouseEvent` classes correspond to possible contents of the union. The `getType` function acts like a discriminator field in the structure, indicating the current type of the contents of the union.

This design can be extended to include a third type of event, `ScrollEvent`, which was mentioned in the previous section. This type of event contains the direction and amount of scrolling that is requested. However, this isn't enough information to perform scrolling. If a text window contains two scroll bars and receives a `ScrollEvent`, it doesn't know whether to scroll the text vertically or horizontally unless it knows which scroll bar sent the message. Therefore, a `ScrollEvent` also contains a pointer to the `ScrollBar` that created it. Its interface is as follows:

```
class ScrollEvent : public Event
{
public:
   EventType getType() { return SCROLL_EVENT; }
   int getDirection();
   int getDistance();
   ScrollBar *getSource();
private:
   int direction;
   int distance;
   ScrollBar *source;
};
```

Notice that you can interpret a `ScrollEvent` to have a meaning other than "scroll text." You can use scroll bars to adjust the volume of a beeper, change the shading of a colored panel, or perform other similar actions.

A window's `handleEvent` function must use a **switch** statement to determine the type of event received. For example:

```
void Win::handleEvent( Event &action )
{
   switch( action.getType() )
   {
   case KBD_EVENT:
      KbdEvent &keyAction = (KbdEvent &)action;  // Cast
      // Respond to keyboard event
      break;
   case MOUSE_EVENT:
      MouseEvent &mouseAction = (MouseEvent &)action;  // Cast
      // Respond to mouse event
      break;
   //...
   };
}
```

Once the type of the event is found, the `handleEvent` function converts the base class pointer to a derived class pointer. (This is not generally a safe conversion in C++, but it works properly in this case because the return value of `getType` guarantees the type of the object.) The conversion to a derived class pointer allows `handleEvent` to access the information specific to that type of event and respond with the appropriate action.

To add a new type of event, you must derive a new class from `Event`. If you want an existing window class to respond to the new type of event, you have to change that class's `handleEvent` function. You can extend the **switch** statement by adding a **case** clause for the new event type.

Limitations of Polymorphism in C++

The event-handling scheme described above sounds like a situation that calls for polymorphism. It seems like it should be possible to replace the **switch** statement with virtual functions. For example, you could give each subclass of `Event` a

virtual `respond` function. A different action would be taken for each type of event, and you could then write the following:

```
// Hypothetical example with polymorphic Events

void Win::handleEvent( Event &action )
{
   action.respond();
}
```

However, this would mean that the actions taken for each event would be the same for every type of interactor. A scroll bar would behave in the same way as a text window. This is clearly unsatisfactory.

What is needed is a way to vary the behavior of `handleEvent` on two parameters: the type of interactor and the type of event. In C, it would look like this:

```
handleEvent( myInteractor, currAction );
```

Such a function in C would contain a giant **switch** statement based on the type of interactor, with each **case** clause containing another **switch** statement based on the type of event. Through polymorphism in C++, the interactor parameter can be removed by giving the interactor class a virtual `handleEvent` function, as follows:

```
myInteractor->handleEvent( currAction );
```

Unfortunately, C++ does not allow an additional level of polymorphism, which would allow you to remove the event parameter. To retain the polymorphism already in place, you must use a **switch** statement for the event parameter.

This illustrates a situation where language limitations influence the design. Because C++ doesn't support an ideal solution, this example program has to use a technique that's less than optimal. You should remember that, in general, it's not good practice to use a **switch** statement to examine an object's type. Only when your design calls for multiple levels of polymorphism is the technique permissible. (A technique that avoids the use of **switch** statements is outlined in the comments to the source code.)

Remember that the classes' public interfaces are not final until the implementation phase. For the final interfaces and the implementation code for this example, see the files in the sample program directory.

Expanding the Hierarchies

The classes are already in small hierarchies, but now that you have a first approximation of the class interfaces, it's appropriate to take another look at those hierarchies and see if they need to be restructured. The goals during this stage are to maximize reuse of code and to take advantage of polymorphism. These goals can often be achieved by adding new base classes, so you should look for opportunities to expand the hierarchies.

New Window Classes

The scrollable text window is useful, but the client may want other kinds of windows as well. The client should be able to define a new window class while still taking advantage of `WinMgr` and the existing event-passing scheme.

You can support this by changing `Win` to an abstract class descended from `Interactor`. We'll change the name of our text window class to `TextWin` and derive it from `Win`. `Win` acts as the base for any type of window, including those the client defines. `WinMgr` treats all windows it manages as generic `Win` objects and passes events to them. Furthermore, a scroll bar considers its parent to be a generic `Win`, so it can be a child of any type of window, not just text windows.

Thus, `Win` provides a polymorphic interface through which `WinMgr` and `ScrollBar` can access all windows. To a lesser degree, `Win` can also permit code reuse if you give it the common characteristics of all windows (for example, a data member to store a title and a function to set the title). The revised window hierarchy is shown in Figure 10.6.

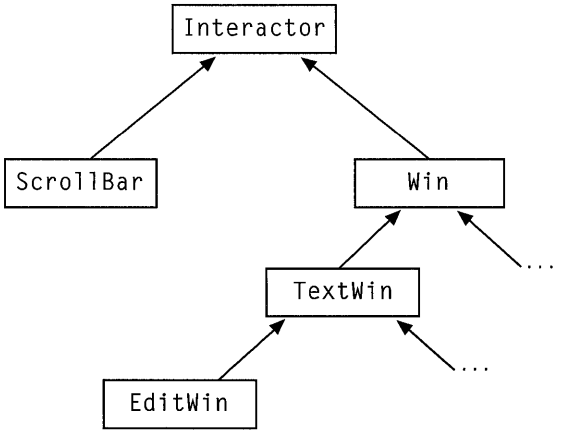


Figure 10.6 Revised Window Class Hierarchy

Suppose you want an editable text window. Such a window should have all the scrolling and cursor movement capabilities of the noneditable window already defined, plus the ability to accept text from the user. This situation calls for inheritance.

You can define a class `EditWin` that derives from `TextWin`. Because a pointer to a derived class object can be treated like a pointer to a base class object, you can insert these editable windows into `WinMgr`.

The main difference between `TextWin` and `EditWin` is in their behavior. When a printable character is received from the keyboard, `TextWin` ignores it. `EditWin` responds by writing that character at the current cursor location and moving the cursor one space to the right. `EditWin` and `TextWin` respond identically to all other events, such as mouse clicks or cursor key presses.

The new behavior must be implemented in the `handleEvent` function of `EditWin`, overriding the `handleEvent` function of `TextWin`. However, it is not necessary to reimplement all the behavior that the windows share. The `handleEvent` function of `EditWin` can call the `handleEvent` function of `TextWin` and pass it all events that it doesn't handle itself. For example:

```
EditWin::handleEvent( Event &action )
{
   switch ( action.getType() )
   {
      KBD_EVENT:
         KbdEvent &keyAction = (KbdEvent &) action;
         if ( printable( keyAction.val() ) )
            // Modify text buffer, move cursor
         else                          // Pass on other keyboard events
            TextWin::handleEvent( action );
         break;
      default:                         // Pass on all other events
         TextWin::handleEvent( action );
         break;
   };
};
```

In this manner, `EditWin` can reuse selected portions of `TextWin`'s event handling. This technique, having a derived class's member function call the base class's member function, can be used in many situations.

New Control Elements

A window can contain other interactive elements besides scroll bars. For example, a window can contain a button that performs some action when pressed. You can create a new class, `PushButton`, to represent this type of interactor.

What information does a `PushButton` store, besides that stored by all `Interactor` objects? All buttons have a label indicating what they do; a button must retain the text string that it displays as its label. What can a `PushButton` do? Any button should momentarily reverse its displayed colors to give the user some feedback. Beyond that, the action performed as a result of pressing the button is entirely determined by the client. Just as a scroll bar can be used for different purposes, a button can be used for almost anything. The only thing a button by itself can do is inform its parent window that it has been pressed. To do this, a button sends its parent a new type of event, a `PushEvent`.

A `PushEvent` contains a pointer to the button that created it, so the parent window can identify the button that was pressed. It is then up to the parent to perform some function in response.

You could define `PushButton` as a descendant of `Interactor`, but it is very similar to the `ScrollBar` class. Both are always children of windows and thus contain a pointer to a parent of class `Win`. You can create an abstract base class for both `ScrollBar` and `PushButton` and put the pointer in that class. This new class is named `Control`, because its descendants are all control elements for windows, and it itself is a descendant of `Interactor`.

By isolating the common features of all control elements, `Control` provides some code reuse. In the current design, `Control` isn't used for polymorphism. However, in a possible alternate design, `Win` objects could contain a collection of generic `Control` objects, instead of a fixed number of specific `Controls`, such as two scroll bars. In such a design, `Control` would provide a useful abstract interface to scroll bars, buttons, and any other interactive elements you define.

The revised hierarchy is shown in Figure 10.7.

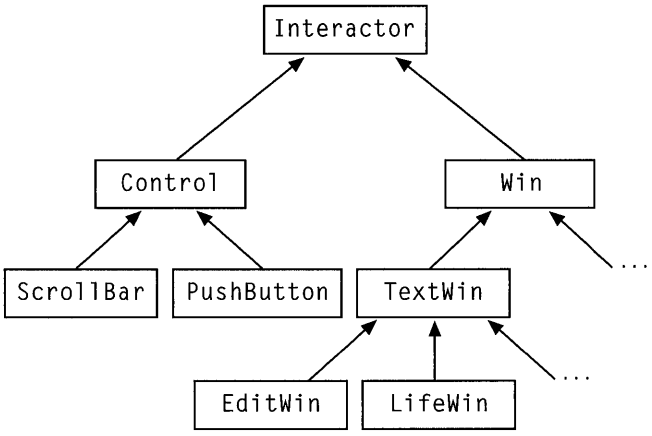


Figure 10.7 Final Window Class Hierarchy

You can now define new types of windows, descended from `Win`, that contain buttons. For example, imagine a class `LifeWin` that runs the game of Life in a window. This class can inherit the scrolling and cursor movement functionality of `TextWin`, but it can respond differently to mouse actions in the window, as well as provide commands through buttons. This is illustrated in Figure 10.8.

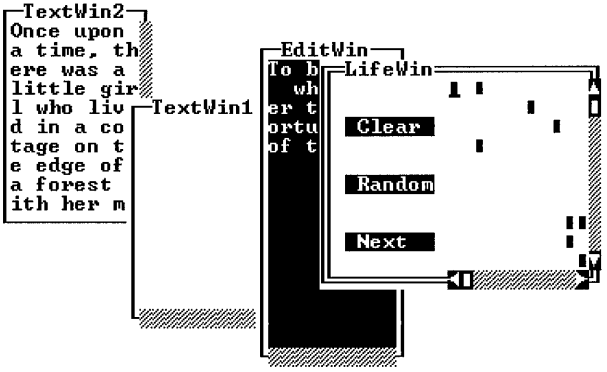


Figure 10.8 Window with Buttons

What Doesn't Fit in This Hierarchy

Is it possible to define pull-down menus as a type of window? Consider a menu's characteristics. A menu's position is restricted to the top of the screen. Its size is determined by the number of items it contains and the width of the longest item. A menu appears only when an entry on the menu bar is activated, and it disappears when the command is executed. This type of behavior doesn't fit easily into the current windowing hierarchy. Pull-down menus thus require a separate hierarchy, which is beyond the scope of this chapter.

Remember that the classes developed here are simply a small example for demonstrating the process of object-oriented design. To write a professional windowing package, you would need to design a much larger class hierarchy. If you want to write applications with windowing capabilities for the Microsoft® Windows™ operating system, see the *Class Library User's Guide* and the reference documentation for the Microsoft Foundation Class Library. Also, note that the design presented in this chapter is only one of several possible designs for the windowing package described here.

Index

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

V

W

Class Library User's Guide →

← C++ Tutorial

Class Library User's Guide

For the Microsoft® Foundation Class Library

Contents

Appendixes

Figures and Tables

Figures

Figures

Tables

Introduction

This manual contains a tutorial for the Microsoft Foundation Class Library. The class library is a set of C++ classes that encapsulate the functionality of applications written for the Microsoft® Windows™ operating system.

Among the features covered in Chapters 1 through 10 of the tutorial are the following:

- Using AppWizard to create a skeletal starter application upon which to build your program.
- Implementing a document class to manage your application's data and to write to and read from files.
- Implementing a "view" class to display your document and manage all user interaction with it.
- Using App Studio to create and edit your application's resources.
- Using ClassWizard to connect user-interface objects, such as menu items, buttons, and accelerator keys, to handler functions in your source code.
- Using App Studio to create dialog-template resources and ClassWizard to encapsulate those resources in dialog classes. Also, arranging for automatic transfer of data between the controls in a dialog box and member variables of the dialog class and for automatic validation of data that the user enters.
- Implementing scrolling and splitter windows, enhancing the default printing capabilities, and adding context-sensitive help to your application.

Chapters 11 through 18 explore additional features of programming with the Microsoft Foundation Class Library:

- Using the diagnostic and exception-handling facilities.
- Working with files and the "serialization" of objects to and from persistent storage.
- Using the collection classes: arrays, lists, and maps.
- Using VBX controls, compatible with Microsoft Visual Basic™.
- Using Object Linking and Embedding (OLE).

Document Conventions

This book uses the following typographic conventions:

Example	Description
STDIO.H	Uppercase letters indicate filenames, segment names, registers, and terms used at the operating-system command level.
char, **_setcolor**, **__far**	Bold type indicates C and C++ keywords, operators, language-specific characters, and library routines. Within discussions of syntax, bold type indicates that the text must be entered exactly as shown.
	Many functions and constants begin with either a single or double underscore. These are part of the name and are mandatory. For example, to have the **__cplusplus** manifest constant be recognized by the compiler, you must enter the leading double underscore.
expression	Words in italics indicate placeholders for information you must supply, such as a filename. Italic type is also used occasionally for emphasis in the text.
[[*option*]]	Items inside double square brackets are optional.
#pragma pack {**1** | **2**}	Braces and a vertical bar indicate a choice among two or more items. You must choose one of these items unless double square brackets ([[ ]]) surround the braces.
`#include <io.h>`	This font is used for examples, user input, program output, and error messages in text.
CL [[*option...*]] *file...*	Three dots (an ellipsis) following an item indicate that more items having the same form may appear.

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

Example	Description
CTRL+ENTER	Small capital letters are used to indicate the names of keys on the keyboard. When you see a plus sign (+) between two key names, you should hold down the first key while pressing the second. The carriage-return key, sometimes marked as a bent arrow on the keyboard, is called ENTER.
"argument"	Quotation marks enclose a new term the first time it is defined in text.
`"C string"`	Some C constructs, such as strings, require quotation marks. Quotation marks required by the language have the form `" "` and `' '` rather than " " and ' '.
Color Graphics Adapter (CGA)	The first time an acronym is used, it is usually spelled out.
▸	Lines of example code to add to your program are marked with this symbol.

CHAPTER 1

Introducing the Class Library

In this tutorial, you'll learn about the class library features and functions by creating an application for Microsoft Windows called Scribble.

In this Chapter

The chapter begins with an overview of the Microsoft Foundation Class Library and its supporting tools. It also covers the following topics:

- Designing programs with the class library
- Using the tutorial
- Working with the incremental versions of the tutorial program

In Chapters to Come

Use the first ten chapters as an introduction to the Microsoft Foundation Classes or as a step-by-step tour through the fundamentals of writing applications for Windows with the class library. Use the remaining chapters to deepen your knowledge and to learn specific programming techniques for such topics as:

- Managing memory
- Deriving classes from the root base class, **CObject**
- Accessing run-time information about the class objects in your application
- Using collection classes for aggregates of data
- "Serializing" files to and from disk
- Using the facilities for diagnostics and robustness
- Handling exceptions
- Importing custom controls from Microsoft Visual Basic
- Programming for Object Linking and Embedding (OLE)

Use Appendix A for a survey of new features in the Microsoft Foundation Class Library, general instructions for installing the class library, and the locations of

sample programs and technical notes. Use Appendix B for information about the prebuilt libraries for the class library and about building other versions.

What You Need to Know

The tutorial assumes that you have some knowledge of C++ and of programming for Windows. If you've already programmed with the Microsoft Foundation Class Library, you should find the transition straightforward. For information about migrating from version 1 of the class library to version 2, see Technical Note 19 in MFCNOTES.HLP. However, if you've programmed with Windows in C, or with another C++ environment for Windows, you may find a different program structure to get used to. But inside that structure much of what you do resembles what you've done before.

You may wish to consult texts on C++ and on Windows. Microsoft Visual C++™ includes the *C++ Tutorial*, and Microsoft Press® publishes Kaare Christian's *Microsoft Guide to C++ Programming* and Charles Petzold's *Programming Windows 3.1* (third edition). Another good C++ text is Stanley B. Lippman's *C++ Primer* (Addison-Wesley, second edition).

With that base of knowledge, you should be able to follow the Microsoft Foundation Class Library tutorial, especially if you have some C-language background.

Beyond those fundamental texts, available shortly after the release of Visual C++ from Microsoft Press, is David J. Kruglinski's *Microsoft Visual C/C++ Programming for Windows*. That book supplements the tutorial in this manual and in some areas provides deeper coverage of programming with the Microsoft Foundation Class Library.

Guide to Related Documentation

For a quick overview of the process of using Microsoft Visual C++, see *Presenting Visual C++: In Our Own Words.*

For an overview of the class library, see Chapters 1 through 6 in the *Class Library Reference.*

For reference information about the classes, global functions, and macros that make up the class library, see the alphabetic reference in the *Class Library Reference.*

Programmers using the Microsoft Foundation Class Library rely on several key tools. AppWizard creates a starter application. App Studio lets you construct your user interface and edit resources. ClassWizard helps you connect user-interface objects such as menus to code.

- For information about using AppWizard, see Chapter 2 in this manual and Chapter 13 in the *Visual Workbench User's Guide*.
- For information about using App Studio, see the *App Studio User's Guide*.
- For information about using ClassWizard, see Chapters 6 and 7 in this manual, Chapter 9 in the *App Studio User's Guide*, and Chapter 13 in the *Visual Workbench User's Guide*.

For information about using the Visual C++ programming environment, see the *Visual Workbench User's Guide*.

For C++ and C run-time library reference information, see the *C++ Language Reference*, the *C Language Reference*, and the *Run-Time Library Reference*.

The Class Library

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

For a summary of what's new in the class library since version 1, see Appendix A.

In the tutorial, you'll learn to:

- Quickly construct a user interface that follows the Windows *Design Guide*.
- Easily implement both single document interface (SDI) and multiple document interface (MDI) applications.
- Implement features that until now were considered difficult or tedious, such as printing, toolbars, scrolling, splitter windows, print preview, and context-sensitive help.
- Take advantage of the many built-in or reusable components of the class library.

This section lists the main features of the class library and introduces the application framework and its associated visually-oriented programming tools: App Studio, AppWizard, and ClassWizard.

Class Library Features

The Microsoft Foundation Class Library has the following features:

- Powerful, visual tools that simplify programming:
  - AppWizard, a tool for creating a skeleton starter application on which to build your application-specific functionality.
  - App Studio, a tool for constructing your user interface and editing your resources.
  - ClassWizard, a tool for connecting Windows messages and user-interface objects such as menus to code.
- A flexible program framework based on document objects that manage data, and view objects that manage user interaction with their documents.
- Classes that encapsulate the Windows API: windows, dialog boxes, device contexts, GDI drawing objects, controls, and more.
- Implementations for standard menus, including the following commands and their associated standard dialog boxes:
  - File menu: New, Open, Save, Save As, Print, Print Setup, Print Preview, Exit commands. Also support for the most-recently-used list of files. Open, Save, and Save As are supported by the framework's standard serialization mechanism.
  - Edit menu: Paste, Paste Link, and Links commands for OLE. Also support for OLE verbs.
  - View menu: Commands to toggle the display of a toolbar and status bar.
  - Window menu: New Window, Tile, Cascade, Hide, Show commands for multiple document interface (MDI) applications.
  - Help menu: Standard Index and Using Help commands.
- Support for routing commands from user-interface objects, such as menu items and toolbar buttons, to command handlers.
- Support for toolbars and status bars.
- Support for scroller and splitter windows.
- Support for forms and text editing windows.
- Support for printing and print preview.
- A simple program interface for initializing, validating, and accessing the data in dialog-box controls.
- VBX custom controls, giving access to controls created for Visual Basic.
- Support for Object Linking and Embedding (OLE) integrated with the document/view architecture and support for OLE user interfaces.
- Support for context-sensitive help.

- Diagnostic facilities and exception handling.
- Collection classes for managing aggregates of data.
- File classes.
- Compatibility with version 1 of the Microsoft Foundation Class Library.

The tutorial demonstrates many of these features, and the Visual C++ environment makes them easy and productive to use. The features are embodied in the framework and the tools.

The Framework

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

Note In the documentation, you'll see the terms "application framework" and "framework" used interchangeably. The classes that make up the framework are listed and explained in the *Class Library Reference*. Chapters 1 through 6 of the *Class Library Reference* explain how the framework works.

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

Key Concepts

The following are central concepts in the application framework:

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

  The application object manages a list of documents and dispatches commands to other objects in the program.

- The unit of data that the user works with is a document.

  The document maintains, loads, and stores its data.

- The user interacts with a document through a "view" on the document.

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

Figure 1.1 shows the relationship between a document and its view.

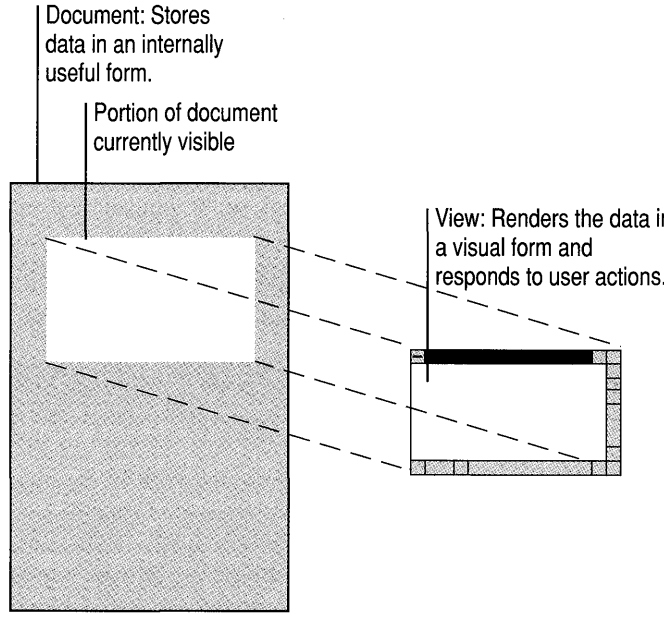


Figure 1.1 Document and View

Working with the Framework

Your main tasks in using the framework are:

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

The general process is to use:

- AppWizard to create the files for a skeleton starter application.

  The classes in these files are derived from classes in the class library. The files contain, in particular, a document class, a view class, a frame window class, and an application class. AppWizard also creates an initial resource file and other supporting files.
- App Studio to construct the user interface by creating and editing resources.
- ClassWizard to connect user-interface objects to message-handler functions.
- The Microsoft Visual Workbench editor and browser to implement the message-handlers — such as for menu commands.
- The Visual Workbench editor also to edit all of your classes and to control the development cycle.
- ClassWizard to define automatic processing of dialog data.

The tools described above are summarized in Chapter 13 of the *Visual Workbench User's Guide*.

The Partnership

Your work with the Microsoft Foundation Class Library is a partnership. The source code you add is your part. This includes code to:

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

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

- Using App Studio to create and edit resources that define the user-interface elements of your program.

- Optionally, editing the Rich-Text Format (RTF) files containing help topics for context-sensitive help.

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

Benefits

The Microsoft Foundation Class Library provides a thorough foundation that allows you to spend most of your programming effort writing the code that handles your data rather than reinventing the graphical user interface. For more information about the application framework, see Chapters 2 through 6 in the *Class Library Reference*.

Figure 1.2 shows schematically how your code fits into the framework.

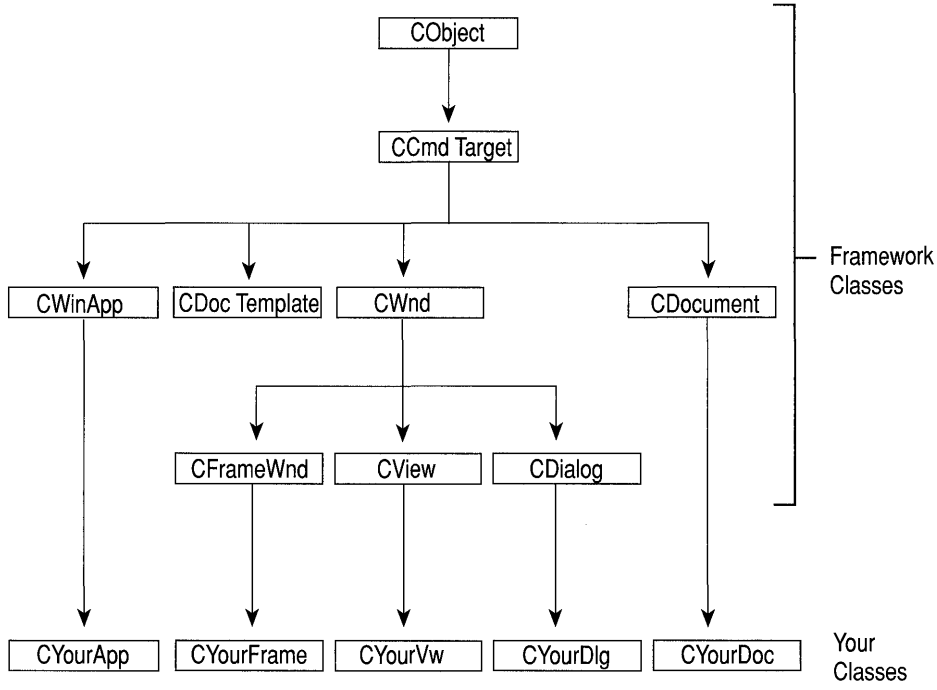


Figure 1.2 Your Code in the Application Framework

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

Introducing Scribble

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

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

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

Figure 1.3 shows what Scribble looks like on the screen.

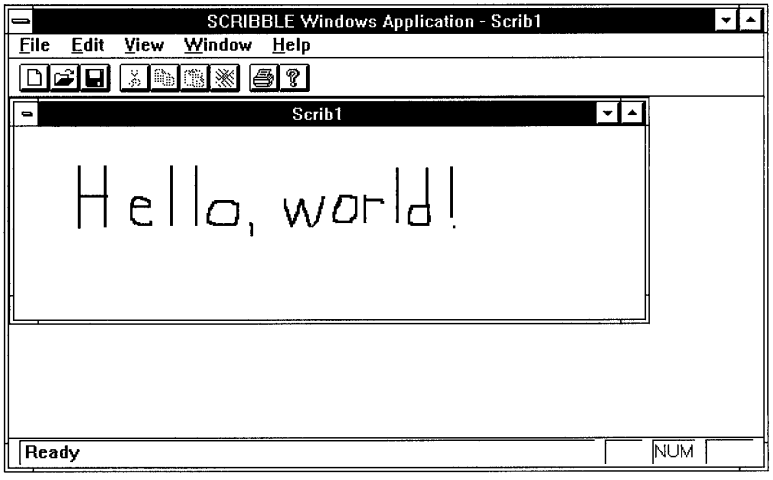


Figure 1.3 Scribble in Action

Using the Tutorial

Some readers may wish to read the tutorial and study the supplied source code before venturing on a first framework application of their own. Others may wish to learn by adding code as they go. The tutorial can be used in either way. For details on procedure, see "Read Along" on page 12 or "Work Along" on page 12.

Important You must complete a chapter of the tutorial before the version of Scribble developed in that chapter will run.

Tutorial Conventions

The tutorial uses the following textual conventions:

- The names of all classes, functions, macros, and other items supplied as part of the Microsoft Foundation Class Library appear in boldface print. For example:

  CDocument

  CDocument::OnOpenDocument

  DECLARE_MESSAGE_MAP

- The names of all classes, functions, and other items developed as part of the tutorial — and not supplied by the library — appear in monotype. For example:

  ```
  CScribDoc
  ```

  ```
  CScribDoc::DeleteContents
  ```

You'll often see these conventions used to distinguish items from the two domains.

Project Makefiles and STEP Directories for Scribble

Source code files, project files (also known as makefiles), and other necessary files for the tutorial are supplied in a group of subdirectories under the MFC\SAMPLES\SCRIBBLE directory. The tutorial develops the Scribble application in seven steps; there are six subdirectories, named STEP0 through STEP5, representing the first six steps. Each subdirectory contains the files needed for one step. For convenience, Table 1.1 correlates chapters, steps, and chapter content. The second column gives the step completed by the end of the corresponding chapter. Each chapter begins where you left off in the previous step.

Table 1.1 Tutorial Steps

Chapter	Step Completed	Content
2	0	Starter application (AppWizard)
3		Scribble's document.
4	1	Scribble's view
5		Menus and toolbar (App Studio)
6	2	Handlers for commands (ClassWizard)
7	3	Dialog boxes (App Studio, ClassWizard)
8	4	Scrolling and splitting
9	5	Printing and print preview
10	6	Context-sensitive help

For each version of Scribble, the project file, called SCRIBBLE.MAK, is stored in the appropriate subdirectory for the step. Use Table 1.1 to locate the right subdirectory for each chapter.

The Files You Work With

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

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

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

For chapters that use App Studio (5–7), you'll work with SCRIBBLE.RC, the application's resource file.

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

Note For Chapter 2, simply follow instructions to create the skeleton starter application with AppWizard. You can do so easily even if you aren't planning to add the tutorial code yourself, and it's a good way to learn to use this tool.

Descriptions of the two alternative styles of using the tutorial follow.

Read Along

Use the following procedure if you don't want to type in the code:

▶ **To read along without typing code**

1. Print out source files from the STEP*n* directory appropriate for the chapter.

   For example, in Chapter 3, print files for step 1 (see Table 1.1 on page 11).

   You can also examine the files by opening them with the Visual Workbench editor.

   For comparison, you can look at the code for the previous step to see what it looked like before the current chapter's additions.

2. As you read the chapter, examine the code that is added, replaced, or deleted in the chapter.

   The chapters surround the pieces of code they discuss with enough context to help you locate them in the files.

Work Along

Probably the most effective way to use the tutorial is to work along:

Work in your own subdirectory. You'll create this subdirectory — call it MYSCRIB — in Chapter 2 by running AppWizard. You might, for example, make MYSCRIB a subdirectory of MFC\SAMPLES\SCRIBBLE. As you work along, you can compare your files in MYSCRIB with the files for the step you're doing in the appropriate STEP*n* subdirectory.

As you work on the code for each chapter, use the next two procedures. The first procedure explains how to use the Visual C++ project file for each Scribble step. The second procedure explains how to work with the Visual Workbench editor to add the tutorial code.

▶ **To open the Visual C++ project for a Scribble step**

1. Start Visual Workbench.
2. Choose the Open command on the Project menu.
3. Move to your MYSCRIB subdirectory and select the project file: SCRIBBLE.MAK. Choose the OK button.
4. Open and edit files as needed, using the File menu and the editor in Visual Workbench. For more information about opening, editing, and saving files with Visual Workbench, see the *Visual Workbench User's Guide*.

▶ **To work along, typing in the code yourself**

1. Begin with the code files you developed in the previous step.

The idea is to begin with the code as it exists at the end of the step for the previous chapter and add the new chapter's code to it.

For example, as you begin Chapter 3, your files should match those in the STEP0 directory. When you finish the chapter, your code should be essentially identical to that in the STEP1 directory.

2. As you read the chapter, type in all code as it appears in the text.

   Be sure to type in all code exactly as it appears, unless you're instructed not to. Code to be added is marked in the left margin of the tutorial with the special symbol ▸. For example, you might be instructed to type the marked lines in the following function:

```
// OnOpenDocument, then ...

CStroke* CScribDoc::NewStroke( )
{
   CStroke* pStrokeItem = new CStroke( m_nPenWidth );
   m_StrokeList.AddTail( pStrokeItem );
   return pStrokeItem;
}
```

   Don't add code that isn't marked.

   The comment above the function supplies contextual information to help you see where to put the new lines. You'll be told which file to add the lines to.

   Sometimes the discussion following a piece of code explains that some lines were deleted or replaced rather than added. As part of the development of Scribble, observe why these changes occurred.

3. Also perform all App Studio and ClassWizard steps.

   In chapters that take you through the steps of using one of these tools, invoke the tool as instructed and follow all steps. Compare the resulting changes to your source code with the illustrative code in the text. Also, for example, compare the dialog box you create with the one shown in the tutorial.

   These tools are an integral part of programming with the Microsoft Foundation Class Library.

4. Complete the chapter (in most cases) by compiling the code you worked on as directed in Chapter 2 of the *Visual Workbench User's Guide*.

Note This procedure also lets you selectively work on a step. You can begin with the previous step for any chapter. For example, to do your first work at step 2 (Chapters 5 and 6), begin with the code for step 1, whether you did step 1 yourself or not.

Each chapter begins with a short reminder of these procedures.

Scribble Build Information

This section explains a few things you'll need to know when you prepare to build Scribble. General procedures for compiling and linking framework programs in Visual C++ and running the executable program under Windows are given in Chapter 2 of the *Visual Workbench User's Guide*.

The Right Directory

If you're simply reading along with the tutorial without adding code, you can still compile Scribble at each step to see what it looks like and how it behaves. In this case, go to the STEP subdirectory for the step indicated at the beginning of a chapter. For example, in Chapter 4, go to the STEP1 subdirectory. Open the project from the Visual Workbench Project Open dialog box by double-clicking the file SCRIBBLE.MAK.

If you're working along, adding code as you read, compile the project in which you've been editing the files. You should already be in your MYSCRIB subdirectory and have the project open.

Setting Options

For Scribble, you'll normally want to use the default debug-mode setting. However:

▶ **To select debug or release build options**

1. From the Options menu in Visual Workbench, choose Project.
2. In the Build Mode group, choose either the Debug or Release radio button.
3. Make sure the check box labeled "Use Foundation Classes" is checked.
4. Choose OK.

Recall that the project type for Microsoft Foundation Class Library programs is "Windows Application." You can verify this in the Project Options dialog box. Figure 1.4 shows the Project Options dialog box.

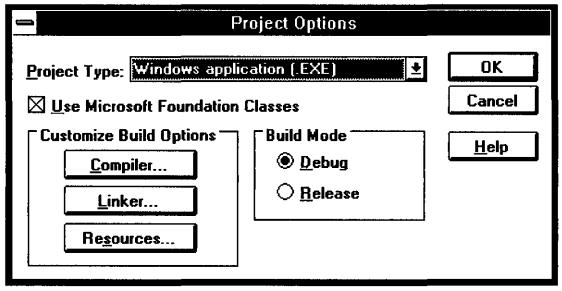


Figure 1.4 The Project Options Dialog Box

Once you've moved to the right directory and set your options (if needed), you're ready to build Scribble. Choose one of the Build commands on the Project menu in Visual Workbench.

Chapter 2 begins the tutorial proper. You'll create a skeleton application with AppWizard. In later chapters, you'll build a more powerful Scribble application upon that skeleton.

CHAPTER 2

Creating a New Application With AppWizard

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

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

To create a new Visual C++ project based on the Microsoft Foundation Class Library, you'll choose the AppWizard command from the Project menu in Visual Workbench.

AppWizard speeds your work in beginning a new project. In seconds it creates a set of Visual C++ files that declare skeletal versions of the classes that make up your application. Key parts of the code that implements these classes are supplied by AppWizard, based on the options you choose in the AppWizard dialog box. These files comprise a starter application that you can compile and run immediately. The starter application contains all the code required to display the windows in which users will interact with your application.

Once you've created the starter application with AppWizard, you'll complete the rest of the steps listed above using App Studio to construct the menus and other user-interface objects, ClassWizard to make connections between those objects and the code you write to respond to them, and Visual Workbench to edit, compile, browse, and debug your source-code files.

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

This chapter shows you how to create a set of starter files for the Scribble application that is developed throughout the tutorial. These files contain skeletal

code for several C++ classes—an "application class," a "document class," a "view class," and a "frame window class." The concepts behind these classes are discussed fully in Chapter 2, "Using the Classes to Write Applications for Windows," in the *Class Library Reference*. You'll also learn more about them in Chapters 3 and 4 of this manual. Details about the created files are available in a text file, README.TXT, that is created along with the starter files. The contents of the starter files are discussed in later chapters as needed. For additional information about the starter files, see Chapter 13 in the *Visual Workbench User's Guide*.

Without adding a line of code, you can compile the starter application you created with AppWizard and run the resulting program, which exhibits much of the standard functionality you expect from a program written for the Windows operating system. The steps needed to compile and run the program are given in the section "Compile the Starter Application" on page 23 and "Run the Starter Application" on page 24.

Chapters 3 and 4 show you how to add the application-specific code for Scribble, the sample application developed in the rest of the tutorial. Chapter 5 shows you how to construct Scribble's user interface with App Studio. From Chapter 6 and the chapters that follow, you'll iteratively add more features to Scribble, then test, revisit App Studio, and so on.

This chapter covers step 0 of the tutorial. If you're working along, adding code as you read, follow all directions in this chapter. When you finish, you'll have a full set of starter files in your own subdirectory. On the other hand, if you're reading along without adding any code, it's still a good idea to work through this chapter to familiarize yourself with AppWizard. If you prefer, however, you can simply study the set of files in the MFC\SAMPLES\SCRIBBLE\STEP0 subdirectory, which are identical to those created by AppWizard .

Create the Starter Application for Scribble

This section shows you how to use AppWizard to create the starter application that forms the beginnings of Scribble. AppWizard lets you specify a number of options. Then it creates a set of source-code files based on these options from which you develop your application. This saves a great deal of time and effort and lets you focus on the application-specific parts of your program.

The procedure described for Scribble applies equally to your own applications. Just change the names and other values as needed.

Figures 2.1 through 2.4 show AppWizard's dialog boxes with the correct values entered for Scribble. The following procedure describes how to enter these values.

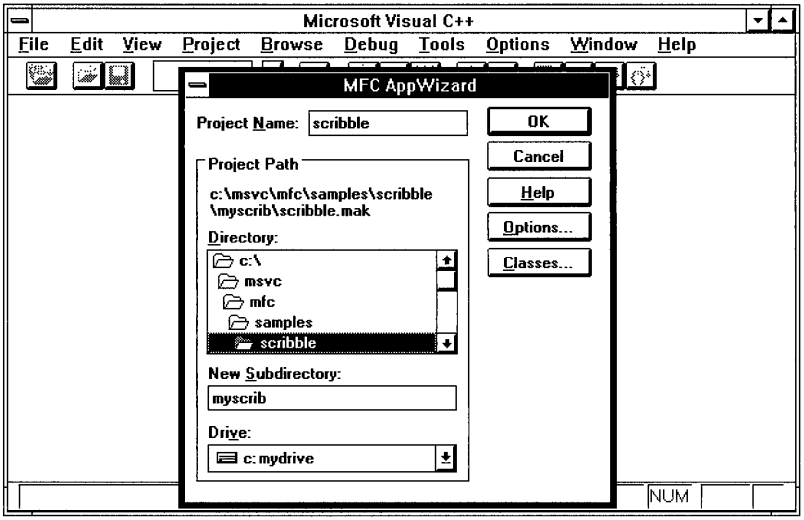


Figure 2.1 AppWizard Dialog Box

▶ To create starter files for Scribble

1. Start Visual Workbench by double-clicking its icon in the Windows Program Manager.
2. On the Project menu in Visual Workbench, choose the AppWizard command.

   Complete the AppWizard dialog box as described below. You can also run AppWizard from Program Manager.
3. In the Project Name text box, type **scribble**

   The application's project file will be given this name: in this case, SCRIBBLE.MAK.
4. In the New Subdirectory text box, delete "scribble" and type **myscrib**

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

   AppWizard will create this directory if it doesn't exist. For Scribble, MYSCRIB is a new directory.
5. Specify the path to the project's subdirectory.

   Use the list box provided to navigate through the directories on the selected drive.

   As you navigate through the directory structure, the path listed in the dialog box changes to show where the named subdirectory (MYSCRIB) should be placed. When the path suits you, stop navigating.

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

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

AppWizard creates a new MYSCRIB subdirectory in the SCRIBBLE directory.

6. Choose the Classes button to check and modify class names and filenames.

   The Classes dialog box lets you edit the names of your program's classes and the files they're in. The dialog box is shown for two different classes in Figures 2.2 and 2.3.

   For Scribble, some of the class names must be changed from the defaults that AppWizard suggests. To edit the information for a class, select the class name in the drop-down list box at the top of the Classes dialog box.

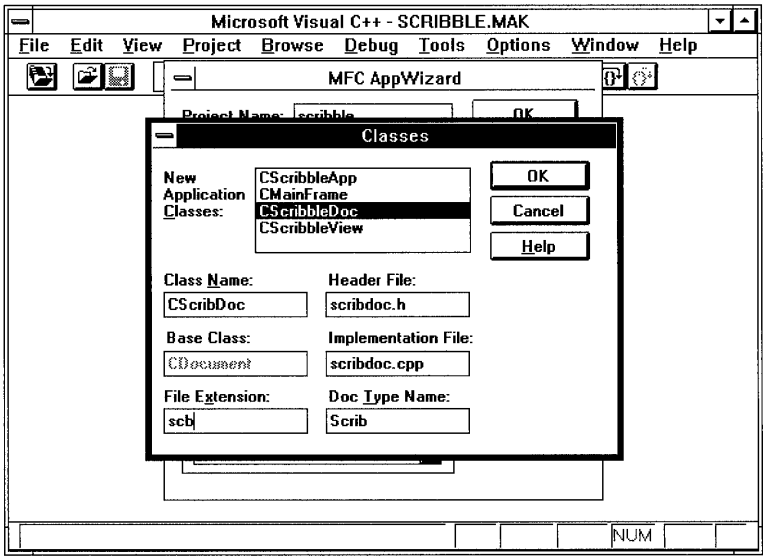


Figure 2.2 The Document Class in the Classes Dialog Box

7. Select class "CScribbleDoc" and edit its information as follows:
   - Change the class name from "CScribbleDoc" to "CScribDoc."
   - Type the file extension **scb** (don't supply a period).

     This is the extension that is appended by default to the names of files that the user saves with Scribble.
   - Change the Document Type Name from "Scribb" to "Scrib."

     These are the characters (up to 6) used wherever Scribble's native document type is referred to. For example, the document type name is used to name the

default file, SCRIB1.SCB, the Windows shell registration name ("Scrib"), and constants referred to in the code created by AppWizard, such as **IDR_SCRIBTYPE**.

Figure 2.3 shows the dialog box as it appears with the document class selected.

When you select a class, the available text boxes in the area at the bottom of the dialog box change to reflect the nature of the chosen class. For all classes, these extra text boxes include Class Name, Base Class, Header File, and Implementation File. For document classes, additional boxes specify a document File Extension and a Document Type Name.

You can edit any box whose contents are not dimmed (grayed).

The changes you type will make using the tutorial easier by keeping the names of your classes and files synchronized with those in the subdirectories provided for the Scribble steps. What you see on the screen as you work will also match the figures in the tutorial.

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

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

   Figure 2.3 shows the dialog box as it appears with the view class selected.

9. Choose the OK button in the Classes dialog box when you finish.

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

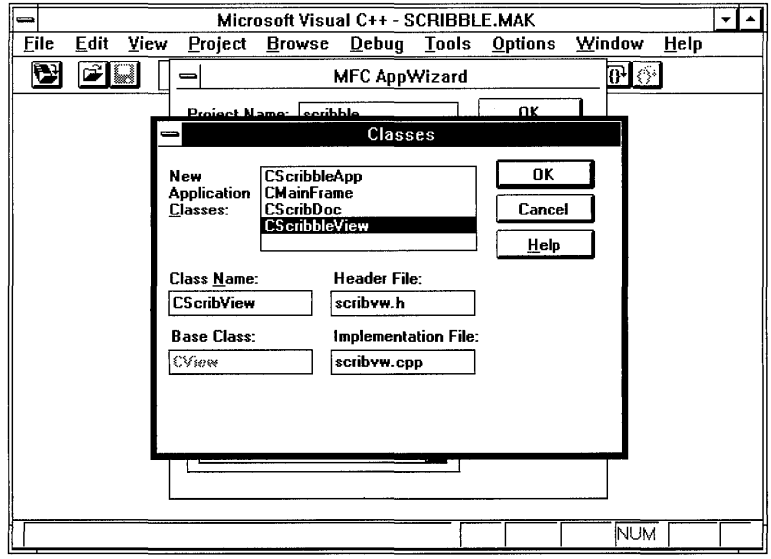


Figure 2.3 The View Class in the Classes Dialog Box

10. Accept the default options.

    Scribble uses AppWizard's default options. To accept these options, do nothing.

    With these options, AppWizard will add code for printing and print preview and to support a toolbar, and it will supply comments throughout the files it creates to help you understand where you need to add your own code. By default, the source code will also support the multiple document interface (MDI).

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

    If you want to see what options are available, you can examine options by choosing the Options button. Figure 2.4 shows the Options dialog box. For more information about the options, see Chapter 4 in the *Visual Workbench User's Guide*.

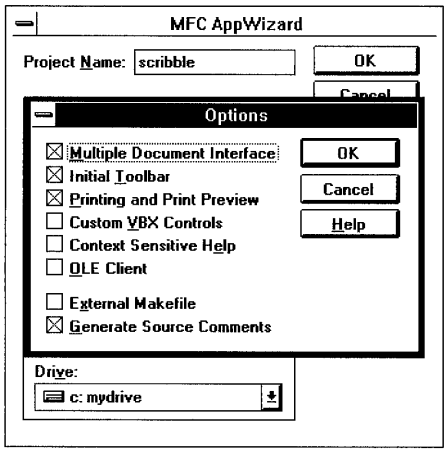


Figure 2.4 AppWizard Options Dialog Box

11. Choose the OK button in the AppWizard dialog box.

    AppWizard creates the specified subdirectory if it doesn't exist. Then it creates all necessary files in the directory. It then opens the project in Visual Workbench.

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

Compile the Starter Files

The next two sections show you what the new application files do when compiled—without adding a single line of code.

The starter application you created provides the skeleton of a working application for the Windows operating system. When you compile the starter files—without adding a thing—the result is an application that runs, opens and closes windows, and lets you perform other operations on the windows. Of course, at this stage the windows have nothing in them. So far, Scribble doesn't scribble.

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

▶ To compile the starter application

1. Make sure you've set up the environment as explained in Appendix A.
2. Build the application in Visual Workbench.

   On the Project menu, choose Build or Rebuild All as shown in Figure 2.5.
3. The starter application is built, producing the file SCRIBBLE.EXE in your new MYSCRIB subdirectory.

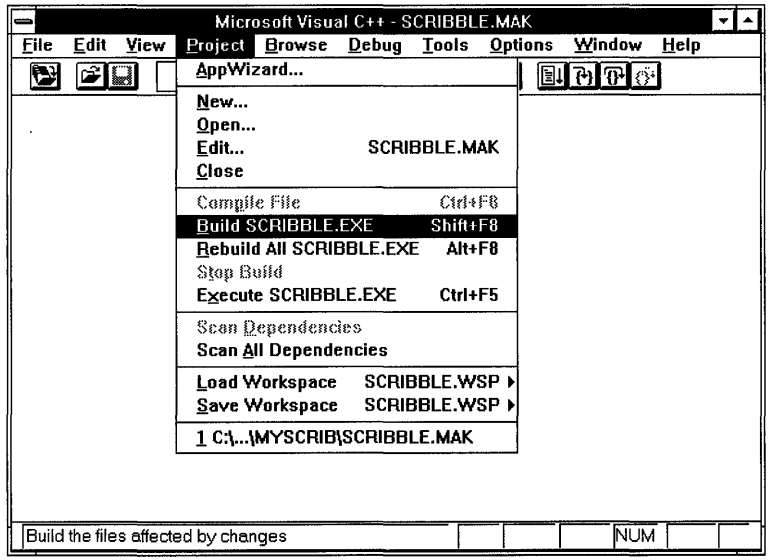


Figure 2.5 Compiling in Visual Workbench

Run the Starter Application

After you compile the starter application, you can run it from Visual Workbench, either with breakpoints set or not. You can also run the application from the Windows Program Manager.

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

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

▶ **To run Scribble without debugging**

- From the Project menu, choose the Execute Target command.

When the starter application runs, an MDI application window appears with a menu bar containing File, Edit, View, Window, and Help menus and a default toolbar. The application window contains one open document window, as shown in Figure 2.6.

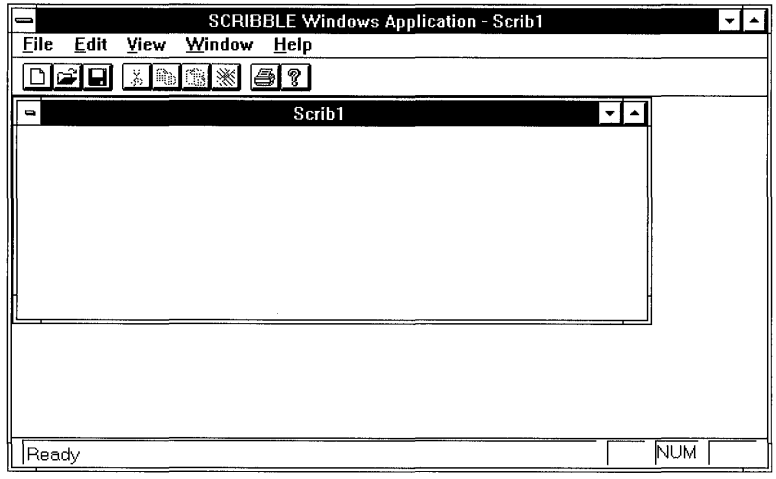


Figure 2.6 The Starter Application

The document window is empty, of course, because you've added no application-specific code yet. But you can move, resize, minimize, maximize, and close the document window and the application window. You can also use the New command on the File menu to open new windows. The Open, Save, and Save As commands are partially functional: at this point, they save empty files. You haven't added all of the code yet to support these commands. The About command on the Help menu brings up an About dialog box with default text in it. The default toolbar is partially functional too: the Open and Save/Save As buttons do the same things as the

corresponding menus. And the status bar at the bottom of the application window displays a description string when you move the mouse pointer over any menu command.

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

You'll undoubtedly want to examine the source code files you created. To orient you, AppWizard also creates a text file, README.TXT, in your new application directory. This file explains the contents and uses of the other new files created by AppWizard.

CHAPTER 3

Creating the Document

In this chapter and the next, you'll add code to the starter application you created in Chapter 2 with AppWizard. By the end of Chapter 4, you can compile and run the Scribble program.

Figure 3.1 shows what the Scribble application developed in the tutorial will look like at the end of Chapter 4.

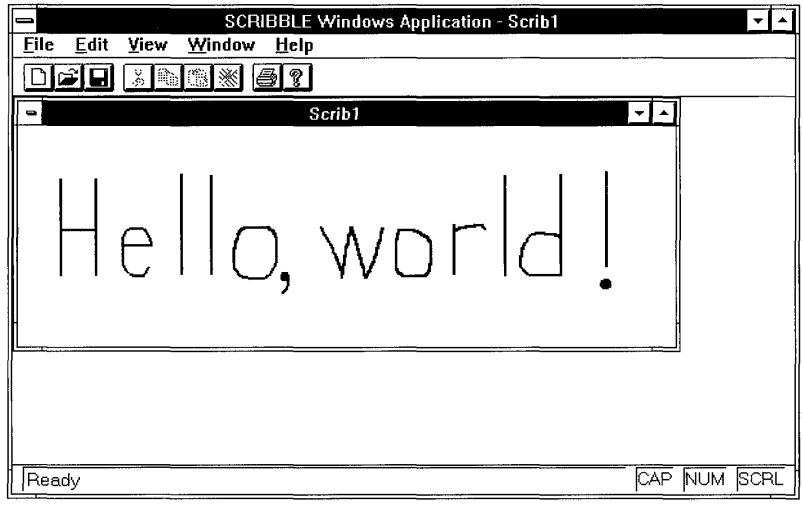


Figure 3.1 Scribble in Action

This chapter introduces documents and develops Scribble's document class, an application-specific class derived from class **CDocument**, called `CScribDoc`. Chapter 4 introduces views and develops the view class. The two chapters together introduce many of the fundamental concepts of the framework: documents, serialization, views, drawing, and messages. Because documents and views are intimately related, you need to implement both before Scribble is fully functional.

In later chapters, you'll incrementally add new features to Scribble: menus, a working toolbar, a dialog box with automatic initialization and validation of its controls, scrolling, splitter windows, enhanced printing, and context-sensitive help.

Your tour of Scribble's code begins with the starter files created by AppWizard in the previous chapter. You'll add a lot of functionality to Scribble with a small amount of code. Among the things you'll develop in this chapter are:

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

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

This chapter and Chapter 4 cover step 1 of Scribble. If you want to work along, adding the code as you go, begin with the files you created with AppWizard in your MYSCRIB subdirectory in Chapter 2. At this point, your files should consist entirely of the set of generic starter files that AppWizard created. As you read this chapter, add or change all lines of code marked with the ▸ symbol in the left margin. At the end of Chapter 4, your files should essentially resemble those in the SCRIBBLE\STEP1 subdirectory.

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

Note If you have trouble locating the correct place to add code, try looking at the corresponding source files in the subdirectory for the completed step. For this chapter and Chapter 4, use the SCRIBBLE\STEP1 subdirectory for this purpose.

If you want to preview Scribble, load the version in SCRIBBLE\STEP1 in Visual Workbench and choose the Build command on the Project menu. Run the program with the Execute Target command on the Project menu.

Documents

At the heart of Scribble are its document and its view. This section explains the role of the document and introduces Scribble's document class and its members.

At run time, an application written with the Microsoft Foundation Class Library is a group of cooperating objects that communicate by sending and receiving Windows messages and by calling each other's member functions. Documents are created by

document template objects and managed by an application object. Users interact with a document through a view object, which is framed by a document frame window object. Figure 3.2 shows graphically the relationships between these key objects.

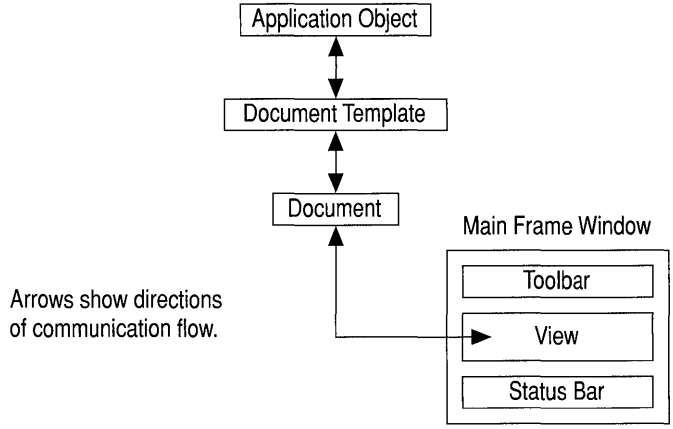


Figure 3.2 Objects in Scribble

Table 3.1 shows how the document and other objects are created and managed in a framework application.

Table 3.1 Key Objects in an Application

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

Definition

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

Figure 3.3 shows the general relationship between a document and its view and frame window.

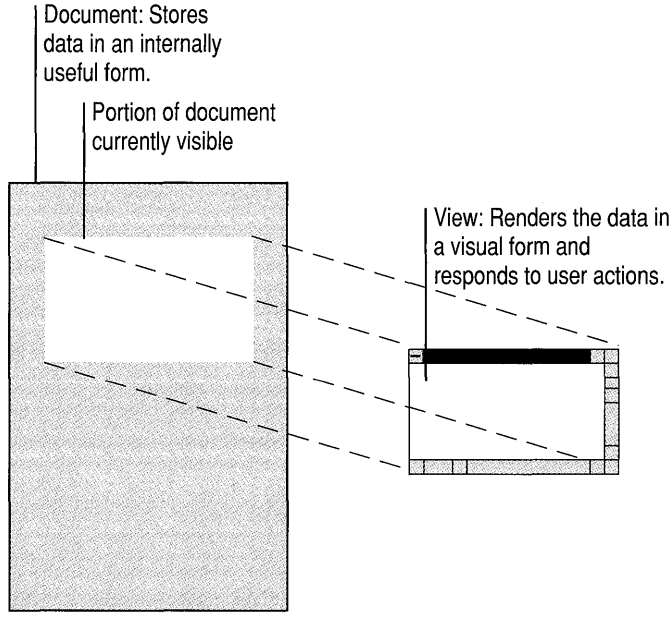


Figure 3.3 Document and View

Documents In the Framework

In the framework, the data and the user's operations on it are managed by two separate objects. A document is an object that stores your data in its member variables and reads and writes it through a member function called `Serialize`. The user interacts with the document through a separate object called a view. The view fills the client area of a frame window, where it displays the data and accepts user input and editing operations. Documents know how to manage data; views know how to display it and accept operations on it. Figure 3.3 shows this important relationship graphically.

Document Creation

When the user opens a document—existing or new—the framework creates a document object and its associated frame window and view objects. If the document is associated with a file, the document reads the file and stores its data. The view obtains data from the document and displays it. Figure 3.4 shows the general process of creating a new document and its view and frame window. For more details, see Chapters 2 and 4 in the *Class Library Reference*.

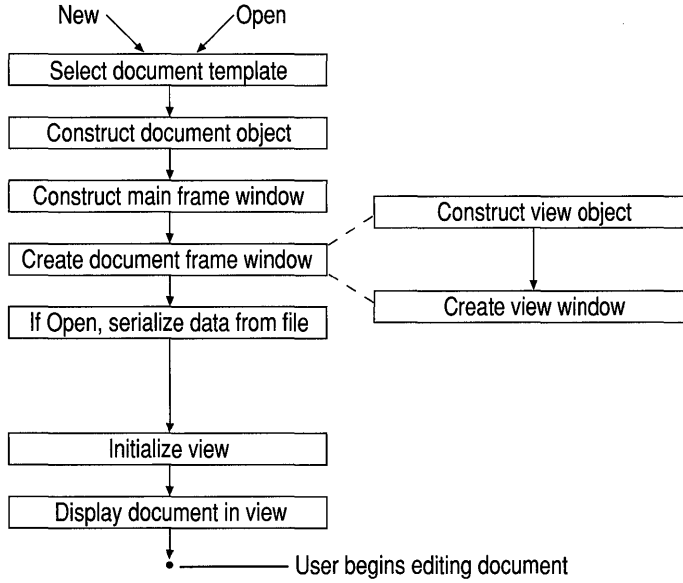


Figure 3.4 Creating a Document

Document/View Interaction

When the user modifies data through the view, the view notifies the document. In turn, the document tells all of its views (some documents have multiple views) to update their displays with the new information, and the views respond by redrawing all or part of the visible portion of the document. You'll learn more about the view's part in this process in Chapter 4.

In the Microsoft Foundation Class Library, documents are based on class **CDocument**. To use **CDocument**, you derive your own document class from it. For more detailed information about documents, see Chapters 2 through 4 and class **CDocument** in the *Class Library Reference*.

You and the Document

Table 3.2 shows your responsibilities and those of the framework in implementing a document.

Table 3.2 Document Implementation Responsibilities

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

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

Scribble's Document: Class CScribDoc

Scribble is a simple drawing program. Documents in Scribble store the lines, or "strokes," that make up a drawing. Because a drawing is typically made up of many strokes, the document stores a list of all the strokes the user has drawn. Figure 3.5 shows a single stroke drawn in a Scribble document's view.

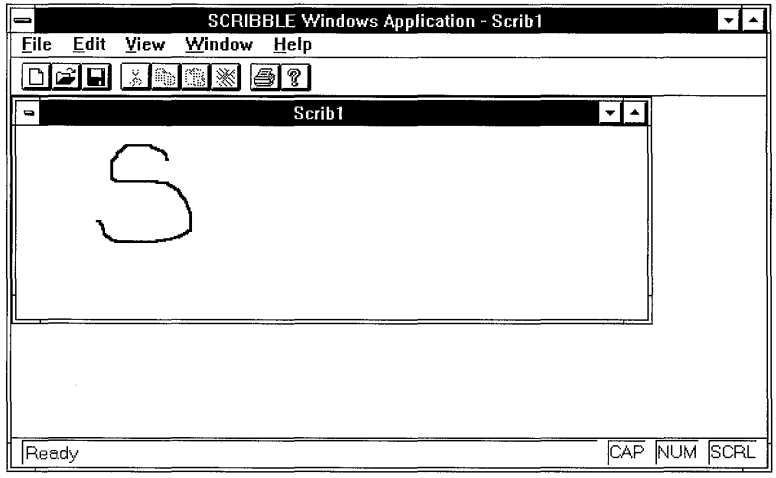


Figure 3.5 One Stroke in Scribble

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

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

▶ To add member declarations to Scribble's document class

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

```
▸  // Forward declaration of data structure class
▸  class CStroke;
   // The document class
   class CScribDoc : public CDocument
   {
   protected: // Create from serialization only
      CScribDoc( );
      DECLARE_DYNCREATE( CScribDoc )

   // Attributes
▸  protected:
▸     CObList    m_strokeList;   // Each element is a CStroke

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

▸     UINT    m_nPenWidth;    // Current user-selected width
▸     CPen    m_penCur;       // Pen created according to
▸                             // user-selected pen style (width)
▸  public:
▸     CPen*   GetCurrentPen( ) { return &m_penCur; }
```

```
// Operations
public:
   void DeleteContents( );
   CStroke* NewStroke( );
   POSITION GetFirstStrokePos( );
   CStroke* GetNextStroke( POSITION& pos );

// Implementation
public:
   virtual ~CScribDoc( );
   virtual void Serialize( CArchive& ar );  // Overridden for
                                            // document i/o
#ifdef _DEBUG
   virtual   void AssertValid( ) const;
   virtual   void Dump( CDumpContext& dc ) const;
#endif
protected:
   void InitDocument( );
   virtual BOOL OnNewDocument( );
   virtual BOOL OnOpenDocument( const char* pszPathName );

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

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

Table 3.3 describes the member variables of class `CScribDoc`.

Table 3.3 CScribDoc Data Members

Member	Description
`m_strokeList`	A list of strokes. Each item in the list is an object of class `CStroke`. The list itself is an object of class **CObList**, one of the collection classes supplied by the class library.
`m_penCur`	A **CPen** object used to do the drawing. Its main attribute is its width. The pen is created when the document is constructed and is used during the creation of new strokes.
`m_nPenWidth`	The current width of the lines drawn by the pen.

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

Table 3.4 describes the member functions.

Table 3.4 CScribDoc Member Functions

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

You'll add code for most of these member functions in later sections of this chapter. You'll learn more about `Serialize` under "Serializing the Data" on page 43. For more information about **AssertValid** and **Dump**, see Chapter 15, "Diagnostics." You won't add code to these functions for Scribble.

The Document's Data: Class CStroke

In Scribble, a stroke consists of an array of points. As the user drags the mouse to draw, Scribble collects points and stores them as part of the current stroke. Points collected from the time the left mouse button is pressed to the time it's released form one stroke of a Scribble drawing. Figure 3.6 shows Scribble's data structure schematically. Scribble uses an object of class **CPen** for drawing.

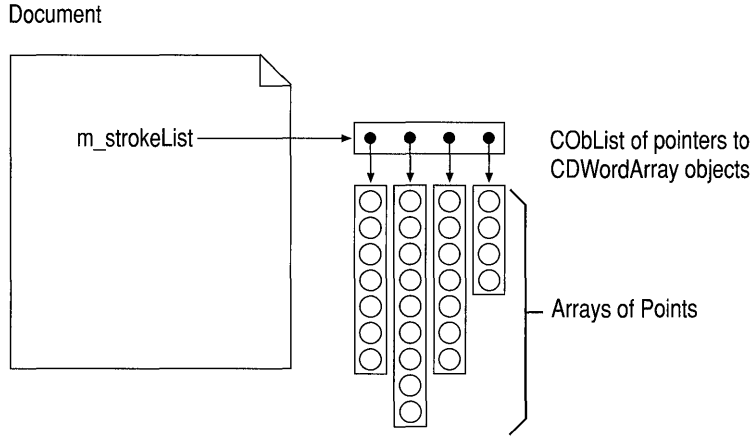


Figure 3.6 Scribble's m_strokeList Data Structure

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

▶ To add the CStroke class

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

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

protected:
   CStroke( );
   DECLARE_SERIAL( CStroke )

// Attributes
   UINT   m_nPenWidth;  // One width applies to entire stroke
   CDWordArray   m_pointArray;  // Series of connected points

// Operations
public:
   void AddPoint( CPoint pt );
   BOOL DrawStroke( CDC* pDC );

// Helper functions
protected:
   CPoint GetPoint( int i ) const
             { return CPoint(m_pointArray[i]); }

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

This code declares a C++ class of stroke objects. The member variables and functions define and manipulate the data of a stroke and serialize it when the document is serialized. You'll add the member function definitions in the next section.

Table 3.5 lists `CStroke`'s member variables.

Table 3.5 CStroke Data Members

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

Table 3.6 lists CStroke's member functions.

Table 3.6 CStroke Member Functions

Member	Description
CStroke	The class defines two constructors, one protected and one public.
AddPoint	Adds a new point to the stroke. The point is represented by a **CPoint** object that defines the coordinates of the mouse during drawing. Class **CPoint** is supplied by the class library.
DrawStroke	When the view object redraws the document's data, it calls upon each stroke object in the stroke list to draw itself by calling its DrawStroke member function.
GetPoint	Returns the **CPoint** object at a given index in the array of points that defines this stroke.
Serialize	To assist the document in making its data persistent, typically on disk, CStroke also overrides the **Serialize** member function of **CObject** to define how a single stroke serializes its points and other data. For more information about point serialization, see "Serializing the Data" on page 43.

Building and Storing Strokes

Your next step is to add definitions for CStroke's member functions.

▶ To add implementation code for the CStroke members

1. Add the following definitions for CStroke's two empty constructors to the SCRIBDOC.CPP file:

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

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

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

2. Add the `AddPoint` member function to the SCRIBDOC.CPP file. The function adds the latest mouse location to the current stroke's list of points:

```
// Constructor declarations, then ...
void CStroke::AddPoint( CPoint pt )
{
   m_pointArray.Add( MAKELONG( pt.x, pt.y ) );
}
```

A `CStroke` object stores its array of points in the `m_pointArray` data member—an object of class **CDWordArray**, one of the collection classes supplied by the class library. A **CDWordArray** stores 32-bit doublewords. A **CPoint** stores its horizontal and vertical coordinates in two 16-bit words, so a **CPoint** fits into a doubleword. The `AddPoint` member function invokes a Windows macro, **MAKELONG**, to convert the coordinates of a point to a doubleword and passes the result to the **Add** member function of class **CDWordArray**. **Add** stores the point as a doubleword at the end of the array.

At this point, class `CStroke` is not quite complete. You'll add code for the two remaining member functions, `GetPoint` and `DrawStroke`, in Chapter 5. These member functions are used by the view object to draw the data.

Managing the Document

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

Initializing and Cleaning Up

Because a document can be created with either the New command or the Open command on the File menu, `CScribDoc` overrides both the **OnNewDocument** and **OnOpenDocument** member functions of **CDocument** to perform necessary document initialization. However, for Scribble, both initializations are the same, so both overrides call the new member function `InitDocument`.

The framework automatically calls `OnNewDocument` when a new document is created or `OnOpenDocument` when a document is opened. AppWizard creates a skeletal version of `OnNewDocument` for you; you must supply `OnOpenDocument` yourself.

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

▶ To implement initialization for Scribble's documents

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

```
// Empty constructor and destructors, then ...
void CScribDoc::InitDocument( )
{
   m_nPenWidth = 2;  // Default 2-pixel pen width
   // Solid black pen
   m_penCur.CreatePen( PS_SOLID, m_nPenWidth, RGB(0,0,0) );
}
```

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

2. Add the following to the override of **OnNewDocument** created by AppWizard in file SCRIBDOC.CPP:

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

3. Finally, add the following override of **OnOpenDocument** to file SCRIBDOC.CPP:

```
// OnNewDocument, then ...
BOOL CScribDoc::OnOpenDocument( const char* pszPathName )
{
   if( !CDocument::OnOpenDocument( pszPathName ) )
      return FALSE;
   InitDocument( );
   return TRUE;
}
```

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

▶ **To implement document cleanup**

- In file SCRIBDOC.CPP, add the following definition for `DeleteContents`, which overrides the **DeleteContents** member function of **CDocument**:

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

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

Scribble's override of `DeleteContents` iterates through the stroke list. For each stroke object, the function invokes the **delete** operator. This destroys the strokes. **RemoveHead**, a member function of class **CObList**, then clears the pointers in the list. Alternatively, this cleanup code could be placed in the destructor, but `DeleteContents` is reused later in other functions.

Managing the Data

As the user works with documents, the document object must manage a pen and its list of strokes in the current drawing. `GetCurrentPen` is a helper function that gives restricted, type-safe public access to the protected data member `m_penCur`. `NewStroke`, `GetFirstStrokePos`, and `GetNextStroke` provide public access to the protected data structure in `m_strokeList`.

▶ **To implement document members for managing Scribble's data**

1. Here's the definition for `GetCurrentPen` (it's already defined inline in file SCRIBDOC.H, so don't add this code):

```
// Declaration of CPen m_penCur, then ...
public:
   CPen*   GetCurrentPen( ) { return &m_penCur; }
```

`GetCurrentPen` simply returns a pointer to the current pen. Here a public member function is used to access the pen instead of making `m_penCur` public.

2. Add the NewStroke member function to file SCRIBDOC.CPP. NewStroke creates a new stroke object and adds it to the stroke list:

```
// Dump member function, then ...
CStroke* CScribDoc::NewStroke( )
{
   CStroke* pStrokeItem = new CStroke(m_nPenWidth);
   m_strokeList.AddTail( pStrokeItem );
   SetModifiedFlag( ); // Mark document as modified
                       // to confirm File Close.
   return pStrokeItem;
}
```

NewStroke uses the C++ **new** operator to construct a new CStroke object dynamically, initializing it with the pen width. It uses the **CObList** member function **AddTail** to add the new stroke to the list. Then it calls the **CDocument** member function **SetModifiedFlag** to flag the changes to the document and returns a pointer to the stroke.

Note that the **new** operator never returns NULL. Instead, an exception is thrown if memory could not be allocated. This would be a good place to implement an exception handler with the **TRY** and **CATCH** macros. For more information about exception handling, see Chapter 16, "Exceptions."

3. Add the following definition for GetFirstStroke (file SCRIBDOC.CPP), which returns the index of the first stroke in m_strokeList:

```
// NewStroke, then ...
POSITION CScribDoc::GetFirstStrokePos( )
{
   return m_strokeList.GetHeadPosition( );
}
```

GetFirstStrokePos calls the **GetHeadPosition** member function of class **CObList** and returns a pointer of type **POSITION** to the first stroke object in the list. **POSITION** is defined in the class library.

4. Finally, add this definition for GetNextStroke (file SCRIBDOC.CPP), which returns the stroke object at a given index in the stroke list:

```
// GetFirstStrokePos, then ...
CStroke* CScribDoc::GetNextStroke( POSITION& pos )
{
   return (CStroke*)m_strokeList.GetNext( pos );
}
```

GetNextStroke calls the **GetNext** member function of **CObList**, which returns a pointer to a **CObject**. The pointer must be cast to a pointer to a CStroke object (CStroke is derived from **CObject**).

The purpose of providing `GetFirstStrokePos` and `GetNextStroke` is to make the stroke list easily accessible in a type-safe manner. These member functions mimic the interfaces of the class library's list classes. This makes it easy to iterate through the list of strokes with clean code like the following (don't add this code to Scribble):

```
POSITION pos = pScribDoc->GetFirstStrokePos( );
while( pos != NULL )
{
   CStroke* pStroke = pScribDoc->GetNextStroke( pos );
   // Do something with the stroke ...
}
```

Serializing the Data

This section adds code to define file input/output for Scribble documents. The default I/O implementation in the Microsoft Foundation Class Library is called "serialization." It provides a mechanism for making a document's data persistent between work sessions with the program. Given the code added here, serialization is automatic when the user chooses the Open, Save, or Save As commands from the File menu.

Note You don't have to write any code to process the Open, Save, and Save As commands on the File menu—such as putting up the dialog boxes. The framework supplies this code.

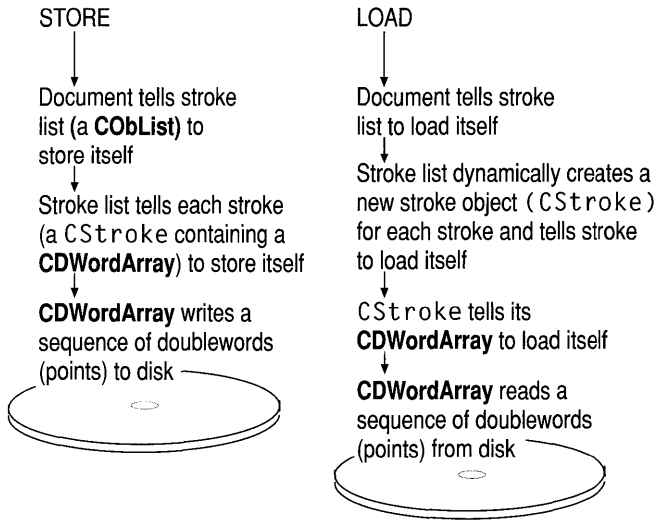


Figure 3.7 Serialization in Scribble

The CScribDoc class declaration in file SCRIBDOC.H begins with the following lines—the lines contain an important macro invocation needed for serialization (don't add this code):

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

AppWizard wrote this code for you.

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

Serializing the Document

Serializing a document occurs in two stages. First, the framework calls the document's Serialize member function. Second, that Serialize function calls the **Serialize** function of the stroke list. If you're working along, add the marked lines to the indicated files.

▶ **To implement serialization for Scribble documents**

- Fill in the Serialize member function for class CScribDoc. The SCRIBDOC.CPP file defines a skeletal version of the function. Here's Serialize, with one marked line added to adapt the serialization mechanism to Scribble:

```
// OnOpenDocument, then ...
void CScribDoc::Serialize( CArchive& ar )
{
   if ( ar.IsStoring( ) )
   {
   }
   else
   {
   }
▶  m_strokeList.Serialize( ar );

}
```

AppWizard creates the skeleton of the `Serialize` member function in class `CScribDoc`; you simply fill in the code shown. Later you'll add code in both branches of the **if** statement. AppWizard also generates comments of the form

```
// TODO: Add storing code here
```

You will typically remove these comments when the functionality has been implemented.

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

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

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

```
CObList m_strokeList;
```

rather than as a pointer, like this:

```
CObList* m_pStrokeList;
```

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

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

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

Serializing Strokes

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

▶ To implement serialization for stroke objects

1. Add the **IMPLEMENT_SERIAL** macro for `CStroke`.

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

   ```
   // End of CScribDoc code in file SCRIBDOC.CPP
   IMPLEMENT_SERIAL( CStroke, CObject, 1 )
   ```

   If you're working along, add the **IMPLEMENT_SERIAL** macro for `CStroke` as shown. The third argument is the schema number, set to 1 for Scribble step 1.

   The **IMPLEMENT_SERIAL** macro complements the **DECLARE_SERIAL** macro invoked in file SCRIBDOC.H. The two macros prepare a class for serialization.

2. Add a `Serialize` override for class `CStroke` in the SCRIBDOC.CPP file. Like `CScribDoc`, `CStroke` also overrides the **Serialize** member function of its base class. When the stroke-list object is called to serialize itself, it calls each stroke object in turn to serialize itself. Here's the code for `CStroke`'s version of `Serialize`:

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

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

If the archive object is for loading, the stroke's data must be read in the same order it was written: first the pen width, then the array of points. The **else** branch of the **if** statement declares a local variable to receive the width, then copies that value to `m_nPenWidth`. It then calls upon the point array to load its data. See Figure 3.7 on page 43.

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

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

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

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

In the Next Chapter

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

At this point, Scribble is about half ready to compile. In Chapter 4, you'll complete the basic Scribble application by developing a view on the document. The view displays strokes and manages all user input. At the end of that chapter, you'll compile and test Scribble.

CHAPTER 4

Creating the View

In Chapter 3, you completed Scribble's document class. In this chapter, you'll add a view class that provides a "view on the document." Scribble's view class displays the strokes of a drawing and accepts user input from the mouse. By the end of this chapter, you can compile and run Scribble.

Among the things you'll develop in this chapter are:

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

You'll also get your first hands-on experience with ClassWizard. ClassWizard lets you map Windows messages to message-handler member functions in your classes. As you'll see in Chapter 6, it also lets you map the commands generated by user-interface objects such as menu items, toolbar buttons, and accelerator keys to message-handler functions.

The code that you must add to fill out the framework in this chapter is mainly in the following files: SCRIBVW.H and SCRIBVW.CPP. You'll also add two more member functions to class `CStroke` in SCRIBDOC.CPP.

This chapter and Chapter 3 cover step 1 of Scribble. If you want to work along, adding the code as you go, begin with the files you worked on in Chapter 3 in your MYSCRIB subdirectory. At this point, your files should consist of the starter files you created with AppWizard in Chapter 2 and modified in Chapter 3. As you read this chapter, add all lines of code marked with the ▸ symbol in the left margin. At the end of this chapter, your files should closely resemble those in the SCRIBBLE\STEP1 subdirectory.

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

Instructions for compiling Scribble are given at the end of this chapter.

Views

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

As you saw in Chapter 3, when a new document is created in response to a New or Open command from the File menu, the framework also creates a "document frame window" and creates a view inside the frame window's client area as a child window. The view displays the document's data and responds to mouse actions, keystrokes, menu commands, and other actions as the user works on the document. It's your task to specify how the view draws your application-specific data and what it does in response to user actions.

Figure 4.1 illustrates the view's role in relation to the document.

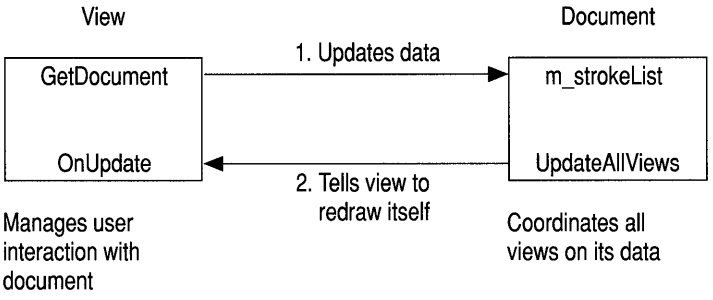


Figure 4.1 The View and the Document

Definition

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

Views in the Framework

In the framework, the document manages data, but the view displays it and acts as intermediary between the user and the document for all input, selection, and editing in the document. Typically, a document has only one view, although it is possible for a document to have multiple views, as in the case of splitter windows. A given

view is always associated with only one document. Each Scribble document uses only one view.

View Creation

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

Drawing the View's Contents

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

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

Document/View Interaction

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

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

You and the View

Table 4.1 shows your responsibilities and those of the framework in implementing a view on a document.

Table 4.1 View Implementation Responsibilities

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

Other view classes include **CFormView** and **CEditView**. For more information about views, see Chapters 2 and 4 and class **CView** and its derived classes in the *Class Library Reference*.

Scribble's View: Class CScribView

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

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

AppWizard writes a skeletal `CScribView` class for you, but you need to implement the `OnDraw` member function and add a few other things. In the following procedure, as you did in Chapter 3, you'll add the indicated code to a named file. Add the lines marked with the ▸ symbol in the left margin.

▶ To define the working data used by the view

- Add Scribble-specific lines to class `CScribView`. File SCRIBVW.H declares class `CScribView`. Lines are added to the code generated by AppWizard to define some Scribble-specific data items:

```
class CScribView : public CView
{
protected: // Create from serialization only
   CScribView( );
   DECLARE_DYNCREATE( CScribView )

// Attributes
public:
   CScribDoc* GetDocument( );

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

// Operations
public:
// Implementation
protected:

public:
   virtual ~CScribView( );
   virtual void OnDraw( CDC* pDC );  // Overridden to draw
                                     // this view
#ifdef _DEBUG
   virtual void AssertValid( ) const;
   virtual void Dump( CDumpContext& dc ) const;
#endif

// Printing support
protected:
   virtual BOOL OnPreparePrinting( CPrintInfo* pInfo );
   virtual void OnBeginPrinting( CDC* pDC, CPrintInfo* pInfo );
   virtual void OnEndPrinting( CDC* pDC, CPrintInfo* pInfo );
// Generated message-map functions
protected:
   //{{AFX_MSG(CScribView)
      // NOTE - the ClassWizard will add and remove member
      // functions here. DO NOT EDIT what you see in these
      // blocks of AppWizard code !
   //}}AFX_MSG
   DECLARE_MESSAGE_MAP( )
};

#ifndef _DEBUG   // Debug version in scribvw.cpp
inline CScribDoc* CScribView::GetDocument( )
    { return (CScribDoc*)m_pDocument; }
#endif
```

This code declares class `CScribView`, the view on the Scribble document's data. The added lines declare two new protected member variables.

Table 4.2 describes the member variables of class `CScribView`.

Table 4.2 CScribView Data Members

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

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

Table 4.3 describes the member functions of class CScribView.

Table 4.3 CScribView Member Functions

Member	Description
`CScribView`, `~CScribView`	With nothing to initialize and no data to destroy, the view's constructor and destructor are empty.
`OnDraw`	`OnDraw` updates the view by redrawing its contents. (It's used to draw both on the screen and on a printer.)
`GetDocument`	Defined inline in file SCRIBVW.H, `GetDocument` retrieves a type-safe pointer to the document attached to this view. The view uses the pointer to call document member functions, which it must do to access the data it displays.
`AssertValid`, `Dump`	These diagnostic functions simply call the base-class functions they override.
`OnPreparePrinting`, `OnBeginPrinting`, `OnEndPrinting`	These virtual functions override the versions in **CView** to specify the application's printing behavior. See Chapter 9 for more information about how Scribble prints.

AppWizard creates the constructor and destructor, `GetDocument`, `AssertValid`, `Dump`, `OnPreparePrinting`, `OnBeginPrinting`, and `OnEndPrinting` for you. You won't need to alter any of these functions for the tutorial, so they are not shown in this chapter.

Notice the inline definition of `GetDocument` after the class declaration above. The debug version of this member function calls the **IsKindOf** member function defined in class **CObject** and uses the **RUNTIME_CLASS** macro to retrieve the run-time class name of the document. For more information about those topics, see class **CObject** in the *Class Library Reference* and Chapter 12, "The CObject Class," in this manual.

Redrawing the View

When the view, or some part of it, must be redrawn, the framework calls your override of the `OnDraw` member function.

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

- Add `OnDraw` to file SCRIBVW.CPP, as defined below:

```
// Constructor and destructor, then ...
void CScribView::OnDraw( CDC* pDC )
{
   CScribDoc* pDoc = GetDocument();

   // The view delegates the drawing of individual strokes to
   // CStroke::DrawStroke( ).
   for( POSITION pos = pDoc->GetFirstStrokePos( );
                                                pos != NULL; )
    {
      CStroke* pStroke = pDoc->GetNextStroke( pos );
      pStroke->DrawStroke( pDC );
   }
}
```

The view calls upon the individual stroke objects to draw themselves. To do this, the view needs access to the stroke data stored in the document, so the view's first task is to obtain a pointer to its document, using `GetDocument`. The view then uses the pointer to iterate through the stroke list, telling each stroke to draw itself. You saw the `CStroke` member functions `GetFirstStrokePos` and `GetNextStroke` in Chapter 3. When `OnDraw` calls `DrawStroke` for a given stroke object, it passes along the device-context object it received as a parameter. (Having the data draw itself is only one possible strategy.)

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

▶ To add drawing code for strokes

1. Add the DrawStroke member function to file SCRIBDOC.CPP. Its declaration is already in place in SCRIBDOC.H. `DrawStroke`, called when the view redraws itself (as described above) looks like this:

```
// AddPoint, then ...

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

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

`DrawStroke` next constructs a new **CPen** object and initializes it with the current properties by calling the pen's **CreatePen** member function—note that this two-stage construction is typical of framework objects. Then `DrawStroke` calls **SelectObject** to select the pen into the device context (saving the existing pen as `pOldPen`) and calls **MoveTo** to position the pen. To initialize the pen position, `DrawStroke` calls the stroke object's `GetPoint` with an argument of 0. `GetPoint` returns the first point in the stroke's array.

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

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

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

---

2. The `GetPoint` member function called by `DrawStroke` (and already defined inline in file SCRIBDOC.H) is a helper function provided to retrieve points from the stroke's point array. It looks like this (don't add this code):

```
// Declaration of DrawStroke in class CStroke, then ...
protected:
   CPoint GetPoint( int i ) const
            { return CPoint(m_pointArray[i]); }
```

`GetPoint` retrieves the doubleword stored at a given index in the point array. It casts the doubleword to a **CPoint**.

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

Handling Windows Messages in the View

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

When the user presses the left mouse button while the pointer is in a Scribble window, Windows sends the window a **WM_LBUTTONDOWN** message. When the user subsequently releases the mouse button within the window, Windows sends the window a **WM_LBUTTONUP** message. Meanwhile, if the user moves the mouse—as in drawing—Windows sends a **WM_MOUSEMOVE** message.

How does Scribble handle these messages? They're sent to a window, in this case the currently active view. The view uses its "message map" to determine whether it has a member function that can handle the message. For example, on receiving a **WM_LBUTTONDOWN** message, the view finds that it has a "handler" associated with that message name and calls the handler. The handler is a member function of class `CScribView`.

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

What do the message-handler functions do? They track mouse activity, drawing in the view. They also call member functions of the document to update its data. As the user draws a stroke, the points that make up the stroke are stored in the document's stroke list.

Connecting Messages to Code

This section takes you through the steps required to connect the three mouse-related messages needed in Scribble to message-handler member functions of class `CScribView`.

This step will be different from the previous ones. Instead of opening a file in the Visual Workbench editor and adding lines of code, you'll invoke ClassWizard and use it to make the connections between Windows messages and their handler functions. ClassWizard lets you make the connections and generate the handler functions with a few clicks of the mouse. ClassWizard writes an entry in the message map for class `CScribView` and writes a default member function definition in the SCRIBVW.CPP file for the handler function. You fill in the function's code.

You'll learn much more about ClassWizard, messages, message maps, and handler functions in Chapter 6. For now, if you're working along, you'll invoke ClassWizard to connect the three mouse-related messages to handler names and to generate the handlers. Then you'll use the Visual Workbench editor to fill in the handler functions. If you're reading along instead, you can still try out ClassWizard on the starter code you created with AppWizard in Chapter 2. For more information about ClassWizard, see Chapter 6 in this manual, Chapter 13 in the *Visual Workbench User's Guide*, and Chapter 9 in the *App Studio User's Guide*.

▶ To connect the messages to Scribble's code

1. With your Scribble project open in Visual Workbench, invoke ClassWizard by choosing the ClassWizard command from the Browse menu. The ClassWizard dialog box appears.
2. In the Class Name edit box, make sure class `CScribView` is selected. The name "CScribView" and a number of predefined command IDs appear in the Object IDs list box. Figure 4.2 shows ClassWizard's main dialog box.

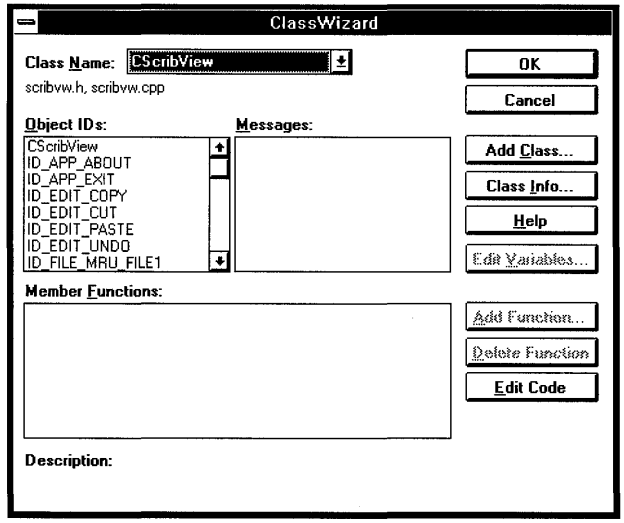


Figure 4.2 The Main ClassWizard Dialog Box

3. In the Object IDs list box, select the name "CScribView." A list of Windows messages that the view can receive appears in the Messages list box. Figure 4.3 shows ClassWizard with the list of messages.

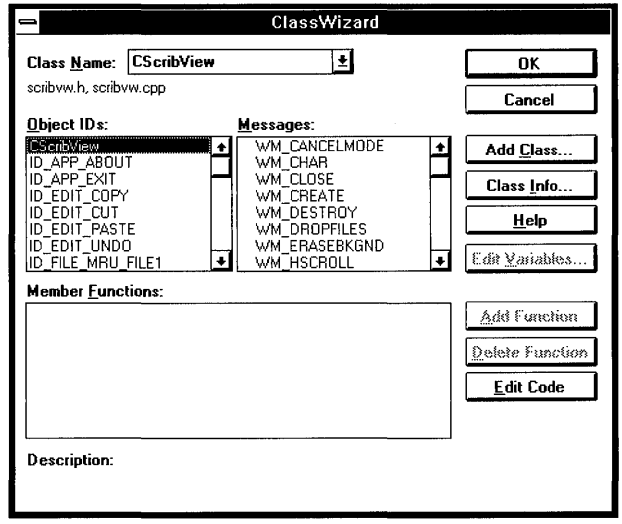


Figure 4.3 Available Windows Messages in ClassWizard

4. In the Messages list box, select the name of a Windows message for which you want to define a handler. To begin, select **WM_LBUTTONDOWN**.
5. To define a handler function for the message, choose the Add Function button.

   The Member Functions list box now lists the member function name **OnLButtonDown** and the corresponding message-map macro name **ON_WM_LBUTTONDOWN**. A small hand-shaped icon next to the **WM_LBUTTONDOWN** message in the Messages list box shows that the connection has been made.
6. Repeat steps 4 and 5 for each additional message: first **WM_LBUTTONUP**, then **WM_MOUSEMOVE**.
7. Choose the OK button to close the main ClassWizard dialog box.

At this point, ClassWizard has done the following things to associate each of the three messages with its handler and to greatly simplify your work:

- Added a function declaration for the handler to the `CScribView` class declaration in file SCRIBVW.H.
- Added a "message-map entry" for the message-to-handler connection in `CScribView`'s message map in file SCRIBVW.CPP.

- Added a function definition with a default body—to file SCRIBVW.CPP. For example, the default function definition for `OnLButtonDown` looks like this (don't add this code):

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

  Notice that ClassWizard embeds a comment reminding you what to do and adds a call to the **OnLButtonDown** member function of class **CView**, the base class of `CScribView`.

You'll learn more about message maps, message-handler functions, and their uses in Chapter 6. For more information about these topics, see Chapter 3 in the *Class Library Reference*.

Adding the Message-Handler Functions

With the connections made, it's time to fill in the bodies of the handler functions. If you're working along, add the marked lines of code.

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

1. Choose the ClassWizard command from the Browse menu in Visual Workbench.
2. To write the handler code for the **WM_LBUTTONDOWN** message, select the `OnLButtonDown` handler name in the Member Functions list box.
3. Choose the Edit Code button. The Visual Workbench editor appears, scrolled to a function definition for `OnLButtonDown`. Its body is selected for editing. Figure 4.4 shows `OnLButtonDown` in the editor.

   ClassWizard creates the following default implementation, which you'll replace later:

```
void CScribView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CView::OnLButtonDown( nFlags, point);
}
```

4. Fill in the member functions as described in the procedures that follow.

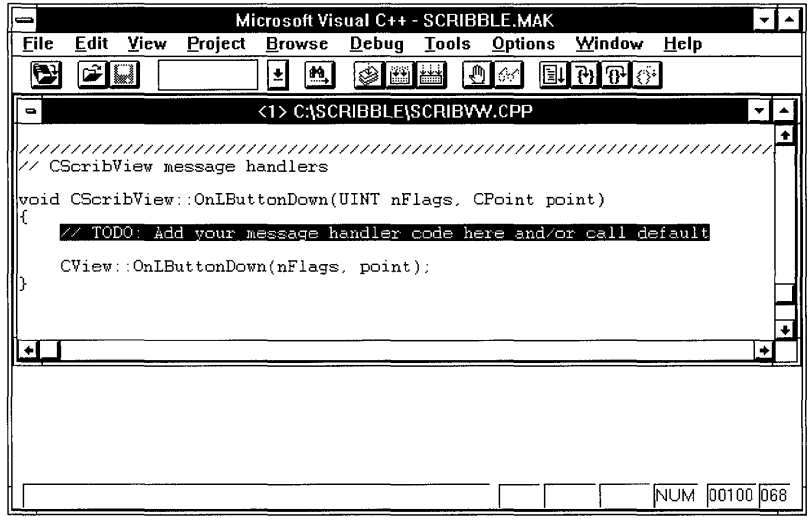


Figure 4.4 The Visual Workbench Editor

Initiate Stroke Drawing

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

▶ To add code for OnLButtonDown

1. If you haven't already done so, use the Edit Code button in ClassWizard (or the Visual Workbench editor) to move to the function definition for `OnLButtonDown`.
2. Replace the default implementation of the `OnLButtonDown` function body with the marked lines shown here:

```
void CScribView::OnLButtonDown( UINT nFlags, CPoint point )
{

   // When the user presses the mouse button, she may be
   // starting a new stroke, or selecting or de-selecting a
   // stroke.

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

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

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

3. Remove the first parameter name, *nFlags*, from the declaration of `OnLButtonDown` in order to avoid a compiler warning that this parameter is not referenced.

```
void CScribView::OnLButtonDown( UINT, CPoint point )
```

Once in the editor, you can complete your other message handlers or return to ClassWizard and select another function.

Terminate Stroke Drawing

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

▶ To add code for OnLButtonUp

1. Use the editor to scroll to the `OnLButtonUp` function definition in the same file.
2. Replace the default implementation of the `OnLButtonUp` function body with the marked lines shown here:

```
// OnLButtonDown, then ...
void CScribView::OnLButtonUp( UINT nFlags, CPoint point )
{

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

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

   CScribDoc* pDoc = GetDocument( );

   CClientDC dc( this );

   CPen* pOldPen = dc.SelectObject( pDoc->GetCurrentPen( ) );
   dc.MoveTo( m_ptPrev );
   dc.LineTo( point );
   dc.SelectObject( pOldPen );
   m_pStrokeCur->AddPoint( point );

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

3. Remove the first parameter name, *nFlags*, from the declaration of `OnLButtonUp` in order to avoid a compiler warning that this parameter is not referenced.

```
void CScribView::OnLButtonUp( UINT, CPoint point )
```

Draw While the Mouse Button is Down

Between the time that the mouse button goes down and the time that it's released, Scribble tracks the mouse and draws a trace of its movements in the view. `OnMouseMove`, shown below, is called as the user moves the mouse while drawing the current stroke. The function connects the latest mouse location with its previous location and saves the new location as the previous point for the next time the function is called. To do the drawing, `OnMouseMove` constructs a local **CClientDC** object used to draw in the window's client area.

▶ To add code for OnMouseMove

1. Scroll to the `OnMouseMove` function definition in the same file.
2. Replace the default implementation of the `OnMouseMove` function body with the marked lines shown here:

```
// OnLButtonUp, then ...
void CScribView::OnMouseMove( UINT nFlags, CPoint point )
{

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

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

   CClientDC dc( this );

   m_pStrokeCur->AddPoint( point );

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

3. Remove the first parameter name, *nFlags*, from the declaration of `OnMouseMove` in order to avoid a compiler warning that this parameter is not referenced.

```
void CScribView::OnMouseMove( UINT, CPoint point )
```

Together, these three member functions handle the three phases of mouse drawing: beginning to track the mouse, tracking the mouse and connecting points, and ending mouse tracking.

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

Compile and Test Scribble

In this section, if you've been working along, you'll compile your completed code and try out the program. If you're simply reading along, use the Scribble project supplied in your Visual C++ installation as described in step 2 following.

▶ To try out your work in Chapters 3 and 4

1. If you don't have Visual Workbench running, start it by choosing the Visual Workbench icon in the Program Manager.
2. Open the SCRIBBLE.MAK project (if it's not already open) by choosing Open on the Project menu and completing the dialog box. The following describes where to find the appropriate version of the project:

   If you're building Scribble after working through the chapter, use the SCRIBBLE.MAK file in your MYSCRIB directory.

   –or–

   If you're simply previewing Scribble without adding code, use the SCRIBBLE.MAK file provided by Visual C++ in the MFC\SAMPLES\SCRIBBLE\STEP1 subdirectory.
3. On the Project menu, choose Build to compile and link Scribble.
4. If necessary, use the Visual Workbench debugging facilities to find and correct any errors you made.

   For debugging information, see Chapter 11 in the *Visual Workbench User's Guide*.

After compiling Scribble, try out its features. When Scribble runs, an MDI application window appears with a menu bar containing File, Edit, View, Window and Help menus and a toolbar and status bar. It has one document window open as shown in Figure 4.5.

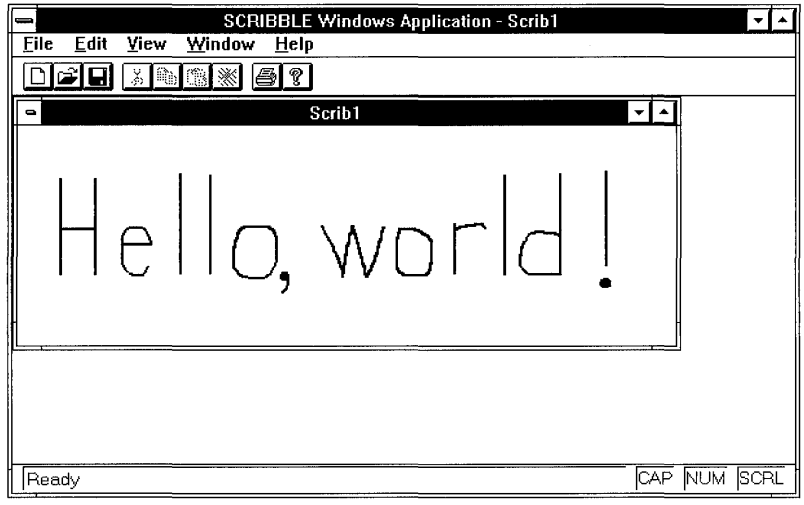


Figure 4.5 Scribble Step 1

▶ To give Scribble a try

1. On the Visual Workbench Project menu, choose the Execute Target command to run Scribble.
2. On the File menu, use the New command to create a new document.
3. Move, resize, minimize, and maximize the document window.
4. Draw "Hello, World!" (or anything) in the window. Then save the file as HELLO.SCB.
5. Try the Print Preview and Print commands on the File menu.

Note Your printout may show the strokes of your scribbling at a reduced size. This is because the current version of Scribble uses the **MM_TEXT** mapping mode instead of the **MM_LOENGLISH** mapping mode. Unlike **MM_LOENGLISH**, which Scribble will use at a later stage, **MM_TEXT** doesn't acknowledge that a pixel on the printer is much smaller than a pixel on the screen. Printing will improve later in the tutorial.

---

6. Close HELLO.SCB and reopen it with the Open button on the toolbar.
7. Create a new document with the New button on the toolbar and draw in the new document. (Save the new document if you like.)
8. Exit Scribble.

Note During drawing, Scribble samples points as fast as it can (as soon as the mouse moves). When Scribble's view redraws a stroke, the playback represents approximately the speed at which the stroke was originally drawn.

This concludes your quick introduction to Scribble. You've seen how to implement the document with serialization and the view with message handling. In the next chapter, you'll learn to construct some additional user-interface components with App Studio. Later chapters add more (and more interesting) code to Scribble.

CHAPTER 5

Constructing the User Interface With App Studio

Now that you've implemented enough of Scribble by hand to see how it works, it's time to explore the tools that eliminate much of the handwork.

The next three chapters show how to use some of the powerful tools supplied with the Microsoft Foundation Class Library.

- This chapter explains how to use App Studio to visually construct Scribble's menus and toolbar.
- In Chapter 6, you will use ClassWizard and Visual Workbench to bind menu items and toolbar buttons to commands and define message-handler functions to process the commands.
- Chapter 7 shows how to use both App Studio and ClassWizard to create a dialog box and connect it to a menu command.

App Studio, Class Wizard and Visual Workbench automatically add these new features to the resource and source files that AppWizard created in Chapter 3 and that you augmented in Chapter 4.

This chapter and Chapter 6 cover step 2 of Scribble. If you want to work along, adding the code as you go, begin with the files from Chapter 4 in your MYSCRIB directory. At this point, your files should be very similar to the files in the SCRIBBLE\STEP1 subdirectory. As you read this chapter, perform all App Studio steps. At the end, your resource file, SCRIBBLE.RC, should closely resemble the same file in the SCRIBBLE\STEP2 subdirectory. Also in Visual Workbench you'll make a small addition to the MAINFRM.CPP file. You can compile the new version of Scribble at the end of Chapter 6.

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

Even if you don't want to add code, however, it's a good idea to work along in this chapter to familiarize yourself with App Studio and the Visual C++ programming process. You can begin by making your own copy of the SCRIBBLE\STEP1 subdirectory.

Edit Scribble's Menus

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

Thanks to AppWizard, Scribble starts out with a skeleton resource file with no effort on your part. You can look at the file SCRIBBLE.RC among the files generated in Chapter 2 to see what's there.

Default Menus

The menus AppWizard generates by default include:

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

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

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

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

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

Scribble's New Menu Commands

The goal in this section is to add a new Clear All command to the Edit menu as well as a completely new Pen menu with two commands, Thick Line and Pen Widths. The Clear All command clears the current drawing and deletes its stroke data. The Thick Line command toggles the thickness of the lines used to draw subsequent strokes. They can be thick or thin. The Pen Widths command brings up a dialog box that lets the user set the thick and thin widths in pixels for subsequent drawing.

In the discussion that follows, you'll see how to create the new menu items with App Studio. In the next chapter, you'll see how to use ClassWizard to connect the menus to code. And in Chapter 7 you'll see how to create the Pen Widths dialog box and connect it to the menu command.

Adding the Menus

This section adds all of Scribble's new menus and demonstrates the fundamental techniques for editing menus with App Studio.

Designing

Because App Studio is highly visual, it is easy for you to design your application's user interface. For example, you can drag menus around on the menu bar, drag and drop new controls on a dialog box, use tools to align the elements, and so on.

Beyond that, App Studio lets you test the behavior of your menus and dialog boxes. While you're editing menus, simply click the menu bar to see a menu drop down. While you're editing a dialog box, invoke App Studio's Test command to simulate the actions of the dialog box and its controls. You'll see the Test command in use in Chapter 7.

Examining and Editing Resources

App Studio lets you work with many resource types: menus, dialog boxes, bitmaps, string tables, and more. Use the resource browser to select a particular resource type and then to select a specific resource of that type. The resource browser is the first window you see when you run App Studio—it's the upper window in Figure 5.1. The figure shows the Menu resource type selected.

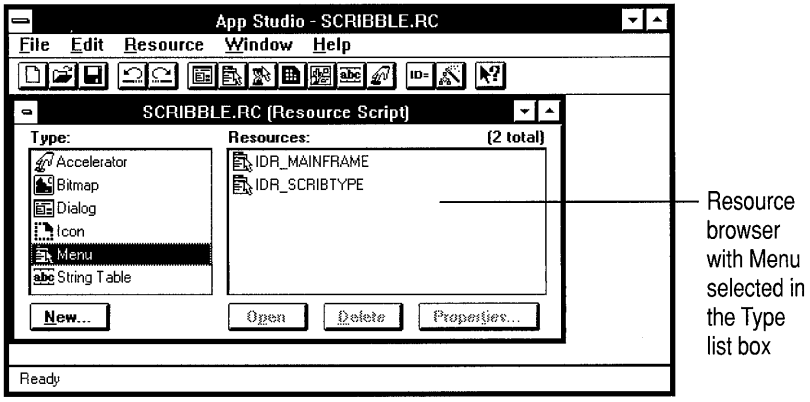


Figure 5.1 The App Studio Resource Browser

In the resource browser, you can click a resource type to view a list of the specific resources of that type. Each resource has an ID and an icon that shows its type. Then you can click a resource ID and choose the Open button to invoke an editor for the selected resource. Or you can create a new resource of the selected type by choosing the New button.

When you select and open a specific resource, such as a menu, App Studio invokes an editor for that type. For example, the menu editor displays a menu bar showing all menus defined by the open menu resource. Although you're in editing mode, you can simulate menu operation by choosing a menu with the mouse (keyboard shortcuts are not available for a menu-in-progress). You can also drag menus to new

locations, add new menus or menu items, and edit the properties of existing menus. Figure 5.2 shows a menu-editor window.

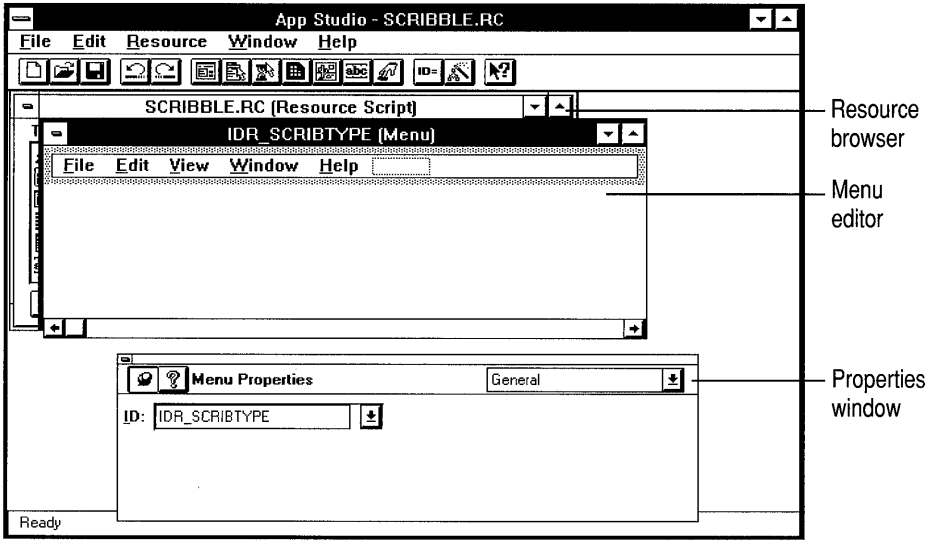


Figure 5.2 The Menu Editor

As you work with the editor for a resource, the Properties window floats nearby, showing a "property page" for the resource. (To keep it there, use the "pushpin" control in the upper left corner, as shown in Figure 5.3.) Figure 5.3 shows the property page for a menu resource. A property page displays information about the resource and lets you edit the information. For example, a menu's property page displays the ID of the command associated with the menu, the menu's caption as it appears when the menu drops down, and other properties that relate to the menu's visibility or status. The property page also shows a command-prompt string displayed when the user moves the mouse over the menu item. In framework programs, the prompt tells the user what the menu command does. Some resources have multiple property pages, which you can access through the drop-down list box in the upper-right corner of the Properties window.

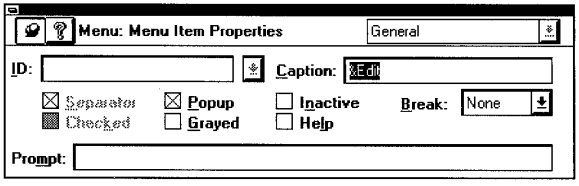


Figure 5.3 A Property Page

Properties for a selected resource are displayed automatically in the Properties window. When you select the ID with a single click, check boxes for general properties appear—such as Preload, Moveable, Discardable, and Pure. Usually you can ignore these. When you open the resource to edit it and select a menu in the editor window, the property page reflects the properties of the chosen menu item. Typically, you'll edit these.

When you use App Studio to define a command ID for a menu, App Studio assigns each such ID a unique value and writes a **#define** line for it in the file you specify or RESOURCE.H by default. Automatic definition of such symbols is one of the advantages of using App Studio. If you enter a duplicate ID, App Studio alerts you so you can change it.

The command ID and a caption string are required menu properties. For more information about menu property pages, see the *App Studio User's Guide*.

Add the Clear All Command to Scribble's Edit Menu

How App Studio handles menus is illustrated when you add the new menu item, the Clear All command, to Scribble. If you're working along, use the following procedure:

▶ To add Scribble's Clear All Menu Command

1. Start up App Studio: With your Scribble project open in Visual Workbench, open the resource file SCRIBBLE.RC from the File menu.

   If you're working along, use the version of SCRIBBLE.RC in the directory MFC\SAMPLES\SCRIBBLE\MYSCRIB. If you're reading along, use the version in the directory MFC\SAMPLES\SCRIBBLE\STEP1.

   This launches App Studio, which automatically uses the resource compiler to compile the file's resources.

   When the file has been compiled, the App Studio resource browser lists Scribble's resource types.

   Important Although you can run App Studio as a stand-alone program, it's best to run it from the Tools menu in Visual Workbench with your project already open. Or you can run App Studio by opening a .RC file from the Visual Workbench File menu. Running App Studio from Visual Workbench in either of these ways allows Visual Workbench to pass useful information about the project to App Studio.

   ---

2. Choose Menu from the Type list box of the resource browser.

Two menu IDs appear in the right-hand list box: **IDR_MAINFRAME** and IDR_SCRIBTYPE. **IDR_MAINFRAME** is the menu resource for the multiple document interface (MDI) frame window when no documents are displayed in its child windows. IDR_SCRIBTYPE is the menu resource for the frame window when a document in a child window has the focus. (Under the multiple document interface (MDI), the available menus vary depending on context.) This ID was defined for you by AppWizard. The ID name is based on the Document Type Name you supplied in AppWizard.

3. Open the IDR_SCRIBTYPE menu resource.

   Select the ID in the Resource list box and choose the Open button, or double-click the ID. A menu-editing window appears. You see the menus much as you would see them in the running application. If the Properties window does not appear, select Show Properties from the Window menu. The Properties window now shows the properties of this particular menu resource. Figure 5.4 shows the menu editor and a general property page for the resource.

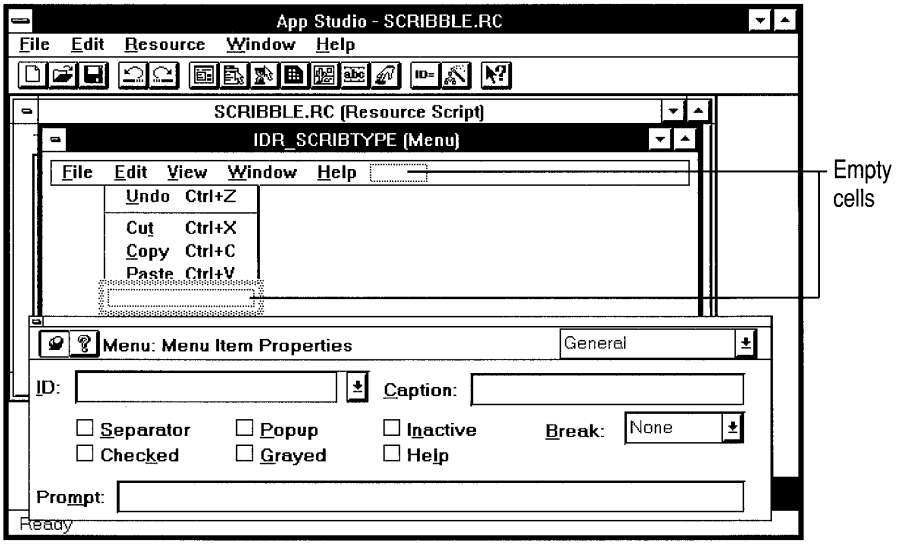


Figure 5.4 Menu Editor for IDR_SCRIBTYPE

4. Click the Edit menu.

   The menu drops down as it would in the application. An empty cell outlined with a dotted line sits below the last menu item, as shown in Figure 5.4. The cell defines where you add the next menu item.

   For a taste of what you can do with menus, see the box "Drag and Drop" on page 77.

5. Click the cell at the bottom of the Edit menu.

The cell is highlighted with a fuzzy gray outline. This is where you'll see the menu's caption after you type it in the Properties window.

6. Type the caption **Clear &All** in the Caption edit box of the Properties window.

   As soon as you start typing, the text you enter appears in the ragged-outlined cell on the menu. You don't have to select the Caption edit box first.

   Type the ampersand character (**&**) in front of the letter to be used in an access key combination. As you type **&A**, for example, the letter A appears underlined in the menu.

Note If you wanted to specify an accelerator or shortcut key for the menu item, you'd append its specifier after the caption. For example, to specify CTRL+O as the accelerator for an Open command, the caption string would read "Open...\tCTRL+O" where "\t" signifies a tab to align the column.

---

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

   As soon as you start typing the ID, the drop-down list box scrolls to the first ID that matches the letters you've typed so far. This behavior occurs because **ID_EDIT_CLEAR_ALL** is a command ID predefined by the class library. App Studio ensures that the ID you enter is unique.

   Two IDs that begin with "ID_EDIT_C" appear in the list box. Select the second one: **ID_EDIT_CLEAR_ALL**.

   Figure 5.5 shows the property page after you've selected the ID.

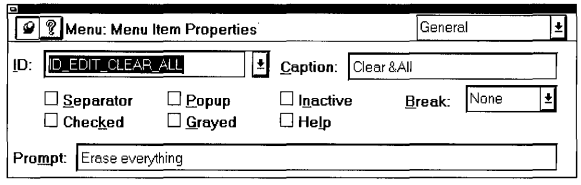


Figure 5.5 Property Page with ID

You can define your own command IDs, of course. You'll see an example under "Add Scribble's Pen Menu" on page 74.

8. As soon as you select the predefined ID, the following string appears in the Prompt edit box: "Erase everything." Change the wording to "Clears the drawing."

   The prompt string is displayed in the status bar, if the application has one, when the user navigates up and down the menu using the keyboard.

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

That's it. You've added the Clear All command to the Edit menu. It appeared in the menu as soon as you started typing its caption. Figure 5.6 shows the finished product.

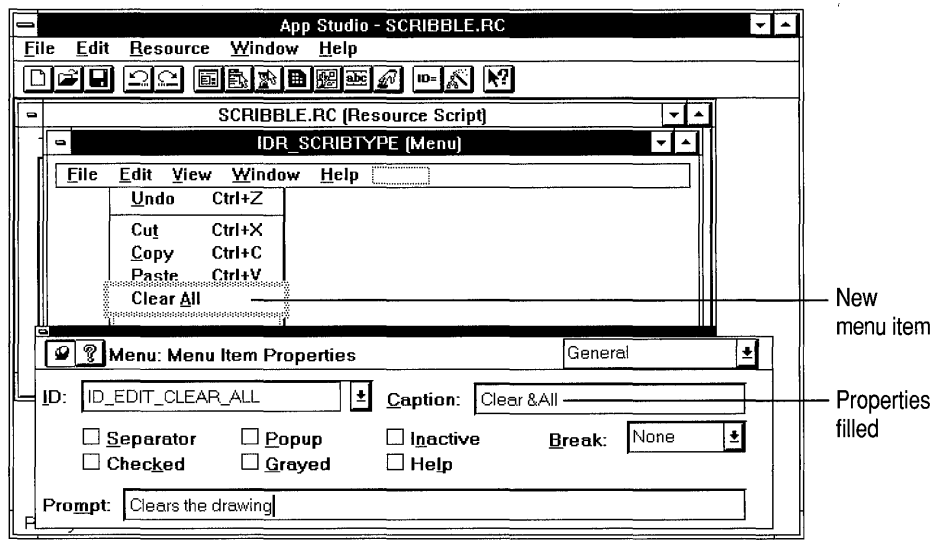


Figure 5.6 Adding the Clear All Menu Item

Note You don't have to press ENTER or click any buttons to conclude your menu editing. App Studio automatically stores the edited resource in the resource file.

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

Add Scribble's Pen Menu

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

▶ To add Scribble's Pen menu

1. With the menu-editor window still showing, click in the ragged-outlined cell at the right-hand end of the Scribble menu bar (after the Help menu).

This cell serves the same purpose for the top-level menus as the other ragged-outlined cell does for items within a pop-up menu.

2. To position the menu entry, drag the ragged-outlined cell to the left and drop it between the Edit and View menus. (See the box "Drag and Drop" on page 77.)
3. Type the new menu's caption, **&Pen**, in the Caption edit box of the Properties window.

   This is the only step needed to define the Pen menu as a whole. The next step is to define the menu items on the Pen menu.

   Figure 5.7 shows the new Pen menu after it has been dragged to its new location and the menu caption typed in.

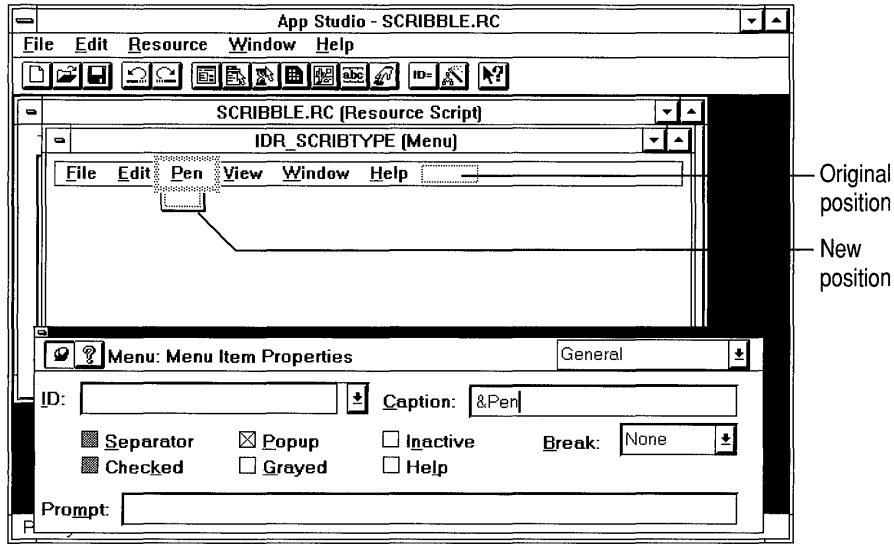


Figure 5.7 The Pen Menu Dragged into Position

4. Press the ENTER key to advance to the first menu item or click the ragged-outlined cell that descends beneath the word "Pen."
5. As you did for the Clear All command, type a caption for the Thick Line command in the Caption edit box: **Thick &Line.**
6. Select the ID combo box and type a command ID: **ID_PEN_THICK_OR_THIN.**
7. Type a command prompt string in the Prompt edit box: **Toggles the line thickness between thin and thick.**

   No default string appeared because ID_PEN_THICK_OR_THIN is not a predefined command ID.
8. Click the ragged-outlined cell at the bottom of the Pen menu, below "Thick Line."

9. As you did for the Thick Line command, type a caption for the Pen Widths command: **Pen &Widths.**

   The ampersand (&) appears before the character to be used as an access key. The ellipses (. . .) following a menu item's text lets the user know that the item brings up a dialog box.

10. Type a command ID: **ID_PEN_WIDTHS**
11. Type a command prompt string: **Changes the size of the thin and thick pen**

That's all it takes to create the Pen menu. Figure 5.8 shows the completed menu as it appears in the menu editor.

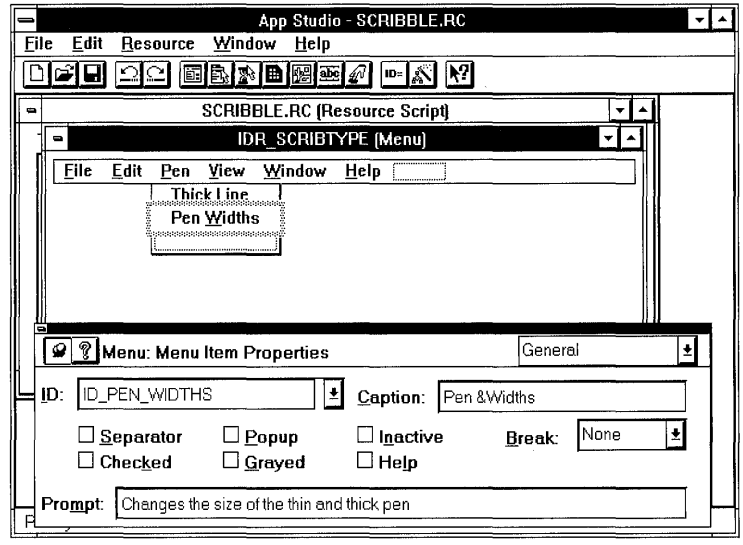


Figure 5.8 The Completed Pen Menu

Drag and Drop

Drag and drop is a common technique in App Studio. You'll find it in the user interfaces of many of the resource editors. For example, in Chapter 7, you'll see it used to drag various kinds of controls from a controls palette and drop them into a dialog box in the dialog editor. Try experimenting in the menu editor: position the mouse over an outlined box under a menu item; press the left mouse button, and drag the box up or down the list and drop it where you like by releasing the mouse button. Notice that when you start to drag, an insertion point appears to orient you for dropping. You can also drag a whole menu to some other location in the menu structure, or you can drag a top-level menu to a lower level, to create a hierarchical menu. If you change your mind, drag the menu back or choose Undo from the Edit menu.

Connect the Menus to Code

Typically, at this point you would invoke ClassWizard from App Studio's Resource menu and use it to bind the menu commands to message-handler functions. That step is postponed until the next chapter in order to keep this chapter focused on App Studio and constructing the user interface. If you like, you can skip ahead, perform the command-binding steps in Chapter 6 and then return to this chapter to edit Scribble's toolbar.

Edit Scribble's Toolbar

The resource file that AppWizard gives you also includes a toolbar bitmap. Figure 5.9 shows the toolbar bitmap. In this section you'll use App Studio's bitmap editor to add a new button to the bitmap for Scribble.

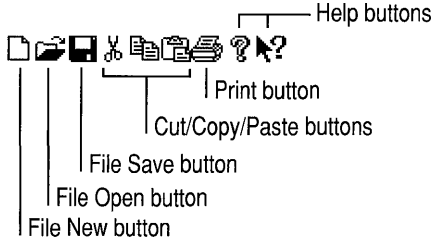


Figure 5.9 The Default Toolbar Bitmap

Earlier in the chapter, you added the Pen menu. One of its menu items is the Thick Line command. In this section, you'll add a Thick Line button to Scribble's toolbar. Then, in Chapter 6, you'll use ClassWizard to connect both the Thick Line menu item and the Thick Line toolbar button to the same handler member function. Thus the Thick Line toolbar button will become an alternative user interface for the Thick Line menu item. That is, both user-interface objects will have the same command ID, so they generate the same command message, which invokes the same handler function.

When the user chooses either the menu item or the toolbar button, the chosen item sends a command message that toggles Scribble's drawing pen between thin and thick lines. Figure 5.10 shows Scribble as it appears with the finished toolbar. The Thick Line button is the eighth button from the left.

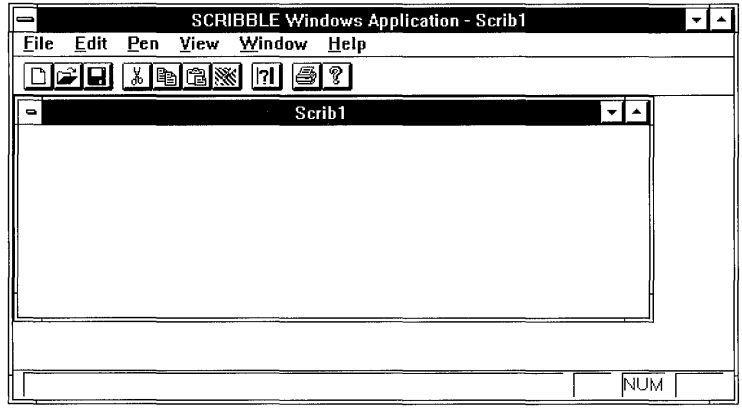


Figure 5.10 Scribble with Its Edited Toolbar

About the Toolbar

Some of the buttons on Scribble's toolbar already work, as you saw in Chapter 4. The buttons for opening and saving files are already connected to handlers defined by the framework. All you had to do to make the file operations functional was write the Serialize functions for the document and the stroke data structure. The print button is supported by default.

The Cut, Copy, and Paste buttons on the toolbar will not be implemented for Scribble. The Help button will not be connected up until a later chapter in the tutorial.

Although this chapter simply shows you how to add one new button, you could easily add others or delete unused buttons from the toolbar bitmap.

Add the Thick Line Button to Scribble's Toolbar Bitmap

You'll use App Studio's graphics editor for this task.

▶ To edit Scribble's toolbar

1. Start up App Studio from Visual Workbench.

   You saw how to do this earlier in the chapter.

2. In the resource browser, choose Bitmap in the type list box.

   If the Properties window is not open, open it by choosing the Properties button in the resource browser window. Click the pushpin so the Properties window will remain open.

   Figure 5.11 shows the resource browser at this stage. In the Properties window, you see a Preview box with an actual-size image of the bitmap.

3. Select the IDR_MAINFRAME resource ID in the Resources list box.

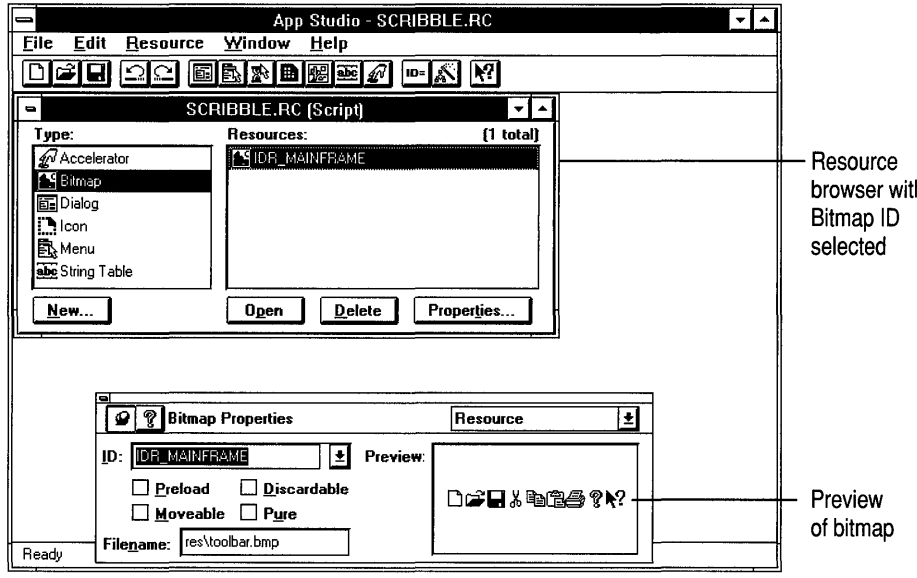


Figure 5.11 Bitmap Selection in the Resource Browser

4. Choose the Open button to Open the **IDR_MAINFRAME** bitmap resource.

   An image window opens showing the bitmap. Below it is the Properties window. A graphics palette opens as well, showing the tool box, a color indicator, the color palette, and the option selector. Figure 5.12 shows this configuration. You may see more or less of the image window than shown in the figure depending on your screen size.

You may want to drag the graphics palette out of the way on the left-hand side of the image window.

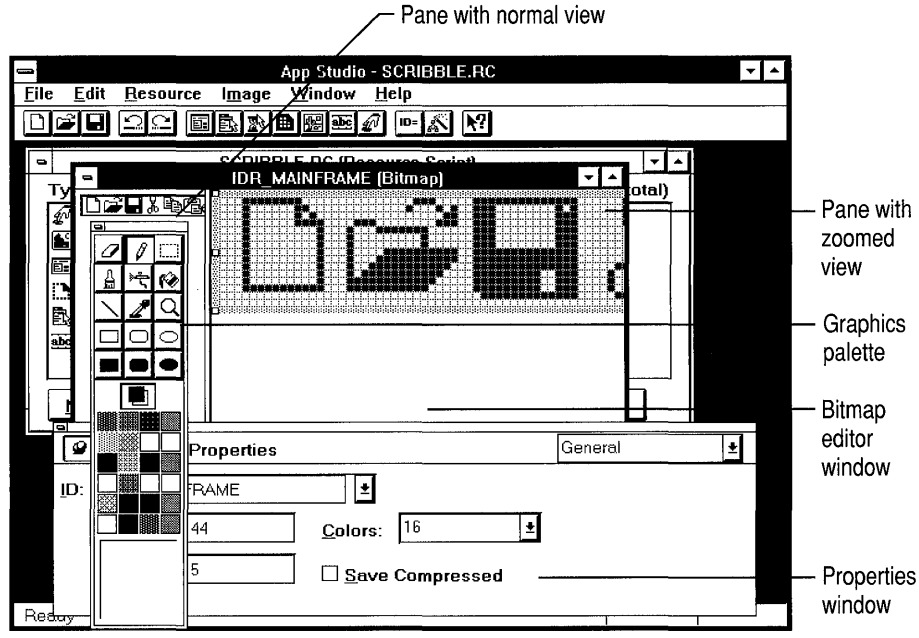


Figure 5.12 The Bitmap Image Window

5. Use the window's maximize button to enlarge the image window that shows the bitmap.
6. From the Image menu Choose Grid Settings.
7. In the Grid Settings dialog box, set the Tile Grid checkbox and choose the OK button. (Figure 5.13 shows the Grid Settings dialog box.) Leave Pixel Grid checked as well. The size boxes show a grid size of 16 pixels by 15 pixels. This is the size of a "tile"—one of the buttons on the toolbar.

   Turning on the Tile Grid option places a thin blue rectangle, a guide, around each tile in the bitmap. These guides make it easier to select and work with a tile.

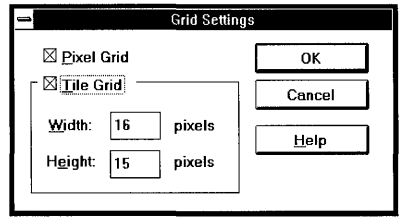


Figure 5.13 The Grid Settings Dialog Box

8. Scroll the right-hand pane of the image window (which shows a zoomed image of the bitmap) until the right-hand end of the bitmap is visible in the center of the pane.

   Notice that the bitmap is outlined by a selection rectangle. In the Properties window, the Width text box shows the bitmap's current width to be 144 pixels. Beyond the end of the bitmap is enough space to expand the bitmap by several tiles.

9. Lengthen the bitmap by one tile: Drag the resizing handle on the right-hand end of the bitmap to the right by the width of one tile.

   The bitmap grows a full tile at a time. Notice that as you drag your bitmap, the rightmost indicator on the App Studio status bar shows new bitmap dimensions of "160 x 16." This makes the bitmap wide enough to accommodate another button.

   Beyond the current end of the bitmap you see a white area the size of a tile at the end, overlaid with the grid. Figure 5.14 shows the bitmap as it appears after you've lengthened it.

   Notice that the Width text box in the Properties window now shows the new width of the toolbar bitmap to be 160 pixels.

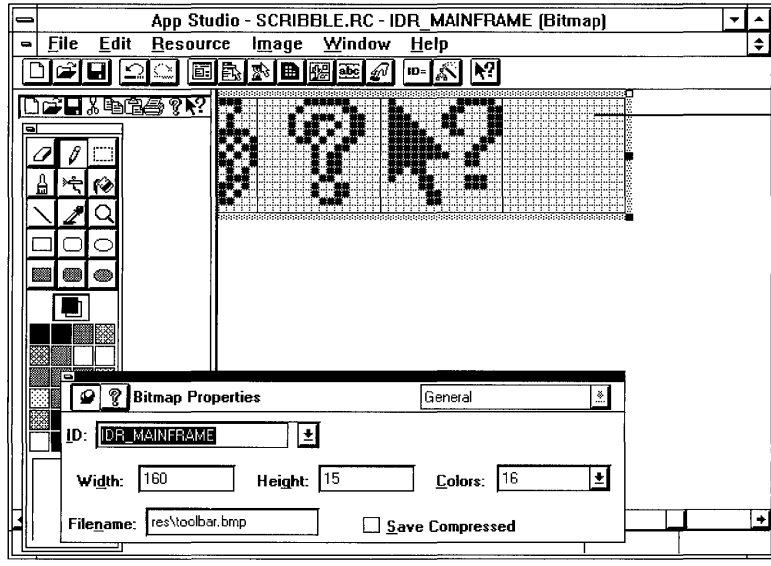


Figure 5.14 The Scrolled Bitmap

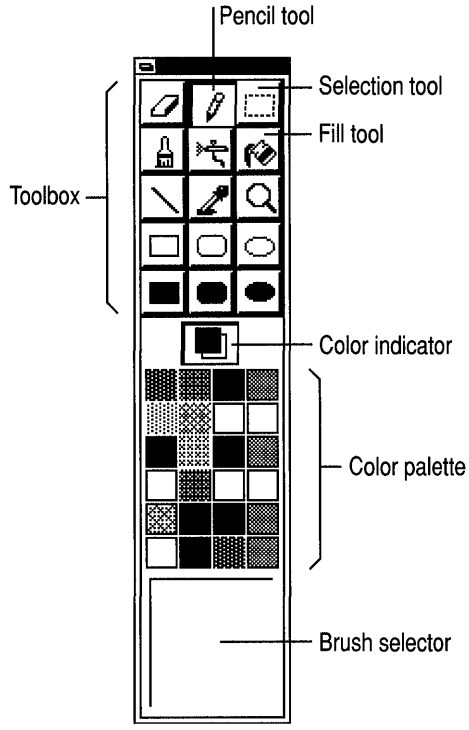


Figure 5.15 The Graphics Palette

10. Enclose the three rightmost button images—a printer, a question mark, and an arrow next to a question mark—with the selection rectangle.

    Click the selection tool in the upper-right corner of the graphics palette.

    Figure 5.15 shows the graphics palette and the three tools you'll be using: the selection tool, the pencil tool, and the fill tool.

    Position the mouse pointer at the upper-left corner of the printer image and drag to the lower-right corner of the question-mark image. Place the lines of the crosshairs just inside the blue guides.

11. Drag the selected images one tile to the right to open up a space for a new button.

    Figure 5.16 shows the bitmap after dragging.

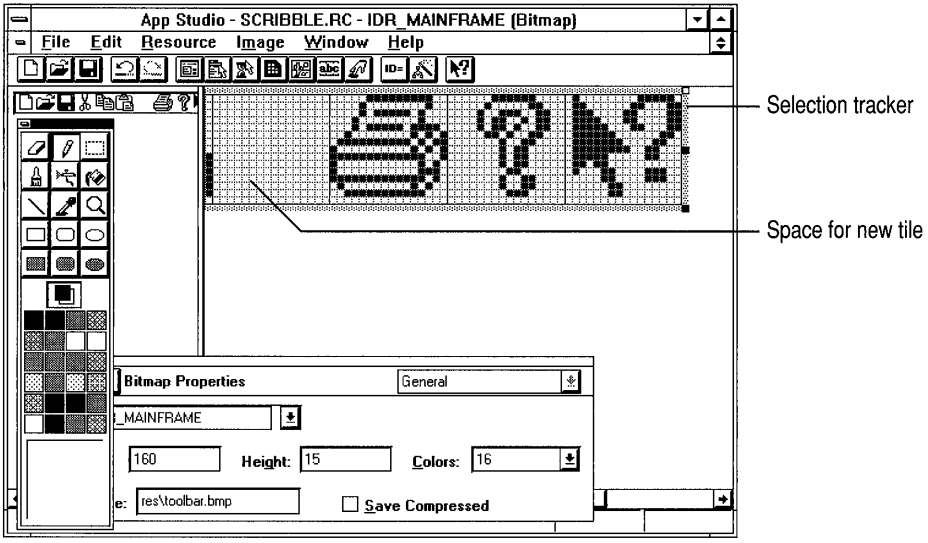


Figure 5.16 The Bitmap Dragged to the Right

12. Choose the pencil tool from the graphics palette.
13. Draw the bitmap shown in Figure 5.17. It doesn't have to be exact. If you make a mistake, choose the eraser tool.

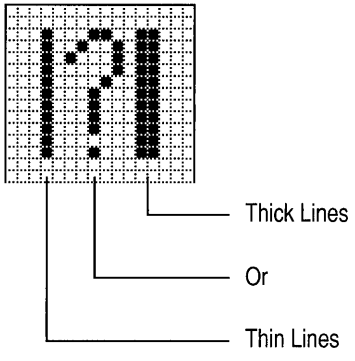


Figure 5.17 Bitmap for the Thick Line Button

14. Choose the paint-bucket fill tool from the graphics palette.
15. Choose light gray from the color palette.
16. Click the fill tool in the new button image to fill its background with the selected color.

Figure 5.18 shows the completed bitmap as it appears in Scribble.

Figure 5.18 The Edited Bitmap

17. From the File menu choose Save.
18. Exit App Studio.

That completes the task of editing Scribble's toolbar bitmap. In order for the new button to work, it must be associated with a command ID. In this case, the Thick Line button will be bound to `ID_PEN_THICK_OR_THIN`. You defined that ID earlier in the property page for the Thick Line menu command, so App Studio has already written a **#define** for the ID in a file called RESOURCE.H. Your only task at this point is to associate the ID with the button. If you are working along, add lines of code marked with ▸.

Note If you had just created a new button for which there was no existing command ID, you'd use the App Studio symbol editor to define a symbol such as `ID_PEN_THICK_OR_THIN`. For information about the symbol editor, see Chapter 2 in the *App Studio User's Guide*.

▶ To associate the button with its command ID

1. Start up Visual Workbench if it's not open.
2. From the File menu in the Visual Workbench editor, open the MAINFRM.CPP file.
3. Scroll to the static array called "buttons." The code maps the buttons to their command IDs.

```
// Arrays of IDs used to initialize control bars

// Toolbar buttons - IDs are command buttons
static UINT BASED_CODE buttons[] =
{
   // Same order as in the bitmap 'toolbar.bmp'
   ID_FILE_NEW,
   ID_FILE_OPEN,
   ID_FILE_SAVE,
      ID_SEPARATOR,
   ID_EDIT_CUT,
   ID_EDIT_COPY,
   ID_EDIT_PASTE,
      ID_SEPARATOR,
```

```
    ID_PEN_THICK_OR_THIN,
       ID_SEPARATOR,
    ID_FILE_PRINT,
    ID_APP_ABOUT,
};
```

This array definition provides a 1-to-1 mapping based on the positions of the button tiles in the toolbar bitmap. The **ID_SEPARATOR** entries denote small amounts of extra space used to group the button tiles.

4. Save the file.

Now the Thick Line button generates precisely the same command as the Thick Line menu item. Because of this, the same handler function serves for both.

Summary

Scribble's resource needs are simple, so this chapter introduced only a few of the things you can do with App Studio. For information about its many capabilities, see the *App Studio User's Guide*.

After editing your application's menus and toolbar with App Studio, the next step is to connect them to code using ClassWizard. That step is explained in Chapter 6.

CHAPTER 6

Binding Visual Objects to Code Using ClassWizard

The version of Scribble you saw in Chapters 3 and 4 added a small amount of code to the skeleton starter files created by AppWizard—much less code than you would normally have to write to get a comparable application up and running without the framework. Considering the small amount of work, the program does quite a lot: drawing, saving, printing, even print preview.

Like all applications written for Windows, Scribble is "message driven." A keystroke, mouse click, or other event causes messages to be sent to some part of the application that can respond to the event. In Chapter 4, for example, you saw how Scribble implements mouse drawing by detecting and responding to messages generated by mouse clicks and drags.

This chapter introduces a category of messages called "commands," which are messages to your application from menu items, toolbar buttons, and accelerator keys. The expanded version of Scribble developed in this chapter adds two menu items that generate commands to toggle the line thickness for drawing and to clear all strokes from the current document. The command that toggles line thickness is also duplicated by a button on Scribble's toolbar.

You created the resources for Scribble's new menu items and its new toolbar button in Chapter 5. Now you can use ClassWizard to assign a user-interface object, such as a menu item, to a command and map the command to a function that handles it.

In this chapter, you will:

- Learn the fundamentals of commands and how the framework routes them to various "command target" objects in the program for handling.
- Extend your knowledge of ClassWizard, begun in Chapter 4.
- Add new command-handling code for Scribble.
- Connect a toolbar button and a menu item to the same command.
- Learn how to keep your user-interface objects (menus and toolbar buttons) updated since a menu item may be enabled or disabled, a button checked or unchecked.

This chapter and the previous chapter cover step 2 of Scribble. If you want to work along, adding the code as you go, begin with the files from Chapter 5 in your MYSCRIB directory. At this point, your files should be very similar to the files in the SCRIBBLE\STEP1 subdirectory, plus the resource changes you made in Chapter 5 with App Studio. As you read this chapter, perform all ClassWizard steps and add all lines of code marked in the left margin with the symbol ▸. At the end of the chapter, your files should closely resemble the files in the SCRIBBLE\STEP2 subdirectory.

If, on the other hand, you want to read along without adding code, you can print or examine the files in the SCRIBBLE\STEP2 subdirectory. However, even if you don't want to add code, it's a good idea to work along in this chapter to familiarize yourself with ClassWizard and the Visual C++ programming process. You can begin by making your own copy of the SCRIBBLE\STEP1 subdirectory.

Using ClassWizard to Bind Commands

In the previous chapters, you created a working Scribble application. In Chapter 4, you used ClassWizard to make connections between several standard Windows messages related to mouse actions and the code that responds to those actions. Now you'll use ClassWizard to add new commands to Scribble.

In this chapter, you will use ClassWizard to connect menu items, toolbar buttons, and other user-interface objects to the code that responds when the user clicks them. In Chapter 7, you willll use ClassWizard again to create new classes and perform other chores (see the following box, "What ClassWizard Can Do").

What ClassWizard Can Do

ClassWizard is one of the Visual C++ tools that you'll use most frequently. It helps you:

- Connect standard Windows messages to message-handler functions.
- Connect user-interface objects to message-handler functions.
- Create new classes, such as dialogs and extra views, documents, or frame windows.
- Add member variables to dialog classes and specify how those variables are to be initialized and validated when the dialog box is displayed.

For more information about the capabilities of ClassWizard, see Chapter 9 in the *App Studio User's Guide*.

When the ClassWizard dialog box appears, a drop-down list shows all of the classes in your program that can handle commands or Windows messages. When you select a class to which you want to map commands, a list box shows you all of the visual objects associated with that class:

- Classes, such as dialogs, views, and frame windows, which can receive standard Windows messages.
- Dialog-box controls, which can generate Windows control-notification messages.
- Command IDs, which are associated with menu items, toolbar buttons, or accelerator keys.

When you select a visual object, a second list box displays a list of the Windows messages or commands to which you can connect the visual object.

To make a connection, you select one of the available messages or commands to connect the object to and choose the Add Function button. Another list box shows the message or command and the name of its handler function.

The resulting connection is called a "command binding." Figure 6.1 shows the ClassWizard dialog box after a command has been connected to a handler. In some cases, a second dialog box appears to let you accept or modify a proposed function name.

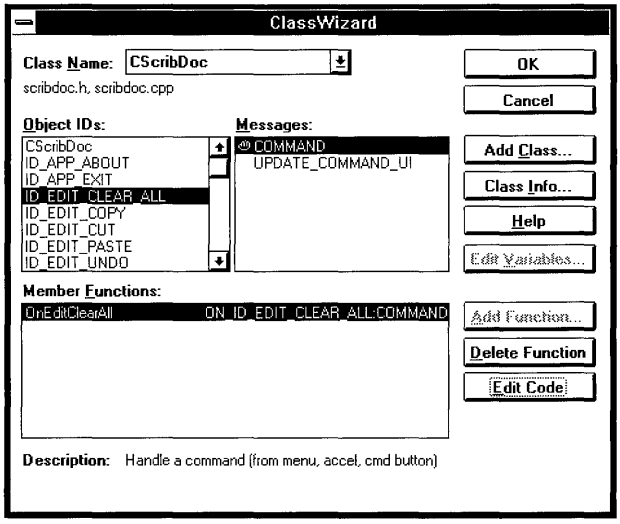


Figure 6.1 ClassWizard Dialog Box

After you create a message-handler function, ClassWizard writes an entry for the command in the chosen class's "message map" and adds a function declaration to the class. Also, ClassWizard writes a function template—a complete member

function definition with an empty function body—in the source files that contain the class. ClassWizard then lets you jump directly to the Visual Workbench editor to fill in the function template. You'll learn more about message maps and message-handler functions later in this chapter.

Note ClassWizard automatically writes its changes to your files to disk. You need to save files explicitly only if you have edited them yourself, as for example when you use the Visual Workbench editor to fill in a handler's code.

You can also use ClassWizard to edit existing message maps and message-handler functions. ClassWizard follows a conservative set of rules in writing to your files, writing only a few kinds of code to predictable places, so it's safe and easy to use.

Warning If you delete a command binding with ClassWizard, its message-map entry is deleted, but the message-handler function, and any references to it in your other code, are not deleted. You must edit those items by hand.

Because you may want to bind commands either while you're editing user-interface objects that generate commands or while you're editing code, you can invoke ClassWizard from the Resource menu in App Studio or from the Browse menu in Visual Workbench.

Typically, you will invoke ClassWizard from App Studio while you're editing menus, dialog boxes, and other user-interface objects. ClassWizard uses the current selection in App Studio to jump quickly to the appropriate class for the selected visual object.

You may want to work something like this: create several menu items or other user-interface objects, call up ClassWizard and connect the first object to a handler, and jump to the Visual Workbench editor to write the handler. Then return to ClassWizard and repeat the process for each of the other objects. Or you may prefer to create the user-interface objects, use ClassWizard to make all of the connections, and then jump to Visual Workbench to fill in all of the handler functions. The tools are flexible enough to support whichever working style you prefer.

Adding Handlers for Commands

Once you've created the user-interface objects—such as menus, accelerators, and toolbar buttons—that generate commands, you must bind them to the code that carries out the commands. This section explains the fundamentals of commands in the framework and explains how to use ClassWizard to bind user-interface objects to commands and commands to code.

Command Fundamentals

Before using ClassWizard to bind commands, you'll need to know more about the related concepts and terms. In addition to this section, you'll find more information about these concepts in Chapter 2 of the *Class Library Reference*.

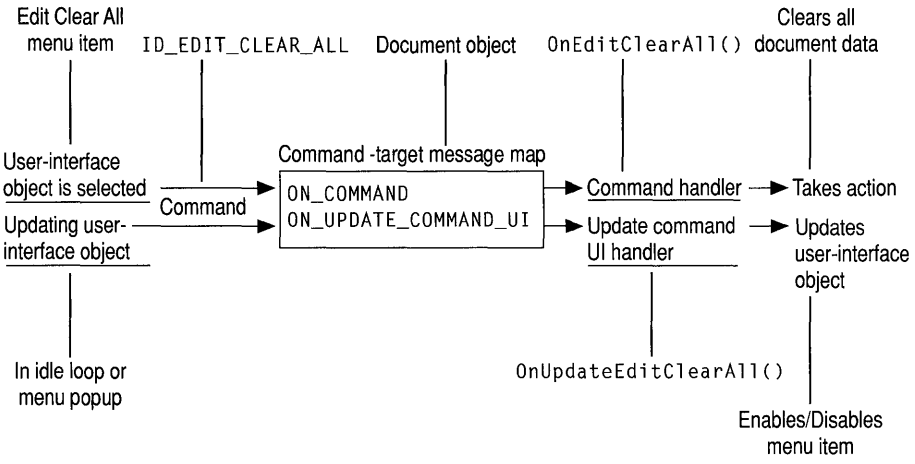


Figure 6.2 Command Architecture

Command Concepts and Terms

Figure 6.2 illustrates the concepts that make up the command architecture of the Microsoft Foundation Class Library. Among the concepts covered in this chapter are:

- Commands, messages, and control notifications

  A command is an instruction to your program to perform a certain action. Unlike a function call, a command is a "message" that is routed to various "command target" objects, each of which has an opportunity to carry out the instruction. Examples of commands include the Open, Save, and Save As commands on the File menu.

  Important To the class library, a command is identical to its ID. A menu item, toolbar button, or dialog box control is bound to a command by giving the item the same ID.

  Commands are based on the **WM_COMMAND** Windows message. Commands can be sent to frame windows, documents, views, the application itself, and other kinds of objects. These objects are discussed as "command targets" on the next page. For more about commands and other Windows messages, see Chapter 2 in the *Class Library Reference*.

- User-interface objects (command generators) and updating

  Menu items, buttons, and similar elements of the user interface can cause Windows to generate commands. The framework routes commands to the other objects in your program that carry them out. Keep in mind that these user-interface objects are not C++ objects.

  You can also use commands to update the visual state of menu items and buttons—for example, enabling them if they're available to the user in a given situation and disabling them if they're unavailable. For more information, see "Updating User-Interface Objects" on page 108.

- Command targets

  Many objects in your program can receive and respond to commands. These objects are called "command targets" and are derived from class **CCmdTarget**. Command-target objects contain their own message maps and message-handler functions. Figure 6.3 shows most of the hierarchy of command-target classes in the class library. For a complete list see *Class Library Reference*, Chapter 1.

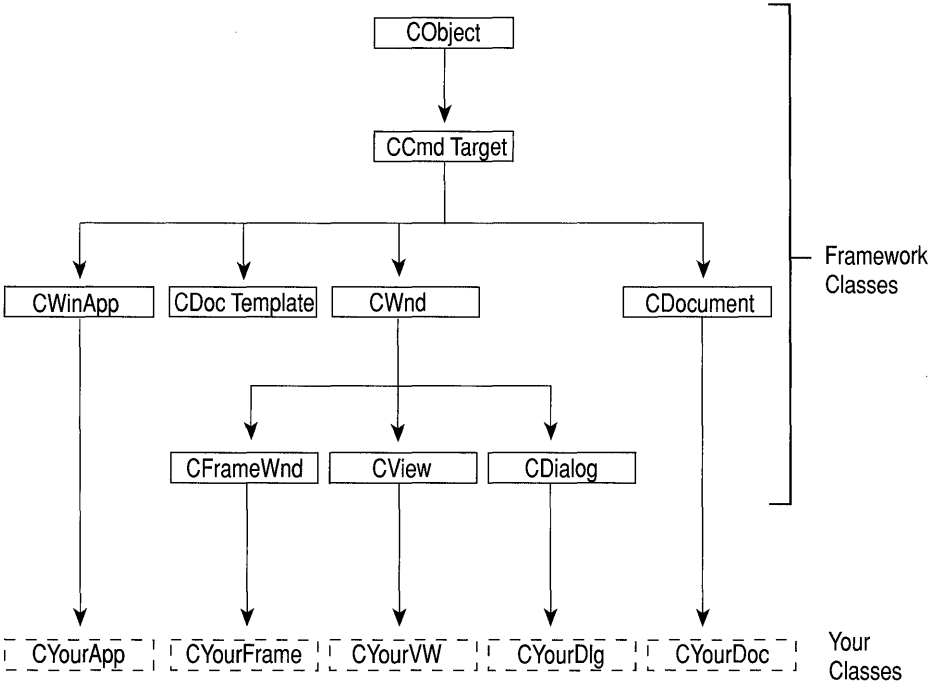


Figure 6.3 Command Target Class Hierarchy

- Command bindings

  A binding associates two things, such as a menu item and the command it invokes. Commands are assigned at one end of the command routing to the user-interface objects that generate them. This is done, for example, by equating the command's ID with the ID of a menu item. At the other end of the routing, commands are mapped to the name of a message-handler function. This is done by connecting the command ID and the name of the function.

  You normally do this task with ClassWizard, which you can invoke from either App Studio or Visual Workbench. Usually, you'll bind some commands as you construct the user interface with App Studio. Later, you'll probably revisit ClassWizard as you work with code in the Visual Workbench editor.

  Note You can assign a command ID to more than one user-interface object. Scribble binds its Thick Line command to both a menu item and a toolbar button. The same message-handler function and message-map entry work for both sources of the command.

- Message maps

  How does a command target know it can handle a command? For that matter, how does any object know it can handle a message? The answer is the message map of the object's class.

  Each command-target object has a message map. Message maps are tables that connect commands with the names of the member functions that handle the commands. These functions are called "message handlers." When a command target object receives a message, the object's message map is used to determine which handler function to call for the message. Message maps are used in dispatching Windows messages.

  You don't typically have to write message-map entries manually—ClassWizard does the job for you. For more information, see "Message Maps" on page 99.

- Message handlers

  Command-target classes have member functions designated to handle any commands to which the target can respond. These functions are called "message handlers." Message handlers are explained under "Message Handlers" on page 100. In this tutorial, the terms "handler," "message handler," and "message-handler function" are equivalent.

  A considerable part of writing an application is writing message-handler functions that determine how a document, view, or other class responds to a command.

- Command routing

The class library provides a standard mechanism for routing commands from the user-interface objects that generate them to the command targets that handle them. This standard routing ensures that your objects receive the commands they need to handle. For details, see "Command Routing" on page 95. Table 6.1, on page 96, illustrates the standard command routing.

Message Maps

Each command-target class defines a message map that contains an entry for each command or other message that the target can handle. Here's a sample from Scribble's document class, `CScribDoc` (this is code that ClassWizard writes, as you'll see shortly):

```
BEGIN_MESSAGE_MAP( CScribDoc, CDocument )
    //{{AFX_MSG_MAP(CScribDoc)
    ON_COMMAND( ID_EDIT_CLEAR_ALL, OnEditClearAll )
    ON_COMMAND( ID_PEN_THICK_OR_THIN, OnPenThickOrThin )
    // ... More entries
    //}}AFX_MSG_MAP
END_MESSAGE_MAP( )
```

ClassWizard places this code in file SCRIBDOC.CPP. The **BEGIN_MESSAGE_MAP** and **END_MESSAGE_MAP** macros bracket the map and provide code that takes care of creating the map at run time. Between the bracketing macros are message-map entries, one per command that the document can handle.

The **BEGIN_MESSAGE_MAP** macro specifies two arguments: the name of the class that the message map belongs to (`CScribDoc` in the example) and the name of the base class from which that class is derived (**CDocument** in the example).

Notice the special comment lines:

```
//{{AFX_MSG_MAP(CScribDoc)
//}}AFX_MSG_MAP
```

The ClassWizard tool uses these comments to locate the message map of a class for which it is binding commands. All message-map entries are written between these comment lines.

Important Use ClassWizard to create and edit all message-map entries. If you add them manually, you may not be able to edit them with ClassWizard later. If you add them outside the bracketing comments, ClassWizard can't edit them at all. You can use this feature to add entries you don't want ClassWizard to touch.

A message-map entry looks like this:

```
ON_COMMAND( ID_PEN_THICK_OR_THIN, OnPenThickOrThin )
```

Each entry has three parts:

- The **ON_COMMAND** macro.
- The command ID; here it would be `ID_PEN_THICK_OR_THIN`.
- The name of the message-handler function for the command; here it's `OnPenThickOrThin`.

The result of this macro is an entry in the message map whose parts connect the command ID to the handler.

When the target object receives a command message, it checks the message map. If the command matches an entry, the message-handler function, a member function of the target class, is called for that object.

The message map has one other required element. In addition to the **BEGIN_MESSAGE_MAP** and **END_MESSAGE_MAP** macros in your .CPP file, you must use the **DECLARE_MESSAGE_MAP** macro in your class declaration in the .H file. It's a convention of the Microsoft Foundation Class Library to put **DECLARE_MESSAGE_MAP** at the end of the class declaration, but it can go anywhere. The macro is not sensitive to C++ access-control keywords. Message maps are always protected; you should specify the access control for any following sections. Keep in mind that AppWizard and ClassWizard do this work for you.

When you began your Scribble development project with AppWizard, all parts of your message maps and the macros listed above had already been created for you except the entries for specific commands such as **ID_EDIT_CLEAR_ALL** and `ID_PEN_THICK_OR_THIN`. You will add those with ClassWizard.

Command Routing

How does a command from a menu or other user-interface object find its handler function? Commands are routed through a standard sequence of command-target objects, one of which is expected to have a handler for the command.

OnCmdMsg

To accomplish this routing, each command target calls the **OnCmdMsg** member function of the next command target in the sequence. Command targets use **OnCmdMsg** to determine whether they can handle a command and, if they can't, route it to another command target.

Each command-target class overrides the **OnCmdMsg** member function. The overrides let each class route commands to a particular next target. A frame

window, for example, always routes commands to its current child window or view. Table 6.1 shows the standard command routing for all command targets.

The default **CCmdTarget** implementation of **OnCmdMsg** uses the message map of the command-target class to search for a handler function for each command message it receives. If it finds a match, it calls the handler. Message map searching is explained in the section "Searching Message Maps" on page 97.

Different command-target classes check their own message maps at different times. Typically, a class routes the command to certain other objects to give them first chance at the command. If none of those objects handle the command, the original class checks its own message map. Then, if it can't supply a handler itself, it may route the command to yet more command targets. Table 6.1 shows how each of the classes structures this sequence. The general order in which a command target routes a command is:

1. To its children
2. To itself
3. To other command targets

How expensive is this routing mechanism? Compared to what your handler does in response to a command, the cost of the routing is low. Bear in mind that commands are generated, and thus routed, only when the user chooses a menu or a button.

Table 6.1 Standard Command Route

When an object of this type receives a command . . .	. . . it gives itself and other command-target objects a chance to handle the command in this order:
Frame window	1. Active child window or view 2. This frame window 3. Application (**CWinApp** object)
View	1. This view 2. Document attached to the view
Document	1. This document 2. Document template attached to the document
Dialog	1. This dialog 2. Window that owns the dialog 3. Application (**CWinApp** object)

If the command is not handled by any object along the routing, the application object passes it to Windows for default processing.

An Example

To illustrate, consider a command message from the Clear All item in Scribble's Edit menu. In Scribble, the handler function for this command happens to be a member function of class `CScribDoc`. Here's how that command reaches its handler after the user chooses the menu item:

1. The main frame receives the command message first.
2. In the case of an MDI frame window, the main frame gives the currently active MDI child window a chance to handle the command.
3. Because of the standard routing, the frame window now gives its view a chance to handle the command before checking its own message map.
4. Unlike the frame, the view checks its own message map first.
5. Finding no handler, the view next routes the command to its associated document.
6. The document checks its message map and, in this case, does find a handler, which gets called—and the routing stops.

If the document did not have a handler, it would route the command next to its document template. Then the command would return to the view and then to the frame window. Finally, the child frame would check its message map. If that check failed as well, the command would be routed back to the MDI frame and then to the application object—the ultimate destination of unhandled commands.

Searching Message Maps

The previous section focused on commands, a particular category of Windows messages. There are two other categories of messages: standard Windows messages (with the **WM_** prefix) and control-notification messages (such as **BN_CLICKED**, which is sent by a button).

The **OnCmdMsg** routing mechanism uses message maps in command-target classes to search for a handler function for each command message. Message maps are also used to locate handlers for standard Windows messages and control-notification messages, the other categories of Windows messages. In Chapter 4, you saw how the view's message map is used to map three mouse-related Windows messages to their handlers in the view. This section explains how message maps are searched.

Windows Messages

Standard Windows messages and control-notification messages are sent only to windows. Messages in these categories are never routed like commands from one

command target to another. Standard messages are sent to frame windows, dialog boxes, views, and other kinds of windows. Control-notification messages are sent from controls (child windows) to their parent windows.

All windows have message maps. (As do all command-target objects.) Windows can handle a wider variety of messages in their message maps then other command targets such as documents, document templates, and the application object.

The Search

When a window receives an incoming Windows message (not a command), it searches its message map for a corresponding handler function. However, the phrase "searches its own message map" needs a little clarification.

It's possible to make a window an object of class **CWnd** directly, without deriving a new class. But in most cases the window receiving a message is an object of some class derived from **CWnd**. For example, Scribble's main application window class, `CMainFrame`, is derived from **CFrameWnd**, which in turn is derived from **CWnd**. By the nature of C++ class derivation, a `CMainFrame` *is* a **CFrameWnd**. It also *is* a **CWnd**. Because of this, the handler functions for some of the messages that a `CMainFrame` receives might actually be defined in **CFrameWnd** or even **CWnd**.

How does a `CMainFrame` find a message handler defined not in `CMainFrame` itself but in one of its base classes? The answer is that part of the definition of any message map, including `CMainFrame`'s, is the name of the immediate base class. Recall the way you use the **BEGIN_MESSAGE_MAP** macro:

```
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
```

The base class named in the second argument gives the receiving window a way to locate the message map of the base class. That message map, in turn, provides the name of the next-higher base class, and so on.

Thus, when it receives a standard Windows message, a `CMainFrame` is able to search not only the `CMainFrame` message map but also the message maps defined for **CFrameWnd** and **CWnd**. Figure 6.4 illustrates the search process.

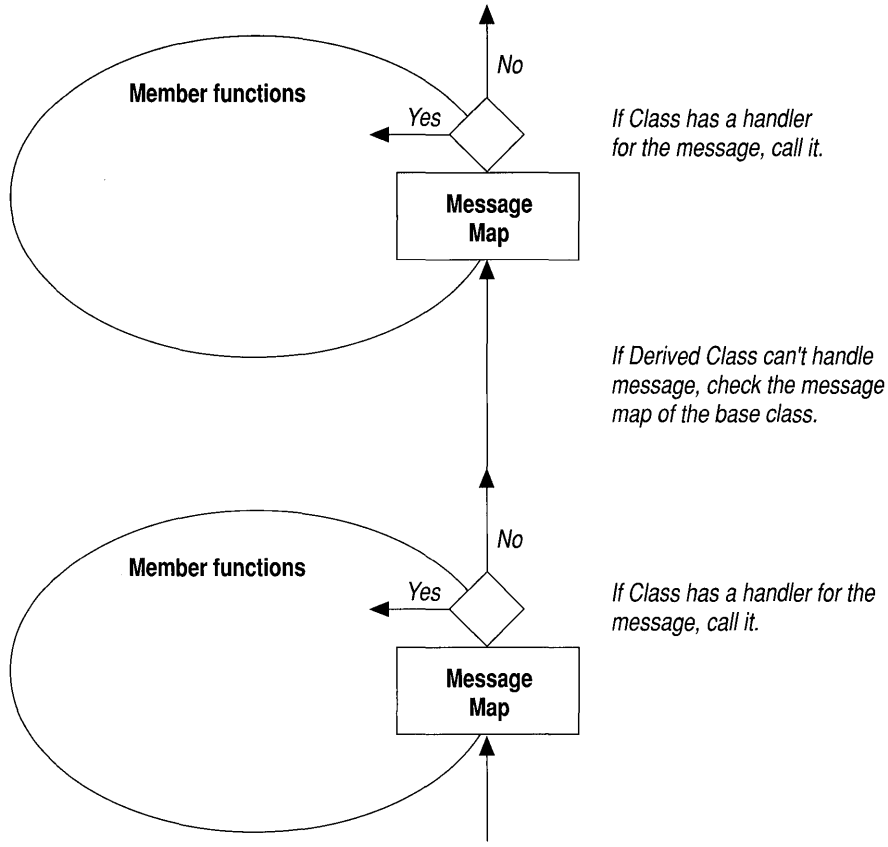


Figure 6.4 Searching Message Maps

There are a few exceptions to the rule that a window handles noncommand messages itself. Control-notification messages, scrolling messages, and a few **WM_** messages are actually delegated to the control that sent the message or to the view for possible handling before the parent window checks its message map(s). This is done to support self-drawing controls, VBX controls, and scrolling views.

Command-Target Message Maps

Because command-target classes other than windows also have message maps, you might expect them to be searched in the same way. Indeed they are. The difference, of course, is that the only messages being searched for in this case are commands. When the search will take place in a given command target depends on the standard command routing and how the command target's class implements its **OnCmdMsg** override. But the search still encompasses not only the message map of the command-target class but also of its base class(es).

For example, in Scribble, when a `CScribDoc` searches its message map—at the appropriate point in the routing, as shown in Table 6.1—it also searches the message map of **CDocument**, the base class of `CScribDoc`, if needed.

Message Handlers

A message handler is a member function of a command-target class. Its purpose is to respond when called through the message map.

You declare message handlers as member functions of your classes. For example, in Chapter 4 you saw three message-handler functions for mouse actions: `OnLButtonDown`, `OnMouseMove`, and `OnLButtonUp`. These were declared as members of class `CScribView`, Scribble's view class. In this chapter, you'll see handlers for several commands, including the Clear All and Thick Line commands, declared as members of `CScribDoc`, the document class.

Chapter 2 in the *Class Library Reference* explains some rules for the names and parameter signatures of message-handler functions. In this chapter you'll see how to create message handlers from ClassWizard.

For more information about commands, messages, message maps, and the command architecture in the Microsoft Foundation Class Library, see Chapter 2 in the *Class Library Reference*.

Binding Scribble's Commands

This section explains the issues and procedures involved in binding Scribble's Clear All and Thick Line commands to their handlers using ClassWizard. (The Pen Widths command is bound in the next chapter.)

Which Command-Target Class Gets the Handler?

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

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

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

the data. Redrawing the view afterwards is secondary. Hence, it makes sense to locate the `OnEditClearAll` handler in the document.

Scribble's Thick Line command is more interesting. This command toggles the current value of a line thickness variable between thick and thin. Should the handler for Thick Line be located in the view because it affects how Scribble's data is drawn? This seems reasonable, but consider what happens when, in a later chapter, Scribble gets splitter window functionality. In that case, each pane of the splitter window is really a new view on the same data. Should each of these views house its own line thickness information (and its own pen)? It seems a better solution to store that information in the document instead, where all of the views can access it. Keep in mind that this is a decision specific to Scribble's user interface, where it's desirable that the pen width commands apply to all views, not just the one with the current focus. You might choose to organize things differently in another application.

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

Here are some guidelines that may help:

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

Bind Scribble's Clear All Command

As discussed above, Scribble's Clear All command is bound to the document class. If you're working along, use the following procedure:

▶ To bind Scribble's Clear All command

1. Invoke ClassWizard (from App Studio or from Visual Workbench).
2. Use the Class Name list box in the ClassWizard dialog box to select the `CScribDoc` class.

   If you invoke ClassWizard from App Studio, the currently selected object, such as a menu or dialog resource, is automatically selected in ClassWizard.

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

After you select a class, the Object IDs list box shows all of the visual objects managed by the class—the available items that can be mapped to functions. These might include controls (for a dialog resource) and commands from menus and accelerator tables. The list may also include class names.

3. In the Object IDs list box, select the **ID_EDIT_CLEAR_ALL** command.

   You see **COMMAND** and **UPDATE_COMMAND_UI** in the Messages list box. For commands, these are always the choices you see. In other cases, you might see other things listed—a list of Windows messages, for example, when the selected item is the name of a window or view class.

4. In the Messages list box, select **COMMAND**.

   Later in the chapter, you'll see how **UPDATE_COMMAND_UI** is used.

   Figure 6.5 shows the selections from steps 2, 3, and 4.

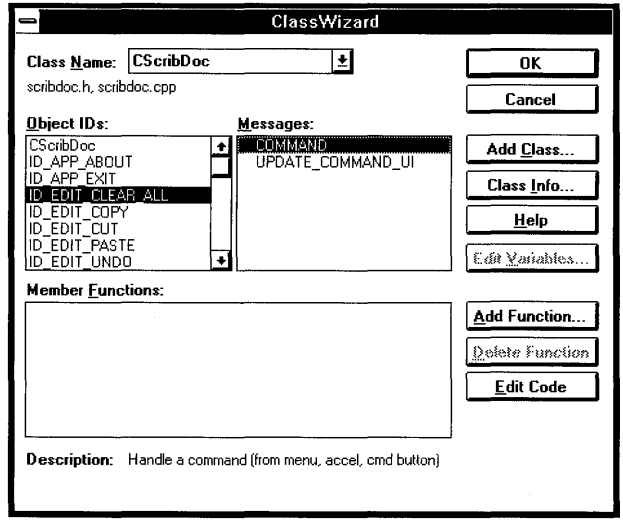


Figure 6.5 Clear All in ClassWizard

5. Choose the Add Function button.

   This brings up a dialog box with a proposed name for the new handler function.

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

   Although you could change that name, don't. The name fits the handler's functionality and its connection to the menu item very well. The proposed name is synthesized from the command name and message type.

   Several things are added to your source files when you add a member function:

   - A message-map entry is added to the class's message map (in the .CPP file for the class).

- A member function declaration is added to the class declaration (in the .H file for the class).
- A member function template definition is added to the .CPP file.

These changes are made to your source files after you finish editing the class.

After you add the function, its name appears in the Member Functions list box beside the ID to which it maps.

7. In the ClassWizard dialog box, select the new handler's name, `OnEditClearAll`, from the Member Functions list box.
8. Choose the Edit Code button.

   The Visual Workbench editor opens file SCRIBDOC.CPP with the function template for `OnEditClearAll` displayed. Figure 6.6 shows the function definition template.

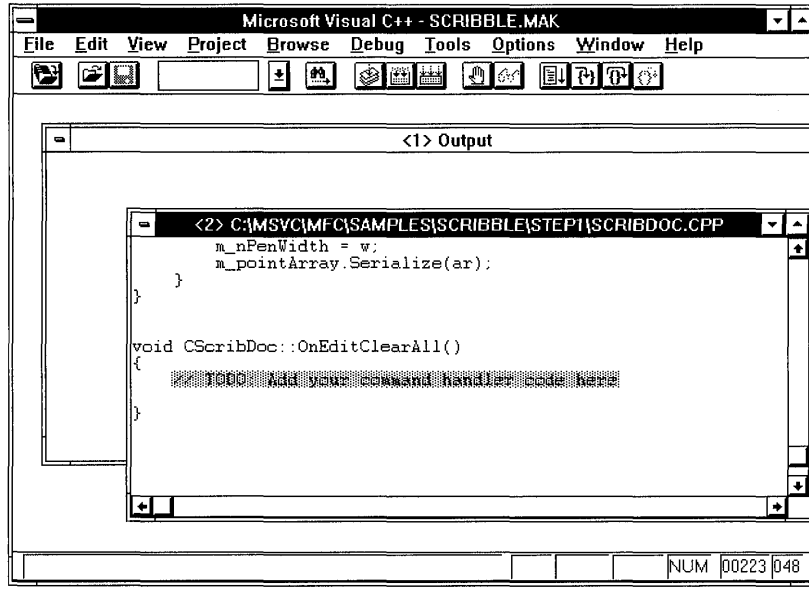


Figure 6.6 The OnEditClearAll Function Template

9. Fill in the `OnEditClearAll` message-handler function.

   The function looks like this (you add the lines marked with ▸):

```
void CScribDoc::OnEditClearAll( )
{
    DeleteContents( );
    SetModifiedFlag();
    UpdateAllViews( NULL );
}
```

SetModified Flag is a member function of class **CDocument**. It marks the document as changed so the framework won't prompt the user to save the document when it closes.

10. In the File menu choose the Save command to save changes to the SCRIBDOC.CPP file.

    This is the only file you changed by adding the new member function.

How do the commands work? The new message handler first calls `DeleteContents` to destroy the document's stroke data. (This version of `DeleteContents`, from Chapter 4, overrides **CDocument**'s **DeleteContents** member function.) Then `OnEditClearAll` calls the **UpdateAllViews** member function inherited from **CDocument** to cause all views of the data to be updated. The document's view is redrawn, this time with no data. **UpdateAllViews** takes a **NULL** argument because the document is modifying itself. The parameter is normally used to pass a pointer to the view that modified the document, but that doesn't apply here.

The `DeleteContents` member function iterates through the list of strokes. For each stroke, it gets the next stroke and invokes the **delete** operator on it. For more information about working with list classes, see Chapter 13, "Collections."

When you finish adding `OnEditClearAll`, you're still in the Visual Workbench editor. To continue binding commands, invoke ClassWizard again from Visual Workbench's Browse menu.

Bind Scribble's Thick Line Command

Like the Clear All command, the Thick Line command will be handled by the document. Recall the discussion under "Which Command-Target Class Gets the Handler?" on page 100.

▶ To bind Scribble's Thick Line command

1. If you're not in ClassWizard, invoke it from Visual Workbench's Browse menu.
2. Select the `CScribDoc` class.
3. In the Object IDs list box, select the **ID_PEN_THICK_OR_THIN** command.
4. In the Messages list box, select **COMMAND**.
5. Choose the Add Function button.

6. In the Add Member Function dialog box, accept the name `OnPenThickOrThin` by choosing the OK button.
7. In the ClassWizard dialog box, choose the Edit Code button.

   If Visual Workbench is not running, you get a warning. Start Visual Workbench from the Windows Program Manager.
8. Fill in the `OnPenThickOrThin` message-handler function.

   The function looks like this (you add the lines marked with ▸):

```
void CScribDoc::OnPenThickOrThin( )
{
▸    // Toggle the state of the pen between thin and thick.
▸    m_bThickPen = !m_bThickPen;

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

   The `OnPenThickOrThin` message handler first toggles the state of a Boolean variable, `m_bThickPen`. If the variable is now TRUE, the pen will be thick. Otherwise, it will be thin. Then the handler calls a helper function, `ReplacePen`, to reset the current pen to the new width.
9. Next, use the Visual Workbench editor to add `ReplacePen` to file SCRIBDOC.CPP (`ReplacePen` is not a message handler, so you don't add it with ClassWizard). Here it is:

```
// OnPenThickOrThin, then ...
▸void CScribDoc::ReplacePen( )
▸{
▸    m_nPenWidth = m_bThickPen ? m_nThickWidth : m_nThinWidth;
▸    // Change the current pen to reflect the new width.
▸    m_penCur.DeleteObject( );
▸    m_penCur.CreatePen( PS_SOLID, m_nPenWidth, RGB(0,0,0) );
▸}
```

   The `ReplacePen` member function uses the C++ conditional operator (**?:**) to determine the pen width and return its value. Then it calls the **DeleteObject** member function of the current pen object and creates a new solid black pen with **CreatePen**, setting its attributes, including its width.
10. Besides adding the `ReplacePen` function definition to SCRIBDOC.CPP, you must also add a function prototype to the `CScribDoc` class declaration in file SCRIBDOC.H. If you're working along, you can open that file with Visual Workbench, locate the class declaration as partially shown immediately following, and add the marked line:

```
class CScribDoc : public CDocument
{
protected: // Create from serialization only.
...
// Attributes
...
// Operations
public:
   void DeleteContents( );
...
// Implementation
protected:
   void ReplacePen( );
...
};
```

11. To add this improved way of updating the pen, locate the `InitDocument` member function in SCRIBDOC.CPP and change the function to match this code:

```
void CScribDoc::InitDocument()
{
   ReplacePen();   // Initialize pen according to current width
}
```

To match the function as shown, besides adding the marked line you must delete the following lines (added in Chapter 4):

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

The line you added calls `ReplacePen` to set up the pen with its new width.

12. Choose the Save All command on Visual Workbench's File menu to save changes to both the SCRIBDOC.H and SCRIBDOC.CPP files.

Bind the Toolbar Button to the Thick Line Command

Scribble's Thick Line command is now bound to the Thick Line toolbar button as well as to the Thick Line menu item. Either user-interface object generates precisely the same command. This duplication is accomplished simply by giving the menu item (above) and the button (as in Chapter 5) the same ID as the command: **ID_PEN_THICK_OR_THIN**.

Add New Member Variables to Scribble

In addition to storing the current pen width in `m_nPenWidth`, class `CScribDoc` needs to keep track of whether the pen is currently thick or thin and how "thick"

and "thin" are defined (in pixels). In Chapter 7, you will add code to allow the user to define these values with a dialog box. For now, defaults are hard coded.

▶ **To add the new data members, use the Visual Workbench editor, as shown in Figure 6.7**

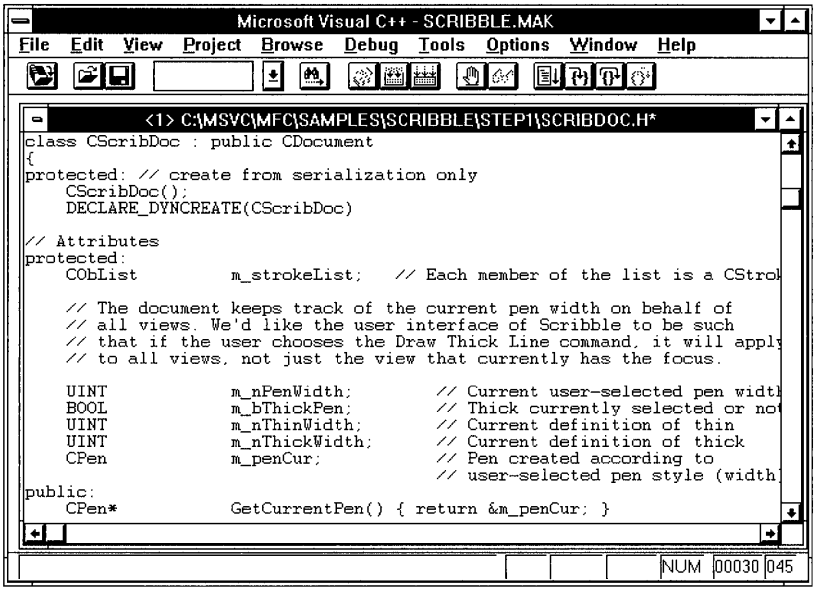


Figure 6.7 Adding the Data Members

1. Open the file SCRIBDOC.H.
2. Find the class declaration for class `CScribDoc`.

   It begins:

   ```
   class CScribDoc : public CDocument
   { ...
   ```

   Locate the section labeled "`// Attributes:`".
3. Add the following marked lines after the **protected** keyword and the existing `m_strokeList` and `m_nPenWidth` declarations:

```
// Attributes:
public:
protected:
CObList    m_strokeList;   // Each member of list is a stroke
// ... comments ...
UINT    m_nPenWidth;   // Current user-selected pen width
BOOL    m_bThickPen;   // Thick currently selected or not
UINT    m_nThinWidth;  // Current definition of thin
UINT    m_nThickWidth; // Current definition of thick
CPen    m_penCur;      // Pen created according to
                       // user-selected pen style (width)

// Additional code ...
```

4. Add the following lines to `InitDocument` in SCRIBDOC.CPP:

```
void CScribDoc::InitDocument()
{
   m_bThickPen = FALSE;
   m_nThinWidth = 2;     // Default thin pen is 2 pixels wide
   m_nThickWidth = 5;    // Default thick pen is 5 pixels wide
   ReplacePen();   // Initialize pen according to current width
}
```

   The added lines specify that the pen is initially thin and define the meanings of "thin" and "thick."

5. Save files SCRIBDOC.H and SCRIBDOC.CPP.

Library Support for Writing Message Handlers

To implement your handlers, call upon the many services and classes provided by the Microsoft Foundation Class Library.

The classes, supplemented by standard C run-time functions and Windows Application Programming Interface (API) functions that aren't encapsulated by the library, let you use object-oriented programming methods in your message handlers and customize the functionality to suit your needs. For more information about these classes, see Chapters 1 through 6 in the *Class Library Reference*. The general-purpose (non-Windows) classes are also covered in more detail in the later chapters of this manual.

Updating User-Interface Objects

When a menu drops down in your application, the user expects to see some menu items enabled (available) and others dimmed (grayed to show they're unavailable) depending on the current context. Some menu items may have a check mark. Similarly, the user expects to see some toolbar buttons enabled and others disabled

and perhaps checked (depressed). The framework provides a direct, command-based way to set the state of the menus and toolbar buttons as conditions in the program change.

Update a Command's User Interface

To update a menu in traditional programming for Windows, you get a handle to the menu and call the Windows **EnableMenuItem** function. Under the Microsoft Foundation Class Library, however, you can use messages to let the most appropriate command-target object update the menu status automatically. This delegates the task of enabling and disabling menu items and buttons to the object that possesses the most contextual information relating to the menu or button.

The idea is to search the default command routing for objects that can enable or disable menu items before the user actually sees the menu. (Updating for buttons will be explained shortly.)

The command-routing mechanism is fast and costs you very little as long as the update handlers you write don't do a lot of calculation.

Update Menu Items

When the user clicks the mouse in the menu bar, the framework—before showing the menu items—sends out command update messages to the command targets. A message is sent for each item on the menu. The important thing to realize is that all items on the menu are updated before the menu drops down and the user sees it.

Figure 6.2 on page 91 shows how menu items are updated and menu commands are carried out.

Messages to update user-interface objects are routed and handled like commands. The framework searches the standard command routing as if the command was actually being executed. If it finds a message-map entry for the initialization message, it passes a pointer to a **CCmdUI** object representing the menu item—or other user-interface object—to the command target. The target should use this pointer in its command user-interface handler to update the menu.

The items on a particular menu might be handled by one or several command-target objects. If you've mapped the items to handlers, the handlers will be found along the routing and called to update the items' state. Items with no handlers are not updated.

To have your application handle user-interface initialization messages, your command-target objects—application, windows, views, documents—in general must do two things:

1. Use ClassWizard to include an **ON_UPDATE_COMMAND_UI** entry in their message maps with arguments for the command ID and the name of the message-handler function.

   Usually you'll pair this message-map entry with one for carrying out the command. For example, Scribble's document includes message-map entries for

   ```
   ON_UPDATE_COMMAND_UI( ID_PEN_THICK_OR_THIN, OnUpdatePenThickOrThin )
   ON_COMMAND( ID_PEN_THICK_OR_THIN, OnPenThickOrThin )
   ```

   and

   ```
   ON_UPDATE_COMMAND_UI( ID_EDIT_CLEAR_ALL, OnUpdateEditClearAll )
   ON_COMMAND( ID_EDIT_CLEAR_ALL, OnEditClearAll )
   ```

   The way **ON_UPDATE_COMMAND_UI** is handled is almost identical for the Clear All and Thick Line commands except that for Clear All the menu item is enabled or disabled, while for Thick Line the menu item is checked or unchecked.

2. Provide a message-handler function that does the menu updating.

   The `OnUpdatePenThickOrThin` handler uses its pointer to a **CCmdUI** object to call an **Enable** member function. An object of class **CCmdUI** is associated with each menu item (and toolbar button). This object provides an interface through which to update the menu or button. The framework passes your update-handler function a pointer to the object.

   Table 6.2 summarizes the command and message-handler naming conventions used in the source-code files of the class library and sample applications, including the conventions for user-interface updating.

Table 6.2 Command and Message-Handler Naming Conventions

Item	Convention	Example
Command ID	ID_XXX	ID_EDIT_CLEAR_ALL
Message handler	OnXxx	OnEditClearAll
Command UI handler	OnUpdateXxx	OnUpdateEditClearAll

Update Toolbar Buttons

Like menus, toolbar buttons show their availability or status (some controls have more than two possible states) with some kind of visual feedback. You can use almost the same update mechanism to update these user-interface objects.

Menu initialization is automatic in the framework, occurring when the application receives a **WM_INITMENUPOPUP** message. For buttons, searching the command routing is done in the idle loop. When there are no pending messages in its message queue, a window enters an idle loop. During the idle loop, the command routing is searched for buttons in much the same way as for menus. (You can also do other useful processing in the idle loop. See "Idle Loop Processing" in Chapter 2 in the *Class Library Reference* and the **OnIdle** member function of class **CWinApp**.)

The CCmdUI Structure

The **CCmdUI** structure defined by the class library supplies a common interface to both menus and controls. That is, for either kind of user-interface object, you can call the same member functions of **CCmdUI** to update the object.

This common program interface is one major advantage of initializing your user-interface objects with **ON_UPDATE_COMMAND_UI**. You can, for example, readily replace or duplicate a menu command with a command generated by a toolbar button. The same message-map entry and handler function in the command target will work.

Another major advantage of initializing user-interface objects using **CCmdUI** is that much of the process is automated and all commands are handled in a uniform way. The same object handles both the command and updating of the user-interface object.

Note For advanced programmers: **CCmdUI** also works for indicators in status bars and normal buttons in dialog bars.

Update Scribble's Clear All Menu Item

This section presents the steps you will take to prepare the code required to update the Clear All menu item on Scribble's Edit menu. This command is handled by the document object, which has the necessary information on whether there are any strokes in the current drawing to clear. If you're working along, the following procedure guides you through the process.

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

1. Invoke ClassWizard.
2. Select class `CScribDoc`.
3. In the Object IDs list box, select the **ID_EDIT_CLEAR_ALL** command.
4. In the Messages list box, select **UPDATE_COMMAND_UI**.

Figure 6.8 shows the selections made in steps 2, 3, and 4:

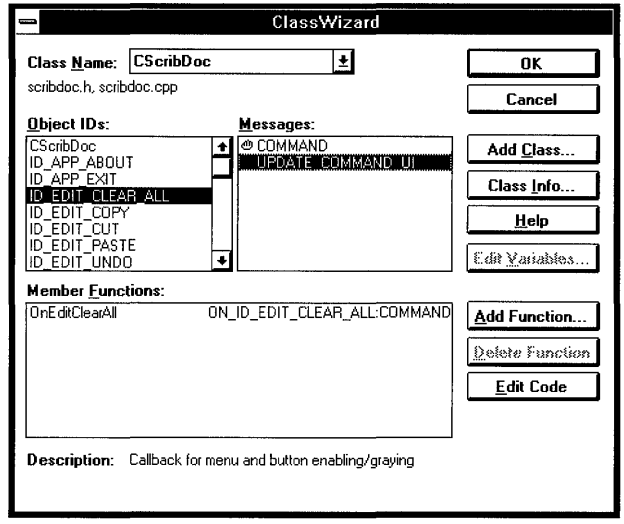


Figure 6.8 ClassWizard Selections for OnUpdateEditClearAll

5. Choose the Add Function button.
6. In the Add Member Function dialog box, accept the name `OnUpdateEditClearAll` by choosing the OK button.
7. In the ClassWizard dialog box, choose the Edit Code button.
8. Fill in the `OnUpdateEditClearAll` update handler function when the Visual Workbench editor opens.

   The function looks like this (you add the marked lines):

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

9. Save changes to file SCRIBDOC.CPP.

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

The pointer to a **CCmdUI** object, `pCmdUI`, is used to access a **CCmdUI** member function, **Enable**. **Enable** takes one Boolean argument. In this code, the expression `!m_strokeList.IsEmpty( )` evaluates to zero if the document has at least one

stroke to clear. If the expression evaluates to nonzero (no strokes), the menu item is disabled (and dimmed or grayed).

Note When the user pulls down a menu, the update handlers for all items on the menu are called before the user sees the menu displayed. It's important, then, to make your update handlers fast.

When you add an update command for the ClearAll menu item, ClassWizard also writes a message-map entry in the document's message map in file SCRIBDOC.CPP that looks like this:

```
BEGIN_MESSAGE_MAP( CScribDoc, CDocument )
   ON_UPDATE_COMMAND_UI( ID_EDIT_CLEAR_ALL, OnUpdateEditClearAll )
   // Other message-map entries
END_MESSAGE_MAP( )
```

The **ON_UPDATE_COMMAND_UI** macro resembles the **ON_COMMAND** macro that you saw earlier for the `OnEditClearAll` message handler.

In addition, ClassWizard adds a new member function declaration for `OnUpdateEditClearAll` to the `CScribDoc` class declaration in file SCRIBDOC.H. The function declaration looks like this:

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

Update Scribble's Thick Line Menu Item

Updating the Thick Line menu is very similar to updating the Clear All menu. In this case, however, rather than enabling or disabling the menu item, the handler puts a check mark beside the item or removes an existing check mark. If you're working along, do the following procedure.

▶ **To add an update handler for the Thick Line menu**

1. Invoke ClassWizard.
2. Select class `CScribDoc`.
3. In the Object IDs list box, select the **ID_PEN_THICK_OR_THIN** command.
4. In the Messages list box, select **UPDATE_COMMAND_UI**.
5. Choose the Add Function button.
6. In the Add Member Function dialog box, accept the name `OnUpdatePenThickOrThin` by choosing the OK button.
7. In the ClassWizard dialog box, choose the Edit Code button.
8. Fill in the `OnUpdatePenThickOrThin` update handler function when the Visual Workbench editor opens.

The function looks like this (you add the marked lines):

```
void CScribDoc::OnUpdatePenThickOrThin( CCmdUI* pCmdUI )
{
   // Add check mark to Pen Thick Line menu item if the current
   // pen width is "thick."
   pCmdUI->SetCheck( m_bThickPen );
}
```

9. Save changes to file SCRIBDOC.CPP.

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

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

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

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

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

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

```
BEGIN_MESSAGE_MAP( CScribDoc, CDocument )
 ON_UPDATE_COMMAND_UI(ID_PEN_THICK_OR_THIN,OnUpdatePenThickOrThin)
 // Other message-map entries
END_MESSAGE_MAP( )
```

ClassWizard also adds a member function declaration to the document class declaration in file SCRIBDOC.H:

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

Compiling the New Scribble

How does Scribble behave with these new commands in place? Compile the new step-2 version of Scribble and find out.

Run the new version of Scribble from the Project menu.

Draw some strokes with the default thin pen. Then change the line thickness with the Thick Line command on the Pen menu and draw some new strokes. Clear all strokes from the drawing with the Clear All command on the Edit menu. Figure 6.9 shows this version of Scribble.

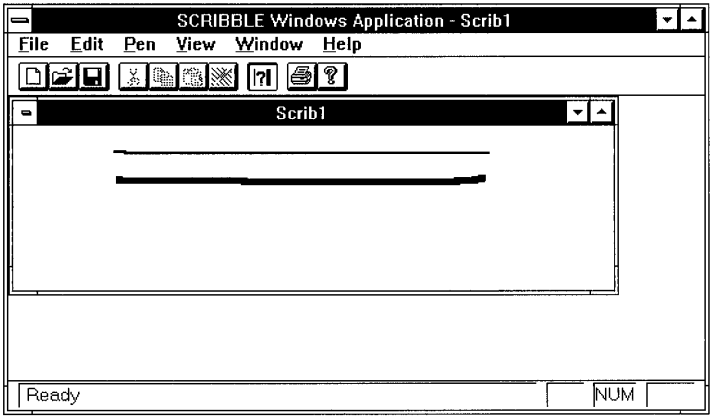


Figure 6.9 Scribble Step 2

Exit Scribble.

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

In the next chapter, you'll implement a command that invokes a dialog box and then processes the results in its message handler.

CHAPTER 7

Adding a Dialog Box

Chapters 5 and 6 added new commands to Scribble in two steps: first, by using App Studio to add new menu items; and second, by using ClassWizard to define message handlers and bind them to the commands. Recall that in Chapter 5, you added menu items for three new commands: Edit Clear All, Thick Pen, and Pen Widths. In contrast, Chapter 6 discussed binding only the first two of these commands.

The reason for this omission is that the Pen Widths command is somewhat different from the other two commands. Both the Edit Clear All and Thick Pen commands execute to completion as soon as the user selects them. By contrast, the Pen Widths command requires more information from the user. The command invokes a dialog box, one that lets the user specify exactly how thin the Thin Pen should be and how thick the Thick Pen should be for subsequent drawing. Before you can write a message handler for this command, you have to design the dialog box that it invokes and define a new class to manage the dialog box. That's what you'll do in this chapter.

This chapter develops a modal dialog box using the same general procedure that chapters 5 and 6 used for adding menu commands: first using App Studio to design the dialog box's appearance, and then using ClassWizard to define message handlers and bind them to the dialog box. Along the way, you'll see a feature of ClassWizard that greatly simplifies the process of gathering data from a dialog box and checking the data's validity.

This chapter describes the following topics:

- Designing a dialog box using App Studio.
- Using ClassWizard to connect a class to a dialog box.
- Invoking a dialog box from your application.

This chapter covers step 3 of Scribble. If you want to work along, adding the code as you go, begin with the files from Chapter 6 in your SCRIBBLE\MYSCRIB subdirectory. At this point, these files should closely resemble those in the SCRIBBLE\STEP2 subdirectory. As you read the chapter, perform all ClassWizard

steps and add all the code that's marked with the symbol ▸. At the end, your files should closely resemble the files in the SCRIBBLE\STEP3 subdirectory.

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

Designing a Dialog Box

Figure 7.1 shows the Pen Widths dialog box that you will create.

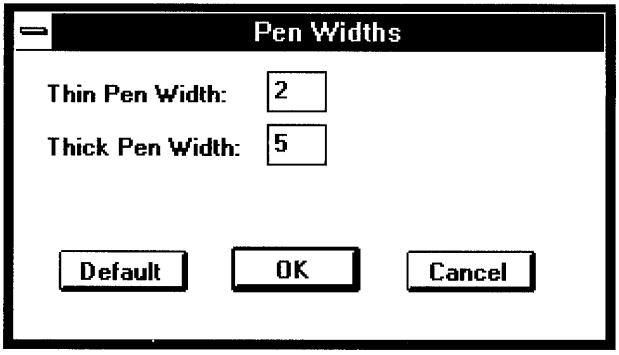


Figure 7.1 Scribble's Pen Widths Dialog Box

The Pen Width dialog box will have the following behavior: for either the Thin or Thick Pen width, the user can enter any number between 1 and 20. If the user enters a value outside of this range, Scribble displays a message box stating the legal range; after dismissing the message box, the user can enter new values. To reset the pen widths to their default values, the user chooses the Default button. To use the currently displayed widths for any subsequent drawing, the user chooses the OK button. To cancel the operation, the user chooses the Cancel button.

Adding the Controls

App Studio provides a graphical editor for designing the appearance of a dialog box. This editor displays a palette of available controls (such as radio buttons, check boxes, and pushbuttons) and an empty dialog box, which is the starting point for the dialog box you're designing. You select controls from the palette and position them on your dialog box. You can move the controls around or resize them directly using the mouse.

There are two steps in designing a dialog box: the first is adding the controls you want, and the second is editing the captions, IDs, and other properties for the controls.

▶ **To add controls to the Pen Widths dialog box**

1. With your Scribble project open in Visual Workbench, choose the Open command from the File menu and load the resource file SCRIBBLE.RC. This launches App Studio, and a resource browser window appears.
2. Choose the New button to create a new resource.
3. Select Dialog from the list of resource types and choose OK. A dialog editor window appears displaying a dialog box that contains two buttons labeled OK and Cancel. The palette of the available controls appears nearby.
4. Add two edit controls. You can add and reposition controls using the "drag and drop" method described in Chapter 5.
5. Add two static text controls to contain the descriptions for the two edit controls. Again, you can use the drag and drop method.
6. Add a third pushbutton to the ones already present.

Modifying the Controls' Properties

To customize the captions and IDs for the controls you've added, you must open the App Studio property page for each control. In Chapter 5, you saw that menus and individual menu items have property pages; in the same way, dialog boxes and dialog controls have property pages describing their attributes.

▶ **To edit the captions and IDs for the controls in the Pen Widths dialog box**

1. If the property page is not currently displayed, double-click the dialog box itself and then click the push-pin button on the property page to keep it open. If the property page is already displayed, click the dialog box to display its properties. Change its ID to IDD_PEN_WIDTHS and change its caption to "Pen Widths."
2. Click the first edit box to display its property page. Change its ID to IDC_THIN_PEN_WIDTH. Then click the second edit box to display its property page. Change its ID to IDC_THICK_PEN_WIDTH.
3. Click the first text box to display its property page. Change the caption of the text box to read "Thin Pen Width:". Then bring up the property page for the other text box, and change its caption to read "Thick Pen Width:". Resize each text box so that the entire caption is visible; you can do this by dragging on the sizing handles on the sides of the text box.

   You won't have to refer to the IDs of the text boxes, so you can leave them with their default values (both have the value IDC_STATIC).
4. Click the pushbutton you added to display its property page. Change its caption to "Default" and its ID to IDC_DEFAULT_PEN_WIDTHS.

Note that App Studio has predefined the OK and Cancel buttons. If you want to look at the property pages for these buttons, click on them in turn. They have **IDOK** and **IDCANCEL**, respectively, as their IDs. Notice that the OK button has the Default Button check box chosen; this makes the OK button the default if the user immediately presses the ENTER key.

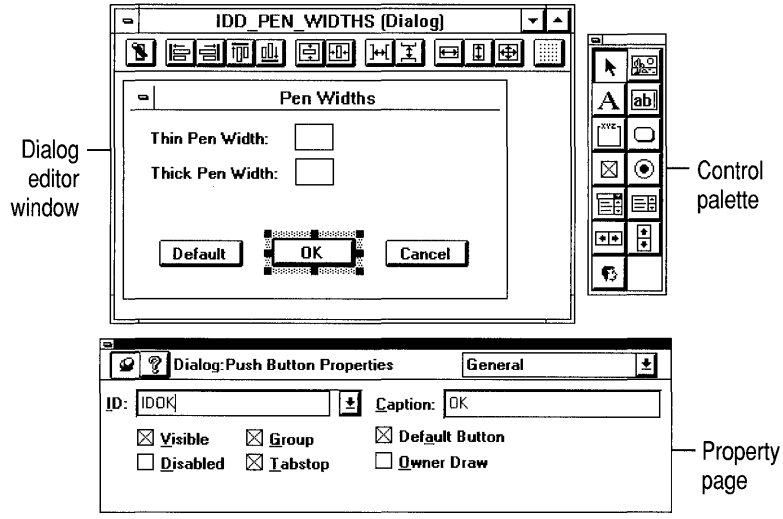


Figure 7.2 Designing the Pen Widths Dialog Box with App Studio

You can clean up the dialog box's appearance by selecting one or more of the controls and using the commands on the Layout menu to align them, make them the same size, etc. If you want to see what the dialog box will look like when it's actually invoked, choose the Test command from the Resource menu. This displays the dialog box as it will appear in Scribble. Exit Test mode by choosing either the OK or Cancel button on the dialog box or by pressing the ESC key.

When you're satisfied with the way your dialog box looks, choose the File Save command. App Studio saves the template for your dialog box in the .RC file that you're working in, SCRIBBLE.RC in this case.

For more information about editing dialog boxes with App Studio, see Chapter 3 of the *App Studio User's Guide*.

Connecting a Class to a Dialog Box

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

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

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

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

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

Declaring the Class

As you saw in Chapter 6, ClassWizard lets you connect a graphical object (such as a menu command or a toolbar button) to a class. The commands described in that chapter are handled by the `CScribDoc` class, the document class generated by AppWizard.

AppWizard doesn't provide any classes for dialog boxes, so you must declare a new class to control the Pen Widths dialog box.

▶ To declare a new dialog class

1. At this point, the App Studio dialog editor window is still open, displaying the Pen Width dialog box. Pull down the Resource menu and select the ClassWizard command. The ClassWizard dialog box appears, and then on top of it the Add Class dialog box appears. ClassWizard knows that a class hasn't been defined yet, so it displays this dialog box to allow you to define one.
2. For the class name, enter "CPenWidthsDlg." Notice that the class type is already set to "CDialog," which is the type ClassWizard assumes because you were using the dialog editor.
3. For the header file, ClassWizard offers the candidate name "penwidth.h"; it has created this name based on the class name. Change the name to "pendlg.h."
4. For the implementation file, change the name to "pendlg.cpp."
5. Choose the Create Class button. The ClassWizard dialog box regains the focus, now displaying the name "CPenWidthsDlg" and the IDs for the controls in the Pen Widths dialog box, as shown in Figure 7.3.

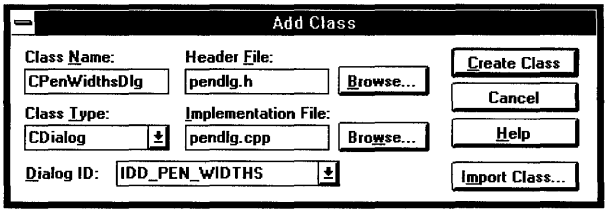


Figure 7.3 The Add Class Dialog Box

In the previous chapters, you've been adding code to the SCRIBDOC.H/CPP and SCRIBVW.H/CPP files exclusively, which were created initially by AppWizard. In this chapter you will work with two new files: PENDLG.H and PENDLG.CPP, which you specified in the Add Class dialog box. ClassWizard automatically adds these files to the project.

Here's the initial version of PENDLG.H that ClassWizard generates once you've completed the Add Class dialog box:

```
class CPenWidthsDlg : public CDialog
{
// Construction
public:
    CPenWidthsDlg(CWnd* pParent = NULL);    // standard constructor

// Dialog Data
    //{{AFX_DATA(CPenWidthsDlg)
    enum { IDD = IDD_PEN_WIDTHS };
    // NOTE: the ClassWizard will add data members here
    //}}AFX_DATA

// Implementation
protected:
    virtual void DoDataExchange(CDataExchange* pDX);// DDX/DDV support

    // Generated message map functions
    //{{AFX_MSG(CPenWidthsDlg)
        // Note: the ClassWizard will add member functions here
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

This file contains a declaration for `CPenWidthsDlg`, the class that implements the Pen Widths dialog box. At this point, the class contains two member functions: a constructor and the `DoDataExchange` function, which is described later on.

As described in Chapter 6, the file contains comment lines that begin `//{{AFX_` and `//}}AFX_`. Recall that ClassWizard uses those comment lines to find the sections of code that it maintains. There are two such sections in the header file,

each delimited by slightly different comments: the AFX_DATA section, containing the declarations of the dialog data members; and the AFX_MSG section, containing the declarations of the message handlers. In general, you shouldn't manually edit any declarations that appear in these sections.

Here's the initial version of PENDLG.CPP that ClassWizard generates once you've completed the Add Class dialog box:

```
#include "stdafx.h"
#include "scribble.h"
#include "pendlg.h"

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

/////////////////////////////////////////////////////////////////////////
// CPenWidthsDlg dialog

CPenWidthsDlg::CPenWidthsDlg(CWnd* pParent /*=NULL*/ )
        : CDialog(CPenWidthsDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CPenWidthsDlg)
        // Note: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT
}

void CPenWidthsDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CPenWidthsDlg)
        // Note: the ClassWizard will add DDX and DDV calls here
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CPenWidthsDlg, CDialog)
    //{{AFX_MSG_MAP(CPenWidthsDlg)
        // Note: the ClassWizard will add message map macros here
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////
// CPenWidthsDlg message handlers
```

This file contains an empty message map and empty function definitions for the constructor and the DoDataExchange member function. The DoDataExchange function will be described later in this chapter.

Notice that the constructor has a base initializer for **CDialog**. The **CDialog** constructor that it invokes creates a modal dialog box, and it takes two parameters: the ID of the dialog resource and a pointer to the parent window. For the first parameter ClassWizard has specified `CPenWidthsDlg::IDD`. This is an enumerated value that is defined in the `AFX_DATA` section in the class declaration. This enumerated value is equal to IDD_PEN_WIDTHS, the ID you specified in the section "Modifying the Controls' Properties" on page 119. Thus the dialog class is associated with the dialog resource you created.

Also notice that the implementation file, like the header file, contains sections delimited by `//{{` and `//}}`, in which ClassWizard will insert code later.

Declaring the Message-Handling Functions

The **CDialog** class defines default handlers for the OK and Cancel buttons. The Pen Widths dialog box contains a third pushbutton, the Default button. For `CPenWidthsDlg` to respond when the user chooses this button, you must define a new message handler and bind it to the Default pushbutton.

Binding a message handler to a control in a dialog box is essentially the same as binding a message handler to a menu command, which was described in Chapter 6; both are accomplished by adding an entry to a class's message map using ClassWizard.

After completing the Add Class dialog box in the previous section, the ClassWizard dialog box is active. This is the same dialog box that was described in Chapter 6, but notice the following differences:

- The dialog class, not the document or view class, is the one that will handle the message. Consequently, the Class Name box displays "CPenWidthsDlg."
- The Object IDs list box displays the IDs of all the controls in the dialog box, not the commands in a menu.
- The message being handled is a Windows control notification message, not an application-specific command. As a result, the Message box displays more than just **COMMAND** and **UPDATE_COMMAND_UI**; it displays all the messages that can be sent by the object that's highlighted in the Object IDs box. For example, if IDC_THIN_PEN_WIDTH — which is the ID of the first edit box — is highlighted in the Object IDs box, the Message box displays all the control notification messages that an edit box can generate, such as **EN_SETFOCUS**, **EN_KILLFOCUS**, and **EN_UPDATE**.

Despite these differences, the procedure for adding a message handler is the same.

▶ To add a message handler for the Default button

1. In the Object IDs list box, highlight IDC_DEFAULT_PEN_WIDTHS. Recall that this is the ID of the Default button. Notice that the Message list box shows all the notification messages that a pushbutton can send, that is, **BN_CLICKED** and **BN_DOUBLECLICKED**.
2. In the Messages list box, highlight the **BN_CLICKED** message. Notice the description in the status bar: "Indicates the user clicked a button."
3. Choose the Add Function button. The Add Member Function dialog box appears, displaying the candidate name, "OnClickedDefaultPenWidths." ClassWizard has synthesized this name based on the object's ID and the message name.
4. Change the name to "OnDefaultPenWidths" and then choose the OK button. The name "OnDefaultPenWidths" appears in the Member Function list box and a hand-shaped icon appears next to the entry for **BN_CLICKED** in the Message box.

At this point, you could choose the Edit Code button to fill in the definition of the `OnDefaultPenWidths` message handler, the way you did with the message handlers for menu commands in Chapter 6. However, the purpose of this function is to manipulate member variables of the dialog class. Right now the `CPenWidthsDlg` class doesn't have any member variables defined; those members will be defined in the next section. Consequently, you will implement `OnDefaultPenWidths` later in the chapter, after you've added the member variables.

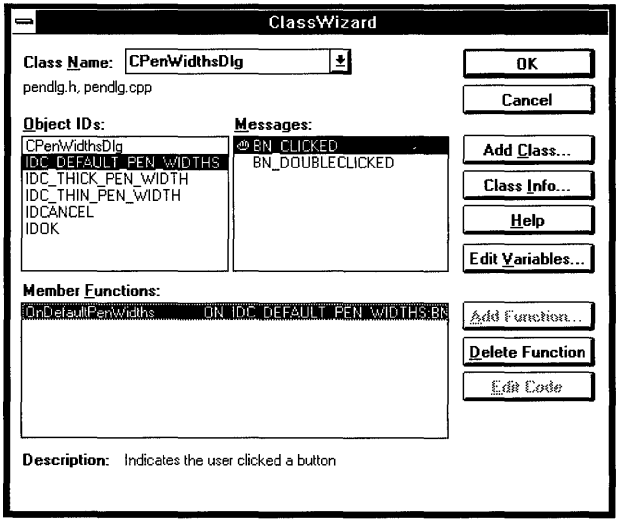


Figure 7.4 The ClassWizard Dialog Box

Here are the changes that ClassWizard makes to PENDLG.H after you've defined the message handler (these changes are saved to the file when you close the ClassWizard dialog box):

```
class CPenWidthsDlg : public CDialog
{
// Construction
public:
    CPenWidthsDlg(CWnd* pParent = NULL);

// Dialog Data
    //{{AFX_DATA(CPenWidthsDlg)
    enum { IDD = IDD_PEN_WIDTHS };
    // Note: the ClassWizard will add data members here
    //}}AFX_DATA

// Implementation
protected:
    virtual void DoDataExchange(CDataExchange* pDX);// DDX/DDV support

    // Generated message map functions
    //{{AFX_MSG(CPenWidthsDlg)
    afx_msg void OnDefaultPenWidths();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

Notice that ClassWizard has inserted a prototype for a member function named `OnDefaultPenWidths`.

ClassWizard makes the following changes to PENDLG.CPP after you've defined the message handler (these changes are saved to the file when you close the ClassWizard dialog box):

```
#include "stdafx.h"
#include "scribble.h"
#include "pendlg.h"

// ...

BEGIN_MESSAGE_MAP(CPenWidthsDlg, CDialog)
    //{{AFX_MSG_MAP(CPenWidthsDlg)
    ON_BN_CLICKED(IDC_DEFAULT_PEN_WIDTHS, OnDefaultPenWidths)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

```
////////////////////////////////////////////////////////////////////
// CPenWidthsDlg message handlers

void CPenWidthsDlg::OnDefaultPenWidths()
{
    // TODO: Add your control notification handler code here
}
```

Notice that ClassWizard has inserted an entry in the message map indicating that the member function `OnDefaultPenWidths` is a message handler that is invoked whenever the control **IDC_DEFAULT_PEN_WIDTHS** sends a **BN_CLICKED** message. ClassWizard has also generated an empty function definition for the message handler. You'll fill in the implementation for the function later in the chapter.

Mapping the Controls to Member Variables

Scribble must be able to retrieve the values that the user enters in the Thin Pen and Thick Pen edit boxes. The Microsoft Foundation Class Library defines a mechanism that automates the process of gathering values from a dialog box; this mechanism is called a "data map." In the same way that a message map binds a user-interface element with a member function, a data map binds a dialog-box control with a member variable. The value of the member variable reflects the status or the contents of the control. By adding entries to `CPenWidthsDlg`'s data map, you can retrieve the values entered in the Thin Pen and Thick Pen edit boxes.

For Scribble, the widths of the thin and thick pens must be between 1 and 20. You can enforce these conditions by using the automated data validation that data maps provide. If the user enters values that fall outside this range, the application displays a message box stating the legal range and allows the user to enter new values.

At this point, the ClassWizard dialog box is still on the screen. Choose the Edit Variables button. The Edit Member Variables dialog box appears, as shown in Figure 7.5. This dialog box contains a list box displaying the mapping between controls and member variables. At the moment the box displays only the IDs for the controls, because you haven't yet specified which member variables the controls correspond to.

▶ To map the controls of the Pen Widths dialog box to member variables

1. Select **IDC_THIN_PEN_WIDTH** and then choose the Add Variable button. The Add Member Variable dialog box appears.
2. In the Member name edit box, enter "m_nThinWidth."
3. In the Variable Type list box, choose "int."

4. Choose OK to add the member variable to the class. Notice that the member name and type you specified now appear in the Control list box and two new edit boxes appear to receive the validation parameters appropriate for an integer.
5. In the Minimum and Maximum edit boxes, enter "1" and "20" respectively.
6. Repeat steps 1 through 5 for the control **IDC_THICK_PEN_WIDTH**. Enter "m_nThickWidth" for the member name, choose "int," and enter lower and upper limits of 1 and 20.
7. Choose the Close button to exit the Add Member Variable dialog box.

You've now completed the data map connecting the Pen Widths dialog box to the `PenWidthsDlg` class.

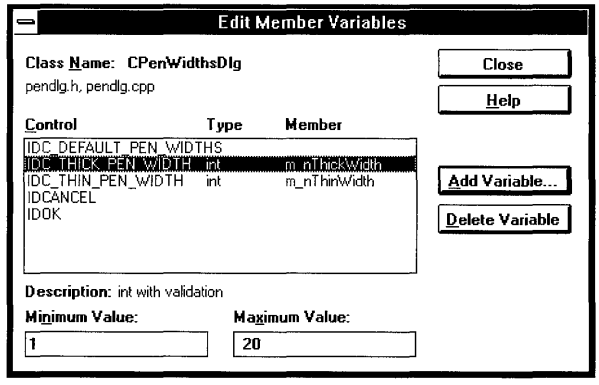


Figure 7.5 The Edit Member Variables Dialog Box

ClassWizard makes the following changes to PENDLG.H after you've mapped the controls to member variables (these changes are saved to the file after you close the ClassWizard dialog box):

```
class CPenWidthsDlg : public CDialog
{
// Construction
public:
    CPenWidthsDlg(CWnd* pParent = NULL);

// Dialog Data
    //{{AFX_DATA{CPenWidthsDlg)
    enum { IDD = IDD_PEN_WIDTHS };
    int           m_nThinWidth;
    int           m_nThickWidth;
    //}}AFX_DATA
```

```
// Implementation
protected:
    virtual void DoDataExchange(CDataExchange* pDX);// DDX/DDV support

    // Generated message map functions
    //{{AFX_MSG(CPenWidthsDlg)
    afx_msg void OnDefaultPenWidths();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

Notice that ClassWizard has inserted declarations of member variables in the data map. These are the member variables you specified in the Add Member Variable dialog box.

ClassWizard makes the following changes to PENDLG.CPP after you've mapped the controls to member variables (these changes are saved to the file when you close the ClassWizard dialog box):

```
#include "stdafx.h"
#include "scribble.h"
#include "pendlg.h"

/////////////////////////////////////////////////////////////////////////
// CPenWidthsDlg Dialog

// ...

CPenWidthsDlg::CPenWidthsDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CPenWidthsDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CPenWidthsDlg)
    m_nThinWidth = 0;
    m_ThickWidth = 0;
    //}}AFX_DATA_INIT
}

void CPenWidthsDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CPenWidthsDlg)
    DDX_Text(pDX, IDC_THIN_PEN_WIDTH, m_nThinWidth);
    DDV_MinMaxInt(pDX, m_nThinWidth, 1, 20);
    DDX_Text(pDX, IDC_THICK_PEN_WIDTH, m_nThickWidth);
    DDV_MinMaxInt(pDX, m_nThickWidth, 1, 20);
    //}}AFX_DATA_MAP
}

// ...
```

Notice that ClassWizard has initialized the member variables in the constructor and provided an implementation for the `DoDataExchange` function. The framework calls `DoDataExchange` whenever values have to be moved between the member variables in the class and the controls in the dialog box on screen (for example, when first displaying the dialog box on the screen or when the user closes the dialog box by choosing OK).

The `DoDataExchange` function is implemented using **DDX** and **DDV** function calls. A **DDX** (for Dialog Data eXchange) function specifies which control in the dialog box corresponds to a particular member variable and transfers the data between the two. A **DDV** (for Dialog Data Validation) function specifies the validation parameters for a particular member variable, ensuring that its value is legal. The **DDX** and **DDV** function calls shown above reflect the mapping and validation parameters you specified with ClassWizard.

Notice that the **DDV** function call for a given member variable immediately follows the **DDX** function call for that variable. This is a rule you must follow if you choose to manually edit the contents of the data map.

For more information about ClassWizard, see Chapter 9 of the *App Studio User's Guide*.

Implementing the Message Handler

Recall that ClassWizard provided an empty function definition for the `OnDefaultPenWidths` message handler, which is called when the user chooses the Default button. Now that the `CPenWidthsDlg` class contains the necessary member variables, it's time to fill in that function definition. This function sets the contents of the edit boxes to the default widths of the thin and thick pens.

▶ To implement the message handler for the Default button

1. From the ClassWizard dialog box, highlight "OnDefaultPenWidths" in the Member Functions list box and then choose the Edit Code button. (The Edit Code button may be disabled if you are running App Studio without running Visual Workbench; in this case, launching Visual Workbench will enable the button.) The Edit Code button transfers you to Visual Workbench, opens PENDLG.CPP, and displays the definition for `OnDefaultPenWidths`.
2. Fill in the `OnDefaultPenWidths` function with the following code:

```
void CPenWidthsDlg::OnDefaultPenWidths()
{
    m_nThinWidth = 2;
    m_nThickWidth = 5;
    UpdateData(FALSE);  // causes DoDataExchange()
        // bSave=FALSE means don't save from screen, rather, write
        // to screen
}
```

The function sets `m_nThinWidth` and `m_nThickWidth` to their default values and then calls **UpdateData**, a member function defined by **CWnd** (the base class of **CDialog**).

The **UpdateData** member function calls the `DoDataExchange` function to move values between the member variables and the controls displayed on the screen. The direction in which the data values are moved is specified by the argument to **UpdateData**. The default value of this argument is **TRUE**, which moves data from the controls to the member variables. A value of **FALSE** moves data from the member variables to the controls. The `OnDefaultPenWidths` member function passes **FALSE**, causing the default values to be displayed in the edit boxes on the screen.

3. Save and close the file PENDLG.CPP and return to App Studio. The dialog editor window is still visible, displaying the Pen Widths dialog box you designed. Click on the push-pin button on the property page so that it is unpinned, then double-click the close box on the dialog editor window. Now only the App Studio resource browser window for SCRIBBLE.RC should be visible.

Invoking the Dialog Box

By now you've specified almost everything about the Pen Widths dialog box: its appearance, the data map for its edit controls, and the message handlers for its pushbuttons. There's only one thing that remains to be specified: when the dialog box should be invoked.

At the moment there is no programmatic connection between the Pen Widths menu item and the Pen Widths dialog box; that is, the menu item and the dialog box are not bound together. You must explicitly bind them by invoking the Pen Widths dialog box from within the message handler for the Pen Widths command.

How do you invoke a dialog box? The first step is to declare a `CPenWidthsDlg` object. This doesn't display the dialog box on the screen, it just constructs the C++ object that manages the dialog box. To display the dialog box, you must call the **DoModal** member function defined by the **CDialog** class.

The Pen Widths dialog box is a "modal" dialog, which means that once invoked, it takes control of the application (it puts the program in a different "mode"). The user

can do no other work in the application while the dialog box is displayed and must dismiss the dialog box, typically by choosing the OK or Cancel button, to continue with the application. The **DoModal** function continues executing as long as the dialog box is displayed on the screen. When the user chooses the OK or Cancel button, the **DoModal** function returns **IDOK** or **IDCANCEL**, respectively, and the application can continue.

Now you can write a message handler for the Pen Widths command. Which class should get the handler? Recall that in Chapter 6 you added declarations for the `m_nThickWidth` and `m_nThinWidth` member variables to the `CScribDoc` class, because the document needs to keep track of the widths of the thick and thin pens (this allows multiple views to share the same pen widths). Since the document class has to maintain those values, it should get the handler for the Pen Widths command.

▶ To bind the Pen Widths command

1. From the Type box in the App Studio resource browser window for SCRIBBLE.RC, choose Menu.
2. Open the **IDR_SCRIBTYPE** menu resource you edited in Chapter 5.
3. From the Resource menu, choose the ClassWizard command. The ClassWizard dialog box appears.
4. In the Class Name dropdown list, choose "CScribDoc."
5. In the Object IDs box, highlight the **ID_PEN_WIDTHS** command.
6. In the Messages box, highlight **COMMAND**.
7. Choose the Add Function button.
8. Choose the OK button to accept the candidate name "OnPenWidths."
9. Choose the Edit Code button. This returns you to the Visual Workbench session and opens the file SCRIBDOC.CPP.
10. Make the following additions to SCRIBDOC.CPP:

```
#include "stdafx.h"
#include "scribble.h"
#include "scribdoc.h"

#include "pendlg.h"

// ...

void CScribDoc::OnPenWidths()
{
    CPenWidthsDlg dlg;
    // Initialize dialog data
    dlg.m_nThinWidth = m_nThinWidth;
    dlg.m_nThickWidth = m_nThickWidth;
```

```
        // Invoke the dialog box
        if (dlg.DoModal() == IDOK)
        {
            // retrieve the dialog data
            m_nThinWidth = dlg.m_nThinWidth;
            m_nThickWidth = dlg.m_nThickWidth;

            // Update the pen used by views when drawing new strokes
            // to reflect the new pen widths for "thick" and "thin".
            ReplacePen();
        }
    }
```

11. Save SCRIBDOC.CPP. In App Studio, save SCRIBBLE.RC.

When modifying SCRIBDOC.CPP, it's necessary to include PENDLG.H, so that the message handler has access to the dialog class you've created. The `OnPenWidths` function declares a `CPenWidthsDlg` object and sets the values of the `m_nThickWidth` and `m_nThinWidth` member variables to the current widths of the thick and thin pens. Then the function calls the **DoModal** function, which displays the dialog box on the screen and takes control of the system until the user exits the dialog box. If the user exits the dialog box by choosing the OK button, the function changes the current thick and thin pen widths to the new values; if the user chooses the Cancel button, the old values are retained. Finally, the function calls the `ReplacePen` member function to make the document's pen use the current widths.

When does the application perform the data exchange and validation defined in the `DoDataExchange` function? Recall that `DoDataExchange` is called by the **UpdateData** member function. Just before the dialog box is first displayed on the screen, the framework calls the **UpdateData** function with an argument of **FALSE**, which sets the contents of the edit boxes to the values of the member variables. If the user exits the dialog box by choosing the OK button, the framework calls **UpdateData** with an argument of **TRUE**, which retrieves the contents of the edit boxes and sets the values of the member variables accordingly. (If the user exits by choosing the Cancel button, the framework doesn't call **UpdateData**.)

You don't have to handle the **UPDATE_COMMAND_UI** message for the Pen Widths menu item, because the menu item doesn't need to be updated. The command is never disabled, since it's always legal to change the widths of the pens, and there's no need to add or remove a check mark, because the command isn't a toggle.

Compile the New Scribble

How does Scribble behave now that a dialog box has been added? Compile the new version of Scribble and find out.

▶ To compile Scribble

- From the Project menu in the Visual Workbench, choose the Rebuild All command.

Run the new version of Scribble. Draw some strokes with the default thick pen and the default thin pen. Then use the Pen Widths dialog to change the thickness of the pens and draw some new strokes.

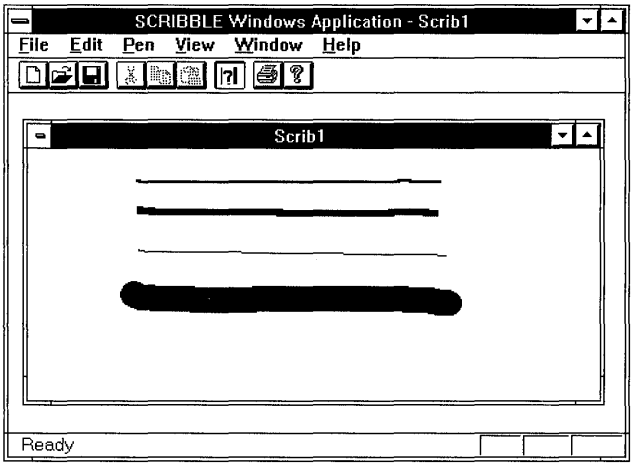


Figure 7.6 Scribble Version 3

Exit Scribble.

This completes step 3 in the tutorial.

In the next chapter you'll implement the updating of multiple views, scrolling, and splitter windows.

CHAPTER 8

Enhancing Views

In the previous chapters, you've seen how a view acts as an intermediary between a document and the user: the view displays a document on the screen and interprets mouse actions as operations on the document. You've also seen how a view cooperates with a frame window so that the frame window implements the generic window behavior while the view provides the application-specific functionality.

However, there are additional benefits to having a view class that is separate from the document and the frame window, benefits which Scribble hasn't demonstrated yet. This chapter describes how to take advantage of the division of labor between these classes to add special features to your application's user interface.

This chapter covers the following topics:

- Updating multiple views on the same document.
- Scrolling a view.
- Splitting a window.

This chapter covers step 4 of Scribble. If you want to work along, adding the code as you go, begin with the files from Chapter 7 in your SCRIBBLE\MYSCRIB subdirectory. At this point, these files should closely resemble those in the SCRIBBLE\STEP3 subdirectory. As you read the chapter, add all the code that's marked with the symbol ▸. At the end, your files should closely resemble the files in the SCRIBBLE\STEP4 subdirectory.

If, on the other hand, you want to read along without adding code, you can print or examine the files in the SCRIBBLE\STEP4 subdirectory.

Updating Multiple Views

Suppose you have a drawing open in Scribble and you choose the New Window command on the Window menu. This action opens a new document window displaying the same drawing. The document object now has two view objects connected to it. Now consider what would happen if you added some new strokes in

one of the document windows. Would the new strokes appear in the other window simultaneously? No, not as Scribble is currently implemented, because each window is unaware of what's happening in the other windows. (This is illustrated in Figure 8.1.) You would have to wait until the other window is repainted (for instance, if you minimized and then restored it). Then its `OnDraw` function would display the drawing again, including the new strokes.

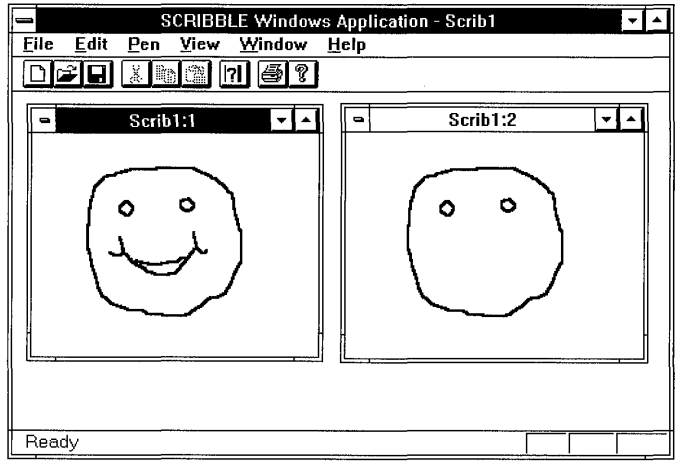


Figure 8.1 Multiple Views on a Document Without Updating

How can you ensure that all the views attached to a document reflect changes to the document as soon as they are made? Each view must notify the other views whenever it has modified the document. The Microsoft Foundation Class Library provides a standard mechanism for notifying views of modifications to a document through the **UpdateAllViews** member function of the **CDocument** class.

The **UpdateAllViews** function traverses the list of views attached to the document. For each view in the list (except the one that made the modifications), it calls the **OnUpdate** member function of the **CView** class. The **OnUpdate** function is where the view responds to changes in the document; the default implementation of the function invalidates the client area of the view, causing it to be repainted. The simplest way for you to use this updating mechanism in your application is to call the document's **UpdateAllViews** function whenever a view modifies a document in response to a user action.

You can also perform more efficient repainting with this updating mechanism if you use the parameters of the **UpdateAllViews** function. Here is the declaration of **UpdateAllViews**:

```
virtual void UpdateAllViews(CView* pSender, LPARAM lHint = 0L,
                            CObject* pHint = NULL);
```

The first argument identifies the view that made the modifications to the document. This is specified to keep the **UpdateAllViews** function from performing a redundant notification; typically the view that made the modifications doesn't need to be told about them. The second two arguments are "hints." You can use these hints to describe the modifications that the view made.

The **UpdateAllViews** function gives the hints to every view attached to the document (except the one that made the modifications) by passing them as parameters to the **OnUpdate** member function. You can override **OnUpdate** to interpret those hints and update only the area of the display that corresponds to the modified portion of the document. Thus, if another view is displaying a completely different portion of the document, it doesn't have to perform any repainting at all.

To inform other views of modifications:

1. Define a type of hint that describes a modification to a document.
2. When a view modifies the document, create a hint describing the modification made and pass it to **UpdateAllViews**.
3. Override **OnUpdate** to use the hint so that only the portion of the screen corresponding to the modification gets updated.

These steps are described in more detail below, using Scribble as an example.

Define a Hint for Scribble

When a stroke is added to a drawing in Scribble, the rectangular region that contains the new stroke is the only area that needs to be updated; the remainder of the drawing can be left alone. Therefore, a logical choice for a hint in Scribble is the bounding rectangle of the new stroke.

▶ To define bounding rectangles for strokes

1. Instead of creating a separate class to represent the hint, it's more convenient to store the bounding rectangle for each stroke in the `CStroke` object itself. Accordingly, load the file SCRIBDOC.H in Visual WorkBench and add the following new member declarations to `CStroke`:

```
class CStroke : public CObject
{
// ...
protected:
// Attributes
// ...
    CRect   m_rectBounding;     // smallest rect that surrounds all
                                // of the points in the stroke
public:
    CRect& GetBoundingRect() { return m_rectBounding; }
// Operations
public:
    void FinishStroke();
// ...
};
```

The protected member variable `m_rectBounding` is a **CRect** object storing the bounding rectangle, and the public member function `GetBoundingRect` allows the rectangle to be retrieved by the view. There is also a new helper function, the `FinishStroke` member function.

2. Load SCRIBDOC.CPP and make the following modifications:

```
// Each time we change what gets serialized, we change
// the schema number.

IMPLEMENT_SERIAL( CStroke, CObject, 2 )
// ...
CStroke::CStroke(UINT nPenWidth) : m_nPenWidth(nPenWidth)
{
    m_rectBounding.SetRectEmpty();
}

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

The changes shown here are needed to manage the addition of the `m_rectBounding` member variable. The first change to be made is incrementing the schema number in the **IMPLEMENT_SERIAL** macro. This is necessary because this version of Scribble changes what's stored in a `CStroke` object by adding a new member variable. Changing the schema number distinguishes strokes saved by this version of Scribble from those of other versions.

The next change initializes the bounding rectangle to an empty rectangle in the `CStroke` constructor. The changes to the `Serialize` member function store and read the `m_rectBounding` member variable.

3. The `FinishStroke` member function calculates the bounding rectangle for a stroke. Add the following function definition to the end of SCRIBDOC.CPP:

```
////////////////////////////////////////////////////////////////////
void CStroke::FinishStroke()
{
    // Calculate the bounding rectangle.  It's needed for smart
    // repainting.

    if( m_pointArray.GetSize() == 0 )
    {
        m_rectBounding.SetRectEmpty();
        return;
    }
    CPoint pt = GetPoint(0);
    m_rectBounding = CRect( pt.x, pt.y, pt.x, pt.y );

    for (int i=1; i < m_pointArray.GetSize(); i++)
    {
        // If the point lies outside of the accumulated bounding
        // rectangle, then inflate the bounding rect to include it.
        pt = GetPoint(i);
        m_rectBounding.left   = min(m_rectBounding.left, pt.x);
        m_rectBounding.right  = max(m_rectBounding.right, pt.x);
        m_rectBounding.top    = min(m_rectBounding.top, pt.y);
        m_rectBounding.bottom = max(m_rectBounding.bottom, pt.y);
    }

    // Add the pen width to the bounding rectangle.  This is needed
    // to account for the width of the stroke when invalidating
    // the screen.
    m_rectBounding.InflateRect(CSize(m_nPenWidth, m_nPenWidth));
    return;
}
```

In this function, the stroke object iterates through its array of points, testing each one's location; if a point falls outside the current bounding rectangle, the stroke

object enlarges the bounding rectangle just enough to contain it. Then the bounding rectangle is expanded on each side by the width of the pen.

Pass the Hint After Modifying the Document

The next step is to pass the hint to the document's **UpdateAllViews** member function. An appropriate time to pass a hint is each time a stroke is completed.

▶ To pass the hint after modifying the document

- The `OnLButtonUp` member function is called when a stroke is finished, so you should call **UpdateAllViews** from there. Load SCRIBVW.CPP and make the following modifications near the end of `OnLButtonUp`:

```
void CScribView::OnLButtonUp(UINT, CPoint point)
{
    // ...
    m_pStrokeCur->AddPoint( point );

    // Tell the stroke item that we're done adding points to it.
    // This is so it can finish computing its bounding rectangle.
    m_pStrokeCur->FinishStroke();

    // Tell the other views that this stroke has been added
    // so that they can invalidate this stroke's area in their
    // client area.
    pDoc->UpdateAllViews(this, 0L, m_pStrokeCur);

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

In this function, the view gets the hint information that it will send to the document. It does this by calling the `FinishStroke` member function for `m_pStrokeCur`; `FinishStroke` computes the bounding rectangle for the current stroke. Then the view calls **UpdateAllViews**, passing two arguments: the **this** pointer, which identifies this view as the one that performed the modification to the document; and `m_pStrokeCur`, whose bounding rectangle is the hint. (The function sends a pointer to the entire `CStroke` object rather than just the bounding rectangle because the hint must be a **CObject** pointer, and **CRect** isn't derived from **CObject**.) The view doesn't need to send any more hint information, so it doesn't pass anything in the **LPARAM** parameter.

The **UpdateAllViews** function iterates through the list of views attached to the document; for each view (except the one that performed the modification), the function calls its **OnUpdate** function and passes the hint as a parameter.

Use the Hint for Efficient Repainting

The last step is to take advantage of the hint so the other views can repaint themselves more efficiently. This involves modifying the `CScribView` class to respond to any hint it receives.

▶ To use the hint for efficient repainting

1. From within Visual Workbench, load SCRIBVW.H and add the following function declaration:

```
class CScribView : public CView
{
// ...
// Implementation
public:
    virtual void OnUpdate( CView* pSender, LPARAM lHint = 0L,
                           CObject* pHint = NULL);
// ...
};
```

   This causes `CScribView` to override the **OnUpdate** function defined by the **CView** class.

2. To define `CScribView`'s version of `OnUpdate`, load SCRIBVW.CPP in Visual Workbench and add the following function definition to the end of the file:

```
//////////////////////////////////////////////////////////////////
void CScribView::OnUpdate(CView*, LPARAM, CObject* pHint)
{
// The document has informed this view that some data has changed.

    if (pHint != NULL)
    {
        if (pHint->IsKindOf(RUNTIME_CLASS(CStroke)))
        {
            // The hint is that a stroke as been added (or changed).
            // So, invalidate its rectangle.
            CStroke* pStroke = (CStroke*)pHint;
            CRect rectInvalid = pStroke->GetBoundingRect();
            InvalidateRect(&rectInvalid);
            return;
        }
    }
    // We can't interpret the hint, so assume that anything might
    // have been updated.
    Invalidate();
    return;
}
```

Recall that this function is called by the **UpdateAllViews** function of `CScribDoc`, which passes it a hint. In this function, the view checks if the hint is a **CStroke** object. If so, the view gets the bounding rectangle for the stroke and marks it as invalid. This rectangle marks the area that must be redrawn. If the hint isn't a **CStroke** object, the view doesn't know what area was modified, so it invalidates the entire client area as a precaution.

After a region has been invalidated, Windows sends a **WM_PAINT** message. The **OnPaint** member function defined by **CView** handles this message by calling the virtual `OnDraw` member function. Consequently, you must modify the `OnDraw` function to take advantage of the invalidated rectangle when redrawing.

3. Make the following changes to the `OnDraw` member function in SCRIBVW.CPP:

```
void CScribView::OnDraw(CDC* pDC)
{
    CScribDoc* pDoc = GetDocument();

    // Get the invalidated rectangle of the view, or in the case
    // of printing, the clipping region of the printer dc.
    CRect rectClip;
    CRect rectStroke;
    pDC->GetClipBox(&rectClip);

    // The view delegates the drawing of individual strokes to
    // CStroke::DrawStroke().
    for (POSITION pos = pDoc->GetFirstStrokePos(); pos != NULL; )
    {
        CStroke* pStroke = pDoc->GetNextStroke(pos);
        rectStroke = pStroke->GetBoundingRect();
        if (!rectStroke.IntersectRect(&rectStroke, &rectClip))
            continue;
        pStroke->DrawStroke(pDC);
    }
}
```

In this function, the view first calls the **GetClipBox** member function of **CDC** to get the invalidated portion of the client area. Then the view iterates through the list of strokes in the document, calling `IntersectRect` for each to determine if any part of the stroke lies in the invalidated region. If so, the view asks the stroke to draw itself. Any strokes that don't intersect the invalidated region don't have to be redrawn.

Adding Scrolling

In the current version of Scribble, you cannot work on a drawing that is larger than the window. It would be more convenient if you could work on a large drawing no matter how small the window is; to do this, Scribble must support scrolling.

The addition of scrolling expands the conceptual role played by a view. Not only does a view produce a visual representation of a document's data, it also acts as a peephole to a document that may be too large to display all at once. This peephole can be moved across the document to reveal different portions of it. This is illustrated in Figure 8.2.

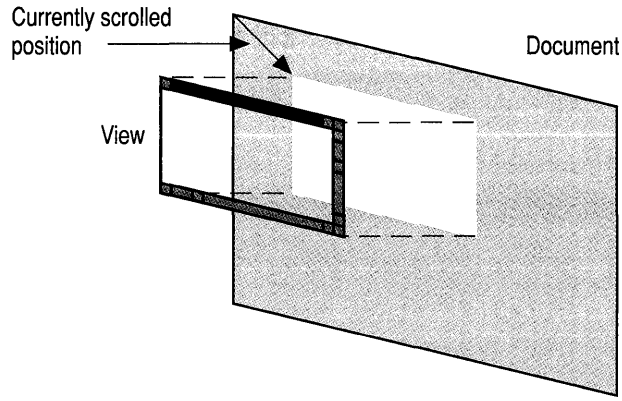


Figure 8.2 A Scrollable View on a Document

Implementing scrolling all by yourself is fairly complicated. However, since a lot of the scrolling code is the same for all applications, the Microsoft Foundation Class Library implements the common scrolling logic in a class called **CScrollView**.

The basic steps for adding scrolling to your application are as follows:

1. Define a size for your documents. This can be a constant, a member stored in each document object, a value calculated at run time, etc.
2. Derive your view class from **CScrollView** instead of **CView**.
3. Pass the document's size to the **SetScrollSizes** member function of **CScrollView** whenever the size may change.
4. Convert between logical coordinates and device coordinates if passing points between graphic device interface (GDI) and non-GDI functions.

The framework's responsibilities are as follows:

- Handle all **WM_HSCROLL** and **WM_VSCROLL** messages, scroll the document in response, and move the scroll box accordingly.

The positions of the scroll boxes reflect where the currently displayed portion of the document resides relative to the rest of the document. If the user clicks on a scroll arrow at either end of the scroll bar, the document is scrolled one "line" (whose meaning depends on the document type). If the user clicks on either side of the scroll box, the document is scrolled one "page." If the user drags the scroll box itself, the document is scrolled accordingly.

- Calculate a mapping between the lengths of the scroll bars and the height and width of the document, adjust this scaling factor when the window is resized or when the size of the document changes, and in turn remove or add scroll bars as needed.

The next section describes how to add scrolling to Scribble. Figure 8.3 shows what Scribble looks like with scroll bars added.

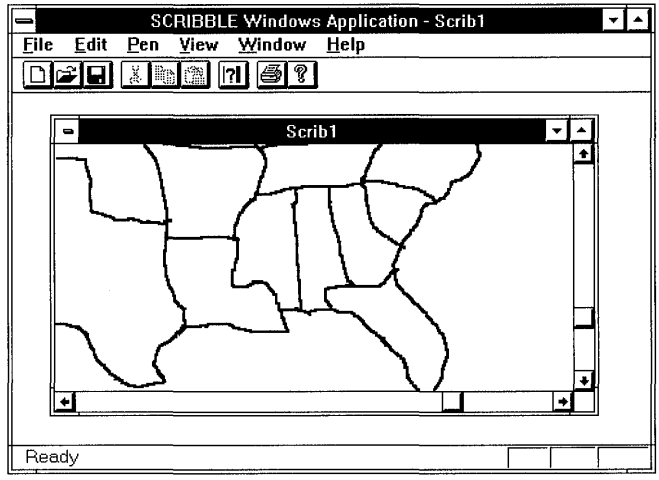


Figure 8.3 Scribble with Scrolling Support

Add Scrolling to Scribble

▶ To add scrolling support to Scribble

1. First define the size of Scribble documents. You can do this by having each document store its dimensions. Launch Visual Workbench and load SCRIBDOC.H. Make the following changes to the declaration of the `CScribDoc` class:

```
class CScribDoc : public CDocument
{
// ...
protected:
    CSize m_sizeDoc;
public:
    CSize GetDocSize() { return m_sizeDoc; }

// Operations
// ...
};
```

The member variable `m_sizeDoc` stores the size of the document in a **CSize** object. This member is protected, so it cannot be accessed directly by the views attached to the document. To let the views retrieve the size of the document, you provide a public helper function named `GetDocSize`. The views base their scrolling limits on the document size.

2. Load SCRIBDOC.CPP and make the following changes:

```
// ...

void CScribDoc::InitDocument()
{
    // ...
    // default document size is 800 x 900 screen pixels
    m_sizeDoc = CSize(800,900);
}

// ...

// CScribDoc serialization

void CScribDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        ar << m_sizeDoc;
    }
    else
    {
        ar >> m_sizeDoc;
    }
    m_strokeList.Serialize(ar);
}
```

The new code in the `InitDocument` member function initializes the `m_sizeDoc` member variable; recall that you use this function whenever a new document is created or an existing document is opened. All Scribble documents are the same size: 800 logical units in width and 900 logical units in height. For simplicity's sake, Scribble doesn't support documents of varying size to accommodate arbitrarily large drawings.

The changes to the `Serialize` member function store and read the `m_sizeDoc` member variable.

3. Next, you must make the view set its scrolling limits according to the size of the document. Load SCRIBVW.H and make the following changes to the declaration of `CScribView`:

```
class CScribView : public CScrollView
{
// ...
// Implementation
public:
    void OnInitialUpdate();
// ...
}
```

By changing the base class of `CScribView` from **CView** to **CScrollView**, you give `CScribView` scrolling functionality without having to implement any of it yourself.

In addition, the `CScribView` class overrides the **OnInitialUpdate** member function, which is called when the view is first attached to the document. By overriding this function, you can inform the view of the document's size as soon as possible.

4. Recall that the Microsoft Foundation Class Library uses message maps as well as C++ inheritance. As a result, modifying the class declaration in the header file isn't enough to give `CScribView` all of **CScrollView**'s functionality; you also have to modify the message-map macros in the implementation file. Load SCRIBVW.CPP and change the following lines:

```
IMPLEMENT_DYNCREATE( CScribView, CScrollView )

BEGIN_MESSAGE_MAP( CScribView, CScrollView )
    //{{AFX_MSG_MAP( CScribView )
        // ...
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

// ...
```

Notice that in the message map macro, `CScribView`'s name is now followed by **CScrollView** instead of **CView**. This instructs the framework to search

CScrollView's message map if it can't find the message handler it needs in `CScribView`'s message map.

5. If you want to use the diagnostic features provided by the Microsoft Foundation Class Library, change the implementations of the `Dump` and `AssertValid` member functions of `CScribView`. These functions simply call their base class versions; change them to call the **CScrollView** versions rather than the **CView** versions.

6. To provide a definition for `OnInitialUpdate`, add the following code at the end of SCRIBVW.CPP:

```
void CScribView::OnInitialUpdate()
{
    SetScrollSizes( MM_TEXT, GetDocument()->GetDocSize() );
}
```

The **SetScrollSizes** member function is defined by **CScrollView**. Its first parameter is the mapping mode used to display the document. The mapping mode that the current version of Scribble uses is **MM_TEXT**; in Chapter 9, Scribble will use the **MM_LOENGLISH** mapping mode for better printing. (For more information on mapping modes, see Chapter 9, or see **CDC::SetMapMode** in the *Class Library Reference*).

The second parameter is the total size of the document, which is needed to determine the scrolling limits. The view uses the value returned by the document's `GetDocSize` member function for this parameter.

SetScrollSizes also has two other parameters, for which Scribble uses the default values. These are **CSize** values that represent the size of one "page" and one "line," the distances to be scrolled if the user clicks the scroll bar or a scroll arrow. The default values are 1/10th and 1/100th of the document size, respectively.

Since Scribble documents are fixed in size, there is no need to make any subsequent calls to **SetScrollSizes**. If your application supports documents of varying size, you should call **SetScrollSizes** immediately after the document's size changes. (You can do this from the **OnUpdate** member function of your view class.)

Notice that the addition of scrolling didn't require you to modify the `OnDraw` member function of `CScribView`. If the drawing function is unchanged, why does the window display different portions of the document depending on where the user has scrolled to? The reason is that the document is displayed using coordinates relative to an origin used by GDI. When this origin was fixed at the upper-left corner of the client area, the part of the document that was visible was always the same. By moving the origin used by GDI, **CScrollView** can adjust what portion of the document is shown in the client area of the window and what portions are hidden.

The origin used by GDI is a characteristic of a device context; it is used by the member functions of the **CDC** class. If you want to make adjustments to the **CDC** object used by your view, you can override the **OnPrepareDC** member function defined by **CView**. **CScrollView** overrides **OnPrepareDC** to move the device context's origin to reflect the currently scrolled position. **OnPrepareDC** is always called by the framework before it calls **OnDraw**; in Scribble, **CScrollView**'s version of **OnPrepareDC** is called before `CScribView`'s `OnDraw` is called. As a result, you don't have to make any changes to the `OnDraw` function to draw a properly scrolled document; all the work needed to do scrolling is done to the device context before `OnDraw` receives it.

It's important to note that changing the device context's origin doesn't affect the coordinates you receive with Windows messages such as **WM_LBUTTONDOWN** or **WM_MOUSEMOVE**; the points accompanying those messages are still specified in coordinates relative to the upper-left corner of the client area. This is because Windows messages are not part of a device context, so they are unaffected by changes to the GDI origin. Thus, `CScribView` must now deal with two types of coordinates:

- The coordinates used for describing the points received with a mouse message. Those points are returned in "device coordinates."
- The coordinates used for drawing with GDI. These are known as "logical coordinates."

When storing the coordinates of strokes, Scribble needs to know where the strokes are relative to the document, not relative to the client area. Consequently, `CScribView` must convert points from device coordinates (relative to the window origin) to logical coordinates (relative to the document origin) before storing them in `CStroke` objects.

▶ To store the strokes using logical coordinates

1. Make the following modifications to the `OnLButtonDown` member function of `CScribView`:

```
void CScribView::OnLButtonDown(UINT, CPoint point)
{
    // CScrollView changes the viewport origin and mapping mode.
    // It's necessary to convert the point from device coordinates
    // to logical coordinates, such as are stored in the document.
    CClientDC dc(this);
    OnPrepareDC(&dc);
    dc.DPtoLP(&point);

    m_pStrokeCur = GetDocument()->NewStroke();
    m_pStrokeCur->AddPoint(point); // add 1st point to the new stroke
    SetCapture();       // Capture the mouse until button up.
    m_ptPrev = point;   // Serves as the MoveTo() anchor for the
                        //   LineTo() the next point as the user
                        //   drags the mouse.
    return;
}
```

In this function, the view receives a point specified in device coordinates. A device context is needed to find the GDI origin, so the function declares a **CClientDC** object, a **CDC** object for the client area of the view, and calls **OnPrepareDC** to adjust its origin. Then the function passes the point to the **DPtoLP** (Device Point to Logical Point) member function of **CDC** to perform the actual conversion. The point added to `m_pStrokeCur` is thus described in logical coordinates (that is, relative to the document origin).

2. Make a similar modification to the `OnMouseMove` member function:

```
void CScribView::OnMouseMove(UINT, CPoint point)
{
    // ...
    if (GetCapture() != this)
        return;   // If this window (view) didn't capture the mouse,
                  // then the user isn't drawing in this window.

    CClientDC dc(this);
    // CScrollView changes the viewport origin and mapping mode.
    // It's necessary to convert the point from device coordinates
    // to logical coordinates, such as are stored in the document.
    OnPrepareDC(&dc);
    dc.DPtoLP(&point);

    m_pStrokeCur->AddPoint(point);
    // ...
}
```

This function already has a device context for drawing the stroke in progress, so the only modifications needed are to call **OnPrepareDC** to move the viewport origin and then **DPtoLP** to convert the point before before adding it.

3. Make the same modification to the `OnLButtonUp` member function:

```
void CScribView::OnLButtonUp(UINT, CPoint point)
{
    // ...
    if (GetCapture() != this)
        return;  // If this window (view) didn't capture the mouse,
                 // then the user isn't drawing in this window.

    CScribDoc* pDoc = GetDocument();
    CClientDC dc(this);

    // CScrollView changes the viewport origin and mapping mode.
    // It's necessary to convert the point from device coordinates
    // to logical coordinates, such as are stored in the document.
    OnPrepareDC(&dc);  // set up mapping mode and viewport origin
    dc.DPtoLP(&point);

    CPen* pOldPen = dc.SelectObject(pDoc->GetCurrentPen());
    // ...
}
```

Like `OnMouseMove`, this function already has a device context to complete drawing the stroke, so the only modifications needed are to call **OnPrepareDC** and then **DPtoLP**.

4. Finally, you must modify the `OnUpdate` member function. Unlike the previous three functions, this function requires a conversion in the opposite direction, that is, from logical coordinates to device coordinates. Recall that `OnUpdate` retrieves the bounding rectangle of a stroke and invalidates that rectangle. The stroke's bounding rectangle is stored in logical coordinates. However, the rectangle passed to **InvalidateRect** must be specified in device coordinates (since **InvalidateRect** is not a GDI function).

   Accordingly, a stroke's bounding rectangle must have its coordinates converted into device coordinates before it can be invalidated. Make the following modifications:

```
void CScribView::OnUpdate(CView*, LPARAM, CObject* pHint)
{
// The document has informed this view that some data has changed.

    if (pHint != NULL)
    {
        if (pHint->IsKindOf(RUNTIME_CLASS(CStroke)))
        {
            // The hint is that a stroke as been added (or changed).
            // So, invalidate its rectangle.
            CStroke* pStroke = (CStroke*)pHint;
            CClientDC dc(this);
            OnPrepareDC(&dc);
            CRect rectInvalid = pStroke->GetBoundingRect();
            dc.LPtoDP(&rectInvalid);
            InvalidateRect(&rectInvalid);
            return;
        }
    }
    // We can't interpret the hint, so assume that anything might
    // have been updated.
    Invalidate(TRUE);
    return;
}
```

The function declares a **CClientDC** object and then calls the **OnPrepareDC** member function to move the viewport origin of the device context to reflect the currently scrolled position. The rectangle is then passed to the **LPtoDP** (Logical Point to Device Point) function of **CDC** to convert its points into device coordinates. (Both **DPtoLP** and **LPtoDP** are overloaded to accept rectangles as well as points.) Once it is converted, the rectangle can be invalidated.

For more information on **CScrollView**, see the *Class Library Reference*.

Adding Splitter Windows

Scrolling lets you work on a document that is larger than the window, but by the same token it means that much of the document is hidden at any one time. Suppose the user needs to refer to two widely separated portions of a document at the same time. One way to do this is to open another window on the same document and scroll them to different locations. However, windows must be resized individually so that they don't overlap. A more convenient solution is to divide a window into separate "panes," each of which can display a different portion of the document. This is illustrated in Figure 8.4.

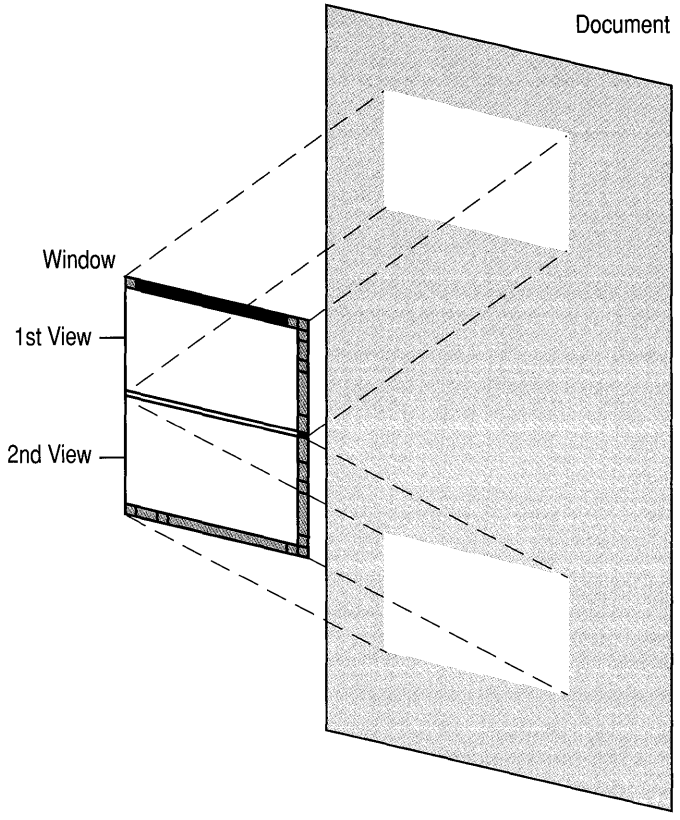


Figure 8.4 A Window with Two Views on a Document

A window that can be divided into multiple panes is called a "splitter window." A splitter window contains split boxes at the top of the vertical scroll bar and at the left of the horizontal scroll bar. By double-clicking a split box, the user can divide a window vertically or horizontally into panes. The panes are separated by a "split bar"; each pane can be scrolled independently to display a different portion of the document. The user can also drag the split bar to resize both panes at once.

Figure 8.5 shows what a Scribble window looks like when it is split into two panes.

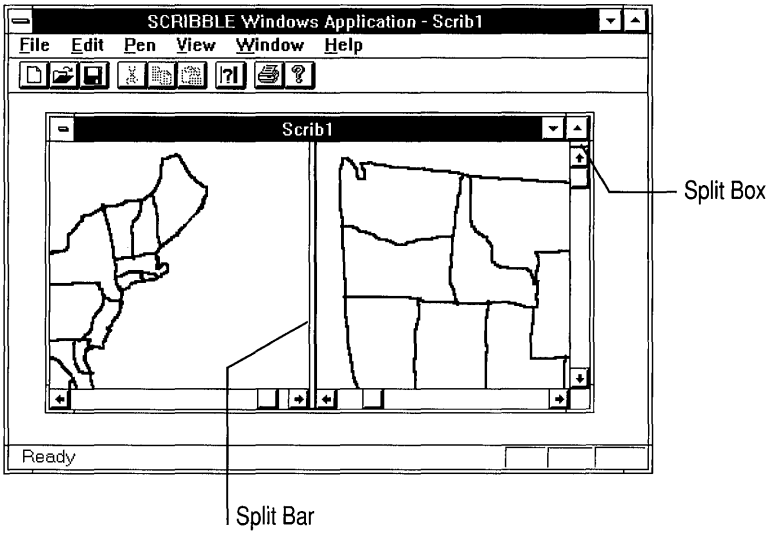


Figure 8.5 Scribble Document Window Split into Two Panes

Each pane in a splitter window represents a separate view object. In Figure 8.5, each pane is an instance of the `CScribView` class, but it's not necessary for the panes to use the same view class; you can use different classes for different panes. This is useful when, for example, you want one pane to display an outline of a document while the other pane displays the full text.

The Microsoft Foundation Class Library provides splitting functionality in a class called **CSplitterWnd**. By using this class, you can support splitting in your application with very little effort.

The basic steps for adding splitter windows to your application are as follows:

1. Derive a frame window class from **CMDIChildWnd** if you are writing a Multiple Document Interface (MDI) application or **CFrameWnd** if you are writing a Single Document Interface (SDI) application. Give this class a member variable of type **CSplitterWnd**.
2. Override the **OnCreateClient** member function of your frame window class to create a **CSplitterWnd**.
3. When defining a document template, use the frame window class you derived instead of **CMDIChildWnd** or **CFrameWnd**.

The following section shows in detail how these steps are accomplished for Scribble.

Add Splitter Windows to Scribble

To make it easy to add splitter windows to your application, ClassWizard provides an option that automatically derives a frame window class and overrides its **OnCreateClient** member function for you.

▶ To add splitter windows to Scribble

1. From Visual Workbench, pull down the Browse menu and choose the ClassWizard command. The ClassWizard dialog box appears.
2. Choose the Add Class button. The Add Class dialog box appears.
3. For the class name, enter "CScribFrame."
4. For the class type, choose "splitter."
5. For the header file, change the default name to "scribfrm.h."
6. For the implementation file, change the default name to "scribfrm.cpp."
7. Choose the Create Class button to generate the new class. The ClassWizard dialog box regains the focus.
8. Choose the OK button to exit the ClassWizard dialog box. As before, the new files are automatically added to the project.

   The file SCRIBFRM.H, which ClassWizard has generated, declares a new class called `CScribFrame`. Here's what the file looks like:

```
////////////////////////////////////////////////////////////////////
// CScribFrame frame with splitter

class CScribFrame : public CMDIChildWnd
{
    DECLARE_DYNCREATE( CScribFrame )
protected:
    CScribFrame();  // protected constructor used by dynamic creation

// Attributes
protected:
    CSplitterWnd  m_wndSplitter;
public:

// Operations
public:

// Implementation
public:
    virtual ~CScribFrame();
    virtual BOOL OnCreateClient( LPCREATESTRUCT lpcs,
                                 CCreateContext* pContext );
```

```
    // Generated message map functions
    //{{AFX_MSG(CScribFrame)
    // NOTE - ClassWizard will add and remove member functions here.
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

`CScribFrame` assumes the role that **CMDIChildWnd** previously played in Scribble. To understand `CScribFrame`'s declaration, it's helpful to review how **CMDIChildWnd** is normally used. Until now, each time you opened a document window in Scribble, there were two objects cooperating to display the document: a **CMDIChildWnd** object, which manages the document window's frame, and a `CScribView` object, which manages the document window's client area. (These two classes were specified when AppWizard created the document template for `CScribDoc` objects.)

For Scribble to support splitting, this organization must change. Objects of three classes must cooperate to displaying a document: a `CScribFrame` object, which manages the document window's frame; a **CSplitterWnd** object, which manages the document window's client area; and one or more `CScribView` objects, each of which manages a pane in the window. The **CSplitterWnd** object is not visible as a distinct entity, but it is responsible for handling the `CScribView` objects as panes, managing their scroll bars, and drawing the split boxes and split bars.

This technique for managing splitter windows is similar to the way MDI is implemented. An MDI application's frame window has a client window managing its entire client area, or workspace. It is this client window that owns the child windows that display documents.

Now consider `CScribFrame`'s declaration. The `CScribFrame` class is derived from **CMDIChildWnd** because Scribble is an MDI application; if Scribble were an SDI application, `CScribFrame` would be derived from **CFrameWnd**. The only constructor for `CScribFrame` is a protected one, because you don't need to explicitly create `CScribFrame` objects; the framework handles their creation for you.

The `CScribFrame` class defines one member variable: a **CSplitterWnd** object. This is the window that covers the frame window's client area. The class also overrides the **OnCreateClient** member function defined by **CFrameWnd** (the base class of **CMDIChildWnd**). The framework calls this function when it first creates the frame window.

The file SCRIBFRM.CPP, which ClassWizard has generated, contains the implementation of the member functions for `CScribFrame`. Here's what it looks like:

```
#include "stdafx.h"
#include "scribble.h"
#include "scribfrm.h"

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

IMPLEMENT_DYNCREATE(CScribFrame, CMDIChildWnd)

CScribFrame::CScribFrame()
{
}

CScribFrame::~CScribFrame()
{
}

BOOL CScribFrame::OnCreateClient( LPCREATESTRUCT /*lpcs*/,
                                  CCreateContext* pContext )
{
    return m_wndSplitter.Create( this,
        2, 2,               // TODO: adjust the number of rows, columns
        CSize( 10, 10 ),    // TODO: adjust the minimum pane size
        pContext );
}

BEGIN_MESSAGE_MAP(CScribFrame, CMDIChildWnd)
    //{{AFX_MSG_MAP(CScribFrame)
       // NOTE - ClassWizard will add and remove mapping macros here.
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////
// CScribFrame message handlers
```

In the `OnCreateClient` member function, the frame window creates the window that will cover its client area by calling the **Create** function of its **CSplitterWnd** member variable. The parameters passed to the **Create** function describe the panes that the splitter window will manage.

The first argument passed to **Create** specifies the parent window for the client window: the function passes the **this** pointer, making the `CScribFrame` window the parent of the **CSplitterWnd** object. The second and third parameters specify the maximum number of rows and columns that the splitter window can have; a value of two is used for each, so Scribble's splitter windows can have up to four panes. The fourth parameter specifies the minimum size of a pane: a square 10 logical units on a side.The fifth parameter is the **CCreateContext** structure that is passed to `OnCreateClient`. This structure is used to determine which view class should be used for each pane in the splitter window.

The **Create** function can also accept an additional three arguments; because Scribble doesn't pass any values for these, the default values are used. The sixth argument specifies the styles to be used for the splitter window. The default value specifies a visible child window with vertical and horizontal scroll bars that supports dynamic splitting. The seventh argument specifies the ID to be assigned to the splitter window. Its default value is **AFX_IDW_PANE_FIRST**, which is the ID of the first pane.

9. Having defined a new frame window class, you must now use it when opening Scribble documents. Open the file SCRIBBLE.CPP and add the code following:

```
#include "stdafx.h"
#include "scribble.h"
#include "mainfrm.h"
#include "scribfrm.h"
#include "scribdoc.h"
#include "scribvw.h"

// ...

BOOL CScribbleApp::InitInstance()
{
    // ...

    AddDocTemplate(new CMultiDocTemplate(IDR_SCRIBTYPE,
            RUNTIME_CLASS(CScribDoc),
            RUNTIME_CLASS(CScribFrame),  // MDI child with splitter wnd
            RUNTIME_CLASS(CScribView)));
    // ...
}
```

First it's necessary to include the header file SCRIBFRM.H so you can access the declaration of the `CScribFrame` class. The major modification occurs in the `InitInstance` member function of `CScribbleApp`. This function calls **AddDocTemplate** to register the `CScribDoc` document type with the application. Recall that a document template connects a document class, a frame window class, and a view class. In previous versions of Scribble, **CMDIChildWnd** was the frame window class used for displaying `CScribDoc`

objects. Now `CScribFrame`, which is derived from **CMDIChildWnd**, is the frame window class. As a result, the windows used for displaying Scribble's documents support splitting.

For more information on **CSplitterWnd**, see the *Class Library Reference*.

Compile the New Scribble

How does Scribble behave with these new enhancements? Compile the new version and find out.

▶ To compile Scribble

- From the Project menu in Visual Workbench, choose the Rebuild All Command.

Run the new version of Scribble.

Draw some strokes, scroll to a new portion of the drawing, and draw some more strokes. Resize the window and scroll back and forth. Click the split box to split the window into two panes. With both panes displaying the same portion of the document, draw some strokes in one pane and see them reflected in the other one. Figure 8.6 shows this version of Scribble.

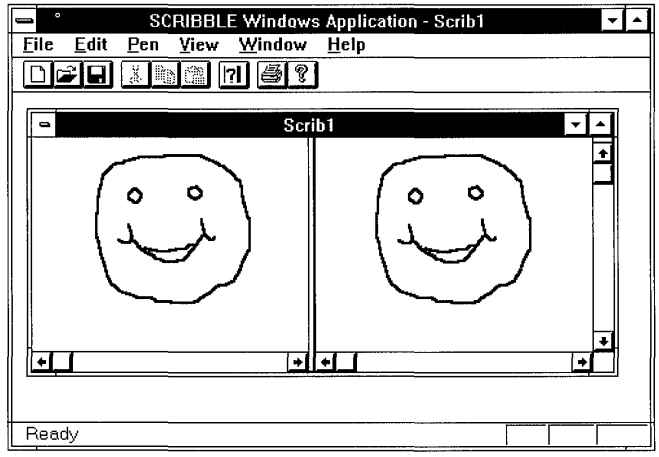


Figure 8.6 Scribble Version 4

Exit Scribble.

This completes step 4 of the tutorial. You now have a basic understanding of the view architecture provided by the Microsoft Foundation Class Library.

In the next chapter you'll enhance Scribble's printing and print preview support.

CHAPTER 9

Enhancing Printing

Scribble has supported printing and print preview ever since Chapter 4, when you first added application-specific code to the starter files created by AppWizard. All the printing and previewing functionality came essentially "for free." None of the code you added dealt specifically with printing; AppWizard and the framework did all the work for you.

While it's nice to get printing and print preview for free, Scribble's current printing support isn't perfect. For example, the printed image is smaller than you might like. In addition, the printed image is very plain; it doesn't include a header or footer. This chapter describes how to enlarge the printed image and implement printing enhancements in your application.

This chapter covers the following topics:

- The printing architecture provided by the Microsoft Foundation Class Library.
- Using a metric mapping mode.
- Dividing a document into pages.
- Defining headers or footers.

This chapter covers step 5 of Scribble. If you want to work along, adding the code as you go, begin with the files from Chapter 8 in your SCRIBBLE\MYSCRIB subdirectory. At this point, these files should closely resemble those in the SCRIBBLE\STEP4 subdirectory. As you read the chapter, add all the code that's marked with the symbol ▸. At the end, your files should closely resemble the files in the SCRIBBLE\STEP5 subdirectory.

If, on the other hand, you want to read along without adding code, you can print or examine the files in the SCRIBBLE\STEP5 subdirectory.

How Default Printing Is Done

Recall how Scribble performs drawing on the screen. In Chapter 4 you saw that the `CScribView` class has a member function named `OnDraw` which contains all the

drawing code. Notice that `OnDraw` takes a pointer to a **CDC** object as a parameter. That **CDC** object represents the device context to receive the image produced by `OnDraw`. When the window displaying the document receives a **WM_PAINT** message, the framework calls `OnDraw` and passes it a device context for the screen (a **CPaintDC** object, to be specific). Accordingly, `OnDraw`'s output goes to the screen.

In programming for Windows, sending output to the printer is very similar to sending output to the screen. This is because the Windows graphics device interface (GDI) is hardware-independent; you can use the same GDI functions for screen display or for printing simply by using the appropriate device context. If the **CDC** object that `OnDraw` receives represents the printer, `OnDraw`'s output goes to the printer.

This explains how Scribble can perform simple printing without requiring extra effort on your part. The framework takes care of displaying the Print dialog box and creating a device context for the printer. When you select the Print command from the File menu, the view passes this device context to `OnDraw`, which draws the document on the printer.

However, there are some significant differences between printing and screen display. When you print, you have to divide the document into distinct pages and display them one at a time, rather than display whatever portion is visible in a window. As a corollary, you have to be aware of the size of the paper (whether it's letter size, legal size, or an envelope). You may want to print in different orientations, such as landscape or portrait mode. The Microsoft Foundation Class Library can't predict how your application will handle these issues, so it provides a protocol for you to add these capabilities.

The Printing Architecture

In an application built with the Microsoft Foundation Class Library, your view class and the framework cooperate in the printing process. Your view class has the following responsibilities:

- Inform the framework how many pages are in the document.
- When asked to print a specified page, draw that portion of the document.
- Allocate and deallocate any fonts or other GDI resources needed for printing.
- If necessary, send any escape codes needed to change the printer mode before printing a given page; for example, to change the printing orientation on a per-page basis.

The framework's responsibilities are as follows:

- Display the Print dialog box.

- Create a **CDC** object for the printer.
- Call the **StartDoc** and **EndDoc** member functions of the **CDC** object.
- Repeatedly call the **StartPage** member function of the **CDC** object, inform the view class which page should be printed, and call the **EndPage** member function of the **CDC** object.
- Call overridables in the view at the appropriate times.

To print a multipage document, the framework and view interact in the following manner. First the framework displays the Print dialog box, creates a device context for the printer, and calls the **StartDoc** member function of the **CDC** object. Then, for each page of the document, the framework calls the **StartPage** member function of the **CDC** object, instructs the view object to print the page, and then calls the **EndPage** member function. If the printer mode must be changed before starting a particular page, the view object sends the appropriate escape code by calling the **Escape** member function of the **CDC** object. When the entire document has been printed, the framework calls the **EndDoc** member function.

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. The following table lists these member functions.

Table 9.1 CView's Overridables for Printing

Name	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.

You can do printing-related processing in other functions as well, but these functions are the ones that drive the printing process.

Figure 9.1 illustrates the steps involved in the printing process and shows where each of **CView**'s printing member functions are called. The following sections explain these steps in more detail.

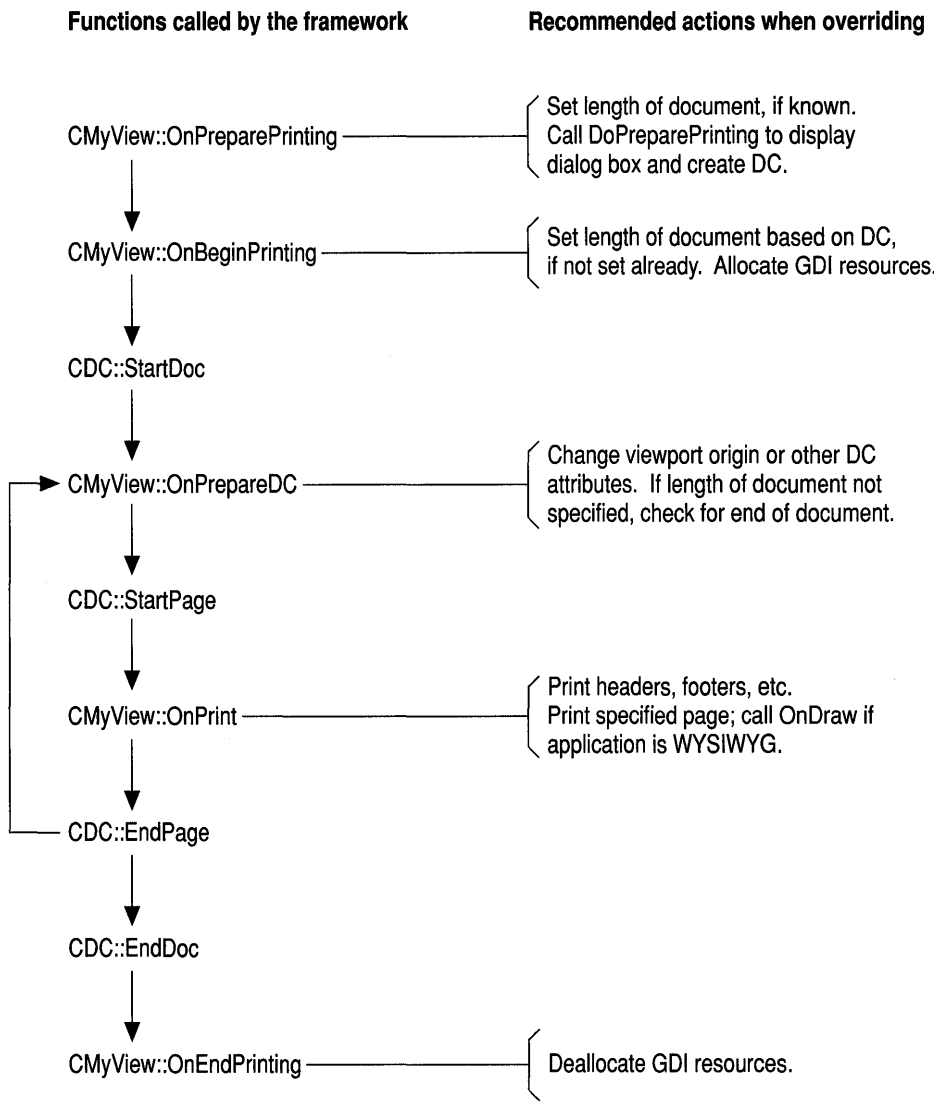


Figure 9.1 The Printing Loop

Pagination

The framework stores much of the information about a print job in an instance of the structure **CPrintInfo**. Several of the values in **CPrintInfo** pertain to pagination; these values are as shown in Table 9.2.

Table 9.2 Page Number Information Stored in CPrintInfo

Name(s)	Page Number Referenced
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 printing.

Page numbers start at 1; that is the first page is numbered 1, not 0. For more information about these and other members of **CPrintInfo**, see the *Class Library Reference*.

At the beginning of the printing process, the framework calls the view's **OnPreparePrinting** member function, passing a pointer to a **CPrintInfo** structure. AppWizard provides an implementation of **OnPreparePrinting** that calls **DoPreparePrinting**, another member function of **CView**. **DoPreparePrinting** is the function that displays the Print dialog box and creates a printer device context.

At this point the application doesn't know how many pages are in the document; it uses the default values 1 and 0xFFFF for the numbers of the first and last page of the document. If you know how many pages your document has, override **OnPreparePrinting** and call **SetMaxPage** for the **CPrintInfo** structure before you send it to **DoPreparePrinting**; this lets you specify the length of your document.

DoPreparePrinting then displays the Print dialog box; when it returns, the **CPrintInfo** structure contains the values specified by the user. If the user wishes to print only a selected range of pages, he or she can specify the starting and ending page numbers in the Print dialog box; the framework retrieves these values using the **GetFromPage** and **GetToPage** functions. If the user doesn't specify a page range, the framework calls **GetMinPage** and **GetMaxPage** and uses the values returned to print the entire document.

For each page of a document to be printed, the framework calls two member functions in your view class, **OnPrepareDC** and **OnPrint**, and passes each function two parameters: a pointer to a **CDC** object and a pointer to a **CPrintInfo** structure. Each time the framework calls **OnPrepareDC** and **OnPrint**, it passes a different value in the **m_nCurPage** member of the **CPrintInfo** structure. In this way the framework tells the view which page should be printed.

Recall that the **OnPrepareDC** member function is also used for screen display; it makes adjustments to the device context before drawing takes place. **OnPrepareDC** serves a similar role in printing, but there are a couple of differences: First, the **CDC** object represents a printer device context instead of a screen device context, and second, a **CPrintInfo** object is passed as a second parameter. (This parameter is **NULL** when **OnPrepareDC** is called for screen display.) Override **OnPrepareDC** to make adjustments to the device context based on which page is being printed; for example, you can move the viewport origin and the clipping region to ensure that the appropriate portion of the document gets printed.

The **OnPrint** member function performs the actual printing of the page. Earlier in this chapter, in the description of default printing, you saw that the framework calls **OnDraw** with a printer device context to perform printing. More precisely, the framework calls **OnPrint** with a **CPrintInfo** structure and a device context, and **OnPrint** passes the device context to **OnDraw**. Override **OnPrint** to perform any rendering that should be done only during printing and not for screen display; for example, to print headers or footers (see the section "Headers and Footers" following for more information). Then call **OnDraw** to do the rendering common to both screen display and printing.

The fact that **OnDraw** does the rendering for both screen display and printing means that your application is WYSIWYG: "What you see is what you get." However, suppose you aren't writing a WYSIWYG application. For example, consider a text editor that uses a bold font for printing but displays control codes to indicate bold text on the screen. In such a situation, you use **OnDraw** strictly for screen display. When you override **OnPrint**, substitute the call to **OnDraw** with a call to a separate drawing function. That function draws the document the way it appears on paper, using the attributes that you don't display on the screen.

Printer Pages vs. Document Pages

When you refer to page numbers, it's sometimes necessary to distinguish between the printer's concept of a page and a document's concept of a page. From the point of view of the printer, a page is one sheet of paper. However, one sheet of paper doesn't necessarily equal one page of the document. For example, if you're printing a newsletter, where the sheets are to be folded, one sheet of paper might contain both the first and last pages of the document, side by side. Similarly, if you're printing a spreadsheet, the document doesn't consist of pages at all; instead, one sheet of paper might contain rows 1 through 20, columns 6 through 10.

All the page numbers in the **CPrintInfo** structure refer to printer pages, The framework calls **OnPrepareDC** and **OnPrint** once for each sheet of paper that will pass through the printer. When you override the **OnPreparePrinting** function to specify the length of the document, you must use printer pages. If there's a one-to-one correspondence (that is, one printer page equals one document page), then this is easy. If, on the other hand, document pages and printer pages do not directly correspond, you must translate between them. For example, consider printing a spreadsheet. When overriding **OnPreparePrinting**, you must calculate how many sheets of paper will be required to print the entire spreadsheet and then use that value when calling the **SetMaxPage** member function of **CPrintInfo**. Similarly, when overriding **OnPrepareDC**, you must translate **m_nCurPage** into the range of rows and columns that will appear on that particular sheet and then adjust the viewport origin accordingly.

Print-Time Pagination

In some situations, your view class may not know in advance how long the document is until it has actually been printed. For example, suppose your application isn't WYSIWYG, so a document's length on the screen doesn't correspond to its length when printed.

This causes a problem when you override **OnPreparePrinting** for your view class: you can't pass a value to the **SetMaxPage** function of the **CPrintInfo** structure, because you don't know the length of a document. If the user doesn't specify a page number to stop at using the Print dialog box, the framework doesn't know when to stop the print loop. The only way to determine when to stop the print loop is to print out the document and see when it ends; your view class must check for the end of the document while it is being printed, and then inform the framework when the end is reached.

The framework relies on your view class's **OnPrepareDC** function to tell it when to stop. After each call to **OnPrepareDC**, the framework checks a member of the **CPrintInfo** structure called **m_bContinuePrinting**. Its default value is **TRUE**; as long as it remains so, the framework continues the print loop. If it is set to **FALSE**, the framework stops. To perform print-time pagination, override the **OnPrepareDC** to check whether the end of the document has been reached, and set **m_bContinuePrinting** to **FALSE** when it has.

The default implementation of **OnPrepareDC** sets **m_bContinuePrinting** to **FALSE** if the current page is greater than 1. This means that if the length of the document wasn't specified, the framework assumes the document is one page long. One consequence of this is that you must be careful if you call the base class version of **OnPrepareDC**; do not assume that **m_bContinuePrinting** will be **TRUE** after calling the base class version.

Headers and Footers

When looking at a document on the screen, the name of the document and your current location in the document are commonly displayed in a title bar and a status bar. When looking at a printed copy of a document, it's useful to have the name and page number shown in a header or footer. This is a common way in which even WYSIWYG programs differ in how they perform printing and screen display.

The **OnPrint** member function is the appropriate place to print headers or footers because it is called for each page, and because it is called only for printing, not for screen display. You can define a separate function to print a header or footer, and pass it the printer device context from **OnPrint**. You may need to adjust the window origin or extent before calling **OnDraw** to avoid having the body of the page overlap the header or footer. You might also have to modify **OnDraw** because the amount of the document that fits on the page could be reduced.

One way to compensate for the area taken by the header or footer is to use the **m_rectDraw** member of **CPrintInfo**. Each time a page is printed, this member is initialized with the usable area of the page. If you print a header or footer before printing the body of the page, you can reduce the size of the rectangle stored in **m_rectDraw** to account for the area taken by the header or footer. Then **OnPrint** can refer to **m_rectDraw** to find out how much area remains for printing the body of the page.

You cannot print a header, or anything else, from **OnPrepareDC**, because it is called before the **StartPage** member function of **CDC** has been called. At that point, the printer device context is considered to be at a page boundary. You can perform printing only from the **OnPrint** member function.

Allocating GDI Resources for Printing

Suppose you need to use certain fonts, pens, or other GDI objects for printing, but not for screen display. Because of the memory they require, it's inefficient to allocate these objects when the application starts up; when the application isn't printing a document, that memory might be needed for other purposes. It's better to allocate them when printing begins, and then delete them when printing ends.

To allocate these GDI objects, override the **OnBeginPrinting** member function. This function is well suited to this purpose for two reasons: the framework calls this function once at the beginning of each print job and, unlike **OnPreparePrinting**, this function has access to the **CDC** object representing the printer device driver. You can store these objects for use during the print job by defining member variables in your view class that point to GDI objects (for example, **CFont** * members, etc.).

To use the GDI objects you've created, select them into the printer device context in the **OnPrint** member function. If you need different GDI objects for different pages of the document, you can examine the **m_nCurPage** member of the **CPrintInfo** structure and select the GDI object accordingly. If you need a GDI object for several consecutive pages, Windows requires that you select it into the device context each time **OnPrint** is called.

To deallocate these GDI objects, override the **OnEndPrinting** member function. The framework calls this function at the end of each print job, giving you the opportunity to deallocate printing-specific GDI objects before the application returns to other tasks.

Enhance Scribble's Printing

Step 5 of Scribble adds the following printing capabilities to the program:

- Enlarging the printed image to a more comfortable size.
- Paginating a Scribble document.
- Adding a page header.

The following sections describe these enhancements in detail.

Enlarge the Printed Image

Recall from Chapter 8 that when you specify a position for a GDI drawing function, you use logical coordinates. Chapter 8 described how **CScrollView** moves the origin of this coordinate system. You can also control the scale of this coordinate system, that is, the physical size of a logical unit. By default, GDI considers logical units to be equal to device units, meaning that 1 logical unit equals 1 pixel on the screen. This interpretation of logical units is called the **MM_TEXT** mapping mode.

Since Scribble uses the **MM_TEXT** mapping mode, it considers a stroke that is 100 units long to be 100 pixels long. The physical size of the stroke depends on the device that displays it. For example, a device unit on a typical laser printer is 1/300 of an inch, which is considerably smaller than a pixel on a typical screen. As a result, the images that Scribble produces on a printer are much smaller than those it displays on the screen.

To keep Scribble from producing tiny images on the printer, you need a mapping mode that ensures a drawing remains the same size no matter what device displays it. Windows provides several such mapping modes, known as "metric" mapping modes. In these modes, GDI considers logical units to be equal to real-world units (or "metrics"), such as millimeters or inches.

In step 5, Scribble changes to the **MM_LOENGLISH** mapping mode, which treats each logical unit as 0.01 inches. In this mode, a stroke that is 100 logical units long is drawn as 1 inch long, no matter which device is used; each device driver determines how many device units are needed to draw a 1-inch stroke.

Once Scribble uses the **MM_LOENGLISH** mode, all coordinates used for GDI drawing are in hundredths-of-an-inch, not pixels. As a result, the images that Scribble displays on the printer are the same size as the ones it displays on the screen. Recall that in Chapter 8 a Scribble drawing was defined to be 800 logical units across and 900 logical units high; now a drawing is 8 inches across and 9 inches high.

Specify the Mapping Mode

You must specify the mapping mode when you call the **SetScrollSizes** member function defined by **CScrollView**. Recall from Chapter 8 that this function sets the view's scrolling limits. **SetScrollSizes** is called from the `OnInitialUpdate` member function of `CScribView`.

▶ **To specify the mapping mode**

- Launch the Visual Workbench and load the file SCRIBVW.CPP. Make the following modification:

```
void CScribView::OnInitialUpdate()
{
    SetScrollSizes( MM_LOENGLISH, GetDocument()->GetDocSize() );
}
```

Recall that `OnInitialUpdate` is called immediately after the view is attached to the document. This lets the view set its mapping mode before `OnDraw` is called.

Reversing the Sign of the Y-Coordinates

Another feature of the **MM_LOENGLISH** mode (as well as the other metric mapping modes) is that its y-axis runs in the opposite direction to that in **MM_TEXT** mode. In **MM_TEXT** mode, y-coordinates increase when you move down, but in all the metric mapping modes, y-coordinates increase when you move up.

Even though Scribble has changed the direction of the y-axis for drawing, most of the code doesn't require any modifications. This is because the **DPtoLP** function performs the conversion for you. Consider: when a point is received with a mouse message, its coordinates are converted by the **DPtoLP** function before being stored in a `CStroke` object. This means its y-coordinates are converted from a positive number of pixels to a negative number of inches. Those coordinates are then passed to the **LineTo** drawing function, and then it's up to the device driver for the screen to determine how many pixels are equivalent to the value that was passed in inches. You never have to directly examine the value of the coordinates.

However, there are some places where the reversal of the y-axis does have an impact. The mapping mode used by GDI is a characteristic of a device context; functions that don't use a device context are unaffected by the mapping mode. The member functions of the **CRect** class don't use the mapping mode; consequently, you must make some adjustments wherever Scribble uses **CRect** functions.

▶ To compensate for the reversal of the y-axis

1. First you must correct the way a bounding rectangle for a stroke is calculated. Load the file SCRIBDOC.CPP in Visual Workbench and make the following modifications to the `FinishStroke` member function of the `CStroke` class:

```
void CStroke::FinishStroke()
{
    // ...
    m_rectBounding = CRect(pt.x, pt.y, pt.x, pt.y);

    for (int i=1; i < m_pointArray.GetSize(); i++)
    {
        // If the point lies outside of the accumulated bounding
        // rectangle, then inflate the bounding rect to include it.
        pt = GetPoint(i);
        m_rectBounding.left     = min(m_rectBounding.left, pt.x);
        m_rectBounding.right    = max(m_rectBounding.right, pt.x);
        m_rectBounding.top      = max(m_rectBounding.top, pt.y);
        m_rectBounding.bottom   = min(m_rectBounding.bottom, pt.y);
    }

    // Add the pen width to the bounding rectangle.  This is needed
    // to account for the width of the stroke when invalidating
    // the screen.
    m_rectBounding.InflateRect(CSize(m_nPenWidth,-(int)m_nPenWidth));
    return;
}
```

When you inflate the top and bottom borders of the bounding rectangle to include each point, the direction in which the borders move is reversed. Also, the y-value of the pen width's sign is reversed before it is added to the bounding rectangle.

You also must make a correction when using the invalid rectangle. Recall that the `OnDraw` member function checks whether the invalid rectangle intersects the bounding rectangle for each stroke. The **IntersectRect** member function of **CRect** assumes that the bottom of a rectangle must have a larger y-coordinate than that of the top; it cannot find the intersection of two rectangles whose bottoms have smaller y-coordinates than their tops.

2. In SCRIBVW.CPP, make the following modifications to the `OnDraw` member function of `CScribView`:

```
void CScribView::OnDraw(CDC* pDC)
{
    CScribDoc* pDoc = GetDocument();

    // Get the invalidated rectangle of the view, or in the case
    // of printing, the clipping region of the printer dc.
    CRect rectClip;
    CRect rectStroke;
    pDC->GetClipBox(&rectClip);
    pDC->LPtoDP(&rectClip);

    // Note: CScrollView::OnPaint() will have already adjusted the
    // viewport origin before calling OnDraw(), to reflect the
    // currently scrolled position.

    // The view delegates the drawing of individual strokes to
    // CStroke::DrawStroke().
    for (POSITION pos = pDoc->GetFirstStrokePos(); pos != NULL; )
    {
        CStroke* pStroke = pDoc->GetNextStroke(pos);
        rectStroke = pStroke->GetBoundingRect();
        pDC->LPtoDP(&rectStroke);
        if (!rectStroke.IntersectRect(&rectStroke, &rectClip))
            continue;
        pStroke->DrawStroke(pDC);
    }
}
```

Both the invalidated rectangle and the bounding rectangle are converted to device coordinates (reversing the signs of their y-coordinates) before being tested for intersection.

Paginate Scribble Documents

If Scribble allowed you to produce arbitrarily large drawings, it would make sense for the program to break up a drawing into pages by dividing it into a grid of *m* by *n* rectangles, the values of *m* and *n* being determined by the size of the drawing. However, Scribble supports drawings of only one size, and each one fits on a single page. To illustrate pagination, step 5 of Scribble prints each drawing as a two-page document: a title page, and the drawing itself.

▶ To add pagination to Scribble

1. Load the SCRIBVW.CPP file in Visual Workbench and make the following additions to `CScribView`'s `OnPreparePrinting` member function:

```
BOOL CScribView::OnPreparePrinting( CPrintInfo* pInfo )
{
    pInfo->SetMaxPage(2);  // the document is two pages long:
                           //  the first page is the title page
                           //  the second page is the drawing
    // default preparation
    return DoPreparePrinting(pInfo);
}
```

   This function specifies the length of the document by calling **SetMaxPage** for the *pInfo* parameter. Since all Scribble documents are two pages long, the function uses a numeric constant rather than a variable to represent the number of the last page of the document. The title page and the drawing page are numbered 1 and 2, respectively. Note that the function still retains a call to **DoPreparePrinting** at the end; this displays the Print dialog box and creates a device context for the printer.

2. Load SCRIBVW.H in Visual Workbench and make the following changes to `CScribView`'s class declaration:

```
class CScribView : public CScrollView
{
public:
    // ...
    virtual void OnUpdate( CView* pSender, LPARAM lHint = 0L,
                           CObject* pHint = NULL );
    void PrintTitlePage(CDC* pDC, CPrintInfo* pInfo);
    void PrintPageHeader(CDC* pDC, CPrintInfo* pInfo,
                         CString& strHeader);
    virtual void OnPrint(CDC* pDC, CPrintInfo* pInfo);
    // ...
}
```

   To perform printing, `CScribView` overrides the **OnPrint** member function and defines two new helper functions: `PrintTitlePage`, which prints the title page, and `PrintPageHeader`, which prints a header on the drawing page.

3. In SCRIBVW.CPP, after the `OnEndPrinting` member function, add the following definition of the `OnPrint` member function:

```
void CScribView::OnPrint(CDC* pDC, CPrintInfo* pInfo)
{
    if (pInfo->m_nCurPage == 1)  // page no. 1 is the title page
    {
        PrintTitlePage(pDC, pInfo);
        return; // nothing else to print on page 1 but the page title
    }
    CString strHeader = GetDocument()->GetTitle();

    PrintPageHeader(pDC, pInfo, strHeader);
    // PrintPageHeader() subtracts out from the pInfo->m_rectDraw the
    // amount of the page used for the header.

    pDC->SetWindowOrg(pInfo->m_rectDraw.left,-pInfo->m_rectDraw.top);

    // Now print the rest of the page
    OnDraw(pDC);
}
```

The behavior of the `OnPrint` member function depends on which of the two pages is being printed. If the title page is being printed, `OnPrint` simply calls the `PrintTitlePage` function and then returns. If it's the drawing page, `OnPrint` calls `PrintPageHeader` to print the header and then calls `OnDraw` to do the actual drawing. Before calling `OnDraw`, `OnPrint` sets the window origin at the upper-left corner of the rectangle defined by **m_rectDraw**; this rectangle was reduced by `PrintPageHeader` to account for the size of the header. This keeps the drawing from overlapping the header.

Notice that the drawing itself isn't divided into multiple pages. Consequently, `OnDraw` never has to display just a portion of the drawing (for example, it never has to display the section that fits on a particular page without displaying the surrounding sections). Either the title page is being printed and `OnDraw` isn't called at all, or else the drawing page is being printed and `OnDraw` displays the entire drawing at once.

This also explains why `CScribView` doesn't override the **OnPrepareDC** member function: there's no need to adjust the viewport origin or clipping region depending on which page is being printed.

4. In SCRIBVW.CPP, below your definition of `OnPrint`, define the `PrintTitlePage` member function as follows:

```
void CScribView::PrintTitlePage(CDC* pDC, CPrintInfo* pInfo)
{
    // Prepare a font size for displaying the file name
    LOGFONT logFont;
    memset(&logFont, 0, sizeof(LOGFONT));
    logFont.lfHeight = 75; //  3/4th inch high in MM_LOENGLISH
                           // (1/100th inch)
    CFont font;
    CFont* pOldFont = NULL;
    if (font.CreateFontIndirect(&logFont))
        pOldFont = pDC->SelectObject(&font);

    // Get the file name, to be displayed on title page
    CString strPageTitle = GetDocument()->GetTitle();

    // Display the file name 1 inch below top of the page,
    // centered horizontally
    pDC->SetTextAlign(TA_CENTER);
    pDC->TextOut(pInfo->m_rectDraw.right/2, -100, strPageTitle);

    if (pOldFont != NULL)
        pDC->SelectObject(pOldFont);
}
```

The `PrintTitlePage` function uses **m_rectDraw**, which stores the usable drawing area of the page, as the rectangle in which the title should be centered.

Notice that `PrintTitlePage` declares a local **CFont** object to use when printing the title page. If you needed the font for the entire printing process, you could declare a **CFont** member variable in your view class, create the font in the **OnBeginPrinting**, and destroy it in **EndPrinting**. However, since Scribble uses the font for just the title page, the font doesn't have to exist beyond the `PrintTitlePage` function. When the function ends, the destructor is automatically called for the local **CFont** object.

Add a Page Header

As mentioned earlier, `CScribView` defines the `PrintPageHeader` function, which is called by `OnPrint` before the drawing itself is printed.

▶ **To add a page header to the drawing**

- In SCRIBVW.CPP, define the `PrintPageHeader` member function as follows:

```
void CScribView::PrintPageHeader(CDC* pDC, CPrintInfo* pInfo,
    CString& strHeader)
{
    // Print a page header consisting of the name of
    // the document and a horizontal line
    pDC->TextOut(0,-25, strHeader);  // 1/4 inch down

    // Draw a line across the page, below the header
    TEXTMETRIC textMetric;
    pDC->GetTextMetrics(&textMetric);
    int y = -35 - textMetric.tmHeight;   // line 1/10th in. below text
    pDC->MoveTo(0, y);                   // from left margin
    pDC->LineTo(pInfo->m_rectDraw.right, y); //  to right margin

    // Subtract from the drawing rectangle the space used by header.
    y -= 25;    // space 1/4 inch below (top of) line
    pInfo->m_rectDraw.top += y;
}
```

The `PrintPageHeader` member function prints the name of the document at the top of the page, and then draws a horizontal line separating the header from the drawing. It adjusts the **m_rectDraw** member of the *pInfo* parameter to account for the height of the header; recall that `OnPrint` uses this value to adjust the window origin before it calls `OnDraw`.

The Print Preview Architecture

Print preview is somewhat different from screen display and printing because, instead of directly drawing an image on a device, the application must simulate the printer using the screen. To accommodate this, the Microsoft Foundation Class Library defines a special class derived from **CDC**, called **CPreviewDC**. All **CDC** objects contain two device contexts, but usually they are identical. In a **CPreviewDC** object, they are different: the first represents the printer being simulated, and the second represents the screen on which output is actually displayed.

When you select the Print Preview command from the File menu, the framework creates a **CPreviewDC** object. Whenever your application performs an operation that sets a characteristic of the printer device context, the framework also performs a similar operation on the screen device context. For example, if your application selects a font for printing, the framework selects a font for screen display that simulates the printer font. Whenever your application would send output to the printer, the framework instead sends the output to the screen.

Print preview also differs from printing in the order each draws the pages of a document. During printing, the framework continues a print loop until a certain range of pages have been rendered. During print preview, one or two pages are

displayed at any time, and then the application waits; no further pages are displayed until the user responds. During print preview, the application must also respond to **WM_PAINT** messages, just as it does during ordinary screen display.

The **OnPreparePrinting** function is called when preview mode is invoked, just as it is at the beginning of a print job. The **CPrintInfo** structure passed to the function contains several members whose values you can set to adjust certain characteristics of the print preview operation. For example, you can set the **m_nNumPreviewPages** member to specify whether you want to preview the document in one-page or two-page mode.

If you know how long the document is and called **SetMaxPage** with the appropriate value, the framework can use this information in preview mode as well as during printing. Once the framework knows the length of the document, it can provide the preview window with a scroll bar, allowing the user to page back and forth through the document in preview mode. If you haven't set the length of the document, the framework cannot position the scroll box to indicate the current position, so the framework doesn't add a scroll bar. In this case, the user must use the Next Page and Previous Page buttons on the preview window's control bar to page through the document.

For print preview, you may find it useful to assign a value to the **m_nCurPage** member of **CPrintInfo**, even though you would never do so for ordinary printing. During ordinary printing, this member carries information from the framework to your view class; this is how the framework tells the view which page should be printed.

By contrast, when print preview mode is started, the **m_nCurPage** member carries information in the opposite direction: from the view to the framework. The framework uses the value of this member to determine which page should be previewed first. The default value of this member is 1, so the first page of the document is displayed initially. You can override **OnPreparePrinting** to set this member to the number of the page being viewed at the time the Print Preview command was invoked. This way, the application maintains the user's current position when moving from normal display mode to print preview mode.

Sometimes you may want **OnPreparePrinting** to perform different initialization depending on whether it is called for a print job or for print preview. You can determine this by examining the **m_bPreview** member variable in the **CPrintInfo** structure; this member is set to **TRUE** when print preview is invoked.

The **CPrintInfo** structure also contains a member named **m_strPageDesc**, which is used to format the strings displayed at the bottom of the screen in single-page and multiple-page modes. By default these strings are of the form "Page *n*" and "Pages *n* - *m*," but you can modify **m_strPageDesc** from within **OnPreparePrinting** and set the strings to something more elaborate. See **CPrintInfo** in the *Class Library Reference* for more information.

Enhance Scribble's Print Preview

The default print preview capabilities are almost sufficient for Scribble's needs. To some extent, Scribble's print preview has already been enhanced when the printing capabilities were enhanced. Recall that in the override of `OnPreparePrinting` you called the **SetMaxPages** function to specify the length of Scribble documents. This allows the framework to add a scroll bar to the preview window.

Another enhancement you can make is to change the number of pages displayed when preview mode is invoked.

▶ To set the number of pages displayed in preview mode

- In SCRIBVW.CPP, add the following line to the `OnPreparePrinting` member function:

```
BOOL CScribView::OnPreparePrinting( CDC* pDC, CPrintInfo* pInfo )
{
    pInfo->SetMaxPage(2);  // the document is two pages long:
                           //  the first page is the title page
                           //  the second page is the drawing
    pInfo->m_nNumPreviewPages = 2;   // Preview 2 pages at a time
    return( CView::OnPreparePrinting(pInfo) );
}
```

The line added here assigns the value 2 to **m_nNumPreviewPages**. This causes Scribble to preview both pages of the document at once: the title page (page 1) and the drawing page (page 2).

Compile the New Scribble

What does Scribble's printing look like now? Compile the new version of Scribble and find out.

▶ To compile Scribble

- From the Project menu in Visual Workbench, choose the Rebuild All command.

Run the new version of Scribble. Draw some strokes, and then choose the File Print Preview command. Switch back and forth between one-page and two-page display mode, or move to the previous or next page. Figure 9.2 shows this version of Scribble.

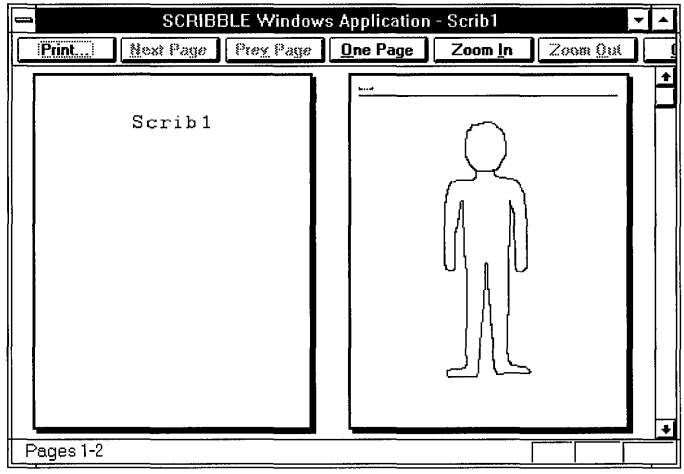


Figure 9.2 Scribble Version 5

Exit Scribble.

This completes step 5 in the tutorial. You now have a basic understanding of the printing architecture provided by the Microsoft Foundation Class Library.

In the next chapter, you'll add context-sensitive help to Scribble.

CHAPTER 10

Adding Context-Sensitive Help

So far, thanks to the Microsoft Foundation Class Library, Scribble implements a number of common user-interface features, such as print preview and splitter windows. To conclude the tutorial, this chapter adds another such feature to Scribble: context-sensitive Windows Help.

Note To complete this chapter, the Windows 3.1 Help Compiler, which is shipped with Microsoft Visual C++ Professional Edition as file HC31.EXE, must be in your path.

Scribble already offers the user some help in the form of prompt strings displayed in the status bar. When the user navigates through a menu using the UP ARROW and DOWN ARROW keys, or uses the mouse to press a toolbar button, Scribble displays a brief description of the command's purpose in the status bar (if the status bar is visible). The framework easily supplies this level of information for commands predefined by the class library. And, as you did in Chapter 5, you can add prompts to the menu items you create in App Studio by filling in a field in the menu's property page. Since prompts are attached to command IDs, Scribble's toolbar buttons, which duplicate commands on the menus, automatically invoke the appropriate prompts.

The level of help described in this chapter, however, goes much further. The user can open Windows Help for your application from the Help menu or invoke context-sensitive help by pressing the F1 key or SHIFT+F1.

This chapter explains how to implement:

- F1 help
- SHIFT+F1 help mode
- Help menu support

The next section explains the three kinds of help listed here.

For a quick preview of how easy it is to add context-sensitive help to your application, follow the instructions described in "See Context-Sensitive Help in Action" on page 186. In that section, you'll create a new application with AppWizard, build the application, and then run it to see the help features you get without adding a single line of code.

The chapter also shows how to add an AppWizard option to your program if you didn't select the option when you originally created your application.

For an overview of the framework's help support, see Chapter 5 in the *Class Library Reference*.

Note You can freely use the help files that AppWizard creates in your applications and freely ship the compiled help.

This chapter covers step 6 of Scribble. Unlike previous steps, the STEP6 subdirectory does not contain the complete source files for this step— it includes only a help source file named PEN.RTF. If you want to see the results of this step, you must follow the directions presented in this chapter, starting with the STEP5 source files.

Division of Labor

To support help, the framework:

- Handles F1 help.

  With an active window, dialog box, or message box, or with a menu item or toolbar button selected, the user can press the F1 key to summon specific help about the selected item.

  For menu items, help is summoned for the item currently highlighted. For toolbar buttons, the user can use the mouse to press the button and press F1 before letting the button up.

  You can define a key other than F1 for help, but it is common among applications for Windows to use F1.

- Handles SHIFT+F1 help mode.

  At any time the application is active, the user can press SHIFT+F1 to put the application into a "help mode." The cursor changes to a help cursor: an arrow beside a question mark.

  While the application is in help mode, clicking any window, dialog box, message box, menu item, or toolbar button summons specific help about the item. Selecting any item for help ends help mode and displays help. Pressing the ESCAPE key or switching to another application and back also ends help mode.

The standard toolbar provided by AppWizard also has a button through which the user can invoke help mode. The graphic on the button resembles the help cursor.

You can define a key combination other than SHIFT+F1, but it is common among applications for Windows to use F1.

- Provides the Index and Using Help commands on the Help menu.

  The Index command causes Windows Help to display an index to the available help topics. The Using Help command causes Windows Help to display information on using Windows Help.

- Provides a starter set of files in Rich-Text Format (RTF) containing standard help topics.

  These include commands on standard menus such as File and Edit, standard information on using help, standard keyboard shortcuts, a standard help index, and more.

To take advantage of this support for help:

- Use the AppWizard Context-Sensitive Help option.
- Write your application-specific help topics in the .RTF files.

  Fill in application-specific details in these help topics, add new topics, and delete unused topics.

- Provide finer-grained context-sensitive help, if desired.

  Fine-tune help further by overriding portions of the class library to support more specific help contexts, such as individual controls in a dialog box. For more information about fine-tuning context-sensitive help, see Technical Note 28 in MFCNOTES.HLP.

Implementing Context-Sensitive Help with AppWizard

Use AppWizard to enable the framework's support for context-sensitive help and the Help menu. The following sections explain how to select this support in AppWizard and what AppWizard creates as a result.

The Context-Sensitive Help Option

When you first create a new application with AppWizard, be sure to select the Context-Sensitive Help option if you plan to support help.

▶ **To select context-sensitive help**

1. In AppWizard, choose the Options button.

2. To select the Context-Sensitive Help option, click the Context-Sensitive Help check box.

   Figure 10.1 shows the Options dialog box in AppWizard with Context-Sensitive Help selected.

3. Select any other options you need.
4. Choose the OK button.
5. Complete your AppWizard session and choose the OK button in the main AppWizard dialog box.

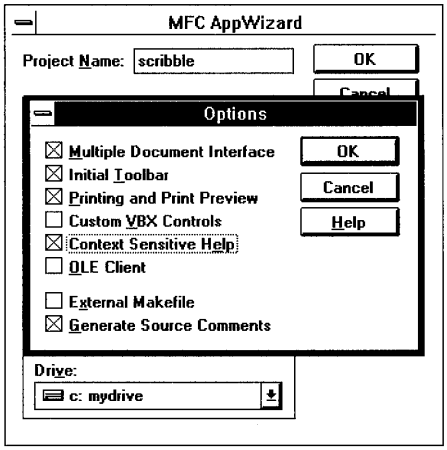


Figure 10.1 Selecting Context-Sensitive Help

When AppWizard creates your skeleton application, it adds the following items:

- Message-map entries in your derived application class for handler functions to handle Help menu items and F1 and SHIFT+F1 help. These handlers are predefined by the framework.
- Index and Using Help items in the menu definitions.
- Status-bar command prompts for the help items. These appear when the user clicks the mouse in one of the menu commands.
- A batch file called MAKEHELP.BAT that you can use to compile your help.
- A Windows Help project file with a .HPJ extension. It's named for your project.
- One or more RTF-format files (.RTF extension) containing standard help contexts. Add application-specific help contexts to these files to customize your help. For more information, see "Editing Scribble's Help Topics" on page 193
- Several bitmap files (.BMP extension) used in the help files.

You can then use the items created by AppWizard, add a few extra steps, and build your help file. These steps are described in later sections.

Note Help project files and Windows Help tools are explained in *Programming Tools for the Microsoft Windows Operating System.*

The Message Map

To support the Help menu commands, F1 help, and SHIFT+F1 help, AppWizard adds five entries to the message map for your **CWinApp**-derived application class. This message map is in the .CPP file named for your project. When you complete the steps outlined in "Adding Help After the Fact" on page 187, the message map for class `CScribbleApp` will look like the following:

```
// CScribbleApp

BEGIN_MESSAGE_MAP(CScribbleApp, CWinApp)
   //{{AFX_MSG_MAP(CScribbleApp)
      ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
      // NOTE - the ClassWizard will add and remove mapping macros here.
      //    DO NOT EDIT what you see in these blocks of generated code !
   //}}AFX_MSG_MAP
   // 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)
   // Global help commands
   ON_COMMAND(ID_HELP_INDEX, CWinApp::OnHelpIndex)
   ON_COMMAND(ID_HELP_USING, CWinApp::OnHelpUsing)
   ON_COMMAND(ID_HELP, CWinApp::OnHelp)
   ON_COMMAND(ID_CONTEXT_HELP, CWinApp::OnContextHelp)
   ON_COMMAND(ID_DEFAULT_HELP, CWinApp::OnHelpIndex)
END_MESSAGE_MAP()
```

The five help-related message map entries follow the comment `// Global help commands`. Table 10.1 explains the purpose of each command ID used in these entries.

Table 10.1 Help-Related Command IDs

Command ID	Purpose
ID_HELP_INDEX	Responds to the Index item on the Help menu by displaying the Windows Help index.
ID_HELP_USING	Responds to the Using Help item on the Help menu by displaying information about using Windows Help.

Table 10.1 Help-Related Command IDs *(continued)*

Command ID	Purpose
ID_HELP	Responds to F1 by displaying a specific topic in Windows Help.
ID_CONTEXT_HELP	Responds to SHIFT+F1 by putting the application into help mode.
ID_DEFAULT_HELP	Used when a specific help context cannot be found.

Notice that all of these commands are mapped to member functions of class **CWinApp**. Unlike most of the other commands you place into the message map, these have handler functions predefined by the class library. Making the message-map entry enables the command.

The application's accelerator table defines F1 for **ID_HELP** and SHIFT+F1 for **ID_CONTEXT_HELP**. You can change the keys used for these help functions by using App Studio to change the key values in the accelerator table.

The Help Project File

AppWizard also creates several help-related files in your project directory, including a help project file (.HPJ extension).

The help project file provides coordinating information used by the Windows Help Compiler. When you complete the steps outlined in "Adding Help After the Fact" on page 187, Scribble's help project file, SCRIBBLE.HPJ, will look like the following:

```
[OPTIONS]
CONTENTS=main_index
TITLE=SCRIBHLP Application Help
COMPRESS=true
WARNING=2

[FILES]
hlp\afxcore.rtf
hlp\afxprint.rtf

[BITMAPS]
; toolbar buttons for File commands
hlp\filenew.bmp
hlp\fileopen.bmp
hlp\fileprnt.bmp
hlp\filesave.bmp
```

```
; toolbar buttons for Edit commands
hlp\editcopy.bmp
hlp\editcut.bmp
hlp\editpast.bmp
hlp\editundo.bmp

...

[ALIAS]
HIDR_MAINFRAME = main_index
HIDR_SCRIBHTYPE = HIDR_DOC1TYPE
HIDD_ABOUTBOX = HID_APP_ABOUT

HID_HT_SIZE = HID_SC_SIZE
HID_HT_HSCROLL = scrollbars
HID_HT_VSCROLL = scrollbars
HID_HT_MINBUTTON = HID_SC_MINIMIZE
HID_HT_MAXBUTTON = HID_SC_MAXIMIZE
AFX_HIDP_INVALID_FILENAME   = AFX_HIDP_default
AFX_HIDP_FAILED_TO_OPEN_DOC = AFX_HIDP_default

...

[MAP]
#include <C:\MSVC\MFC\include\afxhelp.hm>
#include <hlp\scribhlp.hm>
```

This file describes options used by the Windows Help Compiler, topic files with the .RTF extension to be included in the help build, bitmap files to be included in the build, a mapping of context strings to context numbers, and more.

Of particular interest is the [MAP] section, which in this example points to two included files with the .HM (help mapping) extension. The next section explains more about help-context mapping. For more information about Windows Help project files, see *Programming Tools for the Microsoft Windows Operating System.*

The MAKEHELP.BAT File

In Windows Help, a "help context" consists of a string and a number. The help context string is what the help text author uses to identify help topics. The help context number is what the programmer uses to identify help topics. Help author and programmer come together in the [MAP] section of the .HPJ file, which associates the help context string and number. When your application calls Windows Help, Windows Help uses the context your application passes to locate and display the help topic denoted by that context. At run time, the framework manages supplying the appropriate help context.

To facilitate relating the windows, dialog boxes, and commands in your application to Windows Help contexts, the Microsoft Foundation Class Library provides the MAKEHM.EXE tool, which creates the information used in the [MAP] section of the .HPJ file.

AppWizard creates a MAKEHELP.BAT file that you'll use to compile your help. MAKEHELP.BAT calls MAKEHM.EXE and then the Windows Help Compiler.

When you use App Studio to create dialog-template resources, menu commands, and the like, App Studio writes **#define** statements in a file named RESOURCE.H (by default). For example, there might be **#define** statements for such symbols as `IDD_MY_DIALOG` and `ID_PEN_WIDTHS`. For more information about how App Studio adds symbols to RESOURCE.H and how you can view and manipulate them with App Studio's Symbol Browser, see Chapter 2 in the *App Studio User's Guide*.

When you run MAKEHELP.BAT from the MS-DOS® command line, it calls the MAKEHM tool to map the **#define** statements in RESOURCE.H to Windows Help strings in a .HM file. The MAKEHM tool collects **#define** statements from RESOURCE.H and uses an algorithm to map defined symbols to help strings in a .HM file. For the example IDs in the previous paragraph, it would create help strings such as `HIDD_MY_DIALOG` and `HID_PEN_WIDTHS`. These context strings are formed by prefixing an "H" to the symbol found in RESOURCE.H. The algorithm also maps the ID's numeric value to a corresponding number for the help context. An example is shown in section "Help Contexts in Scribble" on page 192.

When MAKEHELP.BAT then runs the Windows Help Compiler, the compiler uses the .HM files pointed to by the .HPJ file to set up the help contexts in your new help file. Once you finish compiling your .HLP file, you can use it from your application.

For more information about MAKEHELP.BAT and MAKEHM.EXE, see the section "Context-Sensitive Help" in Chapter 5 of the *Class Library Reference*.

See Context-Sensitive Help in Action

It isn't necessary to follow the steps described in the next section, "Adding Help to Scribble," to try out the help support provided by the framework and AppWizard. You can try it out now.

▶ **To give the help support a try**

1. Run AppWizard with the help option selected.

   When you run AppWizard, specify a project named MYHELP with a path of MFC\SAMPLES\SCRIBBLE\MYHELP. Select the Context-Sensitive Help option in the Options dialog box. AppWizard creates help-related files for the new application.

2. Build the MYHELP application.

It's not necessary to modify any of the code created by AppWizard. Simply build the MYHELP application that AppWizard just created.

3. Run MAKEHELP.BAT from the MS-DOS command line to build the .HLP file.

   As the Windows Help Compiler runs, it prints a row of dots on the screen.

Run the MYHELP application and try out various help options. Here are some suggestions for what to try:

- Choose Using Help from the Help menu. See the standard help provided by WINHELP's own help file.
- Choose Index from the Help menu. See the standard main help topic that AppWizard has prepared. It describes the standard menus that the framework provides.
- Click the help-mode button on the toolbar, which appears as an arrow beside a question mark. To get help for a menu item, drop down a menu and click a menu item with the mouse. Click the help-mode button again and then click another toolbar button. Finally, enter help mode again by pressing the SHIFT+F1 keys; then click the toolbar itself, or a window's title bar, or some other element of MYHELP's user interface.
- Using the keyboard, drop down a menu and select a menu item using the DOWN ARROW key. Then press the F1 key to get help for the selected item.

Thanks to AppWizard and the framework, you—and your users—get all of this help essentially for free.

Adding Help to Scribble

The normal way to add help to an application is to select the Context-Sensitive Help option when you first run AppWizard, as just described above. However, to simplify the previous chapters this was not done in Scribble step 0, so it's necessary to add the option after the fact.

Adding Help After the Fact

This section explains how to add context-sensitive help at a later stage of program development. The general procedures apply to any AppWizard option.

Merging context-sensitive help support into Scribble at this late stage requires several general steps. Each step is explained in more detail below. The overall steps are:

1. Create a new MYHELP application from which to borrow code and resources for Scribble. See the following procedure "To Create a New MYHELP Application."

The idea is to create a starter application as in Chapter 2 that this time has the help-related files and code.

2. Copy resources from the MYHELP application to Scribble. See the procedure "To Copy Resources to Scribble" later in this section.

   You'll use App Studio to copy the resources.

3. Copy help-related code from the MYHELP application to Scribble. See the procedure "To Copy Help-Related Code to Scribble" later in this section.

   You'll use the Visual Workbench editor (or any text editor) to copy the code.

4. Copy help-related files from the MYHELP directory to your MYSCRIB directory. See the procedure "To Copy Help-Related Files to MYSCRIB" later in this section.

5. Build the new version of Scribble and compile its help file. See the procedure "To Complete Scribble's Help" later in this section.

▶ To create a new MYHELP application

- If you haven't done so already, run AppWizard to create a new MYHELP application, as described in "See Context-Sensitive Help in Action" on page 186.

  It's unnecessary to build the MYHELP application. You're about to borrow code and resources from this application for Scribble.

  The Scribble end of this procedure begins with the files from Chapter 9 (step 5) in your MYSCRIB directory. If you have not done the tutorial step in Chapter 9, you can copy all of the files and subdirectories in the MFC\SAMPLES\SCRIBBLE\STEP5 subdirectory to your MYSCRIB directory.

To perform the steps in the next procedure, you'll use App Studio's menu, accelerator, and string editors. These editors are explained in Chapters 4, 5, and 6, respectively, in the *App Studio User's Guide*.

▶ To copy resources to Scribble

1. Run App Studio and open the resource files for both MYSCRIB and MYHELP.

   You're about to copy resources from MYHELP to MYSCRIB. As you do this, you'll not only learn about adding help to an application after the fact, you'll also learn how easy it is to copy resources from one resource file to another using App Studio.

   You'll copy menu items, accelerator keys, and status-bar prompt strings.

2. Use the App Studio menu editor to open menus from both resource files.

   Arrange the menu editor windows so they don't overlap.

3. Drop down both Help menus.

4. Click on the separator below the Using Help item in MYHELP's Help menu. Then hold down the SHIFT key and click on the Index and Using Help items. Release the SHIFT key.

   This selects the separator and the two menu items.

5. Hold down the CTRL key, click on the highlighted menu items, and drag them to the Help menu in MYSCRIB, above the About Scribble menu item. Release the mouse button and the CTRL key.

   The menu items and the separator are copied to Scribble.

6. Use the App Studio accelerator editor to copy the accelerator keys F1 and SHIFT+F1 for the **ID_HELP** command and the **ID_CONTEXT_HELP** command, respectively.

   The copying procedure is similar to copying menus. To copy the two accelerators, hold down the SHIFT key while selecting them. Then hold down the CTRL key while dragging the accelerators to the new window.

7. Use the App Studio string editor to (a) delete the existing **AFX_IDS_IDLEMESSAGE** string from string segment 0 in MYSCRIB, and (b) copy the following status-bar prompt strings to MYSCRIB: **AFX_IDS_IDLEMESSAGE**, **AFX_IDS_HELPMODEMESSAGE**, **ID_HELP_INDEX**, **ID_CONTEXT_HELP**, **ID_HELP_USING**, and **ID_HELP**.

   The copying procedure is similar to the procedures for copying menus and accelerators. To delete a string, select it in the string editor and choose the Delete button. To copy several strings, hold the SHIFT key down while selecting the strings. Then hold the CTRL key down while dragging the selected strings to the new window.

   For an application without help, AppWizard defines the default status-bar prompt to be "Ready". This is the string that is displayed in the status bar when no other command prompt is being displayed. This string is identified as **AFX_IDS_IDLEMESSAGE**.

   The other strings are command prompts for the Index and Using Help commands on the Help menu and for F1 and SHIFT+F1 help.

8. Save the MYSCRIB resource file, SCRIBBLE.RC. You can close App Studio if you wish.

▶ To copy help-related code to Scribble

1. Copy the help-related entries in the message map for class `CMyhelpApp` to the corresponding message map in class `CScribbleApp`.

   Using the Visual Workbench editor, open MYHELP.CPP and SCRIBBLE.CPP. Copy the help-related lines, marked with the ▸ symbol, from the message map in

MYHELP.CPP and paste them into the same position in the message map in SCRIBBLE.CPP. The message map looks like the following:

```
// CMyHelpApp

BEGIN_MESSAGE_MAP(CMyhelpApp, CWinApp)
   //{{AFX_MSG_MAP(CMyhelpApp)
   ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
      // NOTE - the ClassWizard will add and remove mapping
      // macros here.
      //    DO NOT EDIT what you see in these blocks of
      //    generated code !
   //}}AFX_MSG_MAP
   // 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)

   // Global help commands
   ON_COMMAND(ID_HELP_INDEX, CWinApp::OnHelpIndex)
   ON_COMMAND(ID_HELP_USING, CWinApp::OnHelpUsing)
   ON_COMMAND(ID_HELP, CWinApp::OnHelp)
   ON_COMMAND(ID_CONTEXT_HELP, CWinApp::OnContextHelp)
   ON_COMMAND(ID_DEFAULT_HELP, CWinApp::OnHelpIndex)
END_MESSAGE_MAP()
```

2. Enable the help-mode toolbar button.

   AppWizard includes a toolbar button for help mode in the toolbar bitmap regardless of whether you choose the help option. This button did not appear on the screen when you ran previous versions of Scribble because the button was not mapped to any command in the `buttons` array defined in the MAINFRM.CPP file. The help-mode button has the rightmost position in the toolbar. Up to now there was one less entry in the `buttons` array than there were buttons in the toolbar bitmap.

   To expose the help-mode button, add the command **ID_CONTEXT_HELP** to the end of the list of commands in the `buttons` array in the MAINFRM.CPP file for Scribble. Add the line marked with the ▸ symbol in the left margin, as shown in this code:

```
// toolbar buttons - IDs are command buttons
static UINT BASED_CODE buttons[] =
{
   // same order as in the bitmap 'toolbar.bmp'
   ID_FILE_NEW,
   ID_FILE_OPEN,
   ID_FILE_SAVE,
      ID_SEPARATOR,
   ID_EDIT_CUT,
   ID_EDIT_COPY,
   ID_EDIT_PASTE,
      ID_SEPARATOR,
   ID_PEN_THICK_OR_THIN,
      ID_SEPARATOR,
   ID_FILE_PRINT,
   ID_APP_ABOUT,

   ID_CONTEXT_HELP,
};
```

▶ To copy help-related files to MYSCRIB

1. Copy the MAKEHELP.BAT and MYHELP.HPJ files from the MYHELP directory to the MYSCRIB directory.
2. In the MYSCRIB directory, rename MYHELP.HPJ as SCRIBBLE.HPJ.
3. In the copy of MAKEHELP.BAT in the MYSCRIB directory, change all occurrences of the string "myhelp" to "scribble."
4. In SCRIBBLE.HPJ, make the following changes:
   - Under the [FILES] section, add the line

     ```
     hlp\pen.rtf
     ```

     The source and purpose of the new file PEN.RTF is explained below.
   - Under the [OPTIONS] section, change "CONTENTS=main_index" to "CONTENTS=new_index."

     The new help topic source file, PEN.RTF, will replace the main help topic that AppWizard originally created.
   - Under the [ALIAS] section, change the string "HIDR_MAINFRAME = main_index" to "HIDR_MAINFRAME = new_index."
   - Also under the [ALIAS] section, change "HIDR_MYHELPTYPE" to "HIDR_SCRIBTYPE."
5. Create a subdirectory called HLP in your MYSCRIB directory.
6. Copy the PEN.RTF file from the MFC\SAMPLES\SCRIBBLE\STEP6 directory to MYSCRIB\HLP.

PEN.RTF contains help topics specific to Scribble's Pen menu. The remaining sections of this chapter show you some of the contents of this .RTF file and explain how you would author the help topics using a program that can edit .RTF files, such as Microsoft Word for Windows.

Note that the only file provided in the MFC\SAMPLES\SCRIBBLE\STEP6 directory is PEN.RTF.

▶ **To complete Scribble's help**

1. Run MAKEHELP.BAT to compile your help file.
2. Compile Scribble and test its various help features.

Once you have successfully built Scribble and compiled its help file, run your new version of Scribble and try out its context-sensitive help.

▶ **To try out Scribble's help**

1. Press the SHIFT+F1 keys to enter help mode then click one of the items on Scribble's Pen menu.

   You'll see the custom help that has been provided in PEN.RTF.
2. Select the Index command on the Help menu to see Scribble's custom help index.

Now that you've seen Scribble's help, it's time to examine some of the elements that make it work.

Help Contexts in Scribble

By step 6, Scribble has already defined a number of new IDs (symbols). The following lists Scribble's RESOURCE.H file at this stage:

```
//{{NO_DEPENDENCIES}}
// App Studio generated include file.
// Used by SCRIBBLE.RC
//
#define IDR_MAINFRAME                   1
#define IDR_SCRIBTYPE                   2
#define IDD_ABOUTBOX                    100
#define IDD_PEN_WIDTHS                  101
#define ID_PEN_THICK_OR_THIN            1001
#define ID_PEN_WIDTHS                   1002
#define IDC_THIN_PEN_WIDTH              1000
#define IDC_THICK_PEN_WIDTH             1001
#define IDC_DEFAULT_PEN_WIDTHS          1002
```

Newly defined symbols include `IDR_SCRIBTYPE` (Scribble's menus and other application-specific resources), `IDD_PEN_WIDTHS` (the Pen Widths dialog box

added in Chapter 7), ID_PEN_THICK_OR_THIN (the Thick Line command added in Chapter 5), and so on. Notice that the **ID_EDIT_CLEAR_ALL** command ID doesn't appear in RESOURCE.H because it's a predefined ID in the class library.

MAKEHM will map these symbols to Windows help contexts when you run MAKEHELP.BAT. After you run MAKEHELP.BAT, the SCRIBBLE.HM file looks like the following:

```
// MAKEHELP.BAT generated Help Map file.  Used by scribble.HPJ.

// Commands (ID_* and IDM_*)
HID_PEN_THICK_OR_THIN                   0x103E9
HID_PEN_WIDTHS                          0x103EA

// Prompts (IDP_*)

// Resources (IDR_*)
HIDR_MAINFRAME                          0x20001
HIDR_SCRIBTYPE                          0x20002

// Dialogs (IDD_*)
HIDD_ABOUTBOX                           0x20064
HIDD_PEN_WIDTHS                         0x20065

// Frame Controls (IDW_*)
```

This file contains help contexts for two commands, two resources (menus and other application resources), and two dialog boxes.

Given these files and the framework's help support, the one task remaining is to edit Scribble's .RTF files to add these topics.

Editing Scribble's Help Topics

The framework manages navigation from application user interfaces to help contexts. Implementing further navigation within the help file is the domain of help authoring rather than programming. The purpose of this section is to describe the general process of authoring and editing help topic files.

Editing the help topics for Scribble is too big a task to work through in this chapter, however a few examples will help you get started. The examples in this chapter were edited with Microsoft Word for Windows, but you can use any application that can edit RTF-format files.

The .RTF files that AppWizard creates contain starter help topics for many elements of the Windows user interface. Some of them are fairly complete, while others are skeletal and must be filled out.

If you want to customize the help topics supplied by AppWizard, you must do the following, using all of the .RTF files in MYSCRIB\HLP:

- Globally replace the string "<<YourApp>>" in the .RTF files with the name of the application: "Scribble". This string is a placeholder.

  Figure 10.2 on page 195 shows one place where the string "<<YourApp>>" has been replaced by the string "Scribble".

- From the help topics, remove any references to menu items absent in Scribble.

  For example, file AFXCORE.RTF contains a topic for the Edit Links command, which Scribble doesn't support.

- Replace the helpful directives in the help topics with your own information. These directives are bracketed by << and >> symbols.

  Notice that because the class library predefines **ID_EDIT_CLEAR_ALL**, file AFXCORE.RTF already contains a help topic for the Clear All command that you added to the Edit menu in Chapters 5 and 6. However, its skeletal directive, "<< Write application-specific help here. >>," needs to be replaced with a real description.

- Add topics for Scribble's new commands and its dialog box.

  Examine the listing of RESOURCE.H in the previous section. It lists the following help topics:

  - Two resource-related HIDs (**HIDR_**), for menus and related resources.
  - Two dialog-box HIDs (**HIDD_**), for the About and Pen Widths dialog boxes.
  - Two command HIDs (**HID_**), for the Thick Line and Pen Widths commands.

  New help topics are needed for the Pen Widths dialog box (About already has a topic in AFXCORE.RTF) and for the two commands.

Help topics in an .RTF file are separated by hard page breaks. Each topic has a name and a "footnote" symbol (#).

The footnote symbol # identifies context strings. Other possible footnote symbols identify keywords for searching (K) and topic names ($). Topic text can contain "popups" and "jumps." A popup displays a small window with extra information when its "hot spot" is clicked. A jump takes the user to another topic screen in the help system when its button is clicked. Hot spots for popups and jumps are text strings with special formatting. For more information, see *Programming Tools for the Microsoft Windows Operating System*. If you're using Word for Windows for .RTF files, you can examine the file AFXCORE.RTF with hidden text displayed.

Example Topics

To illustrate the process of adding topics, this section shows the structure needed for a user to jump from the main index screen in help to a screen showing general information about the new Pen menu, and from there to screens that describe the two Pen menu commands. These items are added to the AFXCORE.RTF file.

The Main Index Screen

Figure 10.2 shows the main index screen as it appears in compiled help. This is what the screen looks like after the Pen menu entry has been added.

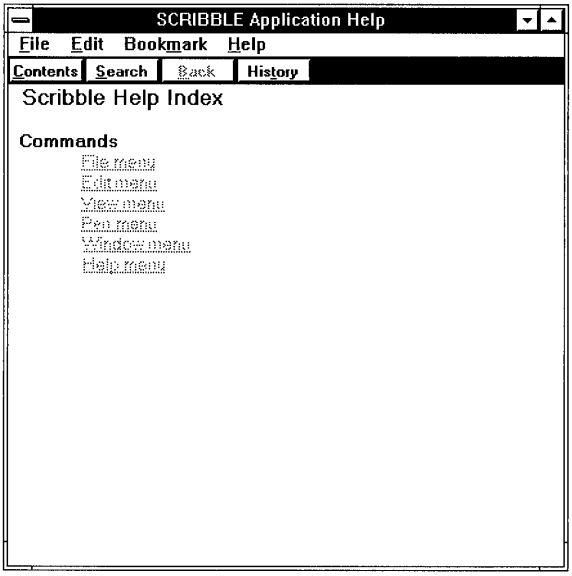


Figure 10.2 The Main Index Screen in Compiled Help

Figure 10.3 shows the help topic for the main index screen as it appears in Microsoft Word for Windows (with hidden text displayed). The screen contains entries for six menus: File, Edit, View, Pen, Window, and Help. The Pen menu is Scribble-specific, so this jump has been added. The others are created by AppWizard.

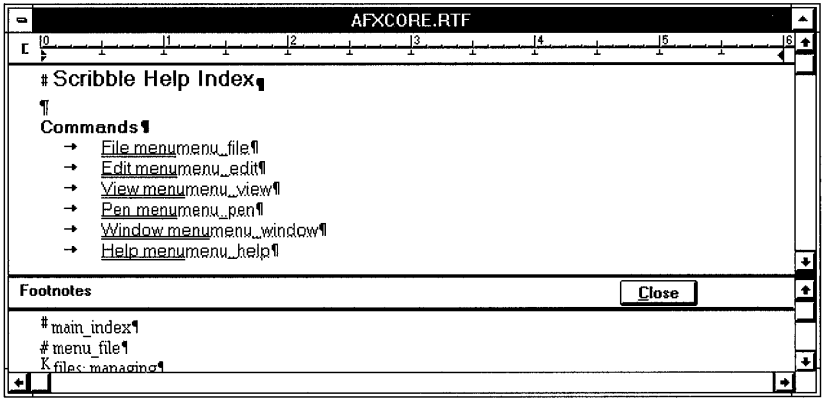


Figure 10.3 The Main Index Screen in AFXCORE.RTF

Each of the menu names on this screen is a "hot spot" that links to another topic. By clicking this hot spot, the user can jump to another screen in the help system. If you examine this screen in the AFXCORE.RTF file using Word for Windows, you see that the menu names—such as "Pen Menu"— are formatted with double underlining. (This might be represented differently by another RTF editor.) Each menu name is followed immediately by a context name formatted as hidden text. For the Pen menu, this text reads "menu_pen."

The Pen Menu Screen

Figure 10.4 shows the text in AFXCORE.RTF for a help screen on the Pen Menu topic. The upper part of the figure shows the text of the file; the lower part shows footnote text. All formatting and hidden text are displayed.

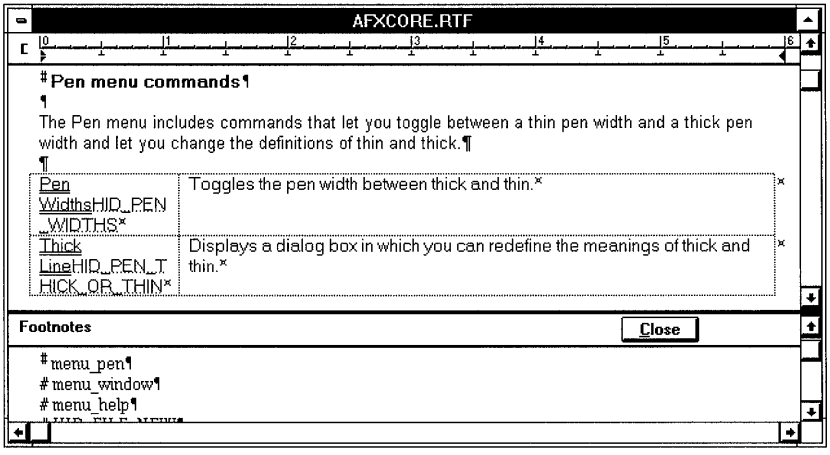


Figure 10.4 The Pen Menu Topic in the .RTF File

The topic screen begins with a hard page break. The next line shows the footnote character, #, followed by the title to be displayed on the user's screen, "Pen menu commands." The rest of the screen contains descriptive text and two more hot spots for jumping to screens about the individual menu items.

The footnote text associated with the # footnote for the Pen menu is "menu_pen," which names the context—the destination of jumps to this screen of the help system.

Setting up this screen requires entering the hard page break and the footnote, then writing the text. In Word for Windows, for example, you'd use the Break command on the Insert menu to enter a hard page break. Then you'd use the Footnote entry on the Insert menu to specify a footnote with the special footnote character #.

Next, you'd type the footnote text in the footnote window. Finally, you'd type the descriptive text and format the hot spots.

The jumps to screens for individual menu items consist of:

- Visible text, such as "Pen Widths," formatted as double underlined, to designate the hot spot on the screen for the jump.
- Hidden text, such as "HID_PEN_WIDTHS," to designate the destination help topic.

In the RTF files that AppWizard supplies, tables (as in Word for Windows) are used to present groups of jumps, but help authors are not required to use tables.

A Screen for a Menu Item

Figure 10.5 shows a screen for the Pen Widths and Thick Line menu items.

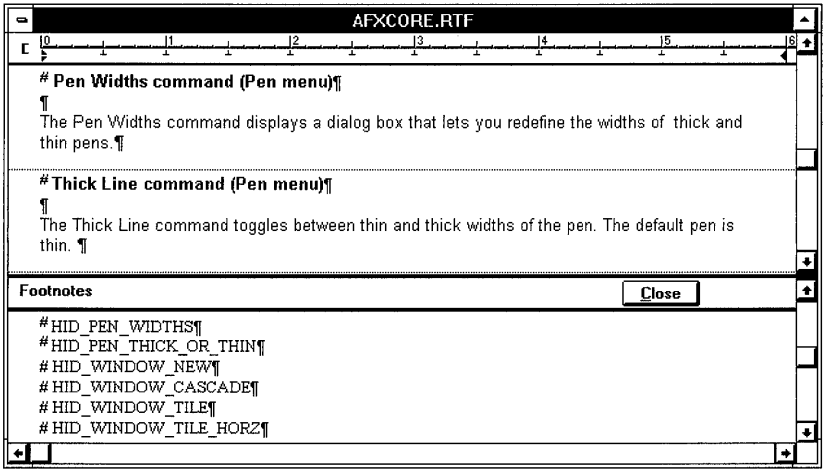


Figure 10.5 The Pen Widths and Thick Line Topics in the .RTF File

The screen is set up similarly to its parent screen for the Pen menu as a whole. Notice the text used for the footnotes in the lower part of Figure 10.5.

Once your help authoring is finished, you can compile your help file. If you've chosen to author Scribble's help files, use the following procedure to see the results.

▶ **To compile and test Scribble's help**

1. To compile help, run MAKEHELP.BAT from the MS-DOS command line.

   The .HLP file is placed in your MYSCRIB directory.

2. To prepare Scribble for testing your help, build Scribble from the Visual Workbench Project menu.

3. To test help, run Scribble and try out the help features.

   The section "See Context-Sensitive Help in Action" on page 186 suggests how to try out the help features.

Conclusion

This concludes the Scribble tutorial portion of the *Class Library User's Guide*. The remaining chapters explore other aspects of the Microsoft Foundation Class Library:

- Memory management (Chapter 11)
- General-purpose classes for dates, times, and strings (Chapter 11)
- Class **CObject**, the root base class of the class library, which offers useful services to classes derived from it (Chapter 12)
- Collection classes, such as arrays, lists, and maps (Chapter 13)
- Files and serialization (Chapter 14)
- Diagnostics (Chapter 15)
- Exception-handling (Chapter 16)
- Using VBX (Visual Basic) controls (Chapter 17)
- Using Object Linking and Embedding (OLE) (Chapter 18)

CHAPTER 11

General-Purpose Classes

The Microsoft Foundation Class Library provides services to make programming easier. These services range from general-purpose memory-management services to more advanced Windows facilities. This chapter describes how to take advantage of the general-purpose services related to memory management, time and date management, and string manipulation.

Memory Management

Memory allocation can be divided into two main categories: frame allocations and heap allocations. One main difference between the two allocation techniques is that with frame allocation you typically work with the actual memory block itself, whereas with heap allocation you are always given a pointer to the memory block. Another major difference between the two schemes is that frame objects are automatically deleted, while heap objects must be explicitly deleted by the programmer.

The following sections describe how to use the capabilities of C and C++ to accomplish memory allocations on the frame and on the heap.

Frame Allocation

Allocation on the frame takes its name from the "stack frame" that is set up whenever a function is called. The stack frame is an area of memory that temporarily holds the arguments to the function as well as any variables that are defined local to the function. Frame variables are often called "automatic" variables because the compiler automatically allocates the space for them.

There are two key characteristics of frame allocations. First, when you define a local variable, enough space is allocated on the stack frame to hold the entire variable, even if it is a large array or data structure. Second, frame variables are automatically deleted when they go out of scope. For local function variables, this scope transition happens when the function exits, but the scope of a frame variable can be smaller than a function if nested braces are used (or larger, in the case of

global variables). This automatic deletion of frame variables is very important. In the case of simple primitive types (such as **int** or **byte**), arrays, or data structures, the automatic deletion simply reclaims the memory used by the variable. Since the variable has gone out of scope, it cannot be accessed anyway. In the case of C++ objects, however, the process of automatic deletion is a bit more complicated.

When an object is defined as a frame variable, its constructor is automatically invoked at the point where the definition is encountered. When the object goes out of scope, its destructor is automatically invoked before the memory for the object is reclaimed. This automatic construction and destruction can be very handy, but you must be aware of the automatic calls, especially to the destructor.

The key advantage of allocating objects on the frame is that they are automatically deleted. When you allocate your objects on the frame, you don't have to worry about forgotten objects causing memory leaks. (For details on memory leaks, see "Detecting Memory Leaks" on page 247.) A disadvantage of frame allocation is that frame variables cannot be used outside their scope. Another factor in choosing frame allocation vs. heap allocation is that for large structures and objects, it is often better to use the heap instead of the stack for storage since stack space is often limited.

Heap Allocation

The heap is reserved for the memory allocation needs of the program. It is an area apart from the program code and from the stack. Typical C programs use the functions **malloc** and **free** to allocate and deallocate heap memory. The Debug version of the Microsoft Foundation Class Library provides modified versions of the C++ built-in operators **new** and **delete** to allocate and deallocate objects in heap memory. When you use **new** and **delete** instead of **malloc** and **free** you are able to take advantage of the Foundation's memory-management debugging enhancements, which can be useful in detecting memory leaks. When you build your program with the Release version of the Microsoft Foundation Class Library, **new** and **delete** still provide an efficient way to allocate and deallocate memory.

Memory Allocation on the Frame and on the Heap

There are three typical kinds of memory allocations:

- An array of bytes
- A data structure
- An object

The following sections describe how the Microsoft Foundation Class Library facilities perform frame allocation and heap allocation for each of these.

Allocation of an Array of Bytes

▶ **To allocate an array of bytes on the frame**

- Define the array as shown by the following code. The array is automatically deleted and its memory reclaimed when the array variable exits its scope.

```
{
    const int BUFF_SIZE = 128;

   // Allocate on the frame
  char myCharArray[BUFF_SIZE];
  int myIntArray[BUFF_SIZE];
  // Reclaimed when exiting scope
}
```

▶ **To allocate an array of bytes (or any primitive data type) on the heap**

- Use the **new** operator with the following array syntax:

```
const int BUFF_SIZE = 128;

// Allocate on the heap
char* myCharArray = new char[BUFF_SIZE];
int* myIntArray = new int[BUFF_SIZE];
```

▶ **To deallocate the arrays from the heap**

- Use the **delete** operator as follows:

```
delete [] myCharArray;
delete [] myIntArray;
```

Allocation of a Data Structure

▶ **To allocate a data structure on the frame**

- Define the structure variable as follows:

```
struct MyStructType {...};
void SomeFunc(void)
{
    // Frame allocation
    MyStructType myStruct;

    // Use the struct
    myStruct.topScore = 297;

    // Reclaimed when exiting scope
}
```

The memory occupied by the structure is reclaimed when it exits its scope.

▶ To allocate data structures on the heap

- Use **new** to allocate data structures on the heap and **delete** to deallocate them, as shown by the following examples:

```
// Heap allocation
MyStructType* myStruct = new MyStructType;

// Use the struct through the pointer ...
myStruct->topScore = 297;

delete myStruct;
```

Allocation of an Object

▶ To allocate an object on the frame

- Declare the object as follows:

```
{
    CPerson myPerson;     // Automatic constructor call here

    myPerson.SomeMemberFunction();     // Use the object

 }
```

The destructor for the object is automatically invoked when the object exits its scope.

▶ To allocate an object on the heap

- Use the **new** operator, which returns a pointer to the object, to allocate objects on the heap. Use the **delete** operator to delete them. The following heap and frame examples assume that the `CPerson` constructor takes no arguments.

```
// Automatic constructor call here
CPerson* myPerson = new CPerson;

myPerson->SomeMemberFunction();     // Use the object

delete myPerson;              // Destructor invoked during delete
```

If the argument for the `CPerson` constructor is a pointer to **char**, the statement for frame allocation is:

```
CPerson myPerson( "Joe Smith" );
```

The statement for heap allocation is:

```
CPerson* MyPerson = new CPerson( "Joe Smith" );
```

Resizable Memory Blocks

The **new** and **delete** operators described above are good for allocating and deallocating fixed-size memory blocks and objects. Occasionally, your application may need resizable memory blocks. You must use the standard C run-time library functions **malloc**, **realloc**, and **free** to manage resizable memory blocks on the heap.

Mixing the **new** and **delete** operators with the resizable memory-allocation functions on the same memory block will result in corrupted memory in the Debug version of the Foundation Class Library. That is, do not allocate a memory block with **new** and deallocate it with **free**. Likewise, you should not use the C++ **delete** operator on a memory block allocated with **malloc** and you should not use **realloc** on a memory block allocated with **new**.

Date and Time

The **CTime** class provides a way to represent date and time information easily. The **CTimeSpan** class represents elapsed time, such as the difference between two **CTime** objects.

Note **CTime** objects cannot be used to represent dates earlier than January 1, 1980. **CTime** objects have a resolution of 1 second.

The first procedure in this section shows how to create a **CTime** object and initialize it with the current time. The next procedure shows how to calculate the difference between two **CTime** objects and get a **CTimeSpan** result.

▶ To get the current time

1. Allocate a **CTime** object, as follows:

   ```
   CTime theTime;
   ```

   Note Uninitialized **CTime** objects are automatically set to an invalid time.

   ---

2. Call the **CTime::GetCurrentTime** function to get the current time from the operating system. This function returns a **CTime** object that can be used to set the value of **CTime**, as follows:

   ```
   theTime = CTime::GetCurrentTime();
   ```

   Since **GetCurrentTime** is a static member function from the **CTime** class, you must qualify its name with the name of the class and the scope resolution operator (**::**), `CTime::GetCurrentTime()`.

Of course, the two steps outlined previously could be combined into a single program statement as follows:

```
CTime theTime = CTime::GetCurrentTime();
```

▶ To calculate elapsed time

- Use the **CTime** and **CTimeSpan** objects to calculate the elapsed time, as follows:

```
CTime startTime = CTime::GetCurrentTime();

// ... perform time-consuming task ...

CTime endTime = CTime::GetCurrentTime();

CTimeSpan elapsedTime = endTime - startTime;
```

Once you have calculated `elapsedTime`, you can use the member functions of **CTimeSpan** to extract the components of the elapsed-time value.

▶ To format a string representation of a time or elapsed time

- Use the **Format** member function from either the **CTime** or **CTimeSpan** classes to create a character string representation of the time or elapsed time, as shown by the following example.

```
CTime t( 1991, 3, 19, 22, 15, 0 ); // 10:15PM March 19, 1991
CString s = t.Format( "%A, %B %d, %Y" );
// s == "Tuesday, March 19, 1991"
```

Strings

The **CString** class provides support for manipulating strings. It is intended to replace and extend the functionality normally provided by the C run-time library string package.

A **CString** object represents a sequence of a variable number of characters. **CString** objects can be thought of as arrays of single-byte characters.

A **CString** object can store up to 32,766 characters. The normal C **char** data type is used to get or set individual characters inside a **CString** object. **CString** objects are automatically growable (that is, you don't have to worry about growing a **CString** object to fit longer strings). A **CString** object also can act like a literal C-style string (a pointer to **const char**).

Basic Operations

The **CString** class provides member functions and overloaded operators that duplicate and, in some cases, surpass the string services of the C run-time libraries (for example, **strcat**). The following sections describe some of the main operations of the **CString** class.

▶ To create CString objects from standard C literal strings

- Assign the value of a C literal string to a **CString** object:

```
CString myString = "This is a test";
```

- Assign the value of one **CString** to another **CString** object:

```
CString oldString = "This is a test";
CString newString = oldString;
```

As explained more completely in the next section on using **CString** objects as values, the contents of a **CString** object are copied when one string is assigned to another **CString** object. Thus, the two strings do not share a reference to the actual characters that make up the string.

▶ To access individual characters in a CString

- You can access individual characters within a **CString** object with the **GetAt** and **SetAt** member functions. You can also use the array element operator ([]) instead of **GetAt** to get individual characters (this is similar to accessing array elements by index, as in standard C-style strings). Index values for **CString** characters are zero-based.

▶ To concatenate two CStrings

- Use the concatenation operators (+ or +=) as follows:

```
CString s1 = "This ";      //Cascading concatenation
s1 += "is a ";
CString s2 = "test";
CString message = s1 + "big " + s2;
//Message contains "This is a big test".
```

At least one of the arguments to the concatenation operators (+ or +=) must be a **CString** object, but you can use a constant character string (such as `"big"`) or a **char** (such as `"x"`) for the other argument.

▶ **To compare two CStrings**

- While the overloaded equality operator (==) and the **Compare** member functions will determine if two **CString** objects are equivalent character for character, you can also use the **CompareNoCase** and **Collate** member functions to do comparisons that are case insensitive and national-language sensitive. The following table shows the three available **CString** comparison functions and their equivalent C run-time string functions.

CString Function	C Run-time Function
Compare	**strcmp**
CompareNoCase	**stricmp**
Collate	**strcoll**

The **CString** class overrides the relational operators (<, <=, >=, >, ==, and !=) to use the **Compare** function, so you can compare two **CStrings** using these operators, as shown here:

```
CString s1( "Tom" );
CString s2( "Jerry" );
if( s1 < s2 )
    ...
```

CString Objects Are Values

Even though **CString** objects are dynamically growable objects, they act like built-in primitive types and simple classes. Each **CString** object represents a unique value. **CString** objects should be thought of as the actual strings rather than as pointers to strings.

The most obvious consequence of using **CString** objects as values is that the string contents are copied when you assign one **CString** to another. Thus, even though two **CStrings** objects may represent the same sequence of characters, they do not share those characters. Each **CString** has its own copy of the character data. When you modify one **CString** object, the copied **CString** object is not modified, as shown by the following example:

```
CString s1, s2;
s1 = s2 = "hi there";

if( s1 == s2 )        // TRUE - they are equal
    ...

s1.MakeUpper();    // Does not modify s2
if( s2[0] == 'h' )    // TRUE - s2 is still "hi there"
```

Notice in the example that the two **CString** objects are considered to be "equal" because they represent the same character string. The **CString** class overloads the equality operator (==) to compare two **CString** objects based on their value (contents) rather than their identity (address).

▶ To specify CString formal parameters correctly

- For most functions that need a string argument, it is best to specify the formal parameter in the function prototype as a pointer to **const char** (**const char*** or **const char FAR***) instead of a **CString**. When a formal parameter is specified as a pointer to **const char**, you can pass either a pointer to a **char** array, a literal string (`"hi there"`), or a **CString** object. The **CString** object will be automatically converted to a pointer to **const char**. Any place you can use a pointer to **char**, you can also use a **CString** object.
- You can also specify a formal parameter as a constant string reference (that is, **const CString&**) if the argument will not be modified. Drop the **const** modifier if the string will be modified by the function. If a default null value is desired, initialize it to the null string (`""`), as shown below:

```
void AddCustomer( const CString& name,
                  const CString& address,
                  const CString& comment = "" );
```

- For most function results, you can simply return a **CString** object by value.

Operations Related to C-Style Strings

It is often useful to be able to manipulate the contents of a **CString** object as if it were a C-style null-terminated string.

▶ To convert to C-style null-terminated strings

- In the simplest case, you can cast a **CString** object to be a pointer to **const char**. The **const char*** type conversion operator returns a read-only pointer to a C-style null-terminated string from a **CString** object.

  The pointer to **char** returned by the implicit conversion shown above points into the data area used by the **CString**. If the **CString** goes out of scope and is automatically deleted or something else changes the contents of the **CString**, the **char** pointer will no longer be valid. You should treat this pointer as a temporary read-only pointer. Do not directly modify the characters to which it points.

- You can use **CString** functions, such as **SetAt**, to modify individual characters in the string object. However, if you need a copy of a **CString** object's characters that you can modify directly, use **strcpy** to copy the **CString** object into a separate buffer where the characters can be safely modified, as shown by the following example:

```
CString theString( "This is a test" );
char* psz = new char[theString.GetLength()+1];
strcpy( psz,theString );
//... modify psz as much as you want
```

Note The second argument to **strcpy** is declared as a constant pointer to **char** (**const char***). The example above passes a **CString** for this argument. The C++ compiler automatically applies the conversion function defined for the **CString** class that converts a **CString** to a **const char***. The ability to define casting operations from one type to another is one of the most useful features of C++.

▶ To work with standard C-library string functions

- In most situations, you should be able to find **CString** member functions to perform any string operation for which you might consider using the standard C run-time library string functions, such as **strcmp**.
- If you find that you must use the C run-time string functions, you can use the techniques described in the previous procedure to copy the **CString** object to an equivalent C-style string buffer, perform your operations on the buffer, and then assign the resulting C-style string back to a **CString** object.

▶ To modify CString contents directly with GetBuffer and ReleaseBuffer

- In most situations, you should use **CString** member functions to modify the contents of a **CString** object or to convert the **CString** to a C-style character string as described in the previous section.
- However, there are certain situations, such as working with operating-system functions that require a character buffer, where it is advantageous to directly modify the **CString** contents.

  The **GetBuffer** and **ReleaseBuffer** member functions allow you to gain access to the internal character buffer of a **CString** object and modify it directly. The following steps show how to use these functions for this purpose:

  1. Call **GetBuffer** for a **CString** object, specifying the length of the buffer you require.
  2. Use the pointer returned by **GetBuffer** to write characters directly into the **CString** object.

3. Call **ReleaseBuffer** for the **CString** object to update all the internal **CString** state information (such as the length of the string). After modifying a **CString** object's contents directly, you must call **ReleaseBuffer** before calling any other **CString** member functions.

▶ To use CString objects with variable argument functions

Some C functions take a variable number of arguments. A notable example is **printf**. Because of the way this kind of function is declared, the compiler cannot be sure of the type of the arguments and cannot determine which conversion operation to perform on the argument. Therefore, it is essential that you use an explicit-type cast when passing a **CString** object to a function that takes a variable number of arguments.

- Explicitly cast the **CString** to a pointer to a constant **char** string, as shown here:

```
CString kindOfFruit = "bananas";
int   howmany = 25;
printf( "You have %d %s\n", howmany, (const char*)kindOfFruit );
```

CHAPTER 12

The CObject Class

CObject is the root base class for most of the Microsoft Foundation Class Library. The **CObject** class contains many useful features that you may want to incorporate into your own program objects, including serialization support, run-time class information, and object diagnostic output. If you derive your class from **CObject**, your class can exploit these **CObject** features.

The cost of deriving your class from **CObject** is minimal. Your derived class will have the overhead of four virtual functions and a single **CRuntimeClass** object.

Deriving a Class from CObject

This section describes the minimum steps necessary to derive a class from **CObject**. Other sections describe the steps needed to take advantage of specific **CObject** features, such as serialization and diagnostic debugging support.

In the following discussions, the terms "interface file" and "implementation file" are used frequently. The interface file (often called the header file, or .H file) contains the class declaration and any other information needed to use the class. The implementation file (or .CPP file) contains the class definition as well the code that implements the class member functions. For example, for a class named `CPerson`, you would typically create an interface file named PERSON.H and an implementation file named PERSON.CPP. However, for some small classes that will not be shared among applications, it is sometimes easier to combine the interface and implementation into a single .CPP file.

There are four levels of functionality from which you can choose when deriving a class from **CObject**:

- Basic functionality that does not include support for run-time class information or serialization but includes diagnostic memory management.
- Basic functionality plus support for run-time class information.
- Basic functionality plus support for run-time class information and dynamic creation.

- Basic functionality plus support for run-time class information, dynamic creation, and serialization.

Classes designed for reuse (those that will later serve as base classes) should at least include run-time class support and serialization support, if any future serialization need is anticipated.

You choose the level of functionality by using specific declaration and implementation macros in the declaration and implementation of the classes you derive from **CObject**.

Figure 12.1 shows the relationship among the macros used to support serialization and run-time information.

	CObject::IsKindOf	**CRuntimeClass:: CreateObject**	**CArchive::operator>> CArchive::operator<<**
Basic **CObject** functionality	No	No	No
DECLARE_DYNAMIC	Yes	No	No
DECLARE_DYNCREATE	Yes	Yes	No
DECLARE_SERIAL	Yes	Yes	Yes

Figure 12.1 Macros Used for Serialization and Run-Time Information

The following sections describe how to specify the level of functionality.

▶ To use basic CObject functionality

- Use the normal C++ syntax to derive your class from **CObject** (or from a class derived from **CObject**).

  The following example shows the simplest case, the derivation of a class from **CObject**:

```
class CPerson : public CObject
{
    // add CPerson-specific members and functions...
}
```

  Typically, however, you may want to override some of **CObject**'s member functions to handle the specifics of your new class. For example, you may usually want to override the **Dump** function of **CObject** to provide debugging output for the contents of your class. For details on how to override **Dump**, see "Dumping Object Contents" on page 242. You may also want to override the **AssertValid** function of **CObject** to provide customized testing to validate the consistency of the data members of class objects. For a description of how to override **AssertValid**, see the section on "Overriding the AssertValid Function" on page 245.

▶ To add run-time class information

CObject supports run-time class information through the **IsKindOf** function, which allows you to determine if an object belongs to or is derived from a specified class. (For more detailed information, see Chapter 14, "Files and Serialization.") This capability is not supported directly by the C++ language. The **IsKindOf** function permits you to do a typesafe cast down to a derived class.

Use the following steps to access run-time class information:

1. Derive your class from **CObject**, as described in the previous section.
2. Use the **DECLARE_DYNAMIC** macro in your class declaration, as shown here:

```
class CPerson : public CObject
{
    DECLARE_DYNAMIC( CPerson )

    // rest of class declaration follows...
};
```

3. Use the **IMPLEMENT_DYNAMIC** macro in the implementation file (.CPP) of your class. This macro takes as arguments the name of the class and its base class, as follows:

```
IMPLEMENT_DYNAMIC( CPerson, CObject )
```

To better understand the relationships among the macros and the functions that support serialization and run-time, see the table on page 212.

Note Always put **IMPLEMENT_DYNAMIC** in the implementation file (.CPP) for your class. The **IMPLEMENT_DYNAMIC** macro should be evaluated only once during a compilation and therefore should not be used in an interface file (.H) that could potentially be included in more than one file.

▶ To add dynamic creation support

CObject also supports dynamic creation, which is the process of creating an object of a specific class at run time. The object is created by the **CreateObject** member function of **CRuntimeClass.** Your document, view, and frame class should support dynamic creation because the framework (through the **CDocTemplate** class) needs to create them dynamically. Dynamic creation is not supported directly by the C++ language. To add dynamic creation, you must do the following:

1. Derive your class from **CObject.**
2. Use the **DECLARE_DYNCREATE** macro in the class declaration.
3. Define a constructor with no arguments (a default constructor).

4. Use the **IMPLEMENT_DYNCREATE** macro in the class implementation file.

▶ **To add serialization support**

Serialization is the process of writing or reading the contents of an object to and from a file. The Microsoft Foundation Class Library uses an object of the **CArchive** class as an intermediary between the object to be serialized and the storage medium. The **CArchive** object uses overloaded insertion (<<) and extraction (>>) operators to perform writing and reading operations.

The following steps are required to support serialization in your classes:

1. Derive your class from **CObject**.
2. Override the **Serialize** member function.

If you call **Serialize** directly, that is, you do not want to serialize the object through a polymorphic pointer, omit these steps:

1. Use the **DECLARE_SERIAL** macro in the class declaration.
2. Define a constructor with no arguments (a default constructor).
3. Use the **IMPLEMENT_SERIAL** macro in the class implementation file.

For more details on how to enable serialization when you derive your class from **CObject**, see "Serialization" on page 229. Each of the steps listed above is described in that section.

Accessing Run-Time Class Information

If you have derived your class from **CObject** and used the **DECLARE_DYNAMIC** and **IMPLEMENT_DYNAMIC,** the **DECLARE_DYNCREATE** and **IMPLEMENT_DYNCREATE,** or the **DECLARE_DYNAMIC** and **IMPLEMENT_DYNAMIC** macros explained previously, the **CObject** class has the ability to determine the exact class of an object at run time.

The ability to determine the class of an object at run time is most useful when extra type checking of function arguments is needed and when you must write special-purpose code based on the class of an object. However, this practice is not usually recommended because it duplicates the functionality of virtual functions.

The **CObject** member function **IsKindOf** can be used to determine if a particular object belongs to a specified class or if it is derived from a specific class. The argument to **IsKindOf** is a **CRuntimeClass** object, which you can get using the **RUNTIME_CLASS** macro with the name of the class. The use of the **RUNTIME_CLASS** macro is shown in the following section.

▶ To use the RUNTIME_CLASS macro

- Use **RUNTIME_CLASS** with the name of the class, as shown here for the class **CObject**:

```
CRuntimeClass* pClass = RUNTIME_CLASS( CObject );
```

You will rarely need to access the run-time class object directly. A more common use is to pass the run-time class object to the **IsKindOf** function, as shown in the next section.

▶ To use the IsKindOf function

The **IsKindOf** function tests an object to see if it belongs to a particular class. The following steps show how to use **IsKindOf**.

1. Make sure the class has run-time class support. That is, the class must have been derived from **CObject** and used the **DECLARE_DYNAMIC** and **IMPLEMENT_DYNAMIC**, the **DECLARE_DYNCREATE** and **IMPLEMENT_DYNCREATE**, or the **DECLARE_SERIAL** and **IMPLEMENT_SERIAL** macros explained previously on pages 212–214.
2. Call the **IsKindOf** member function for objects of that class, using the **RUNTIME_CLASS** macro to generate the **CRuntimeClass** argument, as shown here:

```
// in .H file
class CPerson : public CObject
{
    DECLARE_DYNAMIC( CPerson )
public:
    CPerson(){};

    // other declaration
};

// in .CPP file
IMPLEMENT_DYNAMIC( CPerson, CObject )

CObject* pMyObject = new CPerson;

if( myObject->IsKindOf( RUNTIME_CLASS( CPerson ) ) )
{
   //if IsKindOf is true, then cast is all right
   CPerson* pmyPerson = (CPerson*) pmyObject;
}
```

Note **IsKindOf** returns **TRUE** if the object is a member of the specified class or of a class derived from the specified class. **IsKindOf** does not support multiple inheritance or virtual base classes, although you can use multiple inheritance for your Microsoft Foundation classes if necessary.

▶ To dynamically create an object given its run-time class

- Use the following code to dynamically create an object by using the **CreateObject** function of the **CRuntimeClass**:

```
CRuntimeClass* pRuntimeClass = RUNTIME_CLASS( CMyClass );
CObject* pObject = pRuntimeClass->CreateObject();
ASSERT( pObject->IsKindOf( RUNTIME_CLASS( CMyClass ) ) );
```

For more detailed information on serialization and run-time class information, see Chapter 14, "Files and Serialization."

CHAPTER 13

Collections

The Microsoft Foundation Class Library provides collection classes to manage groups of objects. A collection class is characterized by its "shape" and by the types of its elements. The shape refers to the way the objects are organized and stored by the collection. The Microsoft Foundation Class Library provides three basic collection shapes: lists, arrays, and maps (also known as dictionaries). You can pick the collection shape most suited to your particular programming problem.

Each of the three provided collection shapes is described briefly below, followed by Table 13.1, which compares the features of the shapes to help you decide which is best for your program. In the table, the term "ordered" means that the order of the items in the collection is determined by the order in which they were inserted and deleted. It does not mean that the items are sorted based on their contents. The term "indexed" means that the items in the collection can be retrieved by an integer index, much like a typical array structure.

- List

  The list class provides an ordered, nonindexed list of elements, implemented as a doubly linked list. A list has a "head" and a "tail," and adding or removing elements from the head or tail, or inserting or deleting elements in the middle, is very fast.

- Array

  The array class provides a dynamically sized, ordered, and integer-indexed array of objects.

- Map (also known as a dictionary)

  A "map" is a collection that associates a key object with a value object.

Table 13.1 Shape Features

Shape	Ordered?	Indexed?	Insert an Element	Check for Specified Element	Duplicate Elements?
List	Yes	No	Fast	Slow	Yes
Array	Yes	By int	Slow	Slow	Yes
Map	No	By key	Fast	Fast	No (keys) Yes (values)

How to Make a Type-Safe Collection

The Microsoft Foundation Class Library provides predefined type-safe collections that can be used to contain **CObject**, **UINT**, **DWORD**, and **CString** elements. You can use these predefined collections (such as **CObList**) to hold collections of any objects derived from **CObject**. The Microsoft Foundation Class Library also provides other predefined collections to hold primitive types such as **UINT** and void pointers (**void***). In general, however, it is often useful to define your own type-safe collections to hold objects of a more specific class and its derivatives.

There are three main ways to use collections with the Microsoft Foundation Class Library, as described by the following sections.

▶ To use predefined collections

- The easiest way to use the Microsoft Foundation collection classes is to use a predefined collection type, such as **CWordArray**. You can create a **CWordArray** and add any 16-bit values to it and retrieve them. There is nothing more to do. You just use the predefined functionality.
  - You can also use a predefined collection, such as **CObList**, to hold objects that are derived from **CObject**. A **CObList** is defined to hold pointers to **CObject**. You can put any object that is derived from **CObject** into a **CObList**. When you retrieve an object from the list, you may have to cast the result to the proper type since the **CObList** functions return pointers to **CObject**. For example, if you store `CPerson` objects in a **CObList**, you have to cast a retrieved element to be a pointer to a `CPerson` object. The following example uses a **CObList** to hold `CPerson` objects:

```
class CPerson : public CObject {...};

CPerson* p1 = new CPerson(...);
CObList myList;

myList.AddHead( p1 );   // No cast needed
CPerson* p2 = ( CPerson* )myList.GetHead();
```

- This technique of using a predefined collection type and casting as necessary may be adequate for many of your collection needs. If you need further functionality or more type safety, read the next section.

▶ To derive and extend a collection

- You can also derive your own collection class from one of the predefined collection classes provided with the Microsoft Foundation Class Library. When you derive your class, you can add type-safe wrapper functions to provide a type-safe interface to existing functions.
  - For example, if you derived a list from **CObList** to hold `CPerson` objects, you might add the wrapper functions `AddHeadPerson` and `GetHeadPerson`, as shown below.

```
class CPersonList : public CObList
{
public:
    void AddHeadPerson( CPerson* person )
        {AddHead( person );}

    CPerson* GetHeadPerson()
        {return (CPerson*)GetHead();}
};
```

These wrapper functions provide a type-safe way to add and retrieve `CPerson` objects from the derived list. You can see that for the `GetHeadPerson` function, you are simply encapsulating the casting seen in the previous section.

You can also add new functionality by defining new functions that extend the capabilities of the collection rather than just wrapping existing functionality in type-safe wrappers. For example, a later section describes a function to delete all the objects contained in a list. This function could be added to the derived class as a member function.

▶ To use templates to create new collection classes

- See the technical note that describes the Microsoft Foundation Class Library tool that you can use to create new type-safe collections from template files. These templates and the tool allow you to create a version of an existing collection shape that is customized to hold a specified data type or object type.

The directory (on your distribution disks) that contains sample programs has an application that expands templates defined using a subset of the proposed ANSI template syntax. The Foundation collection classes were generated with this program.

Accessing All Members of a Collection

The Foundation collection classes use a position indicator to describe a given position within the collection. To access one or more members of a collection, first initialize the position indicator and then repeatedly pass that position to the collection and ask it to return the next element. The collection is not responsible for maintaining state information about the progress of the iteration. That information is kept in the position indicator. But, given a particular position, the collection is responsible for returning the next element.

The following examples show how to iterate over the three main types of collections provided with the Microsoft Foundation Class Library.

▶ To iterate an array

- Use sequential index numbers with the **GetAt** member function:

```
CObArray myArray;

for( int i = 0; i < myArray.GetSize();i++ )
{
    CPerson* thePerson = (CPerson*)myArray.GetAt( i );
    ...
}
```

▶ To iterate a list

- Use the member functions **GetHeadPosition** and **GetNext** to work your way through the list:

```
CPersonList myList;

POSITION pos = myList.GetHeadPosition();
while( pos != NULL )
{
    CPerson* thePerson = myList.GetNext( pos );
    ...
}
```

▶ **To iterate a map**

- Use **GetStartPosition** to get to the beginning of the map and **GetNextAssociation** to repeatedly get the next key and value from the map, as shown by the following example:

```
CMapStringToOb myMap;

POSITION pos = myMap.GetStartPosition();
while( pos != NULL )
{
    CObject* pObject;
    CPerson* pPerson;
    CString string;
    // Gets key ( string ) and value ( pObject )
    myMap.GetNextAssoc( pos, string, pObject );
    if( pObject->IsKindOf( RUNTIME_CLASS(CPerson) ) )
    {
        pPerson = (CPerson*)pObject;
        //...
    }
}
```

How to Delete All Objects in a CObject Collection

To delete all the objects in a collection of **CObject**s (or of objects derived from **CObject**), you use one of the iteration techniques described above to delete each object in turn.

Note Objects in collections can be shared. That is, the collection keeps a pointer to the object, but other parts of the program may also have pointers to the same object. You must be careful not to delete an object that is shared until all the parts have finished using the object.

▶ **To delete all objects in a CObList**

1. Use **GetHeadPosition** and **GetNext** to iterate through the list.
2. Use the **delete** operator to delete each object as it is encountered in the iteration.
3. Call the **RemoveAll** function to remove all elements from the list after the objects associated with those elements have been deleted.

The preceding technique deletes all objects in a **CObList** or a list derived from **CObList**.

The following example shows how to delete all objects from a list of `CPerson` objects. Each object in the list is a pointer to a `CPerson` object that was originally allocated on the heap.

```
class CPersonList : public CObList {...};

CPersonList myList
POSITION pos = myList.GetHeadPosition();

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

The last function call, **RemoveAll**, is a list member function that removes all elements from the list. The member function **RemoveAt** will remove a single element.

Notice the difference between deleting an element's object and removing the element itself. Removing an element from the list merely removes the list's reference to the object. The object still exists in memory. When you delete an object, its memory is reclaimed and it ceases to exist. Thus, it is important to remove an element immediately after the element's object has been deleted so that the list won't try to access objects that no longer exist.

▶ To delete all elements in an array

1. Use **GetSize** and integer index values to iterate through the array.
2. Use the **delete** operator to delete each element as it is encountered in the iteration.
3. Call the **RemoveAll** function to remove all elements from the array after they have been deleted.

   The code for deleting all elements of an array is as follows:

```
CObArray myArray;

int i = 0;
while (i < myArray.GetSize() )
{
    delete myArray.GetAt( i++ );
}

myArray.RemoveAll();
```

Like the list example, you can call **RemoveAll** to remove all elements in an array or **RemoveAt** to remove an individual element.

▶ **To delete all elements in a map**

1. Use **GetStartPosition** and **GetNextAssociation** to iterate through the array.
2. Use the **delete** operator to delete the key and/or value for each map element as it is encountered in the iteration.
3. Call the **RemoveAll** function to remove all elements from the map after they have been deleted.

   The code for deleting all elements of a **CMapStringToOb** is as follows. Each element in the map has a string as the key and a `CPerson` object (derived from **CObject**) as the value.

```
CMapStringToOb myMap;
// ... Add some key-value elements ...
// Now delete the elements
pos = myMap.GetStartPosition();
while( pos != NULL )
{
    CObject* pObject;
    CString string;
    // Gets key ( string ) and value ( pObject )
    myMap.GetNextAssoc( pos, string, pObject );
    delete pObject;
}
myMap.RemoveAll();
```

You can call **RemoveAll** to remove all elements in a map or **RemoveKey** to remove an individual element with the specified key.

How to Create a Stack Collection

Because the standard list collection has both a head and a tail, it is easy to create a derived list collection that mimics the behavior of a last-in-first-out stack. A stack is like a stack of trays in a cafeteria. As trays are added to the stack, they go on top of the stack. The last tray added is the first to be removed. The list collection member functions **AddHead** and **RemoveHead** can be used to add and remove elements specifically from the head of the list; thus the most recently added element is the first to be removed.

▶ **To create a stack collection**

- Derive a new list class from one of the existing Foundation list classes and add more member functions to support the functionality of stack operations.

The following example shows you can add member functions to push elements on to the stack, peek at the top element of the stack, and pop the top element from the stack:

```
class CTray : public CObject { ... };

class CStack : public CObList
{
public:
    // Add element to top of stack
    void Push( CTray* newTray )
        { AddHead( newTray ); }

    // Peek at top element of stack
    CTray* Peek()
        { return IsEmpty() ? NULL : (CTray*)GetHead(); }

    // Pop top element off stack
    CTray* Pop()
        { return (CTray*)RemoveHead(); }
};
```

How to Create a Queue Collection

Because the standard list collection has both a head and a tail, it is also easy to create a derived list collection that mimics the behavior of a first-in-first-out queue. A queue is like a line of people in a cafeteria. The first person in line is the first to be served. As more people come, they go to the end of the line to wait their turn. The list collection member functions **AddTail** and **RemoveHead** can be used to add and remove elements specifically from the head or tail of the list; thus the most recently added element is always the last to be removed.

▶ To create a queue collection

- Derive a new list class from one of the predefined list classes provided with the Microsoft Foundation Class Library and add more member functions to support the semantics of queue operations.

The following example shows how you can append member functions to add an element to the end of the queue and get the element from the front of the queue.

```
class CPerson : public CObject { ... };

class CQueue : public CObList
{
public:
    // Go to the end of the line
    void AddToEnd( CPerson* newPerson )
        { AddTail( newPerson ); }    // End of the queue

    // Get first element in line
    CPerson* GetFromFront()
        { return IsEmpty() ? NULL : (CPerson*)RemoveHead(); }
};
```

CHAPTER 14

Files and Serialization

In the Microsoft Foundation Class Library, the **CFile** class handles normal file I/O operations. This chapter explains how to open and close files as well as read and write data to those files. You will also learn about file status operations. For a description of how to use the object-based serialization features of the Microsoft Foundation Class Library as an alternative way of reading and writing data in files, see "Serialization" on page 229.

Files

The **CFile** class provides an interface for general-purpose binary file operations. The **CStdioFile** and **CMemFile** classes are derived from **CFile** to supply more specialized file services.

In the Microsoft Foundation Class Library, the most common way to open a file is a two-stage process.

▶ **To open a file**

1. Create the file object without specifying a path or permission flags.

   You usually create a file object by declaring a **CFile** variable on the stack frame.

2. Call the **Open** member function for the file object, supplying a path and permission flags.

   The return value for **Open** will be nonzero if the file was opened successfully or 0 if the specified file could not be opened.

   The open flags specify which permissions, such as read-only, you want for the file. The possible flag values are defined as enumerated constants within the **CFile** class, so they are qualified with "**CFile::**," as in **CFile::modeRead**. Use the **CFile::modeCreate** flag if you want to create the file.

The following example shows how to create a new file with read/write permission (replacing any previous file with the same path):

```
char* pszFileName = "\\test\\myfile.dat";
CFile myFile;

if ( ! myFile.Open( pszFileName,
                CFile::modeCreate | CFile::modeReadWrite ) )
{
    TRACE( "Can't open file %s\n",pszFileName );
}
```

You may pass an additional **CFileException** object if you require more detailed error reporting.

▶ To read from and write to the file

- Use the **Read** and **Write** member functions to read and write data in the file.

  The **Seek** member function is also available for moving to a specific offset within the file.

  Read takes a pointer to a buffer and a **UINT** specifying the number of bytes to read and returns a **UINT** with the actual number of bytes that were read. If the required number of bytes could not be read because end-of-file (EOF) is reached, the actual number of bytes read is returned. If any read error occurs, an exception is thrown. **Write** is similar to **Read**, but the number of bytes written is not returned. If a write error occurs, including not writing all the bytes specified, an exception is thrown. If you have a valid **CFile** object, you can read from it or write to it as shown in the following example:

```
char     szBuffer[256];
UINT    nActual = 0;

myFile.Write( szBuffer, sizeof( szBuffer ) );
myFile.Seek( 0, CFile::begin );
nActual = myFile.Read( szBuffer, sizeof( szBuffer ) );
```

▶ To close a file

- Use the **Close** member function. This function closes the file-system file and flushes buffers if necessary.

  If you allocated the **CFile** object on the frame (as in the examples above), the object will automatically be closed and then destroyed when it goes out of scope. Note that deleting the **CFile** object does not delete the physical file in the file system.

▶ **To get file status**

- Use the **CFile** class to get and set information about a file. One useful application is to use the **CFile** static member function **GetStatus** to determine if a file exists. **GetStatus** will return 0 if the specified file does not exist.

  Thus, you could use the result of **GetStatus** to determine whether to use the **CFile::modeCreate** flag when opening a file, as shown by the following example:

```
CFile theFile;
char* szFileName = "c:\\test\\myfile.dat";
BOOL bOpenOK;

CFileStatus status;
if( CFile::GetStatus( szFileName, status ) )
{
    // Open the file without the Create flag
    bOpenOK = theFile.Open( szFileName,
                    CFile::modeWrite );
}
else
{
   // Open the file with the Create flag
   bOpenOK = theFile.Open( szFileName,
                   CFile::modeCreate | CFile::modeWrite );
}
```

Serialization (Object Persistence)

Serialization is the process of writing or reading an object to or from a persistent storage medium, such as a disk file. The Microsoft Foundation Class Library provides built-in support for serialization in the class **CObject**. Thus, all classes that are derived from **CObject** can take advantage of **CObject**'s serialization protocol.

The basic idea of serialization is that an object should be able to write its current state, usually indicated by the value of its member variables, to persistent storage. Later, the object can be re-created by reading, or deserializing, the object's state from the storage. Serialization handles all the details of object pointers and circular references to objects that are used when you serialize an object. A key point is that the object itself is responsible for reading and writing its own state. Thus, for a class to be serializable, it must implement the basic serialization operations. As you will see in the following sections, it is easy to add this functionality to a class.

The Microsoft Foundation Class Library uses an object of the **CArchive** class as an intermediary between the object to be serialized and the storage medium. This object is always associated with a **CFile** object, from which it obtains the necessary

information for serialization, including the filename and whether the requested operation is a read or write. The object that performs a serialization operation can use the **CArchive** object without regard to the nature of the storage medium.

A **CArchive** object uses overloaded insertion (<<) and extraction (>>) operators to perform writing and reading operations. For more information, see "Storing and Loading CObjects via an Archive" on page 237.

Note Do not confuse the **CArchive** class with general-purpose iostream classes, which are for formatted text only. The **CArchive** class is for binary format serialized objects.

The following sections cover the two main tasks required for serialization:

- Making a serializable class.
- Serializing an object to and from a file object.

Making a Serializable Class

There are five main steps required to make a class serializable. They are listed below and explained in the following sections:

1. Derive your class from **CObject** (or from some class derived from **CObject**).
2. Use the **DECLARE_SERIAL** macro in the class declaration.
3. Override the **Serialize** member function.
4. Define a constructor that takes no arguments.
5. Use the **IMPLEMENT_SERIAL** macro in the implementation file for your class.

If you call **Serialize** directly rather than through the >> and << operators of **CArchive**, the last three steps are not required for serialization.

Deriving Your Class from CObject and Using the DECLARE_SERIAL Macro

The basic serialization protocol and functionality are defined in the **CObject** class. By deriving your class from **CObject** (or from a class derived from **CObject**), as shown in the following example code, you gain access to the serialization protocol and functionality of **CObject**.

The **DECLARE_SERIAL** macro is required in the declaration of classes that will support serialization, as shown here:

```
class CPerson : public CObject
{
    DECLARE_SERIAL( CPerson )
    // rest of declaration follows...
};
```

Overriding the Serialize Member Function

The **Serialize** member function, which is defined in the **CObject** class, is responsible for actually serializing the data necessary to capture an object's current state. The **Serialize** function has a **CArchive** argument that it uses to read and write the object data. The **CArchive** object has a member function, **IsStoring**, which indicates whether **Serialize** is storing (writing data) or loading (reading data). Using the results of **IsStoring** as a guide, you either insert your object's data in the **CArchive** object with the insertion operator (<<) or extract data with the extraction operator (>>).

Consider a class that is derived from **CObject** and has two new member variables, a **CString** and a **WORD**. The following class declaration fragment shows the new member variables and the declaration for the overridden **Serialize** member function:

```
class CPerson : public CObject
{
public:
    DECLARE_SERIAL( CPerson, )
    // empty constructor is necessary
    CPerson(){};

    CString m_name;
    WORD    m_number;

    void Serialize( CArchive& archive );

    // rest of class declaration
};
```

▶ **To override the Serialize member function**

1. Call your base class version of **Serialize** to make sure that the inherited portion of the object is serialized.

2. Insert or extract the member variables that are specific to your class.

   The insertion and extraction operators interact with the archive class to read and write the data. The following example shows how to implement **Serialize** for the `CPerson` class declared above:

```
void CPerson::Serialize( CArchive& archive )
{
    // call base class function first
    // base class is CObject in this case
    CObject::Serialize( archive );

    // now do the stuff for our specific class
    if( archive.IsStoring() )
        archive << m_name << m_number;
    else
        archive >> m_name >> m_number;
}
```

You can also use the **CArchive::Read** and **CArchive::Write** member functions to read and write large amounts of untyped data.

Defining a Constructor with No Arguments

A default constructor is required by the Microsoft Foundation Class Library when it re-creates your objects as they are deserialized (loaded from disk.) The deserialization process will fill in all member variables with the values required to re-create the object.

This constructor can be declared public, protected, or private. If you make it protected or private, you ensure that it will only be used by the serialization functions. The constructor must put the object in a state that allows it to be safely deleted if necessary.

Note If you forget to define a constructor with no arguments in a class that uses the **DECLARE_SERIAL** and **IMPLEMENT_SERIAL** macros, you will get a "no default constructor available" compiler warning on the line where the **IMPLEMENT_SERIAL** macro is used.

Using the IMPLEMENT_SERIAL Macro in the Implementation File

The **IMPLEMENT_SERIAL** macro is used to define the various functions needed when you derive a serializable class from **CObject**. You use this macro in the implementation file (.CPP) for your class. The first two arguments to the macro are the name of the class and the name of its immediate base class.

The third argument to this macro is a schema number. The schema number is essentially a version number for objects of the class. Use an integer greater than or equal to 0 for the schema number.

The Microsoft Foundation Class Library serialization code checks the schema number when reading objects into memory. If the schema number of the object on disk does not match the schema number of the class in memory, then the library will throw a **CArchiveException**, preventing your program from reading an incorrect version of the object.

The following example shows how to use **IMPLEMENT_SERIAL** for a class, `CPerson`, that is derived from **CObject**:

```
IMPLEMENT_SERIAL( CPerson, CObject, 1 )
```

Serializing an Object

The previous sections showed how to make a class serializable. Once you have a serializable class, you can serialize objects of that class to and from a file via a **CArchive** object. This section first describes a **CArchive** object. It then describes the two ways to create it: (1) let the framework create it for your serializable document or (2) explicitly create the **CArchive** object yourself. Finally, the section describes how to transfer data between a file and your serializable object by using the << and >> operators for **CArchive** or, in some cases, by calling the `Serialize` function of a **CObject**-derived class.

What Is a CArchive Object?

A **CArchive** object provides a type-safe buffering mechanism for writing or reading serializable objects to or from a **CFile** object. Usually the **CFile** object represents a disk file; however, it can be also be a memory file (**CMemFile** object), perhaps representing the Clipboard.

A given **CArchive** object either stores (writes, serializes) data or loads (reads, deserializes) data, but never both. The life of a **CArchive** object is limited to one pass through writing objects to a file or reading objects from a file. Thus, two successively created **CArchive** objects are required to serialize data to a file and then deserialize it back from the file.

When an archive stores objects to a file, the archive attaches the **CRuntimeClass** name to the objects. Then, when another archive loads objects from a file to memory, the **CObject**-derived objects are dynamically reconstructed based on the **CRuntimeClass** of the objects. A given object may be referenced more than once as it is written to the file by the storing archive. The loading archive, however, will reconstruct the object only once. The details about how an archive attaches **CRuntimeClass** information to objects and reconstructs objects, taking into

account possible multiple references, is described in Technical Note 2, which can be found in \MSVC\HELP\MFCNOTES.HLP.

As data is serialized to an archive, the archive accumulates the data until its buffer is full. Then the archive writes its buffer to the **CFile** object pointed to by the **CArchive** object. Similarly, as you read data from an archive, it reads data from the file to its buffer and then from the buffer to your deserialized object. This buffering reduces the number of times a hard disk is physically read, thus improving your application's performance.

Two Ways to Create a CArchive Object

There are two ways to create a **CArchive** object. The most common, and the easiest way is to let the framework create one for your document on behalf of the Save, Save As, and Open commands on the File menu. For example, here is what the framework does when the user of your application issues the Save As command from the File menu:

1. The framework presents the File Save As dialog box and gets the filename from the user.
2. The framework opens the file named by the user as a **CFile** object.
3. The framework creates a **CArchive** object that points to this **CFile** object. In creating the **CArchive** object, the framework sets the mode to "store" (write, serialize), as opposed to "load" (read, deserialize).
4. The framework calls the `Serialize` function defined in your **CDocument**-derived class, passing it a reference to the **CArchive** object.
5. Your document's `Serialize` function writes data to the **CArchive** object, as explained shortly.
6. Upon return from your `Serialize` function, the framework destroys the **CArchive** object and then the **CFile** object.

Thus, if you let the framework create the **CArchive** object for your document, all you have to do is implement the document's `Serialize` function that writes and reads to and from the archive. You also have to implement `Serialize` for any **CObject**-derived objects that the document's `Serialize` function in turn serializes directly or indirectly.

Besides serializing a document via the framework, there are other occasions when you may need a **CArchive** object. For example, you might want to serialize data to and from the Clipboard, represented by a **CMemFile** object. Or, you may want to use a user interface for saving a file that is different from the one offered by the framework. In this case, you can explicitly create a **CArchive** object. You do this the same way the framework does, using the following.

▶ **To explicitly create a CArchive object**

1. Construct a **CFile** object or an object derived from **CFile**.
2. Pass the **CFile** object to the constructor for **CArchive**, as shown in the following example:

```
CFile theFile;
theFile.Open(..., CFile::modeWrite);
CArchive archive(&theFile, CArchive::store);
```

The second argument to the **CArchive** constructor is an enumerated value that specifies whether the archive will be used for storing or loading data to or from the file. The `Serialize` function of an object checks this state by calling the **IsStoring** function for the archive object.

When you are finished storing or loading data to or from the **CArchive** object, close it. Although the **CArchive** (and **CFile**) objects will automatically close the archive (and file), it is good practice to explicitly do so since it makes recovery from errors easier. For more information, see "Catching Exceptions" in Chapter 16.

▶ **To close the CArchive object**

- The following example illustrates how to close the **CArchive** object:

```
archive.Close();
theFile.Close();
```

Using the CArchive << and >> Operators

CArchive provides << and >> operators for writing and reading simple data types as well as **CObject**s to and from a file.

▶ **To store an object in a file via an archive**

- The following example shows how to store an object in a file via an archive:

```
CArchive ar(&theFile, CArchive::store);
WORD wEmployeeID;
...
ar << wEmployeeID;
```

▶ **To load an object from a value previously stored in a file**

- The following example shows how to load an object from a value previously stored in a file:

```
CArchive ar(&theFile, CArchive::load);
WORD wEmployeeID
...
ar >> wEmployeeID;
```

Usually, the places where you store and load data to and from a file via an archive are in the `Serialize` functions of **CObject**-derived classes, which you must have declared with the **DECLARE_SERIALIZE** macro. A reference to a **CArchive** object is passed to your `Serialize` function. You call the **IsLoading** function of the **CArchive** object to determine whether the `Serialize` function has been called to load data from the file or store data to the file.

The `Serialize` function of a serializable **CObject**-derived class typically has the following form:

```
void CPerson::Serialize(CArchive& ar)
{
CObject::Serialize(ar);
    if (ar.IsStoring())
    {
        // TODO:  add storing code here
    }
    else
    {
        // TODO:  add loading code here
    }
}
```

The above code template is exactly the same as the one that AppWizard creates for the `Serialize` function of the document (a class derived from **CDocument**). This code template helps you write code that is easier to review, because the storing code and the loading code should always be parallel, as in the following example:

```
void CPerson:Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        ar << m_strName;
        ar << m_wAge
    }
    else
        {
        ar >> m_strName;
        ar >> m_wAge;
    }
}
```

The library defines << and >> operators for **CArchive** as the first operand and the following data types and class types as the second operand:

CObject* (see discussion following)	**BYTE**
WORD	**LONG**
DWORD	**float**
double	**POINT** and **CPoint**
SIZE and **CSize**	**RECT** and **CRect**
CString	**CTime and CTimeSpan**

The **CArchive** << and >> operators always return a reference to the **CArchive** object, which is the first operand. This enables you to chain the operators, as illustrated below:

```
BYTE bSomeByte;
WORD wSomeWord;
DWORD wSomeDoubleWord;
...
ar << bSomeByte << wSomeWord << wSomeDoubleWord;
```

Storing and Loading CObjects via an Archive

Storing and loading **CObject**s via an archive requires extra consideration. In certain cases, you should call the `Serialize` function of the object, where the **CArchive** object is a parameter of the `Serialize` call, as opposed to using the << or >> operator of the **CArchive**. The important fact to keep in mind is that the **CArchive** >> operator constructs the **CObject** in memory based on **CRuntimeClass** information previously written to the file by the storing archive.

Therefore, whether you use the **CArchive** << and >> operators, versus call `Serialize`, depends on whether you *need* the loading archive to dynamically reconstruct the object based on previously stored **CRuntimeClass** information. Use the `Serialize` function in the following cases:

1. When deserializing the object, you know the exact class of the object beforehand.
2. When deserializing the object, you already have memory allocated for it.

Note If you load the object using the `Serialize` function, then you must also store the object using the `Serialize` function. Don't store using the **CArchive** << operator and then load using the `Serialize` function, or store using the `Serialize` function and then load using **CArchive** >> operator.

The following example illustrates the cases:

```
class CMyObject : public CObject
{
    // ...Member functions
    CMyObject();
    virtual void Serialize( CArchive& ar );

// Implementation
protected:
    DECLARE_SERIAL( CMyObject )
};

class COtherObject : public CObject
{
    // ...Member functions
    COtherObject();
    virtual void Serialize( CArchive& ar );

// Implementation
protected:
    DECLARE_SERIAL( COtherObject )
};

class CCompoundObject : public CObject
{
    // ...Member functions
    CCompoundObject();
    virtual void Serialize( CArchive& ar );

// Implementation
protected:
    CMyObject m_myob;   // Embedded object
    COtherObject* m_pOther; // Object allocated in constructor
    CObject* m_pObDyn;  // Dynamically allocated object
    //..Other member data and implementation


    DECLARE_SERIAL( CCompoundObject )

};


CCompoundObject::CCompoundObject()
{
    m_pOther = new COtherObject; // Exact type known and object already
                //allocated.
    m_pObDyn = NULL;    // Will be allocated in another member function
                // if needed, could be a derived class object.
}
```

```
void CCompundObject::Serialize( CArchive& ar )
{
    CObject::Serialize( ar );   // Always call base class Serialize.
    m_myob.Serialize( ar ); // Call Serialize on embedded member.
    m_pOther->Serialize( ar );  // Call Serialize on objects of known
                exact type.

    // Serialize dynamic members and other raw data
    if ( ar.IsStoring() )
    {
        ar << m_pObDyn;
        // Store other members
    }
    else
    {
        ar >> m_pObDyn; // Polymorphic reconstruction of persistent
                // object
                //load other members
    }
}
```

In summary, if your serializable class defines an embedded **CObject** as a member, then you should *not* use the **CArchive** << and >> operators for that object, but should call the `Serialize` function instead. Also, if your serializable class defines a pointer to a **CObject** as a member, but constructs this other object in its own constructor, then you should also call `Serialize`.

CHAPTER 15

Diagnostics

The Microsoft Foundation Class Library contains many diagnostic features to help debug your program during development. These features, especially those that track memory allocations, will slow down your program. Others, such as assertion testing, will cause your program to halt when erroneous conditions are encountered.

In a commercial retail product, slow performance and program interruption are clearly unacceptable. For this reason, the Microsoft Foundation Class Library provides a method for turning the debugging and diagnostic features on or off. When you are developing your program, you typically build a debug version of your program and link with the Debug version of the Microsoft Foundation Class Library. Once the program is completed and debugged, you build a release version and link with the Release version of the Microsoft Foundation library.

Debugging Features

The following features are included for all classes derived from **CObject** in the Debug version of the Microsoft Foundation Class Library:

- **Dump** member function to dump object contents to debugging output
- Trace output to print or display debugging output to evaluate argument validity
- Assertions and **AssertValid** member function
- Memory diagnostics to detect memory leaks
- **DEBUG_NEW** macro to show where objects were allocated

▶ To enable the debugging features

1. Compile with the symbol **_DEBUG** defined. This is typically done by passing the **/D_DEBUG** flag on the compiler command line. To accomplish this in Visual C++, set the Debug option in the Project Options dialog box, which happens automatically when you turn on the Use Microsoft Foundation Class check box. When you define the **_DEBUG** symbol, sections of code delimited by **#ifdef _DEBUG / #endif** are compiled.

2. Link with the Debug versions of the Microsoft Foundation Class Library. Setting the Debug option in Visual C++ ensures linking with the Debug libraries. The Debug versions of the library have a "D" at the end of the library name. For example, the medium-model Debug version of the Microsoft Foundation Class Library for Windows is named MAFXCWD.LIB, and the Release version (non-debug) is named MAFXCW.LIB.

Dumping Object Contents

When deriving a class from **CObject**, you have the option to override the **Dump** member function and write a textual representation of the object's member variables to a dump context, which is similar to an I/O stream. Like an I/O stream, you can use the insertion (<<) operator to send data to a **CDumpContext**.

You do not have to override **Dump** when you derive a class from **CObject.** However, if you use other diagnostic features for debugging, providing the capability for dumping an object and viewing its contents is very helpful and highly recommended.

▶ **To override the Dump member function**

1. Call the base class version of **Dump** to dump the contents of a base class object.
2. Write a textual description and value for each member variable of your derived class.

The declaration of the `Dump` function in the class declaration looks like:

```
class CPerson : public CObject
{
public:
#ifdef _DEBUG
    virtual void Dump( CDumpContext& dc ) const;
#endif

    CString m_firstName;
    CString m_lastName;
    // etc. ...
};
```

Note Since object dumping only makes sense when you are debugging your program, the declaration of the `Dump` function is bracketed with an **#ifdef _DEBUG / #endif** block.

In the following example from an implementation file for the class `CPerson`, the `Dump` function's first statement calls the **Dump** member function for its base class.

It then writes a short description of each member variable along with the member's value to the diagnostic stream.

```
#ifdef _DEBUG
void CPerson::Dump( CDumpContext& dc ) const
{
    // call base class function first
    CObject::Dump( dc );

    // now do the stuff for our specific class
    dc   << "last name: " << m_lastName << "\n"
        << "first name: " << m_firstName ;
}
#endif
```

Note Again, notice that the definition of the `Dump` function is bracketed by **#ifdef _DEBUG / #endif** directives. If you refer to **afxDump** in a program linked with the nondebug libraries, you will get unresolved externals errors at link time.

▶ **To send Dump output to afxDump**

- You must supply a **CDumpContext** argument to specify where the dump output will go when you call the `Dump` function for an object. The Microsoft Foundation Class Library supplies a predefined **CDumpContext** object named **afxDump** that you will normally use for routine object dumping. The following example shows how to use **afxDump**:

```
CPerson pMyPerson = new CPerson;
// set some fields of the CPerson object...
//..
// now dump the contents
#ifdef _DEBUG
pMyPerson->Dump( afxDump );
#endif
```

  In Windows, **afxDump** output is sent to the debugger, if present. In MS-DOS, **afxDump** output is sent to **stderr**.

  Note **afxDump** is defined only in the Debug version of the Microsoft Foundation Class Library.

The TRACE Macro

The **TRACE** macro can be used during development to print or display debugging messages from a program. **TRACE** prints a string argument to the current diagnostic output device. For programs with character-based output in MS-DOS,

the **TRACE** output will go to **stderr**. For Windows programs, the **TRACE** output will be directed to your debugger (or to the AUX port, which can be captured using DBWIN.EXE, found in \MSVC\BIN).

The **TRACE** macro can handle a variable number of arguments, similar to the way **printf** operates. Following are examples of different ways to use **TRACE** macros:

```
int x = 1;
int y = 16;
float z = 32.0;
TRACE( "This is a TRACE statement\n" );

TRACE( "The value of x is %d\n", x );

TRACE( "x = %d and y = %d\n", x, y );

TRACE( "x = %d and y = %x and z = %f\n", x, y, z );
```

The **TRACE** macro is active only in the Debug version of the library. After a program has been debugged, you can build a Release version to deactivate all **TRACE** calls in the program. For important information on the **TRACE** macro, see "Globals and Macros" in the *Class Library Reference* and Technical Note 7 in \MSVC\HELP\MFCNOTES.HLP.

The ASSERT Macro

The most typical use of the **ASSERT** macro is to identify program errors used during development. The argument given to **ASSERT** should be chosen so that it holds true only if the program is operating as intended. The macro evaluates the argument; and if the argument expression is false (0), prints a diagnostic message and halts program execution. No action is taken if the argument is true (nonzero).

In MS-DOS, the diagnostic message is sent to **afxDump** and has the form:

```
assertion failed in file <name> in line <num>
```

where <name> is the name of the source file and <num> is the line number of the assertion that failed. In Windows, the following message box appears:

```
assertion failed in file <name> in line <num>
Abort Retry Ignore
```

If you choose Abort, program execution terminates. If you choose Retry, the debugger is activated. If you choose Ignore, program execution continues.

The following example shows how the **ASSERT** macro could be used to check the validity of a function's return value:

```
int x = SomeFunc(y);
ASSERT(x == 0);   //  Assertion fails if x not equal to 0
```

ASSERT can also be used in combination with the **IsKindOf** function to provide extra checking for function arguments, such as in the following example. (For a discussion of the **IsKindOf** function, see "Accessing Run-Time Class Information" on page 214).

```
ASSERT( pObject1->IsKindOf( RUNTIME_CLASS( CPerson ) ) );
```

The liberal use of assertions throughout your programs can catch errors during development. A good rule of thumb is that you should write assertions for any assumptions you make. For example, if you assume that an argument is not **NULL**, then you should use an assertion statement to check for that condition.

The **ASSERT** macro will catch errors only when you are using the Debug version of the Microsoft Foundation Class Library during development. It will be turned off (produce no code) when you build your program with the Release version of the library.

Note The expression argument to **ASSERT** will not be evaluated in the release version of your program. If you want the expression to be evaluated in both debug and release environments, use the **VERIFY** macro instead of **ASSERT**. In debug versions, **VERIFY** simply passes its expression argument to **ASSERT**. In release environments, **VERIFY** evaluates the expression argument but does not check the result.

The ASSERT_VALID Macro

Use the **ASSERT_VALID** macro to perform a run-time check of an object's internal consistency. The class of that object should override the **AssertValid** function of **CObject** as described in the next section. The **ASSERT_VALID** macro is a more robust way of accomplishing:

```
pObject->AssertValid();
```

Like the **ASSERT** macro, **ASSERT_VALID** is turned on in the debug version of your program, but turned off in the release version.

Overriding the AssertValid Function

The **AssertValid** member function is provided in **CObject** to allow run-time checks of an object's internal state. **AssertValid** typically performs assertions on all the object's member variables to see if they contain valid values. For example, **AssertValid** can check that all pointer member variables are not **NULL**. If the object is invalid, **AssertValid** halts the program.

Although you are not required to override **AssertValid** when you derive your class from **CObject**, you can make your class safer and more reliable by doing so. The following example shows how to declare the **AssertValid** function in the class declaration:

```
class CPerson : public CObject
{
protected:
    CString m_strName;
    float   m_Salary;
public:
    virtual void AssertValid() const    // Override

    // ...
};
```

When you override `AssertValid`, first call **AssertValid** for the base class. Then use the **ASSERT** macro to check the validity of the members that are unique to your derived class, as shown by the following example:

```
void CPerson::AssertValid()
{
    // call inherited AssertValid first
    CObject::AssertValid()

    // check CPerson members...
    ASSERT( !m_strName.IsEmpty()); \\Must have a name.
    ASSERT( m_Salary ! 0 ); \\Must have an income
}
```

If any of the member variables of your class store objects, you can use the **ASSERT_VALID** macro to test their internal validity (if their classes override **AssertValid**). The following example shows how this is done.

Consider a class `CDataBase`, which stores a **CObList** in one of its member variables. The **CObList** variable, `m_DataList`, stores a collection of `CPerson` objects. An abbreviated declaration of `CDataBase` looks like this:

```
class CDataBase : public CObject
{
   // Constructor and other members ...
   protected:
      CObList * m_DataList;
   // Other declarations ...
   public:
      virtual void AssertValid( ) const; // Override
   // Etc. ...
};
```

The **AssertValid** override in `CDataBase` looks like this:

```
void CDataBase::AssertValid( )
{
   // Call inherited AssertValid
   CObject::AssertValid( );
   // Check validity of CDataBase members
   ASSERT_VALID( m_pDataList );
   // ...

}
```

`CDataBase` uses the **AssertValid** mechanism to add validity tests for the objects stored in its data member to the validity test of the `CDataBase` object itself. The overriding **AssertValid** of `CDataBase` invokes the **ASSERT_VALID** macro for its own **m_pDataList** member variable.

The chain of validity testing might stop at this level, but in this case class **CObList** overrides **AssertValid** too, and the **ASSERT_VALID** macro causes it to be called. This override performs additional validity testing on the internal state of the list. If an assertion failure occurs, diagnostic messages are printed, and the program halts.

Thus a validity test on a `CDataBase` object leads to additional validity tests for the internal states of the stored **CObList** list object. With a little more work, the validity tests could include the `CPerson` objects stored in the list as well. You could derive a class `CPersonList` from **CObList** and override **AssertValid**. In the override, you would call **CObject::AssertValid** and then iterate through the list, calling `AssertValid` on each `CPerson` object stored in the list. `CPerson` already overrides **AssertValid**.

This is a powerful mechanism when you build for debugging, and when you subsequently build for release, the mechanism is turned off automatically.

Users of an `AssertValid` function of a given class should be aware of the limitations of this function. A triggered assertion indicates that the object is definitely bad and execution will halt. However, a lack of assertion only indicates that no problem was found, but the object isn't guaranteed to be good.

Detecting Memory Leaks

A memory leak occurs when you allocate memory on the heap and never deallocate that memory to make it available for reuse, or if you mistakenly use memory that has already been allocated. This is a particular problem for programs that are intended to run for extended periods. In a long-lived program, even a small incremental memory leak can compound itself; eventually all available memory resources are exhausted and the program crashes. Traditionally, memory leaks have been very hard to detect.

The Microsoft Foundation Class Library provides classes and functions that you can use to detect memory leaks during development. Basically, these functions take a snapshot of all memory blocks before and after a particular set of operations. You can use these results to determine if all memory blocks allocated during the operation have been deallocated.

The size of the operation that you choose to bracket with these diagnostic functions is arbitrary. It can be as small as a single program statement, or it can span the entry and exit from the entire program. Either way, these functions will allow you to detect memory leaks and identify which memory blocks have not been deallocated properly.

Memory Diagnostics

▶ **To enable or disable memory diagnostics**

- Call the global function **AfxEnableMemoryTracking** to enable or disable the diagnostic memory allocator. Since memory diagnostics are on by default in the Debug library, you will typically use this function to temporarily turn them off, which increases program execution speed and reduces diagnostic output.

▶ **To select specific memory diagnostic features with afxMemDF**

- If you want more precise control over the memory diagnostic features, you can selectively turn individual memory diagnostic features on and off by setting the value of the Microsoft Foundation Class Library global variable **afxMemDF**. This variable can have the following values as specified by the enumerated type **AfxMemDF**:

Value	Meaning
allocMemDF	Turn on debugging allocator (default).
delayFreeMemDF	Delay freeing memory when calling **delete** or **free**. This will cause maximum memory stress for your program.
checkAlwaysMemDF	Call **AfxCheckMemory** every time memory is allocated or freed.

These values can be used in combination by performing a logical-OR operation, as shown here:

```
afxMemDF |= delayFreeMemDF | checkAlwaysMemDF;
```

Detecting a Memory Leak

The following instructions and examples show you how to detect a memory leak.

▶ To detect a memory leak

1. Create a **CMemoryState** object and call the **Checkpoint** member function to get the initial snapshot of memory.
2. After you perform the memory allocation and deallocation operations, create another **CMemoryState** object and call **Checkpoint** for that object to get a current snapshot of memory usage.
3. Create a third **CMemoryState** object, call the **Difference** member function, and supply the previous two **CMemoryState** objects as arguments. The return value for the **Difference** function will be nonzero if there is any difference between the two specified memory states, indicating that some memory blocks have not been deallocated.

   The following example shows how to check for memory leaks:

```
// Declare the variables needed
#ifdef _DEBUG
    CMemoryState oldMemState, newMemState, diffMemState;
#endif

#ifdef _DEBUG
    oldMemState.Checkpoint();
#endif

    // do your memory allocations and deallocations...
    CString s = "This is a frame variable";
    // the next object is a heap object
    CPerson* p = new CPerson( "Smith", "Alan", "581-0215" );

#ifdef _DEBUG
    newMemState.Checkpoint();
    if( diffMemState.Difference( oldMemState, newMemState ) )
    {
        TRACE( "Memory leaked !\n" );
    }
#endif
```

   Notice that the memory-checking statements are bracketed by **#ifdef _DEBUG / #endif** blocks so that they are only compiled in debug versions of your program.

Dumping Memory Statistics

The **CMemoryState** member function **Difference** determines the difference between two memory-state objects. It detects any objects that were not deallocated from the heap between the beginning and end memory-state snapshots.

▶ To dump memory statistics

- The following example (continuing the example from the previous section) shows how to call **DumpStatistics** to get information about the objects that have not been deallocated:

```
if( diffMemState.Difference( oldMemState, newMemState ) )
{
    TRACE( "Memory leaked !\n" );
    diffMemState.DumpStatistics();
}
```

A sample dump from the example above is shown here:

```
0 bytes in 0 Free Blocks
22 bytes in 1 Object Blocks
45 bytes in 4 Non-Object Blocks
Largest number used: 67 bytes
Total allocations: 67 bytes
```

- The first line describes the number of blocks whose deallocation was delayed if **afxMemDF** was set to **delayFreeMemDF**. For a description of **afxMemDF**, see the procedure "To select specific memory diagnostic features with afxMemDF" on page 248.
- The second line describes how many objects remain allocated on the heap.
- The third line describes how many nonobject blocks (arrays or structures allocated with **new**) were allocated on the heap and not deallocated.
- The fourth line gives the maximum memory used by your program at any one time.
- The last line lists the total amount of memory used by your program.

Dumping All Objects

DumpAllObjectsSince dumps out a description of all objects detected on the heap that have not been deallocated. As the name implies, **DumpAllObjectsSince** dumps all objects allocated since the last **Checkpoint**. However, if no **Checkpoint** has taken place, all objects and nonobjects currently in memory are dumped.

▶ **To dump all objects**

- Expanding on the previous example, the following code dumps all objects that have not been deallocated when a memory leak is detected:

```
if( diffMemState.Difference( oldMemState, newMemState ) )
{
    TRACE( "Memory leaked !\n" );
    diffMemState.DumpAllObjectsSince();
}
```

A sample dump from the preceding example is shown here:

```
Dumping objects ->

{5} strcore.cpp(80) : non-object block at $00A7521A, 9 bytes long
{4} strcore.cpp(80) : non-object block at $00A751F8, 5 bytes long
{3} strcore.cpp(80) : non-object block at $00A751D6, 6 bytes long
{2} a CPerson at $51A4

Last Name: Smith
First Name: Alan
Phone #: 581-0215

{1} strcore.cpp(80) : non-object block at $00A7516E, 25 bytes long
```

The numbers in braces at the beginning of most lines specify the order in which the objects were allocated. The most recently allocated object is displayed first. You can use these ordering numbers to help identify allocated objects.

Interpreting an Object Dump

The preceding dump comes from the original memory checkpoint example in "Detecting a Memory Leak" on page 249. Remember that there were only two explicit allocations in that program—one on the frame and one on the heap:

```
// do your memory allocations and deallocations ...
CString s = "This is a frame variable";
// the next object is a heap object
CPerson* p = new CPerson( "Smith", "Alan", "581-0215" );
```

Start with the `CPerson` object; its constructor takes three arguments that are pointers to **char**. The constructor uses these arguments to initialize **CString** member variables for the `CPerson` class. In the memory dump, you can see the `CPerson` object listed along with three nonobject blocks (3, 4, and 5) that hold the characters for the **CString** member variables. These memory blocks will be deleted when the destructor for the `CPerson` object is invoked.

Block number 2 represents the `CPerson` object itself. After the `CPerson` address listing, the contents of the object are displayed. This is a result of **DumpAllObjectsSince** calling the `Dump` member function for the `CPerson` object.

You can guess that block number 1 is associated with the **CString** frame variable because of its sequence number and its size, which match the number of characters in the frame **CString** variable. The allocations associated with frame variables are automatically deallocated when the frame variable goes out of scope.

In general, you shouldn't worry about heap objects associated with frame variables because they are automatically deallocated when the frame variables go out of scope. In fact, you should position your calls to **Checkpoint** so that they are outside the scope of frame variables to avoid clutter in your memory diagnostic dumps. For example, place scope brackets around the previous allocation code, as shown here:

```
oldMemState.Checkpoint();
{
    // do your memory allocations and deallocations ...
    CString s = "This is a frame variable";
    // the next object is a heap object
    CPerson* p = new CPerson( "Smith", "Alan", "581-0215" );
}
newMemState.Checkpoint();
```

With the scope brackets in place, the memory dump for this example is as follows:

```
Dumping objects ->

{5} strcore.cpp(80) : non-object block at $00A7521A, 9 bytes long
{4} strcore.cpp(80) : non-object block at $00A751F8, 5 bytes long
{3} strcore.cpp(80) : non-object block at $00A751D6, 6 bytes long
{2} a CPerson at $51A4

Last Name: Smith
First Name: Alan
Phone #: 581-0215
```

Notice that some allocation are objects (such as `CPerson`) and some are non-object allocations. "Non-object allocations" are allocations for objects not derived from **CObject** or allocations of primitive C types such as char, int, or long. If the **CObject**-derived class allocates additional space, such as for internal buffers, those objects will show both object and non-object allocations

Notice that the memory block associated with the **CString** frame variable has been deallocated automatically and does not show up as a memory leak. The automatic deallocation associated with scoping rules takes care of most memory leaks associated with frame variables.

For objects allocated on the heap, however, you must explicitly delete the object to prevent a memory leak. To clean up the last memory leak in the previous example, you can delete the `CPerson` object allocated on the heap, as follows:

```
{
    // do your memory allocations and deallocations ...
    CString s = "This is a frame variable";
    // the next object is a heap object
    CPerson* p = new CPerson( "Smith", "Alan", "581-0215" );
    delete p;
}
```

Using DEBUG_NEW to Aid Debugging

The Microsoft Foundation Class Library defines the macro **DEBUG_NEW** to assist you in locating memory leaks. You can use **DEBUG_NEW** everywhere in your program that you would ordinarily use the **new** operator.

When you compile a Debug version of your program, **DEBUG_NEW** keeps track of the filename and line number for each object that it allocates. Then, when you call **DumpAllObjectsSince**, as described in the previous section, each object allocated with **DEBUG_NEW** will be shown with the file and line number where it was allocated, thus allowing you to pinpoint the sources of memory leaks.

When you compile a Release version of your program, **DEBUG_NEW** resolves to a simple **new** operation without the filename and line number information. Thus, you pay no speed penalty in the Release version of your program.

▶ **To use DEBUG_NEW**

- Define a macro in your source files that replaces **new** with **DEBUG_NEW**, as shown here:

  ```
  #define new DEBUG_NEW
  ```

  You can then use **new** for all heap allocations. The preprocessor will substitute **DEBUG_NEW** when compiling your code. In the Debug version of the library, **DEBUG_NEW** will create debugging information for each heap block. In the Release version, **DEBUG_NEW** will resolve to a standard memory allocation without the extra debugging information.

Note You must place the **#define** statement after all statements that call the **IMPLEMENT_DYNCREATE** or **IMPLEMENT_SERIAL** macros in your module, or you will get a compile-time error.

CHAPTER 16

Exceptions

There are three categories of outcomes that can occur when a function is called during program execution: normal execution, erroneous execution, or abnormal execution. Each category is described below.

- Normal execution

  The function may execute normally and return. Some functions return a result code to the caller, which indicates the outcome of the function. The possible result codes are strictly defined for the function and represent the range of possible outcomes of the function. The result code can indicate success or failure or can even indicate a particular type of failure that is within the normal range of expectations. For example, a file-status function can return a code that indicates that the file does not exist. Note that the term "error code" is not used since a result code represents one of many expected outcomes.

- Erroneous execution

  The caller makes some mistake in passing arguments to the function or calls the function in an inappropriate context. This situation causes an error, and it should be detected by an assertion during program development. (For more information on assertions, see "The ASSERT Macro" on page 244.)

- Abnormal execution

  Abnormal execution includes situations where conditions outside the program's control are influencing the outcome of the function, such as low memory or I/O errors. Abnormal situations should be handled by catching and throwing exceptions.

Microsoft Foundation Classes Exception Handling

The Microsoft Foundation Class Library uses an exception-handling scheme that is very similar to one proposed by the ANSI standards committee for C++ 2.1. You set up an exception handler before calling functions that you think might encounter abnormal situations. If your program does run into abnormal conditions, then it

throws an exception. When an exception is thrown, program execution jumps to the exception handler and execution resumes there.

Exceptions are represented as objects derived from the abstract class **CException**. The Microsoft Foundation Class Library provides several predefined kinds of exceptions:

Exception Class	Meaning
CMemoryException	Out-of-memory
CFileException	File exception
CArchiveException	Archive/Serialization exception
CNotSupportedException	Response to request for unsupported service
CResourceException	Windows resource allocation exception
COleException	OLE exceptions
CUserException	Exception that alerts the user with a message box, then throws a generic **CException**

Since many parts of the Microsoft Foundation Class Library, especially those dealing with files and serialization, use exceptions to report abnormal conditions, you will find it useful to use the Microsoft Foundation exception-handling mechanism in the parts of your program that call those types of Microsoft Foundation Class Library functions. For a description of each Microsoft Foundation Class Library function and the exceptions that can possibly be thrown by that function, see the *Class Library Reference*. If you see that a function can throw an exception, you should probably surround it with an exception handler.

Catching Exceptions

The following instructions and examples will show you how to catch exceptions.

▶ **To catch exceptions**

- Use the **TRY** macro to set up a **TRY** block. Execute any program statements that might throw an exception within a **TRY** block.

  Use the **CATCH** macro to set up a **CATCH** block. Place exception handling code in a **CATCH** block. The code in the **CATCH** block is executed only if the code within the **TRY** block throws an exception of the type specified in the **CATCH** statement.

The following skeleton shows how **TRY** and **CATCH** blocks are normally arranged:

```
// Normal program statements
...

TRY
{
    // Execute some code that might throw an exception.
}
CATCH( CException, e )
{
    // Handle the exception here.
    // "e" contains information about the exception
}
END_CATCH

// Other normal program statements
...
```

Note The **END_CATCH** macro marks the end of the **CATCH** blocks.

The **CATCH** macro takes an exception-type parameter, allowing you to selectively handle different types of exceptions with sequential **CATCH** and **AND_CATCH** blocks as listed below:

```
TRY
{
    // Execute some code that might throw an exception.
}
CATCH( CMemoryException, e )
{
    // Handle the out-of-memory exception here.
}
AND_CATCH( CFileException, e )
{
    // Handle the file exceptions here.
}
AND_CATCH( CException, e )
{
    // Handle all other types of exceptions here.
}
END_CATCH
```

Examining Exception Contents

The **CATCH** macro includes an argument that is used to hold a pointer to a **CException** object (or an object derived from **CException**, such as **CMemoryException**). Depending on the exact type of the exception, you can examine the data members of the exception object to gather information about the specific cause of the exception.

For example, the **CFileException** type has the **m_cause** data member that contains an enumerated type that specifies the cause of the file exception. Some examples of the possible return values are **CFileException::fileNotFound** and **CFileException::readOnly**.

▶ **To examine exception contents**

- The following example shows how to examine the contents of a **CFileException**. Other exception types can be examined in a similar way.

```
TRY
{
    // Do something to throw a file exception.
}
CATCH( CFileException, theException )
{
    if( theException->m_cause == CFileException::fileNotFound )
       TRACE( "File not found\n" );
}
END_CATCH
```

Freeing Objects in Exceptions

The exception-handling mechanism of the Microsoft Foundation Class Library can interrupt normal program flow. Thus, it is very important to keep close track of objects that have been created on the heap so that you can properly dispose of them in case an exception is thrown.

There are two primary methods to do this.

- Handle exceptions locally using the **TRY** and **CATCH** macros, then destroy all objects with one statement.
- Destroy any object in the **CATCH** block before the exception is thrown outside for further handling.

These two approaches are illustrated below as solutions to the following problematic example:

```
void SomeFunc()
{
    CPerson* myPerson = new CPerson;

    // Do something that might throw an exception.
    myPerson->SomeFunc();

    // Now destroy the object before exiting.
    delete myPerson;
}
```

As written above, `myPerson` will not be deleted if an exception is thrown by `SomeFunc`. Execution jumps directly to the innermost exception handler, bypassing the normal function exit and the code that deletes the object. The pointer to the object goes out of scope when the exception leaves the function, and the memory occupied by the object will never be recovered as long as the program is running. This is known as a memory leak and would be detected by using the memory diagnostics.

Handle the Exception Locally

The **TRY/CATCH** paradigm provides a defensive programming method for avoiding memory leaks and ensuring that your objects are destroyed when exceptions occur. For example, the previous example could be rewritten as shown below:

```
void SomeFunc()
{
    CPerson* myPerson = new CPerson;

    TRY
    {
        // Do something that might throw an exception.
        myPerson->SomeFunc();
    }
    CATCH( CException, e )
    {
        // Handle the exception locally.
    }
    END_CATCH

    // Now destroy the object before exiting.
    delete myPerson;
}
```

This new example sets up an exception handler to catch the exception and handle it locally. It then exits the function normally and destroys the object. The important aspect of this example is that a context to catch the exception is established with the **TRY/CATCH** blocks. Without a local exception frame, the function would never know that an exception had been thrown and would not have the chance to exit normally and destroy the object.

Throw Exceptions After Destroying Objects

Another way to handle exceptions is to pass them on to the next outermost exception-handling context. In your **CATCH** block, you can do some cleanup of your locally allocated objects and then throw the exception on for further processing. The following example shows how this can be done:

```
void SomeFunc()
{
    CPerson* myPerson = new CPerson;

    TRY
    {
        // Do something that might throw an exception.
        myPerson->SomeFunc();
    }
    CATCH( CException, e )
    {
        // Destroy the object before passing exception on.
        delete myPerson;
        // Throw the exception to the next handler.
        THROW_LAST( );
    }
    END_CATCH

    // On normal exits, destroy the object.
    delete myPerson;
}
```

If you call functions that can throw exceptions, you can use **TRY/CATCH** blocks to make sure that you catch the exceptions and have a chance to destroy any objects you have created. In particular, be aware that many Microsoft Foundation Class Library functions can throw exceptions.

Throwing Exceptions from Your Own Functions

It is possible to use the Microsoft Foundation Class Library exception-handling paradigm solely to catch exceptions thrown by functions in the Microsoft Foundation Class Library or other libraries. In addition to catching exceptions thrown by library code, you can throw exceptions from your own code if you are writing functions that can encounter exceptional conditions.

▶ To throw an exception

- Use one of the Microsoft Foundation Class Library helper functions, such as **AfxThrowMemoryException**, listed in AFX.H. These functions throw a preallocated exception object of the appropriate type.

  When an exception is thrown, execution of the current function is aborted and jumps directly to the **CATCH** block of the innermost exception frame. The exception mechanism bypasses the normal exit path from a function. Therefore, you must be sure to delete those memory blocks that would be deleted in a normal exit. In the following example, a function tries to allocate two memory blocks and throws an exception if either allocation fails:

```
{
    char* p1 = malloc( SIZE_FIRST );
    if( p1 == NULL )
        AfxThrowMemoryException();
    char* p2 = malloc( SIZE_SECOND );
    if( p2 == NULL )
    {
        free( p1 );
        AfxThrowMemoryException();
    }

    // ... Do something with allocated blocks ...

    // In normal exit, both blocks are deleted.
    free( p1 );
    free( p2 );
}
```

  If the first allocation fails, you can simply throw the memory exception. If the first allocation is successful but the second one fails, you must free the first allocation block before throwing the exception. If both allocations succeed, then you can proceed normally and free the blocks when exiting the function.

Exceptions in Constructors

When throwing an exception in a constructor, clean up whatever objects and memory allocations you have made prior to throwing the exception, as explained in the previous section.

Throwing an exception in a constructor is tricky, however, because the memory for the object itself has already been allocated by the time the constructor is called. There is no simple way to deallocate the memory occupied by the object from within the constructor for that object. Thus, you will find that throwing an exception in a constructor will result in the object remaining allocated. For a discussion of how to detect objects in your program that have not been deallocated, see "Detecting Memory Leaks" on page 247.

If you are performing operations in your constructor that can fail, it might be a better idea to put those operations into a separate initialization function rather than throwing an exception in the constructor. That way, you can safely construct the object and get a valid pointer to it. Then, you can call the initialization function for the object. If the initialization function fails, you can delete the object directly.

Frame Variables and Exceptions

Explicitly allocated heap objects must also be deallocated before an exception is thrown. With frame objects, the frame memory will be reclaimed automatically by the exception mechanism. Although the memory occupied by the frame object is reclaimed, the destructor for the frame object is not executed by the exception mechanism.

For most objects, the reclamation of frame space is sufficient to clean up the object. But, for objects that allocate memory in addition to the frame space they occupy, such as **CString** objects, the default exception handling is not sufficient to completely deallocate the object. In addition, objects whose destructors are an integral part of their operations need special handling during exceptions.

When a **CString** object is allocated on the frame, its constructor also allocates memory on the heap to hold the characters of the string. Thus, a **CString** occupies space on the frame and also on the heap. When a **CString** frame variable is destroyed normally, its destructor takes care of deallocating the heap space used by the object. When the normal destruction of the **CString** is bypassed by an exception, this heap space is not deallocated, even though the frame space occupied by the **CString** is reclaimed.

▶ To avoid a CString memory leak

- Call the **Empty** function for any **CString** frame variables when handling an exception. The following example shows how to do this:

```
{
    CString s1 = "This is a test";

    TRY
    {
        char* p1 = new char[ A_BIG_BLOCK ];
    }
    CATCH( CMemoryException,e )
    {
        // Deallocate heap space used by string.
        s1.Empty();

        // Now you can safely throw the exception.
        THROW_LAST( );
    }
    END_CATCH
}
```

The need to explicitly deallocate heap resources for a frame-based object is not limited to **CString** objects. Since the destructors for frame objects are not automatically executed when an exception interrupts normal program flow, any frame-based object where the destructor performs significant tasks will need special attention during exception handling.

CHAPTER 17

Programming with VBX Controls

A wide variety of custom controls have been developed for the Visual Basic programming environment. You can use these controls within Visual C++ to improve the visual appeal and functionality of your programs.

Custom controls are also available from other independent software vendors. Since these controls can be used in programming systems other than Visual Basic, they are referred to generically as "VBX" controls.

With Visual C++, you can:

- Install VBX controls into App Studio and make them part of App Studio's environment.
- Use Class Wizard to define message maps for VBX controls, create member variables, establish dynamic data exchange (DDX), and initialize control properties.
- Use property sheets to set various VBX control properties interactively.
- Use the member functions of class **CVBControl** to create and manipulate VBX controls within your programs.

Additional Background

VBX controls are stored in a standard file format, usually with an extension of "VBX." VBX files are actually Windows DLLs that can be created with Visual C++. More than one custom control can be stored in a single VBX file.

VBX files can be stored anywhere on your hard disk, but a common convention is to store them in the \WINDOWS\SYSTEM directory. This provides a common location in which applications can look to find the controls they must access.

Each custom control has its own set of properties and events. Because every custom control is different, you must have documentation for the control, including which properties it has and which events it sends. This documentation is usually available from the provider of the VBX custom control DLL.

Examples Used in this Chapter

The source code examples used in this chapter are small sections of code taken from the VBCIRCLE sample program, which can be found in the MSVC\MFC\SAMPLES directory.

The VBCIRCLE sample uses a very rudimentary VBX control named CIRC3.VBX that is provided with Visual C++. The CIRC3 control consists only of a circle and a control caption. When the user clicks inside or outside the circle, the VBCIRCLE program displays an appropriate message stating where the user clicked. Although not very sophisticated, it provides a simple example of how to program with VBX controls.

VBCHART is a more sophisticated sample program that is also found in the MSVC\MFC\SAMPLES directory. It uses the GRID.VBX control included with Visual C++ and is also provided with the Visual Basic Professional Toolkit. The VBCHART sample allows the user to enter numbers into a table, and then displays a graphical chart of the data.

Before reading this chapter, you may want to compile and run both of these programs to see what they do. As you read through this chapter, you can look at the source code for both the VBCIRCLE and VBCHART samples.

An Overview of Using VBX Controls

Before you start programming with VBX controls, it is helpful to understand the basic steps involved with using these controls in Visual C++. A typical sequence of steps you will take to incorporate a VBX control in your program is as follows:

1. If you are generating a new "starter" application with AppWizard, make sure the Custom VBX Controls option is checked in AppWizard's Options dialog box. If you are adding custom controls to an existing application, make sure that VBX support in the framework is properly initialized (see the following section).
2. Install the VBX control within App Studio. This places the control on App Studio's control palette, and you can drag it into your dialog boxes, resize it, move it, and set its properties using property sheets.
3. Use ClassWizard to declare a pointer to the control, set up a message map, register the control's events, and create new member functions that handle the control's events.
4. Use the Visual Workbench editor to add the code necessary to create and manipulate the control as well as handle the control's messages. This is done using the member functions of class **CVBControl**.

The first part of this chapter provides an overview of how to use ClassWizard to generate much of the code you need to start programming with VBX controls. The second part of this chapter shows you how to program with the controls.

For more information on using AppWizard to generate starter applications that support VBX, see Chapter 13 of the *Visual Workbench User's Guide*. For more information on installing VBX controls within App Studio, see Chapter 3 of the *App Studio User's Guide*.

Initializing VBX Runtime Support

To use VBX controls in your program, you need to initialize the VBX support within the Microsoft Foundation Class Library. To do this, add the following statement anywhere within your application's **InitInstance** function:

```
EnableVBX();
```

If you use AppWizard with the VBX support option checked, this line will be added automatically to **InitInstance**.

Using ClassWizard to Set Up a VBX Control

ClassWizard simplifies programming with VBX controls by doing much of the preparatory work for you. You use ClassWizard to:

- Create a member variable that points to the control. This pointer is placed in the declaration of the class (usually derived from **CDialog**) that contains the VBX control.
- Register events (messages) for the control.
- Create a message map that associates the control's events to the member functions that will handle the events.
- Create message-handling functions for VBX control notifications in your implementation file.

This section provides an overview of these steps. If you are not familiar with ClassWizard, you should read Chapter 9 of the *App Studio User's Guide* before reading the rest of this chapter.

Declaring the Control Pointer

A VBX control is typically used within a window such as a dialog box or a form. When you set up the member variables and functions for the window containing the control, you use ClassWizard to generate a pointer to the control in your class. This pointer can be used within your message-handler function implementations to invoke the control's member functions.

▶ **To generate a control pointer**

1. From the main ClassWizard dialog box, select the class that contains the VBX control.
2. Click the Edit Variables button.
3. In the Edit Member Variables dialog box, select the VBX control's ID in the list of controls, then click the Add Variable button.
4. In the Add Member Variable dialog box, select Control in the Property list box. The Variable Type list box will display the type CVBControl*.

For example, if you are placing a control in a dialog box such as that used in the VBCIRCLE sample in the MSVC\MFC\SAMPLES directory, the code added by ClassWizard in the header file looks something like the following:

```
class CCircleDialog : public CDialog
{
    DECLARE_DYNAMIC(CCircleDialog)
// Construction
public:
    CCircleDialog(CWnd* pParent = NULL); // standard constructor

    //{{AFX_DATA(CCircleDialog)
    enum { IDD = IDD_CIRCLE_DLG };
    BOOL    m_bBorder;
    CString m_strCaption;
    CVBControl* m_circle;
    int     m_nShape;
    //}}AFX_DATA
```

The **m_circle** member is a pointer to the circle control object and is used within the program to access the control's member functions.

Handling VBX Control Messages

VBX controls have events associated with them. These events are sent as control notification messages to your dialog class. These events can include mouse clicks on certain areas of the control, keystrokes, and other actions. The list of control events is different for each control and you should refer to the control's documentation for more information about these events.

To handle events generated by VBX controls, you use ClassWizard to set up a message map that associates each event with a function that handles it.

▶ To set up a message map for VBX events

1. Within ClassWizard, select the class that contains the control (in the Class Name list box).
2. In the Object IDs list box, select the object ID of the control. A list of all VBX control notification messages for the control will be displayed.
3. Select the notification message to which you want to assign a function handler, then click the Add Function button.
4. In the Add Function dialog box, enter the name of the function that will handle the notification message.
5. Repeat steps 3 and 4 for each notification message you want to handle.

For example, in the VBCIRCLE sample program, ClassWizard generates the following message map in the DIALOG.CPP implementation file:

```
BEGIN_MESSAGE_MAP(CCircleDialog, CDialog)
    //{{AFX_MSG_MAP(CCircleDialog)
    ON_BN_CLICKED(IDC_UPDATE_CIRCLE, OnUpdateCircle)
    ON_VBXEVENT(VBN_CLICKIN, IDC_CIRCLE, OnClickinCircle)
    ON_VBXEVENT(VBN_CLICKOUT, IDC_CIRCLE, OnClickoutCircle)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

This message map contains several **ON_VBXEVENT** macros that map a single VBX control notification message event to a handler function. In this case, the two notification messages involve clicking inside (**VBN_CLICKIN**) or outside (**VBN_CLICKOUT**) the control's circle.

ClassWizard also adds the following event register map in the application's main implementation file:

```
//{{AFX_VBX_REGISTER_MAP()
        UINT NEAR VBN_CLICKIN = AfxRegisterVBEvent("CLICKIN");
        UINT NEAR VBN_CLICKOUT = AfxRegisterVBEvent("CLICKOUT");
//}}AFX_VBX_REGISTER_MAP
```

This map registers the VBX events as control notification messages. These are maintained by ClassWizard and you should not have to edit these entries.

Adding Code to Create and Use VBX Controls

Once ClassWizard has generated the necessary variables, data maps, and message maps in your source files, you can begin using the member functions of class **CVBControl** to create and manipulate the control in your program. This section provides an overview of programming with VBX controls.

Constructing and Creating the Control

If you use a VBX control within a dialog box or form view edited with App Studio, the control is created automatically from the information in the dialog template resource. In this case, there is no need to call the constructor or the **Create** member.

If you use a VBX control in a window other than a dialog box or form view, you must construct the control object with the **CVBControl** constructor and then use the **Create** member function of the **CVBControl** class to create it. For example, in the VBCircle sample, the **Create** function is called as follows:

```
int CVBCircleFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    // Call base class OnCreate() to create the main frame window.
    if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;

m_circle.Create("CIRC3.VBX;CIRC3;text from Create( ) call",
            WS_CHILD | WS_VISIBLE,      CRect(10, 30, 100, 100),
            this, 1);

// Remainder of function goes here
}
```

The first parameter of the **Create** function is an aggregate string that is parsed by the **Create** function. The three parts of this string are: the name of the file containing the control, the control name, and the initial window text. See the Microsoft Foundation Class Library alphabetical function reference for more information on the **CVBControl::Create**.

Manipulating the Control

Once the control has been created, you can manipulate it by using the member functions of class **CVBControl**. Typical operations are:

- Retrieving and setting control properties.
- Moving and resizing the control.
- Refreshing the control to reflect changes.

These operations can be accessed through the control's pointer that was generated by ClassWizard. For example, in the VBCIRCLE program, the function **OnUpdateCircle** uses values input by the user into a dialog box to update the border style, shape, and caption of the circle control:

```
void CCircleDialog::OnUpdateCircle()
{

    UpdateData(TRUE);

    m_circle->SetNumProperty("BorderStyle", m_bBorder);
    m_circle->SetNumProperty("CircleShape", m_nShape);
    m_circle->SetStrProperty("Caption", m_strCaption);
}
```

You can use the control pointer in this manner to access any **CVBControl** member functions.

Destroying the Control

There are several different ways to destroy **CVBControl** objects. If you set the *bAutoDelete* parameter of **CVBControl::Create** to **TRUE**, then the control and its associated C++ object are automatically destroyed when the parent window is destroyed.

If *bAutoDelete* is **FALSE,** you must explicitly destroy the control and delete the associated C++ object. The preferred method of doing this is to use **CWnd::DestroyWindow** to destroy the control and use **delete** to delete the object. For example:

```
m_pMyVBXControl->DestroyWindow();
//other destruction handling...
delete m_pMyVBXControl;
```

You should set *bAutoDelete* to **TRUE** if you allocate a **CVBControl** object on the heap.

Distributing VBX Controls with Applications

When an application needs to access controls in VBX files, the files must be on the system's path or in the same directory as the executable file.

If you want to distribute an application that uses custom controls, it is recommended that your installation procedure copy all required VBX files into the user's \WINDOWS\SYSTEM directory. This directory is the customary place for VBX

controls to be installed into, and is where most applications look to find the controls they need. Keeping controls in a single directory also prevents the proliferation of multiple control versions in different directories.

You can freely distribute the CIRC3.VBX and GRID.VBX control with any application you create with Visual C++.

CHAPTER 18

OLE Support

Object Linking and Embedding (OLE) is a mechanism that allows users to create and edit documents containing data created by multiple applications. This allows a single document to contain text, graphics, spreadsheets, sound, or other types of data. By supporting OLE in your application, you can let users combine data that it creates with data created by other applications.

This chapter describes how to use the Microsoft Foundation Class Library to write an application that supports OLE.

Overview of OLE

Before discussing how the Microsoft Foundation Class Library supports OLE, it's helpful to review the basic concepts involved in OLE. This section describes OLE from the user's point of view and then defines some common OLE terms. For a complete description of OLE, see the *OLE Programmer's Reference* or the *Microsoft Windows Software Development Kit.*

Using OLE

As an example of how OLE changes the way applications work together, consider how you would use a drawing program and a text editor together. You can demonstrate this yourself by using the Paintbrush and Write applications included with Windows version 3.1. Suppose you want to insert a picture in a text document.

▶ **To insert a picture in a text document**

1. Open Paintbrush and draw a picture.
2. Select a portion of the picture and copy it into the Clipboard.
3. Exit Paintbrush.
4. Open Write and enter some text.
5. Choose the Paste command. The portion of the picture you selected earlier is inserted into the document.

Now suppose that, after entering some more text, you want to modify the picture in the document. If Paintbrush and Write didn't support OLE, you would essentially have to repeat the entire process.

▶ **To modify a picture in a text document, without OLE**

1. Open Paintbrush again and reload the original picture (assuming you saved it; otherwise recreate it). Make the desired modifications.
2. Repeat steps 2 through 5 above, replacing the original picture in the document.

However, the Windows 3.1 versions of Paintbrush and Write support OLE. As a result, the process is much simpler.

▶ **To modify a picture in a text document, using OLE**

- Double-click on the picture in the document from within Write.

  Paintbrush is automatically invoked with the picture from your document. Modify the picture as desired and then exit the drawing program. Your document is still open in Write, but the picture has been updated. You can now continue entering text.

With OLE, Write and Paintbrush work together to produce a single document consisting of both text and graphics. You can edit the entire document as a single entity, rather than having to edit its components separately and combine them later. OLE allows you to invoke the appropriate application, as needed, whenever you work on a given portion of the document. A document that contains data created by different applications is called a "compound document."

While you use the Paste command in the same way in OLE and non-OLE applications, the command is clearly performing very different operations in each case. The difference can be clearly illustrated when you consider the action of pasting some cells from a worksheet into a text document. Compare what happens with a non-OLE spreadsheet application versus an OLE spreadsheet application, such as Microsoft Excel:

- If you paste some cells from a non-OLE worksheet into a text document, what gets inserted is a textual representation of the numbers in the cells.
- If you paste some cells from an Excel worksheet into a Write document, what gets inserted are the actual floating-point values used by Excel, plus any associated formulas. The textual representation is still used for display purposes, but when you double-click on those numbers while working within Write, the original floating-point values are sent back to Excel. This is what allows you to edit the cells again without re-entering the data.

In a non-OLE application, the Paste command inserts a representation of the original data using a format that the receiving application knows how to display. In

an OLE application, the Paste command still inserts this "presentation" data, but this is used for display purposes only; the command also inserts the actual data used by the donor application. This data is called the "Native" data for the item.

Embedded vs. Linked Items

Using the Paste command in an OLE application creates an "embedded item." An embedded item is stored as part of the compound document that contains it. In this way, a .WRI file for a Write document can contain not only text, but also bitmaps, floating-point numbers and formulas, or any other type of data.

OLE also provides another way to incorporate data from another application: creating a "linked item." The steps for creating a linked item are similar to those for creating an embedded item, except that instead of using the Paste command, you use the Paste Link command. The difference between a linked item and an embedded item is that with a linked item, the data is stored in a separate file.

For example, if you create a linked item of some cells from an Excel worksheet, the data for that item is stored in the .XLS file for that particular worksheet. The Write document contains only the information that specifies where the item can be found; that is, it contains a link to the item's .XLS file. When you double-click on those cells, Excel is launched and the original worksheet file is loaded in for editing.

A linked item has a couple of advantages over an embedded item:

- The compound document is smaller, since it contains only a link rather than all the data for the item.
- If the original document is updated, the linked item is also updated.

Every OLE item, whether embedded or linked, has a type associated with it; an item's type is based on the application that created it. For example, a Paintbrush item is one type of item, while an Excel item is another type. However, some applications can create more than one type of item; for example, Excel can create worksheet items, chart items, and macrosheet items.

Note The *Microsoft Windows Software Development Kit* refers to embedded and linked items as "objects" and refers to types of items as "classes." This chapter uses the term "item" to distinguish the OLE entity from the corresponding C++ object and the term "type" to distinguish the OLE category from the C++ class.

For more information on what kind of data is stored in embedded and linked items, see the *OLE Programmer's Reference* or the *Microsoft Windows Software Development Kit*.

Clients and Servers

An application that can incorporate embedded or linked items into its own documents is called a "client application." An application that can create items for use by client applications is called a "server application." In the earlier examples, Write is the client application while Paintbrush and Excel are the server applications.

The documents managed by a client application must be able to store and display OLE items as well as data created by the application itself. A client application must also allow users to insert new items or edit existing items. The user-interface requirements of a client application are listed in the section "Implementing a Client Application" later in this chapter.

Some server applications, such as Paintbrush, support the creation of embedded items only, while others, such as Excel, support the creation of both embedded and linked items. All server applications must be able to be invoked by a client application when the user wants to edit an item. If a server application supports linked items, it must also be able to copy its data to the Clipboard so that client can use it to create OLE items. An application can be both a client and a server; that is, it can both incorporate items into its documents, and its data can be incorporated as items into other application's documents.

Clients and servers do not communicate directly. Instead, they communicate indirectly through the OLE system DLLs. These DLLs provide functions that clients and servers call, and the clients and servers define callback functions that the DLLs call.

This system of communication is designed so that a client doesn't need to know anything specific about a server application. This allows a client to accept items created by any server; a client does not have to define what types of servers it can work with. As a result, the user of a client application, such as Write, can take advantage of new applications and data formats created in the future. As long as those new applications are OLE servers, a Write document will be able to incorporate items created by those applications.

Verbs

Double-clicking on an OLE item from within a client application typically launches the server application; this allows you to edit the item using the server. However, certain items don't behave this way. For example, double-clicking on an item created with the Sound Recorder application does not cause the server to be launched; instead it causes the sound to be played.

The reason for this difference in behavior is that Sound Recorder items have a different "primary verb." The primary verb is the action performed when the user double-clicks on an item. For most types of items, the primary verb is Edit, which

launches the server that created the item. For some types of items, such as Sound Recorder items, the primary verb is Play.

Most types of items support only one verb, and Edit is the most common one. However, some types of items support multiple verbs. For example, Sound Recorder items support Edit as a secondary verb.

Any verbs other than the primary verb must be invoked through a menu command when the item is selected. For information on the *typename* Object command, see the section "Implementing a Client Application" later in this chapter.

The verbs that a server application supports are listed in the Windows registration database. Your server application's verbs should be registered when it is installed. See the section "Registering a Server Application" later in this chapter.

The OLE Classes

The Microsoft Foundation Class Library provides nine classes that support OLE, which can be grouped into three categories: general classes, client classes and server classes.

The following are the general classes:

Name	Purpose
COleDocument	Provides generic functionality for a document containing OLE items; serves as base class for **COleClientDoc** and **COleServerDoc**.
CDocItem	Provides generic functionality for a document item; serves as base class for **COleClientItem** and **COleServerItem**.
COleException	Describes an error occurring during an OLE operation.

You do not use the **COleDocument** and **CDocItem** classes directly; they are abstract classes that define an interface for dealing with a document as a list of items. This interface is used by both **COleClientDoc** and **COleServerDoc** to their stored OLE items.

The **COleException** class is a special type of exception thrown during OLE operations. For more information on exception handling, see Chapter 16 of this book and Chapter 6 in the overview of the *Class Library Reference*.

The classes that you will primarily use are the client classes and the server classes. The following are the client classes:

Name	Purpose
COleClientDoc	Document managed by a client application; able to contain **COleClientItem** objects as well as the client application's own data.
COleClientItem	Defines client interface to an embedded or linked item.

To write a client application, you need the following:

- One document class for each type of document that your client application supports. Each class must be derived from **COleClientDoc**.

  Your application needs one object of a document class for each open document.

- One item class, which must be derived from **COleClientItem**. Note that only one item class is needed for embedded and linked items of all types.

  Your application needs one object of the item class for each embedded or linked item.

When writing your client application, you do not have to write any code for managing the specific contents of embedded or linked items. For example, you do not have to write code that manages floating-point numbers in order to accept embedded worksheet items. The framework and the OLE system DLLs manage all of this data for you. This is what allows your client application to accept items from any server.

The following are the server classes:

Name	Purpose
COleServer	Creates and manages server documents of a specific type; used for servers that are only launched by clients.
COleTemplateServer	Creates and manages server documents using a **CDocTemplate** object; used for servers that are launched as stand-alone applications, as well as by clients.
COleServerDoc	Document managed by a server application; able to treat its contents as **COleServerItem** objects.
COleServerItem	Defines server interface to an OLE item.

To write a server application, you need the following:

- One server class for each type of item your application supports. For example, if your application acts as a server for both worksheets and charts, you need two server classes. You can use either **COleTemplateServer** directly as a server class, or you can derive a class from **COleServer**.

  Your application needs only one object of each server class.

- One document class for each type of item your application supports. Each class must be derived from **COleServerDoc**.

  Your application needs one object of a document class for each open document.

- One item class for each type of item your application supports. Each class must be derived from **COleServerItem**.

  Your application needs one object of an item class for each embedded or linked item that is open. However, you may also use objects of an item class for representing data that is not an embedded or linked item.

Note that the **COleClientItem** and **COleServerItem** classes do not represent two different categories of items; they represent different interfaces to the same item. Each OLE item has both a **COleClientItem** interface and a **COleServerItem** interface.

As described earlier, client and server applications interact through the OLE system DLLs. A client application never directly calls a member function of a server class, and similarly, a server application never directly calls a member function of a client class.

Most of the member functions of the client classes are operations; you call them directly from your client application's code. By comparison, most of the member functions of the server classes are overridables; they are called by the OLE system DLLs (via the framework) in response to requests from the client. You typically don't call them from your server application's code.

Implementing a Client Application

A client application typically supports the following user-interface features:

- The Insert New Object command.

  The Insert New Object command typically appears on the Edit menu (though it may appear on the Insert menu, if your application has one). This command lets the user create an embedded item in a document in a more direct manner than the Paste command. When the user selects the Insert New Object command, the Insert Object dialog box appears. This dialog box displays a list of all the types of items available on the system. When the user chooses the type of item to embed, the appropriate server application is launched, allowing the user to create a new item. If the user chooses Update when exiting the server application, a new embedded item appears in the document.

- The Paste command.

  The Paste command appears on the Edit menu. It has the same behavior in OLE applications as it does in non-OLE applications, except that it can create an embedded item if the Clipboard contains data from a server application.

If the Clipboard doesn't contain all the data necessary to create an embedded item, the Paste command creates a "static" item. A static item cannot be edited.

- The Paste Link command.

  The Paste Link command appears on the Edit menu if the client application accepts linked items. It is the most common way to create a linked item. (You can also create linked items using the Paste Special command. However, the Paste Special command has a more complicated user interface than the Paste Link command and is not commonly implemented.)

- Invoking a verb on an item.

  There are two ways for the user to invoke a verb on an item: by double-clicking the item or by using the *typename* Object command.

  When the user of the client application double-clicks an item, typically the item's primary verb is executed. The Edit verb launches the server application and displays the item for editing. The Play verb also launches the server application, but may not display anything (for example, if playing a sound item).

  The *typename* Object command appears on the Edit menu. This command lets the user execute any of the verbs that an item supports. It differs from most other commands in that its appearance and behavior depend entirely on what type of OLE item is selected. The command has the following behavior:

  1. If nothing is currently selected, or if the item selected is not an OLE item, or if more than one OLE item is selected, the menu item reads "Object . . ." and is disabled.
  2. If an OLE item is selected, the caption on the menu item changes. If the selected item has an unnamed verb, the menu item becomes "*typename* Object."
  3. If the selected item has only one verb, the menu item becomes "*verb typename* Object."
  4. If the selected item has more than one verb, the menu item becomes "*typename* Object." Choosing the command opens a submenu listing all the supported verbs.

- The Edit Links command.

  The Edit Links command appears on the Edit menu if the client application accepts linked items. This command lets the user modify properties of the linked items in the document. When the user chooses the Edit Links command, the Links dialog box appears. This dialog box displays a list of all the linked items in the document (the information displayed for each linked item includes the type of the item, the document in which the item is stored, and the name that identifies the item within the document). The user can select one or more of the linked items and change the type of link updating (automatic or manual), update linked items immediately, cancel links, or repair broken links.

- Cutting/copying an item to the Clipboard.

The basic steps in writing a client application are described in more detail in the following sections.

Defining a Client Document Class

A client document class must be able to do everything a non-OLE document class does; it must interact with views and be able to handle commands. However, it must also be designed to accommodate objects of a **COleClientItem**-derived class as well as its own data.

▶ **To define a client document class**

- Derive your document class from **COleClientDoc** instead of **CDocument**.
- Override the **Serialize** member function to enable the document to be stored to disk.

  If you define **CDocItem**-derived classes to represent your non-OLE data, you can use the **COleDocument** list to store both your OLE and non-OLE data. Otherwise, you must manage your non-OLE data using your own data structures, as you would when implementing documents for a non-OLE application.

When a client document is created, the framework automatically registers the document with the OLE system DLL. This allows the DLL to identify the client documents.

Defining a Client Item Class

As mentioned earlier, you do not have to write code that manages the specific contents of embedded or linked items, such as the floating-point numbers in an embedded worksheet item; the framework and the OLE system DLLs manage that for you. However, you may want to define member variables in your item class that store the item's size and position within the compound document.

▶ **To define a client item class**

- Derive a class from **COleClientItem**.
- Implement the **OnChange** member function.

  The **OnChange** function is called by the framework when the user of the server application modifies the item or saves or closes the document containing the item. You must implement this function to respond to changes in the item's state. Typically you update the item's appearance by invalidating the area in which the item is displayed.

The Insert New Object Command

If you choose the OLE Client option, AppWizard adds the Insert New Object command to the Edit menu and gives it the **ID_OLE_INSERT_NEW** command ID. AppWizard also defines a member function in your view class named `OnInsertObject`, which acts as the message handler for this command.

AppWizard provides a skeletal definition of `OnInsertObject`; it declares a **CString** object and passes it to the global function **AfxOleInsertDialog**. This function displays the Insert Object dialog box and returns the type name selected by the user in the **CString** object. You must complete the implementation of `OnInsertObject` by creating an item of the specified type and insert it in your document.

▶ **To complete the implementation**

- Dynamically allocate a new instance of your **COleClientItem**-derived class; this creates the C++ object, but not the OLE item itself. Pass a pointer to the client document as the argument to the base class constructor—that is, the **COleClientItem** constructor. This constructor adds the item to the specified document.
- Create the OLE item itself by calling the **CreateNewObject** member function, passing the name of the server specified by the user and a name that uniquely identifies the item being created.

  This function launches the specified server application that allows the user to create a new item. In a server application written with the Microsoft Foundation Class Library, the **OnCreateDoc** member function of the server class is called at this point. For more information, see the section "Defining a Server Class" later in this chapter.

  When **CreateNewObject** returns, the item is not displayed because it doesn't contain anything yet. When the user updates the item, the framework calls the **OnChange** function of the new item; the client application responds by displaying the item. See the section "Displaying an Item" later in this chapter for information on what you should do in your override of **OnChange**. If the user chooses not to update the item when exiting the server, no item is embedded.

For details on implementing the Insert New Object command, see the OCLIENT sample program.

The Paste and Paste Link Commands

AppWizard provides the Paste command on the Edit menu and gives it the **ID_EDIT_PASTE** command ID. If you choose the OLE Client option when you run AppWizard, it also adds the Paste Link command to the Edit menu and gives it the **ID_EDIT_PASTE_LINK** command ID.

The Microsoft Foundation Class Library defines update handlers for the Paste or Paste Link commands (these are member functions of **COleClientDoc**). These functions enable or disable the Paste and Paste Link commands, respectively, based on the current contents of the Clipboard. The Paste command is enabled if either an embedded or a static item can be pasted. The Paste Link command is enabled if a linked item can be pasted.

▶ **To complete the implementation**

- Define a message handler in your view class. The message handler for the Paste command should be named `OnPaste`, and the one for the Paste Link command should be named `OnPasteLink`. Do the following steps in each message handler:
  1. Open the clipboard by calling the **CWnd::OpenClipboard** member function.
  2. Create an instance of your **COleClientItem**-derived class:
     - For the Paste command, call the **CreateFromClipboard** member function.

       If the function call is unsuccessful, call the **CreateStaticFromClipboard** member function. This function creates a static item; such an item does not contain the Native data used by the server, only the presentation data for displaying the item. As a result, a static item can be displayed but not edited.
     - For the Paste Link command, call the **CreateLinkFromClipboard** member function.
  3. Insert the new item in your document at the current position. The specifics depend on your implementation of compound documents.
  4. Close the clipboard by calling the **CWnd::CloseClipboard** member function.

See the OCLIENT sample program for details on implementing the Paste and Paste Link commands.

Invoking a Verb on an Item

If you want to invoke an item's primary verb when the user double-clicks the item, override the **OnLButtonDblClick** message handler of the view class that displays the item. In your override of the function, call the **DoVerb** member function defined by **COleClientItem**. Even if the selected item supports more than one verb, you should use the primary verb for consistency with other OLE applications. To specify the primary verb, pass **OLEVERB_PRIMARY** to **DoVerb**.

To implement the *typename* Object command if you choose the OLE Client option, AppWizard adds a command to the Edit menu with **ID_OLE_VERB_FIRST** as its

command ID. The Microsoft Foundation Class Library defines both the update handler and the message handler for this command (these are member functions of **COleClientDoc**). The update handler enables/disables the menu item and adjusts its caption depending on what type of OLE item, if any, is currently selected. The **OnCmdMsg** member function of **COleClientDoc** handles the command by calling the **DoVerb** member function of the selected item.

▶ **To complete the implementation of the *typename* Object command**

- Implement some form of item selection. The details of this will depend on your application.
- Override the **IsSelected** member function in your view class; this is needed for the menu command to be enabled properly.

When the user invokes a verb on an item, the server opens the item and then executes the verb. In a server application written with the Microsoft Foundation Class Library, a member function of the item class is called to execute the verb; the particular member function depends on the verb. See the section "Defining a Server Item Class" later in this chapter for information.

See the OCLIENT sample program for details on invoking verbs on embedded or linked items.

The Edit Links Command

If you choose the OLE Client option, AppWizard adds the Edit Links command to the Edit menu and gives it the **ID_OLE_EDIT_LINKS** command ID. The Microsoft Foundation Class Library defines both the update handler and the message handler for this command (these are member functions of **COleClientDoc**). The update handler checks whether there are any linked items in the document and enables or disables the menu item accordingly. The message handler calls the global **AfxOleLinksDialog** function, which displays the Links dialog box. No further work is required.

Displaying an Embedded or Linked Item

As described in the section "Defining a Client Item Class," you are responsible for managing the size and position of embedded and linked items. Use this information when displaying items.

▶ **To display an item**

- Override **OnDraw** in your view class.

- For each item, call the **Draw** member function defined by **COleClientItem**. This function renders an image of the item. If the item's server was written with the Microsoft Foundation Class Library, the function plays the metafile created by the **OnDraw** member function of **COleServerItem**. See the section "Defining a Server Item Class" later in this chapter for information.

To ensure that the items are displayed correctly when the server updates them, override the **OnChange** member function in your **COleClientItem**-derived class. This function is called in response to actions performed by the user while working in the server application. From this function, call the **UpdateAllViews** member function of the document. As described in Chapter 8, this function notifies all the views attached to the document to update their displays. The default behavior for each view is to invalidate their entire display area; this causes the **OnDraw** member function of the view to be called, redisplaying each item.

See the OCLIENT sample program for details on displaying embedded or linked items. The OCLIENT program allows resizing of items and optimizes drawing so that a change in one item does not force all items to be redrawn.

Cutting or Copying Items to the Clipboard

▶ **To implement the Cut or Copy commands**

- Implement some form of item selection. The details of this will depend on your application.
- Override the **IsSelected** member function in your view class; this is needed for the menu commands to be enabled properly.
- Define update handlers for the Cut and Copy commands in your view class. Enable those commands on the menu depending on whether anything in the document is selected.
- Define message handlers for the Cut and Copy commands in your view class. In the message handler, do the following:
  - Open and empty the Clipboard.
  - Call the **CopyToClipboard** member function defined by **COleClientItem** for the currently selected item.
  - Close the clipboard.
  - For the Cut operation, destroy the item.

If your application supports selection of more than one item at a time, or selection of non-OLE data with OLE data, you must define additional Clipboard formats and implement the Cut and Copy commands so they copy the multiple selections to the Clipboard.

Loading/Saving a Compound Document

As described in previous chapters, the Microsoft Foundation Class Library provides a framework for creating, loading, and saving documents through the **CDocument** interface. Because **COleClientDoc** is derived indirectly from **CDocument**, client documents can be managed by the framework in the same way as ordinary documents; **COleClientDoc** overrides certain functions to perform the additional work needed for OLE client documents.

For example, when loading a client document, a client application must register the document with the OLE system DLL. **COleClientDoc** takes care of this by overriding **OnNewDocument** and **OnOpenDocument** to call the **RegisterClientDoc** member function. Similarly, a client application must notify the library when a document is saved; to do this, **COleClientDoc** overrides **OnSaveDocument** to call the **NotifySaved** member function.

▶ **To complete the implementation**

- Override **Serialize** in your **COleClientDoc**-derived class. To save embedded or linked items, use the interface provided by **COleDocument** to iterate through the list of items contained in the document and call **Serialize** for each one. If you are using your own data structures as well, iterate through them and serialize their contents.
- Override **Serialize** in your **COleClientItem**-derived class. Serialize any information you store in your class, such as the position and size of the item. Then call the version of **Serialize** defined by **COleClientItem**; this serializes the data stored in the item itself. That is, for an embedded item, it stores the Native data and the type of the item, and for a linked item, it stores the type of the item and information that specifies where the item can be found.

Implementing a Server Application

Support server functionality requires only minor changes to an application's user interface: when an embedded item is being edited, the Save command on the File menu is replaced by the Update command and the Save As command is replaced by the Save Copy As. This indicates that storing changes made to the document does not involve saving anything to disk, but instead involves updating the item stored in a compound document. The framework updates the menu captions automatically.

Defining a Server Class

The class you use to implement your server depends on the type of server application you want to write.

Mini-Servers vs. Full Servers

There are two types of server applications: mini-servers and full servers. A mini-server can be launched only by a client. The MS-Draw and Graph servers are examples of mini-servers. A mini-server does not store documents as their own files on disk; instead, it reads its documents from and writes them to items in client documents. As a result, such an application can support only embedding, not linking.

A full server is a stand-alone application; it can either be run by itself or be launched by a client application. A full server supports storing documents on disk as their own files. Such an application can support embedding only, or it can support both embedding and linking. A full server typically supports the Cut and Copy commands; these allow the user of a client application to create an embedded item by choosing the Paste command or, in the case of the Copy command, to create a linked item by choosing the Paste Link command.

▶ **To define a class for a mini-server**

- Derive a class from **COleServer**.
- Implement the **OnCreateDoc** member function. This is a pure virtual function, so you must provide an implementation.

  The **OnCreateDoc** function is called when the user of a client application chooses the Insert New Object command. Return a document object from this function; this must be an object of your **COleServerDoc**-derived class.
- Implement the **OnEditDoc** member function. This is a pure virtual function, so you must provide an implementation.

  The **OnEditDoc** function is called when the user of a client application edits an existing embedded item. Return a document object from this function; this must be an object of your **COleServerDoc**-derived class.

If you are writing a full server, you should use the class **COleTemplateServer**. **COleTemplateServer** is derived from **COleServer** and can be used directly. It provides definitions for the pure virtual functions defined by **COleServer** by using the document-creation facilities of a **CDocTemplate** object. This lets your server application take advantage of the document/view architecture provided by the Microsoft Foundation Class Library.

▶ **To use COleTemplateServer as a server class**

- Create a **CDocTemplate** object, specifying a **COleServerDoc**-derived class as the document class. Typically you pass this to the application's **AddDocTemplate** member function.

- Create a **COleTemplateServer** object; you can declare this as a member variable in your **CWinApp**-derived class. From within your application's **InitInstance** member function, call the **RunEmbedded** member function of the **COleTemplateServer** object, passing the **CDocTemplate** object.

For an example of a mini-server, see the MINSVR sample program. For an example of a full server, see the HIERSVR sample program.

SDI Servers vs. MDI Servers

Another issue related to the mini- versus full-server question is whether your application should be a single document interface (SDI) or a multiple document interface (MDI) application. Mini-servers are always SDI applications. Full servers can be either, though they are typically MDI applications.

A server application must be able to support multiple clients simultaneously, in case more than one client wants to edit an embedded or linked item. If the server is an SDI application, it must be possible for multiple instances of the server to be running simultaneously. This allows a separate instance of the application to handle each client's requests.

If the server is an MDI application, it can simply create a new MDI child window each time a client needs to edit an item. In this way, a single instance of the application can support multiple clients.

Your server application must specify whether or not it supports multiple instances when it is first launched. This tells the OLE system DLL what to do if one instance of the server is already running when another client requests its services: whether it should launch a new instance of the server, or whether it should direct all clients' requests to one instance of the server.

The following table summarizes some characteristics of mini- and full servers:

Type of server	SDI/MDI	Number of items per document	Supports multiple instances	Number of documents per instance
Mini-server	SDI	1	Yes	1
Full server that supports embedding only	MDI	1	No	0 or more
Full server that supports embedding and linking	MDI	0 or more	No	0 or more

See the section "Launching a Server Application" later in this chapter for more information.

Defining a Server Document Class

In a server document, there is not the same distinction between OLE items and the application's own data as there is in a client document. This is because an OLE item consists of data created by the server application itself. As a result, the design of your server document class and your server item class are interdependent.

The relationship between the document and item classes depends on whether your server supports embedding only, or linking as well. If your server supports only embedding, a server item is equivalent to a server document. Thus you will need only one object of your item class for each document.

If your server supports linking, designing your document and item classes is potentially more difficult. From the server's point of view, a linked item can be anything that can be selected. There are two basic types of selection: single selection, in which only one element in the document can be selected at a time, and multiple selection, in which an arbitrary number of elements in the document can be selected at a time.

The easiest way to support linking in a server is to allow only single selection. With this approach, you can use **COleServerItem** as the base class for an element in the document. Your document class can use the **COleDocument** list to store your document items. Whenever an element is copied to the Clipboard, you can simply call the **CopyToClipboard** member function for that item. The HIERSVR sample program allows only single selection.

Supporting multiple selection is much more difficult. You must design data structures that allow a **COleServerItem** object to represent an arbitrary number of the elements that make up a document. A full discussion of implementing multiple selection is beyond the scope of this chapter.

For more information, see the section "Defining a Server Item Class" later in this chapter.

▶ **To define a server document class**

- Derive your document class from **COleServerDoc** instead of **CDocument**.
- Implement the **OnGetEmbeddedItem** member function. This is a pure virtual function, so you must provide an implementation.

  OnGetEmbeddedItem is called when the user of a client application creates or edits an embedded item. Return an item representing the entire document. This should be an object of your **COleServerItem**-derived class.

- If you are implementing a full server, override the **Serialize** member function. If you are implementing a mini-server, the documents are never written to disk, so the **Serialize** member function is unnecessary.

The **OnGetLinkedItem** member function of **COleServerDoc** is called when the user of a client application edits a linked item. A default implementation of this function is provided. You need to override it only if you are implementing your own data structures for storing server items instead of using the **COleDocument** list.

When a server document is created, the framework automatically registers the document with the OLE system DLL. This allows the DLL to identify the server documents.

Defining a Server Item Class

Most of the behavior of your server application lies in the item class. The item class contains the Native data that makes up an item; the item class also defines the appearance of the items and the actions taken when verbs are invoked.

▶ **To define a server item class**

- Derive a class from **COleServerItem**.
- Implement the **OnDraw** member function. This is a pure virtual function, so you must provide an implementation.

  The **OnDraw** function draws an image of the item to a metafile, allowing it to be displayed when a client application opens a compound document.
- Implement the **Serialize** member function. This is a pure virtual function, so you must provide an implementation.

  The **Serialize** function reads and writes the Native data for an item, that is, all the information needed to describe it. This allows an embedded item to be transferred between the server and client applications.
- (Optional) Override the **OnShow** member function.

  The **OnShow** function is called when an item is opened for editing. Override the function to scroll the view so that the item is visible and then select the item.
- (Optional) If the type of item supports more than one verb, override the **OnExtraVerb** member function to perform the desired action for each supported verb.

  To inform clients of the verbs that your server supports for a particular type of item, provide entries for those verbs in the Windows registration database. For more information, see the following section, "Registering a Server."

- (Optional) If you want to support **CF_TEXT** as a presentation format, override the **OnGetTextData** member function. A presentation format is a format in which the server can provide an image of an item. By default, servers written with the Microsoft Foundation Class Library provide images of items only as metafiles (that is, the only presentation format supported is **CF_METAFILEPICT**). If you want to provide a text representation of an item, override **OnGetTextData** to do so.

  If you want to support other formats, override the **OnEnumFormats** member function to return the formats and override the **OnGetData** member function to provide the data in these formats.

Registering a Server Application

Every server application installed on the system must have an entry in the Windows registration database indicating that it supports OLE. The registration database is a binary file named REG.DAT located in the Windows directory.

A server application's entry in the registration database contains several pieces of information: a name identifying the type of item that the application supports, the command line to execute to launch the application, the verbs that it supports, and whether an object-handler DLL exists for the application.

The framework and OLE system DLLs use this database to determine what servers are available on the system. For example, when the user of a client application executes the Insert New Object command, the list of servers that appears in the dialog box is constructed by querying the registration database. The OLE system DLLs also use this database to determine how to launch the server application when a linked or embedded object is activated.

You should register your server when it is first installed. You can also have the server update its registration every time it is executed as a stand-alone program. This keeps the registration database up-to-date if the server's executable file is moved.

If you want to register your application during installation, use the REGLOAD utility included with the Windows Software Development Kit. If you include a setup program with your application, you should have the setup program run REGLOAD. Otherwise, instruct the user to run REGLOAD with the name of your server application.

To use REGLOAD, write a text file with the extension .REG for your server. Here is a sample .REG file:

```
REGEDIT
HKEY_CLASSES_ROOT\MySvr = My OLE Server
HKEY_CLASSES_ROOT\MySvr\protocol\StdFileEditing\server =
        C:\MYSERVER\MYSERVER.EXE
HKEY_CLASSES_ROOT\MySvr\protocol\StdFileEditing\verb\0 = Edit
HKEY_CLASSES_ROOT\MySvr\protocol\StdFileEditing\verb\1 = Play
HKEY_CLASSES_ROOT\.MYS = MySvr
```

The .REG file contains the following information:

- An internal name for the item type. For example, "MySvr" is the internal name in the sample .REG file above. This name is for internal use only; try to pick a name that avoids collisions. This name should not contain spaces.
- A user-visible name for the item type. For example, "My OLE Server" is the human-readable name in the sample .REG file above. This is the name that appears in the Insert Object dialog box, as well as in various other places. This name can contain spaces.
- The path name of the server's executable file.
- (Optional) A standard suffix for document files edited by the server. For example, ".MYS" is the suffix in the sample .REG file above.
- All the verbs supported by the item type. The primary verb has the index 0, the secondary verb has the index 1, and so forth. If your server supports multiple verbs, you must specify those verbs here.

REGLOAD merges the contents of the .REG text file into the registration database. You can also use the REGEDIT program to view or modify the contents of the registration database.

If you want to register your server application each time it is executed as a stand-alone program, call the global function **AfxOleRegisterServerName** from the **InitInstance** member function of your **CWinApp**-derived class. **AfxOleRegisterServerName** takes two parameters: the internal and user-visible names for the type of item that the server supports. This function updates the path for the server's executable file. If there are no verbs registered for the item type, the function also adds Edit as the default primary verb.

Launching a Server Application

When a server application is launched by a client application, the OLE system DLL adds the "/Embedding" option to the server's command line. A server application's behavior differs depending on whether or not it was launched by a client, so the first thing an application should do when it begins execution is check for the "/Embedding" or "-Embedding" option on the command line.

If you are writing a mini-server, it will always be launched by a client, by definition. You should still parse the command line to check for the "/Embedding"

option. The lack of a "/Embedding" option on the command line indicates that the user tried to launch the mini-server as a stand-alone application. In this case, you should register the server with the Windows registration database and then display a message box informing the user to launch the application from a client application.

Before running the server, you must perform instance registration. This operation does not add an entry to the registration database; instead, it informs the OLE system DLL that the server is active and ready to receive requests from clients. Register the server by calling the **Register** member function defined by **COleServer**. The **Register** function takes two parameters: the server's internal type name and a flag indicating whether the server supports multiple instances. A mini-server must be able to support multiple instances; that is, it must be possible for multiple instances of the server to run simultaneously, one for each client. Consequently, you should pass **TRUE** for this flag when launching a mini-server.

If you are writing a full server using **COleTemplateServer**, call the **RunEmbedded** to parse the command line. This function updates the server's entry in the Windows registration database and calls the **Register** member function for you, performing instance registration.

When you call **RunEmbedded**, you must pass a flag indicating whether the server supports multiple instances. A full server typically does not support multiple instances, because it is typically an MDI application; a single instance of the server can handle all clients. Consequently, you should pass **FALSE** for this flag when launching a full server.

If a server is launched by a client to edit a linked item, the OLE system DLL passes a filename on the command line following the "/Embedding" option; this is the name of the document that is the source of the linked item. The server should open this document immediately.

Sequences of OLE Function Calls

This section outlines the major steps that occur during common OLE operations and lists the member functions called in each step. For more information on the OLE classes and their member functions, see the descriptions in the *Microsoft Class Library Reference*.

Inserting a New Embedded Item

The following list describes the scenario where the user chooses the Insert New Object command.

1. Your client code calls **COleClientItem::CreateNewObject**.

2. The OLE system DLL searches the Windows registration database to find the executable file for the server, and launches the server with the "/Embedding" option on the command line.
3. **COleClientItem::CreateNewObject** returns at approximately this time, but the item is blank.
4. If the server is a mini-server, your server code calls **COleServer::Register**. If it's a full server, **COleTemplateServer** calls it for you.
5. **COleServer::OnCreateDoc** is called; it returns a document object.
6. **COleServerDoc::OnGetEmbeddedItem** is called; it returns an item.
7. **COleServerItem::OnShow** is called; it displays the item, allowing the user to edit it.
8. The user chooses the Update command. The framework calls **COleServerDoc::NotifySaved**.
9. **COleServerItem::OnGetData** is called to get the item's Native data. This function calls **COleServerItem::Serialize** to write the data.
10. **COleServerItem::OnGetData** is called to get the item's metafile representation. This function calls **COleServerItem::OnDraw** to draw an image of the item to a metafile.
11. **COleClientItem::OnChange** is called, notifying the client that the item was updated. At this point the item can be displayed.
12. The user chooses the Exit command in the server application. If the server is a mini-server, your server code calls **COleServer::BeginRevoke**. If it's a full server, **COleTemplateServer** calls it for you (if there are no other documents open).
13. **COleClientItem::OnChange** is called, notifying the client that the server has closed the document.

Editing an Embedded Item

The following list describes the scenario where the user invokes the Edit verb on an embedded item, either by double-clicking the item or by using the *typename* Object command.

1. Your client code calls **COleClientItem::DoVerb**, which in turn calls **COleClientItem::Activate**.
2. The OLE system DLL searches the Windows registration database to find the executable file for the server, and launches the server with the "/Embedding" option on the command line.
3. If the server is a mini-server, your server code calls **COleServer::Register**. If it's a full server, **COleTemplateServer** calls it for you.
4. **COleServer::OnEditDoc** is called; it returns a document object.

5. **COleServerDoc::OnGetEmbeddedItem** is called; it returns an object of the item class.
6. **COleServerItem::OnSetData** is called to set the item's Native data; this function calls **COleServerItem::Serialize** to read the data.
7. **COleServerItem::OnShow** is called; it displays the item, allowing the user to edit it.
8. **COleClientItem::DoVerb** returns at approximately this time.
9. Steps 8-13 of the Insert New Object procedure occur.

Editing a Linked Item

The following list describes the scenario where the user invokes the Edit verb on a linked item, either by double-clicking the item or by using the *typename* Object command.

1. Your client code calls **COleClientItem::DoVerb**, which in turn calls **COleClientItem::Activate**.
2. The OLE system DLL searches the Windows registration database to find the executable file for the server, and launches the server with the "/Embedding *filename*" option on the command line.
3. **COleTemplateServer** calls **COleServer::Register** for you.
4. **COleServer::OnOpenDoc** is called, loading a file from disk; the function returns a document object.
5. **COleServerDoc::OnGetLinkedItem** is called; it returns an object of the item class.
6. **COleServerItem::OnShow** is called; it displays the item, allowing the user to edit it.
7. **COleClientItem::DoVerb** returns at approximately this time.
8. The user chooses the Save command. The framework saves the document and calls **COleServer::NotifySaved**.
9. **COleServerItem::OnGetData** is called to get the item's metafile representation. This function calls **COleServerItem::OnDraw** to draw an image of the item to a metafile.
10. **COleClientItem::OnChange** is called, notifying the client that the item was saved.
11. The user chooses the Exit command in the server application. **COleTemplateServer** calls **COleServer::BeginRevoke** for you (if there are no other documents open).
12. **COleClientItem::OnChange** is called, notifying the client that the server has closed the document.

APPENDIX A

Getting Started

This appendix provides an overview of the new and changed functions and features in the current version of the Microsoft Foundation Class Library. It also provides information on installing the class library and locating its components.

What's New in the Class Library

Among the new features in version 2 of the Microsoft Foundation Class Library are the following:

- The class library is now integrated with the visual tools in the Visual C++ programming environment: AppWizard, App Studio, ClassWizard, and Visual Workbench.
- The class library gives you a head start with the skeleton starter application created by AppWizard.
- Managing Windows messages and commands from menus, buttons, and accelerator keys is greatly simplified when you use ClassWizard.
- App Studio provides one-stop user-interface design and resource editing.
- Visual Workbench lets you compile and debug your application and browse your class hierarchy.
- At the heart of the class library is its document/view program structure. Documents manage your data. Views manage user interaction with a document.
- The class library implements many standard user-interface elements:
  - standard menu items, such as the New, Open, Save, and Save As commands on the File menu
  - support for toolbars, status bars, and other control bars
  - support for scrolling and splitter windows
  - form-style user interfaces based on dialog templates
  - support for printing and print preview

  - support for VBX (Visual Basic) and other custom controls
  - support for context-sensitive help
- You get a dialog-box interface that makes it easy to initialize and validate dialog-box data and to retrieve data from a dialog box.
- Object Linking and Embedding (OLE) support is integrated with the document/view architecture, and the framework supports OLE user interfaces.
- Complete support for dynamic link libraries (DLLs). This support is more extensive than the support in the previous version.

These and other features make the Microsoft Foundation Class Library more powerful and even easier to use. The current version of the class library is also compatible with the previous version.

Installing the Class Library

For general information and installation instructions, see Chapter 1 in the *Visual Workbench User's Guide.*

The Setup program for Microsoft Visual C++ includes an option to install the Microsoft Foundation Class Library. If you choose that option, you have all of the files and directories you need to compile your programs and most of the sample programs that Setup installs, including prebuilt libraries for standard memory models. If you need other versions of the libraries for other memory models, you'll need to build them yourself. For more information about building libraries, see "Building Libraries" in Appendix B.

On the main screen in the Setup program during installation, be sure the check boxes labeled "Microsoft Foundation Classes" and "Microsoft App Studio: Resource Editor" are selected (they are selected by default). You can also install these components at a later time. For information on installing Visual C++ components after your initial installation, see "Reinstalling Visual C++" in the *Visual Workbench User's Guide.*

Additional Files

In addition to header files and libraries, the MFC directory and its subdirectories created when you install the Microsoft Foundation Class Library contain source code, sample programs, and technical notes.

Source Code

The full source code for the Microsoft Foundation Class Library is supplied on your distribution disks in the MFC\SRC subdirectory. You'll find the source code useful in a number of ways:

- As an aid in debugging your own code when the failure is revealed, say, with an ASSERT inside a class in the class library.
- As the ultimate reference if the documents don't answer all of your questions.
- In very rare cases, you may want to modify the library class implementations, although this is generally not recommended.

For information on building new library versions, see Appendix B, "Versions of the Microsoft Foundation Class Library."

Sample Programs

Sample programs are provided in the MFC\SAMPLES subdirectory. You'll find it instructive to compile and run them and to examine their source code for useful techniques. To install the samples, see "Samples" in Chapter 1 of the *Visual Workbench User's Guide*.

For a helpful overview of the sample programs and what they illustrate, see MFCSAMP.HLP, which you can reach through the main contents screen of MFC.HLP.

Technical Notes

Technical notes are provided to discuss porting issues and the architecture and advanced features of the Microsoft Foundation Class Library.

For easy access to the technical notes, see MFCNOTES.HLP, which you can reach through the main contents screen of MFC.HLP.

APPENDIX B

Versions of the Microsoft Foundation Class Library

This appendix provides information on building different versions of the Microsoft Foundation Class Library.

Prebuilt Libraries

The Visual C++ Setup program automatically provides prebuilt libraries needed for the sample programs. You'll have to build other versions manually as you need them. Table B.1 shows which libraries are installed, already built, by Setup.

Table B.1 Class Library Support for Windows Memory Models

Memory Model	Prebuilt Debug	Prebuilt Release	Supported
Small	No	No	Yes
Medium	Yes	Yes	Yes
Compact	No	No	Yes
Large	Yes	Yes	Yes
Huge	No	No	No
AFXDLL (Large)	Yes	Yes	Yes
USRDLL (Large)	No	No	Yes

A "yes" under "Prebuilt Debug" or "Prebuilt Release" indicates that the prebuilt library is supplied with Microsoft Visual C++. A "yes" under "Supported" means the Microsoft Foundation Class Library supports the model. If a library is not supplied but is supported, you can build it yourself. For information on building libraries, see "How to Build Other Library Versions" on page 303. Memory models are explained in *Programming Techniques*, Chapter 2.

By default, AppWizard uses medium model and defaults to a debug build.

▶ **To switch to a release build**

1. In Visual Workbench, choose the Options command on the Project menu.
2. In the Project Options dialog box, select the option button for Debug.

Library Naming Conventions

Object-code libraries for the Microsoft Foundation Class Library use the following naming conventions. The library names have the form

*m*AFX*cwd*.LIB

where the letters shown in italic lowercase are placeholders for specifiers with the meanings shown in Table B.2.

Table B.2 Library Names

Specifier	Values and Meanings
m	Memory model:
	S = Small, M = Medium, C = Compact, L = Large
c	Type of program to create:
	C = EXE, D = USRDLL
w	Target:
	W = Windows, R = Real mode MS-DOS
d	Debug or Release:
	D = Debug. Omit specifier for release.

DLL Libraries

The Microsoft Foundation Class Library provides complete support for Windows dynamic link libraries (DLLs). You can build applications that use a shared DLL version of the Microsoft Foundation Class Library (called AFXDLL). Or you can use the classes in the Microsoft Foundation Class Library in a Windows DLL that can be used with applications not built with the class library (called USRDLL).

AFXDLL libraries are supplied. You must build your own USRDLL libraries.

The AFXDLL form of the library is named MFC200.LIB or MFC200D.LIB (for debug). These versions require the run-time DLLs MFC200.DLL or MFC200D.DLL (for debug) to run.

For more information on using AFXDLL in Microsoft Foundation Class Library applications, see Technical Note 33. For information on using the USRDLL version, see Technical Note 11. Both technical notes are accessible through MFCNOTES.HLP, which you can reach through MFC.HLP.

How to Build Other Library Versions

A standard MAKEFILE is supplied in MFC\SRC for building new libraries.

To build a new library version, use NMAKE. For information about using NMAKE, see the Tools TechNote Viewer. (To open the Tools TechNote Viewer, click its icon in the Microsoft Visual C++ program group in Windows.)

Table B.3 shows the commands for building debug and release versions of the libraries for all of the standard memory models.

Table B.3 Commands for Building Library Versions

Library	Command to Build the Library
SAFXCWD.LIB	NMAKE MODEL = S TARGET = W DEBUG = 1
SAFXCW.LIB	NMAKE MODEL = S TARGET = W DEBUG = 0
MAFXCWD.LIB	NMAKE MODEL = M TARGET = W DEBUG = 1
MAFXCW.LIB	NMAKE MODEL = M TARGET = W DEBUG = 0
CAFXCWD.LIB	NMAKE MODEL = C TARGET = W DEBUG = 1
CAFXCW.LIB	NMAKE MODEL = C TARGET = W DEBUG = 0
LAFXCWD.LIB	NMAKE MODEL = L TARGET = W DEBUG = 1
LAFXCW.LIB	NMAKE MODEL = L TARGET = W DEBUG = 0

For the naming conventions used for library files, see "Library Naming Conventions" on page 302. Changing the TARGET option to R in an NMAKE command (including those shown in this appendix) builds an MS-DOS version.

Other library options include DLL and CODEVIEW, as explained in the next two sections.

Building DLLs

To build a Windows dynamic link library (LAFXDWD.LIB), add a specification for DLL to your command line. The default value, DLL = 0, builds an .EXE. Specifying the command

```
NMAKE MODEL = L TARGET = W DEBUG = 1 DLL = 1
```

builds a debug version of USRDLL. For information about USRDLL, see "Prebuilt Libraries" on page 301. The value of MODEL must be L for a DLL.

Building Programs with CodeView Information

To build a version of your application with debugging information that the Microsoft CodeView® debugger can use, add a CODEVIEW specification to your command line. CODEVIEW = 2 is the default for debug builds. CODEVIEW = 0 is the default for release builds. The value CODEVIEW = 0 builds with no CodeView information. Specifying the command

```
NMAKE MODEL = M TARGET = W DEBUG = 1 CODEVIEW = 1
```

builds a debug version of your program (medium model, in this example) with full CodeView information. Specifying CODEVIEW = 2 builds a version with minimal CodeView information.

Index

A

B

C

D

E

F

G

H

N

O

P

Q

R

S

T

U

V

W

Programming Techniques →

← Class Library User's Guide

Programming Techniques

Contents

Figures and Tables

Figures

Tables

Tables (Continued)

Introduction

Programming Techniques describes how to take advantage of the special features of Microsoft® Visual C++™. The topics covered by this manual include language extensions, special-purpose library functions, and the interaction between programming strategies and compiler options.

Scope of This Book

Programming Techniques is divided into two parts. Part 1, "Improving Program Performance," helps you write more efficient programs. Chapter 1 provides specific information about precompiled header files—when, why, and how to use them to reduce compilation time during development. Chapters 2 and 3 explain memory management options for C and C++, respectively, and when to use each option. Chapter 4 describes the inline assembler, a feature that makes it possible to mix assembly language with your C and C++ source code, and Chapter 5 describes the floating-point math packages.

Part 2, "Special Environments," covers techniques specific to certain programming situations. Chapter 6 discusses issues related to using the CL compiler command-line driver and the run-time libraries when writing programs for the Microsoft Windows™ operating system. Chapter 7 describes the QuickWin library, using which you can convert MS-DOS programs with simple input and output requirements into programs that have a user interface similar to that of Windows. Chapter 8 shows how to program in mixed languages, and Chapter 9 offers tips to make your programs more portable.

Note For information on Microsoft product support, see the chapter "Microsoft Support Services" in the *Visual Workbench User's Guide*.

Document Conventions

This book uses the following typographic conventions:

Example	Description
STDIO.H	Uppercase letters indicate filenames, segment names, registers, and terms used at the operating-system command level.

Example	Description
char, _setcolor, __far	Bold type indicates C and C++ keywords, operators, language-specific characters, and library routines. Within discussions of syntax, bold type indicates that the text must be entered exactly as shown.
	Many functions and constants begin with either a single or double underscore. These are part of the name and are mandatory. For example, to have the **__cplusplus** manifest constant be recognized by the compiler, you must enter the leading double underscore.
expression	Words in italics indicate placeholders for information you must supply, such as a filename. Italic type is also used occasionally for emphasis in the text.
[[*option*]]	Items inside double square brackets are optional.
#pragma pack {1 | 2}	Braces and a vertical bar indicate a choice among two or more items. You must choose one of these items unless double square brackets ([[ ]]) surround the braces.
`#include <io.h>`	This font is used for examples, user input, program output, and error messages in text.
CL [[*option...*]] *file...*	Three dots (an ellipsis) following an item indicate that more items having the same form may appear.

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

CTRL+ENTER	Small capital letters are used to indicate the names of keys on the keyboard. When you see a plus sign (+) between two key names, you should hold down the first key while pressing the second.
	The carriage-return key, sometimes marked as a bent arrow on the keyboard, is called ENTER.
"argument"	Quotation marks enclose a new term the first time it is defined in text.
`"C string"`	Some C constructs, such as strings, require quotation marks. Quotation marks required by the language have the form " " and ' ' rather than “ ” and ‘ ’.
Color Graphics Adapter (CGA)	The first time an acronym is used, it is usually spelled out.

PART 1

Improving Program Performance

CHAPTER 1

Using Precompiled Headers

The Microsoft C and C++ compilers provide options for precompiling any C or C++ code—including inline code. Using this performance feature, you can compile a stable body of code, store the compiled state of the code (including information from the Microsoft CodeView® debugger) in a file, and during subsequent compilations combine the precompiled code with code that is still under development. Each subsequent compilation is faster by the time not required to recompile the stable code. Use of the fast compile option (/f), the compiler's default action, further speeds compilation.

With Microsoft Visual C++, you can precompile any C or C++ code; you are not limited to precompiling only header files. To precompile headers, you can choose one of two approaches:

- Automatic precompiling. Simply use one option (/YX) and let the compiler decide when to create and use precompiled headers.
- Manual precompiling. Use this when you know that your source files use common sets of header files but do not include them in the same order, or when you want to include source code in your precompilation.

The first choice is quick and easy. The second requires some planning, but it offers significantly faster compilations if you precompile source code other than simple header files.

1.1 When to Precompile Source Code

Precompiled code is useful during the development cycle to reduce compilation time, especially if:

- You always use a large body of code that changes infrequently.
- Your program comprises multiple modules, all of which use a standard set of include files and the same compilation options. In this case, all include files can be precompiled into one precompiled header.

The first compilation—the one that creates the precompiled header file—takes a bit longer than subsequent compilations. Subsequent compilations can proceed more quickly by including the precompiled code.

Precompilation works for C and C++ programs.

You can precompile both C and C++ programs. In C++ programming, it is common practice to separate class interface information into header files. These header files can later be included in programs that use the class. By precompiling these headers, you can reduce the time a program takes to compile.

Note Although you can use only one precompiled header (.PCH) file per source file, you can use multiple .PCH files in a project.

1.2 Creating and Using Precompiled Headers

Either way you can precompile code—automatically or manually—stores the resulting precompiled code in a file called a precompiled header.

The easiest way to precompile code is to simply use the automatic precompiled-header option (/YX) on the command line. The /YX option causes the compiler to either create a precompiled header with a default name of MSVC.PCH or use a precompiled header named MSVC.PCH if one exists. You can also control the name of the precompiled header that is created or used with the /Fp ("specify precompiled header filename") option.

The other way to precompile code is to use the manual precompiled-header options /Yc ("create precompiled header") and /Yu ("use precompiled header"). Use the /Yc option to create a precompiled header. When used with the optional **hdrstop** pragma, /Yc lets you precompile both header files and source code. You can also use the /Fp option with the /Yc and /Yu options to provide an alternative name for the precompiled header.

The following sections describe the precompiled header options and the **hdrstop** pragma in more detail. Section 1.4, page 16, describes a method for using precompiled header files in a project; it includes an example makefile and the code that it manages. For further examples using precompiled headers, see the makefiles used to build the program examples that ship with the Microsoft Foundation Class Library.

1.3 Precompiled Header Compiler Options

The compiler options described in the following sections control the creation and use of precompiled headers. The **hdrstop** pragma, described in "The hdrstop Pragma" on page 10, gives you extra control over the behavior of these options.

The /YX (Automate Precompiled Header) Option

The /YX ("automate precompiled header") option instructs the compiler to use a precompiled header file with a default name of MSVC.PCH if it exists or to create one if it does not. The file is created in the current directory. You can use the /Fp*filename* option to change the default name (and placement) of the precompiled header. The following command line uses /YX to create a precompiled header named MSVC.PCH:

```
CL /YX MYPROG.CPP
```

The following command line creates a precompiled header named MYPROG.PCH and places it in the \PROJPCH directory:

```
CL /YX /Fp\PROJPCH\MYPROG.PCH MYPROG.CPP
```

Using the /YX option limits precompilation to header files only. The precompiled header is created when the compiler encounters the first declaration, definition, **hdrstop** pragma, or **#line** directive that occurs in the source file. In a subsequent compilation, the precompiled header is used at the point in a source file where the compiler makes its final consistency check. For more information, see "Consistency Rules for /YX," page 6.

The actual set of header files precompiled with /YX is determined by the compiler, which may use a subset of the header files available to make the resulting precompiled header useful in more cases.

Although it is usually best to let the compiler determine which header files to use, you can selectively precompile header files by placing a **hdrstop** pragma in the source file between two **#include** directives. All header files before the **hdrstop** pragma are precompiled; those after are not. When used with /YX, any filename specified with the **hdrstop** pragma is ignored. If any subsequent compilation using the precompiled header does not find an identical **hdrstop** pragma at the same point in the source file, the compiler builds a new precompiled header. For more information, see "The hdrstop Pragma," page 10.

Note If either /Yc or /Yu is specified with /YX, a warning is issued. In such cases, /YX is ignored, and /Yc or /Yu takes precedence.

Use of the /YX option implies the /Yd ("duplicate debugging information in all object files") option.

Debugging Code Compiled with /YX

You can use either the /Zi or the /Z7 option with the /YX option to generate debugging information for CodeView or the Visual Workbench integrated debugger.

It is recommended that you use the /Zi option unless you need /Z7 to maintain compatibility with Microsoft C/C++ version 7.0. The /Zi option places the generated debugging information into a program database (PDB); a PDB speeds linking during the debugging process. The /Yd option, which places all debugging information into every object file created using a precompiled header, is ignored unless used with /Z7.

Table 1.1 summarizes results obtained from using the /Zi, /Z7, and /Yd options in combination with /YX.

Table 1.1 Results of Combining Debugging Options with /YX

Option combination	Result
/Zi	Debugging information for both the precompiled header and the rest of the source code is placed in a program database (PDB) with the default name of MSVC.PDB.
/Zi /Yd	/Yd is ignored. Debugging information for both the precompiled header and the rest of the source code is placed in a PDB with the default name of MSVC.PDB.
/Z7	Debugging information for both the precompiled header and the rest of the source code is placed in every object file created using the precompiled header.
/Z7 /Yd	Debugging information for both the precompiled header and the rest of the source code is placed in every object file created using the precompiled header.

For more information on PDBs, /Zi, and /Z7, see Chapter 1, "CL Command Reference," in *Command-Line Utilities User's Guide*; this chapter discusses the Visual C++ compiler, a file called CL.EXE. For information on /Yd, see the "The /Yd (Duplicate Debugging Information in All Object Files) Option," page 14.

Consistency Rules for /YX

If a precompiled header file exists (either MSVC.PCH or one specified by /Fp*filename*), it is compared to the current compilation for consistency. Unless the following requirements are met, a new precompiled header file is created and the new file overwrites the old:

- The current compiler options must match those specified when the precompiled header was created.
- The current working directory must match that specified when the precompiled header was created.

- The order and values of all **#include** and **#pragma** directives must match those specified when the precompiled header was created. These, along with **#define** directives, are checked one by one as they appear during subsequent compilations that use the precompiled header. The values of **#define** directives must match. However, a sequence of **#define** directives need not occur in exactly the same order because there are no semantic order dependencies for **#define** directives. The **#pragma** directives must be nearly identical, with a few exceptions; for example, multiple spaces outside of strings are treated as a single space to allow for different programming styles.
- The value and order of include paths specified on the command line with /I options must match those specified when the precompiled header was created.
- The time stamps of all the header files (all files specified with **#include** directives) used to build the precompiled header must match those that existed when the precompiled header was created.

The /Yc (Create Precompiled Header) Option

The /Yc ("create precompiled header") option instructs the compiler to create a precompiled header file that represents the state of compilation at a certain point. The syntax of this option is:

/Yc[[*filename*]]

Using /Yc with a Filename

If you specify a filename with the /Yc option, the compiler precompiles all code up to and including the specified file. The precompiled code is saved in a file with a name created from the base name of the file specified with the /Yc option and a .PCH extension. You can also use the /Fp option, described in "The /Fp (Specify Precompiled Header Filename) Option," page 15, to specify a name for the resulting precompiled header file.

Consider the following code:

```
#include <afxwin.h>   // Include header for class library
#include "resource.h" // Include resource definitions
#include "myapp.h"    // Include information specific to this
                      //   application
...
```

When compiled with the command

```
CL /YcMYAPP.H PROG.CPP
```

the compiler saves all the results of processing AFXWIN.H, RESOURCE.H, and MYAPP.H in a precompiled header file called MYAPP.PCH.

Using /Yc Without a Filename

If you specify the /Yc option with no filename, the resulting precompiled header saves the compilation state at the end of the base source file or, if the base file contains a **hdrstop** pragma, at the place where the **hdrstop** pragma occurs.

The resulting .PCH file has the same base name as your base source file unless you specify a different filename using the **hdrstop** pragma or the /Fp option.

Note If /Yc*filename* and /Yu*filename* options occur on the same command line and both reference the same *filename*, /Yc*filename* takes precedence, precompiling all code up to and including the named file. This feature simplifies the writing of makefiles.

The /Yu (Use Precompiled Header) Option

The /Yu ("use precompiled header") option instructs the compiler to use an existing precompiled header in the existing compilation. The syntax of this option is:

/Yu[[*filename*]]

Using /Yu with a Filename

The *filename* argument is the name of a header file, which is included in the source file using an **#include** preprocessor directive. The include file's name must be the same for both the /Yc option that creates the precompiled header and any subsequent /Yu option indicating use of the precompiled header.

For /Yc, *filename* specifies the point at which precompilation stops; the compiler precompiles all code though *filename* and names the resulting precompiled header using the base name of the include file and an extension of .PCH. For /Yu, the compiler assumes that all code occurring before *filename* is precompiled. The compiler skips to the specified **#include** directive, uses the code contained in the precompiled header file, and then compiles all code after *filename*.

Consider the following code:

```
#include <afxwin.h>    // Include header for class library
#include "resource.h" // Include resource definitions
#include "myapp.h"     // Include information specific to this
                       //   application
...
```

When compiled with the command line

```
CL /YuMYAPP.H PROG.CPP
```

the compiler does not process the three **#include** statements but uses the precompiled code from the precompiled header MYAPP.PCH, thereby saving the time involved in preprocessing all three of the files (and any files they might include).

Using /Yu Without a Filename

When you specify the /Yu option without a filename, your source program must contain a **hdrstop** pragma. The compiler skips to the location of that pragma and uses the content of the precompiled header file specified by the pragma. If the **hdrstop** pragma does not specify a filename, the name is derived from the base name of the source file with the .PCH extension. You can also use the /Fp option to specify a different .PCH file.

If you specify the /Yu option without a filename and fail to specify a **hdrstop** pragma, an error message is generated and the compilation is unsuccessful.

Note If /Yc*filename* and /Yu*filename* option occur on the same command line and both reference the same *filename*, /Yc*filename* takes precedence, precompiling all code up to and including the named file. This feature simplifies the writing of makefiles.

Debugging Code Compiled with /Yc or /Yu

You can use either the /Zi or the /Z7 option with the /Yc and /Yu options to generate debugging information compatible with CodeView or the debugger that is integrated with the Visual Workbench.

It is recommended that you use the /Zi option unless you need /Z7 to maintain compatibility with Microsoft C/C++ version 7.0. The /Zi option places the generated debugging information into a program database (PDB); a PDB speeds linking during the debugging process.

The /Yd option, which places all debugging information into every object file created using a precompiled header, is ignored unless used with /Z7.

Table 1.2 summarizes results obtained from using the /Zi, /Z7, and /Yd options in combination with /Yc and /Yu.

Table 1.2 Results of Combining Debugging Options with /Yc or /Yu

Option combination	Result
/Zi	Debugging information for both the precompiled header and the rest of the source code is placed in a program database (PDB) with the default name of MSVC.PDB.
/Zi /Yd	/Yd is ignored. Debugging information for both the precompiled header and the rest of the source code is placed in a PDB with the default name of MSVC.PDB.
/Z7	Debugging information for both the precompiled header and the rest of the source code is placed in the first object file created using the precompiled header.
/Z7 /Yd	Debugging information for both the precompiled header and the rest of the source code is placed in every object file created using the precompiled header.

For more information on PDBs, /Zi, and /Z7, see Chapter 1, "CL Command Reference," in *Command-Line Utilities User's Guide*. For information on /Yd, see "The Yd (Duplicating Debugging Information in All Object Files) Option," page 14.

The hdrstop Pragma

The **hdrstop** pragma gives you additional control over precompilation filenames and over the place at which the compilation state is saved. The syntax of the **hdrstop** pragma is

#pragma hdrstop [[("*filename*")]]

where *filename* is the name of the precompiled header file to use or create (depending on whether /Yu or /Yc is specified). If the filename does not contain a path specification, the precompiled header file is assumed to be in the current directory. Any *filename* is ignored when the automatic precompiled header option (/YX) is specified.

Note The **hdrstop** pragma is ignored unless either the /YX option is specified or the /Yu or /Yc compiler option is specified without a filename.

For the /Yc and the /Yu options, if neither of the two compilation options nor the **hdrstop** pragma specifies a filename, the base name of the source file is used as the base name of the precompiled header file. This differs from the precompiled-header naming conventions of /YX; with /YX, the default name of the precompiled header is MSVC.PCH.

The filename specified in the **hdrstop** pragma is a string and is therefore subject to the constraints of any C or C++ string. In particular, you must escape backslashes (\) when specifying paths. For example:

```
#pragma hdrstop( "c:\\c700\\include\\myinc.pch" )
```

You can also use preprocessing commands to perform macro replacement as follows:

```
#define INCLUDE_PATH "c:\\c700\\include\\"
#define PCH_FNAME "PROG.PCH"
.
.
.
#pragma hdrstop( INCLUDE_PATH PCH_FNAME )
```

Placement of the hdrstop Pragma

The following rules govern where the **hdrstop** pragma can be placed:

- It must appear outside any data or function declaration or definition.
- It must be specified in the base file, not within a header file.

Consider the following example:

```
#include <windows.h>                    // Include several files
#include "myhdr.h"

__inline Disp( char *szToDisplay )      // Define an inline function
{
    ...                                 // Some code to display string
}
#pragma hdrstop
```

Use the manual precompilation options to precompile any code.

In this example, the **hdrstop** pragma appears after two files have been included and an inline function has been defined. This might, at first, seem to be an odd placement for the pragma. Consider, however, that using the manual precompilation options, /Yc and /Yu, with the **hdrstop** pragma makes it possible for you to precompile entire source files—even inline code. The Microsoft compiler does not limit you to precompiling only data declarations.

Consistency Rules for /Yc and /Yu

When you use a precompiled header created using the /Yc option, the compiler compares the current compilation environment to the one that existed when you created the .PCH file. You should take care to specify a environment consistent with the previous one (using consistent compiler options, pragmas, and so on) for the current compilation. If the compiler detects an inconsistency, it issues a warning

and identifies the inconsistency where possible. Such warnings don't necessarily indicate a problem with the .PCH file; they simply warn you of possible conflicts. The following sections explain the consistency requirements for precompiled headers.

Compiler Option Consistency

Table 1.3 lists compiler options that might trigger an inconsistency warning when using a precompiled header.

Table 1.3 Compilation Option Consistency

Option	Name	Rule
/A*X* or /A*xxx*	Specify memory model	Must be the same between the compilation that created the precompiled header and the current compilation. If these options differ, a error message results.
/D	Define constants and macros	Must be the same between the compilation that created the precompiled header and the current compilation. The state of defined constants is not checked, but unpredictable results can occur if your files depend on the values of the changed constants.
/E or /EP	Copy preprocessor output to standard output	Precompiled headers do not work with the /E or /EP option.
/Fr or /FR	Generate Microsoft Source Browser information	For the /Fr and /FR options to be valid with the /Yu option, they must also have been in effect when the precompiled header was created. Subsequent compilations that use the precompiled header also generate Source Browser information. Browser information is placed in a single .SBR file and is referenced by other files in the same manner as CodeView information. You cannot override the placement of Source Browser information.
/GA, /GD, /GE, /Gw, or /GW	Windows protocol options	Must be the same between the compilation that created the precompiled header and the current compilation. If these options differ, a warning message results.

Table 1.3 Compilation Option Consistency *(continued)*

Option	Name	Rule
/Zi	Generate complete debugging information	If this option is in effect when the precompiled header is created, subsequent compilations that use the precompilation can use that debugging information. If /Zi is not in effect when the precompiled header is created, subsequent compilations that use the precompilation and the /Zi option trigger a warning. The debugging information is placed in the current object file, and local symbols defined in the precompiled header are not available to the debugger.

Note The precompiled header facility is not intended for use with a file that is not a C or C++ program.

Include Path Consistency

A precompiled header created with /Yc does not contain information about the include path that was in effect when you created the .PCH file. When you use a .PCH file, the compiler always uses the include path specified in the current compilation.

Source File Consistency

When you use a precompiled header, the compiler ignores all preprocessor directives (including pragmas) that appear before the **hdrstop** pragma. The compilation specified by such preprocessor directives must be the same as the compilation used to create the precompiled header file.

Pragma Consistency

Pragmas processed during the compilation of a precompiled header usually affect the file in which the precompiled header is subsequently used. The following pragmas affect only the code within the .PCH file; they do not affect code that subsequently uses the .PCH file:

comment	**message**	**pagesize**	**subtitle**
linesize	**page**	**skip**	**title**

The following pragmas are retained as part of a precompiled header. They do affect the remainder of a compilation that uses the precompiled header:

alloc_text	**code_seg**	**inline_recursion**	**optimize**
auto_inline	**data_seg**	**intrinsic**	**pack**
check_pointer	**function**	**loop_opt**	**same_seg**
check_stack	**inline_depth**	**native_caller**	**warning**

The /Yd (Duplicate Debugging Information in All Object Files) Option

Use the /Yd ("duplicate debugging information in all object files") option with the /Z7 ("generate Microsoft C/C++ version 7.0 compatible CodeView information") option to place complete debugging information in all object files that are ultimately created from the resulting precompiled header (.PCH) file. The /Yd option takes no argument.

Note The /Yd option is obsolete with the advent of the new program database (PDB) files created by the "generate complete debugging information" (/Zi) option. The /Yd option is retained on /Zi, /Zd, and /Z7 for backward compatibility with Microsoft C/C++ version 7.0. For more information, see Chapter 1, "CL Command Reference," in *Command-Line Utilities User's Guide*.

Storing complete debugging information in every object file, including the information that describes only the .PCH file, is sometimes convenient, but it slows compilation and requires considerable disk space.

In contrast, when /Yc and /Z7 are used without /Yd, the compiler stores debugging information describing the .PCH file in the first object file created from that file. The compiler does not insert this common debugging information into object files subsequently created from the .PCH file; rather, it inserts cross-references to the debugging information. Therefore, no matter how many object files use the .PCH file, only one object file contains the common debugging information.

Although this default behavior results in faster build times and reduces disk-space demands, it is undesirable if a small change requires rebuilding the object file containing the common debugging information. In this case, the compiler must rebuild all object files containing cross-references to the original object file. Also, it is difficult to rely on cross-references to a single object file if a common .PCH file is used by different projects.

Examples

Suppose you have two base files, F.CPP and G.CPP, each containing these **#include** statements:

```
#include "windows.h"
#include "etc.h"
```

The following command creates the precompiled header file ETC.PCH and the object file F.OBJ:

```
CL /YcETC.H /Z7 F.CPP
```

The object file F.OBJ includes type and symbol information for WINDOWS.H and ETC.H (and any other header files they include). Now you can use the precompiled header ETC.PCH to compile the source file G.CPP:

```
CL /YuETC.H /Z7 G.CPP
```

The object file G.OBJ does not include the debugging information for the precompiled header but simply references that information in the F.OBJ file. Note that you must link with the F.OBJ.

If your precompiled header was not compiled with /Z7, you can still use it in later compilations using /Z7. However, the debugging information is placed in the current object file, and local symbols for functions and types defined in the precompiled header are not available to the debugger.

The /Fp (Specify Precompiled Header Filename) Option

The /Fp option gives you extra control over the name of the precompiled header (.PCH) file. Use it to specify a precompiled header filename that is different from the default. For example, use the following command to rename the default MSVC.PCH file created and used by the "automate precompiled header" option:

```
CL /YX /FpMYPCH.PCH PROG.CPP
```

This command causes the compiler either to use a precompiled header named `MYPCH.PCH` if it exists or to create the file if it does not exist.

If you want to create a precompiled header file for a debugging version of your program and you are compiling both header files and source code, you can specify a command such as:

```
CL /DDEBUG /Zi /Yc /FpDPROG.PCH PROG.CPP
```

This command assumes the existence of a **hdrstop** pragma in `PROG.CPP` and creates a precompilation of all code up to the **hdrstop** pragma. The precompiled

code is stored in a file called DPROG.PCH. If you need a release version in parallel, you simply change the compilation command to:

```
CL /Yc /FpRPROG.PCH PROG.CPP
```

This command creates a separate precompilation of all code up to the **hdrstop** pragma and stores it in RPROG.PCH.

You can also use the /Fp option with the /Yu and /YX options.

1.4 Using Precompiled Headers in a Project

Previous sections in this chapter present an overview of precompiled headers and reference material describing the manual precompiled-header options, /Yc and /Yu, the /Fp option, and the **hdrstop** pragma. This section describes a method for using the manual precompiled-header options in a project; it ends with an example makefile and the code that it manages.

For another approach to using the manual precompiled-header options in a project, study the makefile located in the \MSVC\MFC\SRC directory that is created during the default setup of Visual C++. The makefile takes a similar approach to the one presented in this section but makes greater use of Microsoft Program Maintenance Utility (NMAKE) macros to offer greater control of the build process.

Note The techniques presented in this section do not apply when using the automate precompiled header option (/YX).

The Build Process

The code base of a software project is usually contained in multiple C or C++ source files, object files, libraries, and header files. Typically, a makefile coordinates the combination of these elements into an executable file. Figure 1.1 shows the structure of a makefile that uses a precompiled header file. Both the NMAKE macro names and the filenames in this diagram coincide with those in the example code found at the end of this section.

Figure 1.1 uses three diagrammatic devices to show the flow of the build process. Named rectangles represent each file or each macro; the three macros represent one or more files. Shaded areas represent each compile or link action. Arrows show which files and macros are combined during the compilation or linking process.

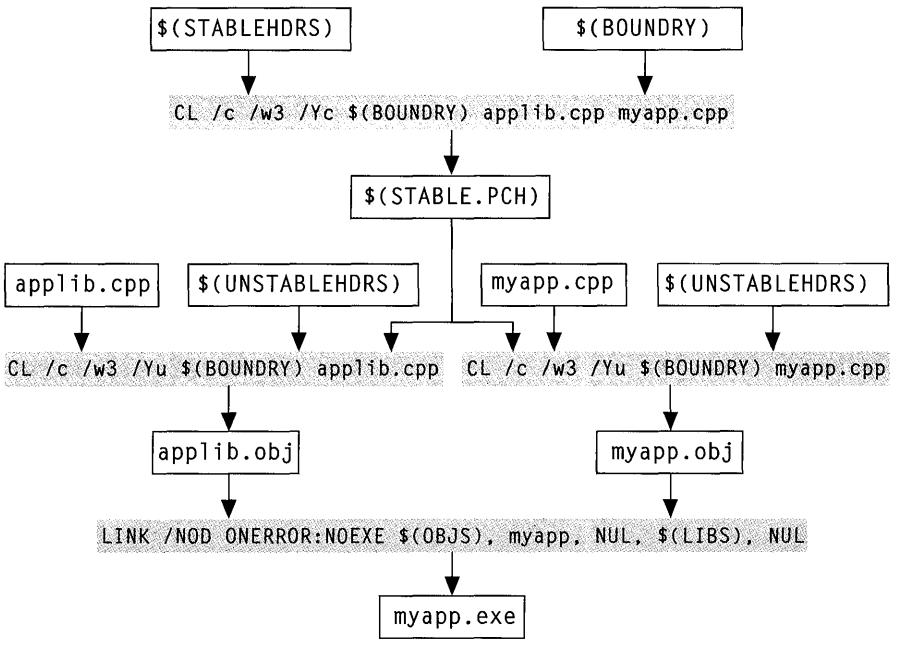


Figure 1.1 Structure of a Makefile That Uses a Precompiled Header File

List files not likely to change in the STABLEHDRS and BOUNDRY macros.

Beginning at the top of the diagram, both STABLEHDRS and BOUNDRY are NMAKE macros in which you list files not likely to need recompilation. These files are compiled using the command string

```
CL /c /W3 /Yc$(BOUNDRY) applib.cpp myapp.cpp
```

only if the precompiled header file (STABLE.PCH) does not exist or if you make changes to the files listed in the two macros. In either case, the precompiled header file will contain only code from the files listed in the STABLEHDRS macro. List the last file you want precompiled in the BOUNDRY macro.

The files you list in these macros can be either header files or C or C++ source files. (A single .PCH file cannot be used with both C and C++ modules.) Note that you can use the **hdrstop** macro to stop precompilation at some point within the BOUNDRY file. The **hdrstop** pragma is discussed in "The hrdstop Pragma," page 10.

List files likely to change in the UNSTABLEHDRS macro.

Continuing down the diagram, APPLIB.OBJ represents the support code used in your final application. It is created from APPLIB.CPP, the files listed in the UNSTABLEHDRS macro, and precompiled code from the precompiled header.

MYAPP.OBJ represents your final application. It is created from MYAPP.CPP, the files listed in the UNSTABLEHDRS macro, and precompiled code from the precompiled header.

Finally, the executable file (MYAPP.EXE) is created by linking the files listed in the OBJS macro (APPLIB.OBJ and MYAPP.OBJ).

Sample Makefile

The following makefile uses macros and an **!IF**, **!ELSE**, **!ENDIF** flow-of-control command structure to simplify its adaptation to your project.

```
# Makefile : Illustrates the effective use of precompiled
#            headers in a project
# Usage:     NMAKE option
# option:    DEBUG=[0|1]
#            (DEBUG not defined is equivalent to DEBUG=0)
#
OBJS = myapp.obj applib.obj
# List all stable header files in the STABLEHDRS macro.
STABLEHDRS = stable.h another.h
# List the final header file to be precompiled here:
BOUNDRY = stable.h
# List header files under development here:
UNSTABLEHDRS = unstable.h
# List all compiler options common to both debug and final
# versions of your code here:
CLFLAGS = /c /W3
# List all linker options common to both debug and final
# versions of your code here:
LINKFLAGS = /NOD /ONERROR:NOEXE
!IF "$(DEBUG)" == "1"
CLFLAGS   = /D_DEBUG $(CLFLAGS) /Od /Zi /f
LINKFLAGS = $(LINKFLAGS) /COD
LIBS      = slibce
!ELSE
CLFLAGS   = $(CLFLAGS) /Oselg /Gs
LINKFLAGS = $(LINKFLAGS)
LIBS      = slibce
!ENDIF
myapp.exe: $(OBJS)
    link $(LINKFLAGS) @<<
$(OBJS), myapp, NUL, $(LIBS), NUL;
<<
# Compile myapp
myapp.obj  : myapp.cpp $(UNSTABLEHDRS)  stable.pch
    $(CPP) $(CLFLAGS) /Yu$(BOUNDRY)    myapp.cpp
# Compile applib
applib.obj : applib.cpp $(UNSTABLEHDRS) stable.pch
    $(CPP) $(CLFLAGS) /Yu$(BOUNDRY)    applib.cpp
# Compile headers
stable.pch : $(STABLEHDRS)
    $(CPP) $(CLFLAGS) /Yc$(BOUNDRY)    applib.cpp myapp.cpp
```

List compiler and linker options in the CLFLAGS and LINKFLAGS macros.

Besides the STABLEHDRS, BOUNDRY, and UNSTABLEHDRS macros shown in Figure 1.1, this makefile also provides a CLFLAGS macro and a LINKFLAGS macro. You must use these macros to list compiler and linker options that apply whether you build a debug or final version of the application's executable file. There is also a LIBS macro where you list the libraries required by your project.

The makefile also uses **!IF**, **!ELSE**, **!ENDIF** to detect whether you define a DEBUG symbol on the NMAKE command line:

```
NMAKE DEBUG=[1|0]
```

This feature makes it possible for you to use the same makefile during development and for the final versions of your program—use DEBUG=0 for the final versions. The following command lines are equivalent:

```
NMAKE
NMAKE DEBUG=0
```

For more information on makefiles, see "Managing Projects with NMAKE" in the Tools TechNote Viewer. For more information on compiler options, see Chapter 1, "CL Command Reference," and Chapter 6, "Optimizing 16-Bit Programs," in *Command-Line Utilities User's Guide*. For more information on the Microsoft Segmented Executable Linker (LINK), see Chapter 2, "Linking Object Files with LINK," in *Command-Line Utilities User's Guide*.

The Example Code

The following examples are used in the makefile described previously. Note that the comments contain important information.

```
// ANOTHER.H : Contains the interface to code that is not
//             likely to change.
//
#ifndef __ANOTHER_H
#define __ANOTHER_H
#include<iostream.h>
void savemoretime( void );
#endif // __ANOTHER_H

// STABLE.H : Contains the interface to code that is not likely
//            to change. List code that is likely to change
//            in the makefile's UNSTABLEHDR macro.
//
#ifndef __STABLE_H
#define __STABLE_H
#include<iostream.h>
void savetime( void );
#endif // __STABLE_H
```

```
// UNSTABLE.H : Contains the interface to code that is
//              likely to change. As the code in a header
//              file becomes stable, remove the header file
//              from the makefile's UNSTABLEHDR macro and list
//              it in the STABLEHDR macro.
//
#ifndef __UNSTABLE_H
#define __UNSTABLE_H
#include<iostream.h>
void notstable( void );
#endif // __UNSTABLE_H

// APPLIB.CPP : This file contains the code that implements
//              the interface code declared in the header
//              files STABLE.H, ANOTHER.H, and UNSTABLE.H.
//
#include"another.h"
#include"stable.h"
#include"unstable.h"
// The following code represents code that is deemed stable and
// not likely to change. The associated interface code is
// precompiled. In this example, the header files STABLE.H and
// ANOTHER.H are precompiled.
void savetime( void )
    { cout << "Why recompile stable code?\n"; }
void savemoretime( void )
    { cout << "Why, indeed?\n\n"; }
// The following code represents code that is still under
// development. The associated header file is not precompiled.
void notstable( void )
    { cout << "Unstable code requires"
           << " frequent recompilation.\n"; }

// MYAPP.CPP : Sample application
//             All precompiled code other than the file listed
//             in the makefile's BOUNDRY macro (stable.h in
//             this example) must be included before the file
//             listed in the BOUNDRY macro. Unstable code must
//             be included after the precompiled code.
//
#include"another.h"
#include"stable.h"
#include"unstable.h"
void main( void )
{
    savetime();
    savemoretime();
    notstable();
}
```

CHAPTER 2

Managing Memory for 16-Bit C Programs

When you develop advanced 16-bit applications with Microsoft Visual C++, you must pay attention to memory management—that is, how data and code are stored and accessed in memory. A well-thought-out memory strategy makes your 16-bit programs run faster and occupy less memory.

You can follow one or more of these memory management strategies:

- Choose a standard memory model.
- Create a mixed-model program with the **__near**, **__far**, **__huge**, and **__based** keywords.
- Create your own customized memory model.
- Allocate memory as you need it with the **malloc** family of functions.
- Use virtual memory with the **_vmalloc** family of functions.

This chapter explains pointers, memory models (including tiny model), variations such as custom memory models and mixed models, based pointers, and virtual memory.

Most of the material covered in this chapter is relevant only to 16-bit programs. The only topic described in this chapter that applies to 32-bit programs is that of pointers based on a pointer.

The next chapter, "Managing Memory for 16-Bit C++ Programs," discusses memory management issues that are specific to C++.

2.1 Pointer Sizes

One of the strengths of the C language is that in it, you can use pointers to directly access memory locations.

Every Visual C++ program has at least two parts: the code (function definitions) and the data (variables and constants). As a program runs, it refers to elements of the code or the data by their addresses. These addresses can be stored in pointer

variables. Pointer variables can fit into 16 bits or 32 bits, depending on the distance of the object to which they refer.

Pointers and 64K Segments

IBM personal computers and compatibles use the Intel 8086, 80186, 80286, 80386, or 80486 processors (collectively called the 80*x*86 family). These processors have a "segmented" architecture, which means they all have a mode that treats memory as a series of segments, each of which occupies up to 64K of memory. An offset from the base of the segment makes it possible to access information within a given segment. Accessing more than one segment at a time requires additional machine code.

The 64K limit is necessary because the 80*x*86 registers are 16 bits (2 bytes) wide. A single register can address only 65,536 (64K) unique memory locations.

A 16-bit pointer can address up to 65,536 locations.

A pointer variable that fully specifies a memory address needs 16 bits for the segment location and another 16 bits for the offset within the segment, a total of 32 bits. However, if you have several variables in the same general area, your program can set the segment register once and treat the pointers as smaller 16-bit quantities.

The 80*x*86 register CS holds the base for the code segment; the register DS holds the base for the data segment. Two other segment registers are available: the stack segment register (SS) and the extra segment register (ES). (The 80386 and 80486 have additional segment registers: FS and GS.)

Near Pointers

If you don't explicitly specify a memory model, Visual C++ defaults to the small model, which allots up to 64K for the code and another 64K for the data (see Figure 2.1).

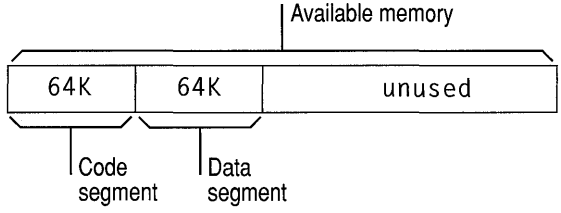


Figure 2.1 Anatomy of a Small-Model Program

When a small-model program runs, the CS and DS segment registers never change. All code pointers and all data pointers contain 16 bits because they remain within the 64K range.

These 16-bit pointers to objects within a single 64K segment are called "near pointers." Accessing a near object is called "near addressing."

Far Pointers

If your program needs more than 64K for code or data, at least some of the pointers must specify the memory segment, which means these pointers occupy 32 bits instead of 16 bits.

These larger 32-bit pointers that can point anywhere in memory are called "far pointers." Accessing a far object is called "far addressing."

Far pointers can address any location, but they are bigger and slower.

Far addressing has the advantage that your program can address any available memory location—up to 640K in MS-DOS®. The disadvantages of the larger far pointers are that they take up more memory (four bytes instead of two) and that any use of the pointers (assigning, modifying, or otherwise accessing values) takes more time.

Allowing either code or data to expand beyond 64K makes your programs larger and slower.

Huge Pointers

A third type of pointer in Visual C++ is the "huge" pointer, which applies only to data pointers. Code pointers cannot be declared as huge.

A huge address is similar to a far address in that both contain 32 bits, made up of a segment value and an offset value. They differ only in the way pointer arithmetic is performed.

For far pointers, Visual C++ assumes that code and data objects lie completely within the segment in which they start, so pointer arithmetic operates only on the offset portion of the address. Limiting the size of any single item to 64K makes pointer arithmetic faster.

Huge pointers overcome this size limitation; pointer arithmetic is performed on all 32 bits of the data item's address, thus allowing data items referenced by huge pointers to span more than one segment.

In the following example, both `hp` and `fp` are incremented:

```
int __huge *hp;
int __far *fp;
.
.
.
hp++;
fp++;
```

The huge pointer is incremented as a 32-bit value that represents the combined segment and offset. Only the offset part of the far pointer (a 16-bit value) is incremented.

Extending the size of pointer arithmetic from 16 to 32 bits causes such arithmetic to execute more slowly. You gain the use of larger arrays by paying a price in execution speed.

Based Addressing

When you declare near, far, and huge variables, the Microsoft compiler and linker automatically manage details such as allocating memory and keeping track of segments.

A "based pointer" is a fourth kind of pointer that operates as a 16-bit offset from a base that you specify. In this respect, based addressing differs from near, far, or huge addressing; you're responsible for naming the base, instead of letting the compiler decide. Based pointers are explained in more detail in Section 2.5, "Using Based Pointers and Data," on page 48.

2.2 Selecting a Standard Memory Model

If you want to choose one size for all pointers, there's no need to declare each variable as near or far. Instead, you select a standard memory model, and your size choice applies to all variables in the program.

A standard memory model provides default sizes for all pointers.

One advantage of using standard memory models is simplicity: You specify the way the compiler allocates storage for code and data only once. Another advantage is that the standard memory models do not require the use of Microsoft-specific keywords such as **__near** and **__far**, which makes them the best choice for writing code that is portable to other (non–MS-DOS) systems.

The disadvantage of standard memory models is that, because they make global assumptions about the environment, they may not provide the most efficient use of memory for a particular program.

The Six Standard Memory Models

The six Microsoft memory models are shown in Table 2.1.

Table 2.1 Memory Models

Model	Maximum memory for code	Maximum memory for data	Maximum memory for data arrays
Tiny	Less than 64K	Less than 64K	Less than 64K
Small	64K	64K	64K
Medium	No limit	64K	64K
Compact	64K	No limit	64K
Large	No limit	No limit	64K
Huge	No limit	No limit	No limit

The Setup program creates the libraries that support the six standard memory models.

When you choose one of the standard memory models, the compiler inserts the name of the corresponding C run-time library in the object file so the linker chooses it automatically. Each memory model has its own library, except for the huge memory model (which uses the large-model library) and the tiny model (which uses the small-model library).

Limitations on Code Size and Data Size

When writing a program with Visual C++, keep in mind two limitations that apply to all six memory models:

- No single source module can generate 64K or more of code. You must break large programs into modules and link their individual .OBJ files to create the .EXE file.
- No single data item can exceed 64K unless it appears in a huge-model program or it has been declared with the **__huge** keyword.

Tiny Memory Model

The tiny memory model resembles the small model with three exceptions:

- The tiny model cannot exceed 64K per program (including both code and data). A small-model program, on the other hand, can occupy up to 128K: 64K for code and 64K for data.
- The tiny model produces .COM, rather than .EXE, files. To produce .COM files, compile with the /AT option. Then link with the /TINY option and link in CRTCOM.LIB by adding it to the linker's *objfile* field.
- The tiny model applies to MS-DOS only; it is not available in Windows.

The tiny model produces the smallest programs but imposes the most severe limits on code and data size. As far as speed is concerned, the tiny model and small model produce equally fast programs, although the tiny model offers a load-time speed advantage over the small model.

If you are programming in C++, you cannot construct static objects under the tiny model. Because the predefined stream objects (like **cin** and **cout**) are static objects, you cannot use iostreams in a tiny-model program.

Figure 2.2 illustrates how memory is arranged for the tiny memory model.

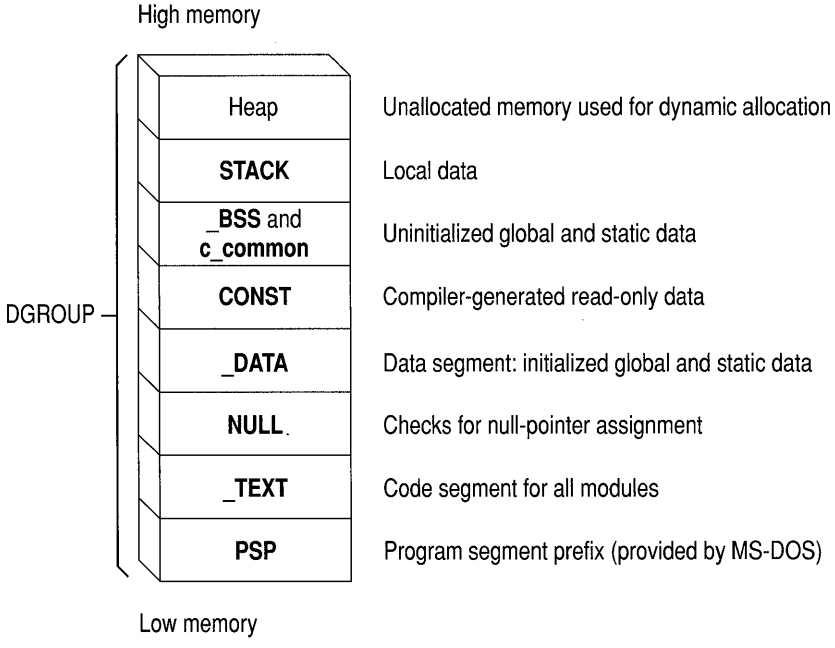


Figure 2.2 Memory Map for the Tiny Memory Model

Creating Small-Model Programs

The small model provides one code segment and one data segment, each limited to 64K. The total size of a small-model program can never exceed 128K. As most programs fit easily into this model, this is the default memory model.

By default, both code and data items in small-model programs are accessed with near addresses. This makes small-model programs faster than those using far addresses. You can override the defaults by using the **__far** or **__huge** keyword for data or by using the **__far** keyword for code.

Figure 2.3 illustrates how memory is arranged for the small memory model.

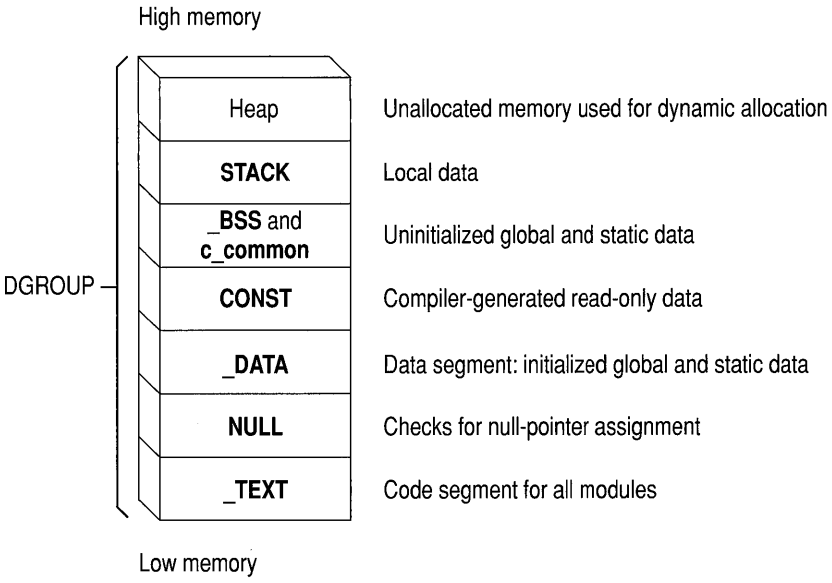


Figure 2.3 Memory Map for the Small Memory Model

Creating Medium-Model Programs

The medium memory model provides a single segment for program data and multiple segments for program code, each limited to 64K. Each source module is given its own code segment.

Medium-model programs typically have a large number of program statements (more than 64K of code) but a relatively small amount of data (less than 64K). Program code can occupy any amount of space and is given as many segments as needed; total program data cannot be greater than 64K.

By default, code items in medium-model programs are accessed with far addresses and data items are accessed with near addresses. You can override the default by using the **__far** or **__huge** keyword for data and the **__far** keyword for code.

The medium model provides a useful trade-off between speed and space for a program that refers more frequently to data items than to code. Figure 2.4 illustrates how memory is set up for the medium memory model.

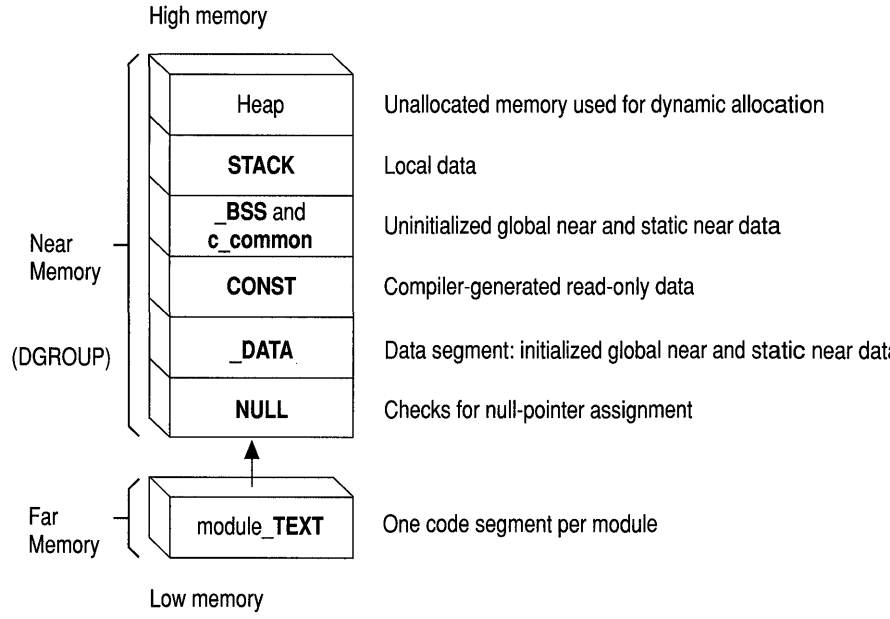


Figure 2.4 Memory Map for the Medium Memory Model

Creating Compact-Model Programs

The compact memory model provides multiple segments for program data but only one segment for the program code. Each segment is limited to 64K.

Compact-model programs typically have a large amount of data but a relatively small number of program statements. Program data can occupy any amount of space and is given as many segments as needed.

By default, code items in compact-model programs are accessed with near addresses and data items are accessed with far addresses. You can override the defaults by using the **__near** or **__huge** keyword for data or by using the **__far** keyword for code.

Figure 2.5 illustrates how memory is arranged for the compact memory model.

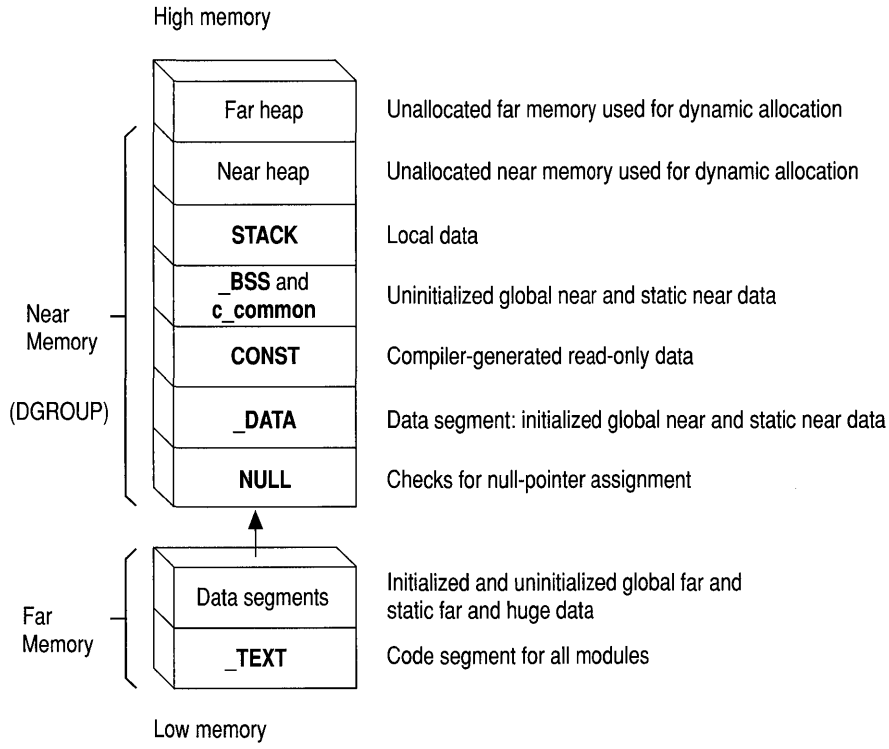


Figure 2.5 Memory Map for the Compact Memory Model

Creating Large-Model Programs

The large memory model provides multiple segments, as needed, for both code and data. Each segment is limited to 64K; no one data item can exceed 64K.

Large-model programs are typically very large and use a large amount of data storage during normal processing.

By default, both code and data items in large-model programs are accessed with far addresses. You can override the defaults by using the **__near** or **__huge** keyword for data or by using the **__near** keyword for code.

Figure 2.6 illustrates how memory is set up for the large and huge memory models.

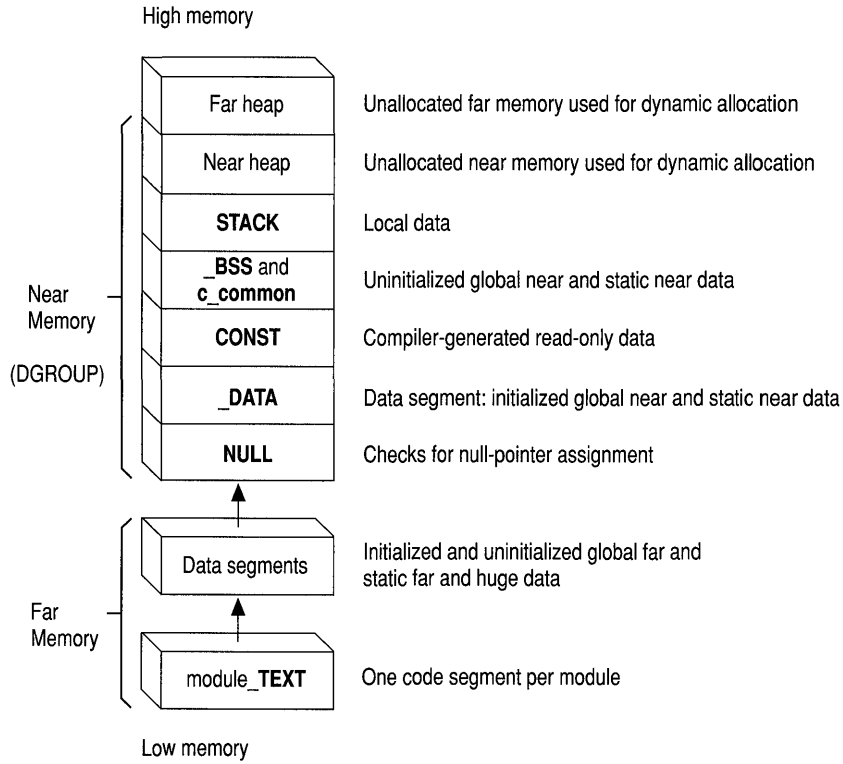


Figure 2.6 Memory Map for the Large and Huge Memory Models

Huge Memory Model

The huge memory model is nearly identical to the large model. The only difference is that the huge model permits individual arrays to exceed 64K in size. For example, an **int** uses two bytes, so an array of 40,000 integers, occupying 80,000 bytes of memory, is permitted in the huge model. All other models limit each array, structure, or other data object to no more than 64K.

Note Automatic arrays cannot be declared huge. Only static arrays and arrays occupying memory allocated by the **_halloc** function can be huge.

The huge model lifts the limits on arrays.

Although the huge model lifts the limits on arrays, some size restrictions do apply. To maintain efficient addressing, no individual array element is allowed to cross a segment boundary. This has the following implications:

- No single element of an array can be larger than 64K. An array can be larger than 64K, but its individual elements cannot.
- For any array larger than 128K, all elements must have a size in bytes equal to a power of 2: 2 bytes, 4 bytes, 8 bytes, 16 bytes, and so on. If the array is 128K or smaller, its elements can be any size, up to and including 64K.

Pointer arithmetic changes within the huge model, as well. In particular, the **sizeof** operator may return an incorrect value for huge arrays. The American National Standards Institute (ANSI) draft standard for C defines the value returned by **sizeof** to be of type **size_t** (which, in the Microsoft C compiler, is an **unsigned int**). The size in bytes of a huge array is an **unsigned long** value, however. To find the correct value, you must use a type cast:

```
(unsigned long)sizeof( monster_array )
```

Similarly, the C language defines the result of subtracting two pointers as **ptrdiff_t** (a **signed int** in the Microsoft C compiler). Subtracting two huge pointers yields a **long** value. The Microsoft C compiler gives the correct result with the following type cast:

```
(long)(ptr1__huge - ptr2_huge)
```

When you select huge model, all **extern** and uninitialized arrays are treated as **__huge**. Operations on data declared as **__huge** can be less efficient than the same operations on data declared as **__far**.

Null Pointers

Within the medium and compact models, code pointers and data pointers differ in size: A code pointer is 32 bits wide, and a data pointer is 16 bits wide. When using these memory models, you should be careful in your use of the manifest constant **NULL**.

NULL represents a null data pointer. The library include files define it as follows for C:

```
#define NULL ((void *) 0)
```

For C++, it is defined as follows:

```
#define NULL 0
```

There can be problems in models where code and data pointers are different sizes.

In memory models where data pointers have the same size as code pointers, the actual size of a null pointer doesn't matter. In memory models where code and data pointers are different sizes, problems can occur. Consider this example:

```
void main()
{
    func1( NULL );
    func2( NULL );
}

int func1( char *dp )
{
    .
    .
    .
}

int func2( char (*fp)( void ) )
{
    .
    .
    .
}
```

In the absence of function prototypes for `func1` and `func2`, the compiler always assumes that **NULL** refers to data and not code.

The previous example works correctly in tiny, small, large, and huge models because, in those models, a data pointer is the same size as a code pointer. In the medium or compact model, however, `main` passes **NULL** to `func2` as a null data pointer rather than as a null code pointer (a pointer to a function), which means the pointer is the wrong size.

To ensure that your code works properly in all models, declare each function with a prototype. For example, before `main`, include these two lines:

```
int func1( char *dp );
int func2( char (*fp)( void ));
```

If you add these prototypes to the example, the code works properly in all memory models. Prototypes force the compiler to coerce code pointers to the correct size. Prototypes also enable strong type checking of parameters.

Specifying a Memory Model

If you do not specify a memory model, Visual C++ defaults to the small model, which is adequate for many small to midsized programs.

You can choose a memory model by including an option on the command line. For example, to compile CLICK.C as a compact-model program, type this:

```
CL /AC CLICK.C
```

The /AC option selects the compact memory model. The six options and four libraries are as follows:

Option	Memory model selected	Library used
/AT	Tiny model	SLIBC*xx*.LIB (plus CRTCOM.LIB)
/AS	Small model	SLIBC*xx*.LIB
/AM	Medium model	MLIBC*xx*.LIB
/AC	Compact model	CLIBC*xx*.LIB
/AL	Large model	LLIBC*xx*.LIB
/AH	Huge model	LLIBC*xx*.LIB

2.3 Mixing Memory Models

In standard memory models, explained in the preceding section, all data pointers are the same size and all code pointers are the same size. A mixed memory model selectively combines different types of pointers within the same program. A mixed model extends the limits of a given memory model while retaining its benefits.

For example, imagine a programming situation in which you add an array to a small-model program, pushing the data segment past the 64K limit. You can solve the problem by moving up from the small to the compact memory model. Doing so bumps all data pointers from two to four bytes. The .EXE file grows accordingly. Execution time slows.

A mixed memory model lets you mix near and far pointers.

A second and perhaps better solution is to stay within the standard small memory model, which uses near pointers, but to declare the new array as far. You mix near pointers and far pointers, creating a mixed model.

Using Visual C++, you can override the standard addressing convention for a given memory model by specifying that certain items are **__near**, **__far**, **__huge**, or **__based**. These keywords are not a standard part of the C language; they are Microsoft extensions, meaningful only on systems that use 80*x*86 microprocessors. Using these keywords may affect the portability of your code.

Note Previous versions of the Microsoft C compiler accepted the keywords **near**, **far**, and **huge** without an initial underscore, in addition to with a single underscore. Because the ANSI standard for C permits compiler implementors to reserve keywords that begin with two underscores, all Microsoft-specific keywords have two initial underscores. To maintain compatibility with existing source code, the compiler still recognizes the obsolescent versions of these keywords.

You can compile a program in the small model, for example, but declare a certain array to be **__far**. At run time, the address of that array occupies four bytes; the program may slow slightly when accessing items in that particular far array. Throughout the rest of the program, all addressing is near. Note that all pointers to elements of an array declared as **__far** must also be declared as **__far**.

Table 2.2 lists the effects of these keywords on data pointers, code pointers, and pointer arithmetic.

Table 2.2 Addressing Declared with Microsoft Keywords

Keyword	Data	Code	Arithmetic
__near	Data resides in the default data segment; addresses are 16 bits.	Functions reside in the current code segment; addresses are 16 bits.	16 bits
__far	Data can be anywhere in memory, not necessarily in the default data segment; addresses are 32 bits.	Functions can be called from anywhere in memory; addresses are 32 bits.	16 bits
__huge	Data can be anywhere in memory, not necessarily in the default data segment. Individual data items (arrays) can exceed 64K in size; addresses are 32 bits.	Not applicable—code cannot be declared **__huge**.	32 bits (data only)
__based	Data can be anywhere in memory, not necessarily in the default data segment; 16-bit addresses plus a known base provide the range of 32-bit addresses.	Functions reside in specified code segment; **__based** can be used with **__near** or **__far**.	16 bits

Pointer Problems

When you declare items to be **__near**, **__far**, **__huge**, or **__based**, you can link with a standard run-time library. Be aware, however, that in some cases, the modified pointers are incompatible with standard library functions. Watch for these problems that affect pointers:

- A library function that expects a 16-bit pointer as an argument does not function properly with modified variables that occupy 32 bits. In other words, you can cast a near pointer to a far pointer, because it adds the segment value and maintains the integrity of the address. If you cast a far pointer to near, however, the compiler generates a warning message because the offset may not lie within the default data segment, in which case the original far address is irretrievably lost.
- A library function that returns a pointer returns a pointer of the default size for the memory model. This is only a problem if you are assigning the return value to a pointer of a smaller size. For example, there may be difficulties if you compile with a model that selects far data pointers, but you have explicitly declared the variable to receive the return value **__near**.

  This warning does not apply to all functions. Visual C++ provides model-independent versions of its string and memory functions such as **_fstrcat**, the far version of **strcat**.
- Based pointers pose a special problem. Based pointers are passed to other functions as is, without normalization. Certain functions expect to receive based pointers, but most do not. Therefore, in most cases, you must either explicitly cast a based pointer to a far pointer or make sure that all functions that receive based pointers are prototyped.

Some run-time library functions support near, far, huge, and based variables. For example, **_halloc** allocates memory for a huge data array.

You can always pass the value (but not the address) of a far item to a small-model library routine. For example:

```
/* Compile in small model */

#include <stdio.h>

long __far time_val;

void main()
{
    time( &time_val );              /* Illegal far address */
    printf( "%ld\n", time_val );    /* Legal value */
}
```

When you use a mixed memory model, you should include function prototypes with argument-type lists to ensure that all pointer arguments are passed to functions correctly.

Declaring Near, Far, Huge, and Based Variables

The **__near**, **__far**, **__huge**, and **__based** keywords can modify either objects or pointers to objects. When using them to declare variables, keep these rules in mind:

- The keyword always modifies the object or pointer immediately to its right. In complex declarations, think of the **__far** keyword and the item to its right as being a single unit.

  For example, in the case of the declaration

  ```
  char __far * __near *p;
  ```

  p is a near pointer to a far pointer to **char**, which resides in the default data segment for the memory model being used.

  By contrast, the declaration

  ```
  char __far * __near p;
  ```

  is a far pointer to **char** that is always stored in DGROUP, regardless of the memory model being used.

- If the item immediately to the right of the keyword is an identifier, the keyword determines whether the item is allocated in the default data segment (**__near**) or a separate data segment (**__far**, **__huge**, or **__based**). For example,

  ```
  char __far a;
  ```

  allocates a as an item of type **char** with a **__far** address.

- If the item immediately to the right of the keyword is a pointer, the keyword determines whether the pointer holds a near address (16 bits), a based address (16 bits), a far address (32 bits), or a huge address (also 32 bits). For example,

  ```
  char __huge *p;
  ```

  allocates p as a huge pointer (32 bits) to an item of type **char**. Any arithmetic performed on the huge pointer p affects all 32 bits. That is, the instruction p++ increments the pointer as a 32-bit entity.

Declaring Near and Far Functions

You cannot declare functions as **__huge**. The rules for using the **__near** and **__far** keywords for functions are similar to those for using them with data:

- The keyword always modifies the function or pointer immediately to its right.
- If the item immediately to the right of the keyword is a function, the keyword determines whether the function is called using a near (16-bit) or far (32-bit) address. For example,

```
char __far fun();
```

defines `fun` as a function with a 32-bit address that returns a **char**. The function may be located in near memory or far memory, but it is called with the full 32-bit address. The **__far** keyword applies to the function, not to the return type.

- If the item immediately to the right of the keyword is a pointer to a function, the keyword determines whether the function is called using a near (16-bit) or far (32-bit) address. For example,

```
char (__far *pfun)();
```

defines `pfun` as a far pointer (32 bits) to a function returning type **char**.

- Function declarations must match function definitions.
- The **__huge** keyword does not apply to functions. That is, a function cannot be huge (larger than 64K). A function can return a huge data pointer to the calling function.
- The **__based** keyword can be used to modify a function declaration, and it can be used in combination with the **__near** and **__far** keywords. Based functions are described in Section 2.6, "Using Based Addressing for Functions," on page 59. A function can return a based pointer unless it is a pointer based on **__self** (for more information, see Section 2.5, "Using Based Pointers and Data," on page 48).

The following example declares `fun1` as a far function returning type **char**:

```
char __far fun1(void);              /* Small model */
char __far fun(void)
{
    .
    .
    .
}
```

Here, the `fun2` function is a near function that returns a far pointer to type **char**:

```
char __far * __near fun2();         /* Large model */
char __far * __near fun()
{
    .
    .
    .
}
```

The following example declares `pfun` as a far pointer to a function that has an **int** return type, assigns the address of **printf** to `pfun`, and prints `Hello world` twice:

```
/* Compile in medium, large, or huge model */

#include <stdio.h>

int (__far *pfun)( char *, ... );

void main()
{
     pfun = printf;
     pfun( "Hello world\n" );
     (*pfun)( "Hello world\n" );
}
```

Pointer Conversions

Passing near or far pointers as arguments to functions can cause automatic conversions in the size of the pointer argument. Passing a pointer to an unprototyped function forces the pointer size to the larger of the following two sizes:

- The default pointer size for that type, as defined by the memory model selected during compilation. For example, in medium-model programs data pointer arguments are near by default and code pointer arguments are far by default.
- The size of the type of the argument.

Note that if you supply a based pointer as an argument to a function and do not specifically cast it to a far pointer type, a 16-bit offset from the base segment is passed.

Function prototypes prevent problems that may occur in mixed memory models.

If you provide a function prototype with complete argument types, the compiler performs type checking and enforces the conversion of actual arguments to the declared type of the corresponding formal argument. However, if no declaration is present or the argument-type list is empty, the compiler converts nonbased pointer arguments automatically to the default type or the type of the argument, whichever is larger. To avoid mismatched arguments, always use a prototype with the argument types.

For example, the following program produces unexpected results in compact-model, large-model, or huge-model programs:

```
void main()
{
    int __near *x;
    char __far *y;
    int z = 1;

    test_fun( x, y, z );     /* x is coerced to far     */
                             /*   pointer in compact,   */
                             /*   large, or huge model. */
}

int test_fun( int __near *ptr1, char __far *ptr2, int a)
{
    printf("Value of a = %d\n", a);
}
```

If the preceding example is compiled as a tiny, small, or medium program, the size of `x` is 16 bits, the size of `y` is 32 bits, and the value printed for `a` is 1.

However, if the example is compiled in compact, large, or huge model, both `x` and `y` are automatically converted to far pointers when they are passed to `test_fun`. Because `ptr1`, the first parameter of `test_fun`, is defined as a near pointer argument, it takes only 16 bits of the 32 bits passed to it. The next parameter, `ptr2`, takes the remaining 16 bits passed to `ptr1`, plus 16 bits of the 32 bits passed to it. Finally, the third parameter, `a`, takes the leftover 16 bits from `ptr2`, instead of the value of `z` in the **main** function.

This shifting process does not generate an error message, because both the function call and the function definition are legal. However, in this case, the program does not work as intended, because the value assigned to `a` is not the value intended.

To pass `ptr1` as a near pointer, you should include a function prototype that specifically declares this argument for `test_fun` as a near pointer, as follows:

```
/* First, prototype test_fun so the compiler
 * knows in advance about the near pointer argument:
 */
int test_fun(int __near*, char __far *, int);

main()
{
    int __near *x;
    char __far *y;
    int z = 1;
```

```
    test_fun( x, y, z );     /* Now, x is not coerced
                              * to a far pointer; it is
                              * passed as a near pointer,
                              * regardless of which memory
                              * model is used.
                              */
}

int test_fun( int __near *ptr1, char __far *ptr2, int a)
{
    printf( "Value of a = %d\n", a );
}
```

2.4 Customizing Memory Models

A third way to manage memory is to combine different features from standard memory models to create your own customized memory model. You should have a thorough understanding of C and C++ memory models and the architecture of 80*x*86 processors before creating your own nonstandard memory models.

In a customized model, you select the size of code pointers and data pointers.

Using the /A*string* option, you can change the attributes of the standard memory models to create your own memory models. The three letters in *string* correspond to the code pointer size, the data pointer size, and the stack and data segment setup, respectively. Because the letter allowed in each field is unique to that field, you can give the letters in any order after /A. All three letters must be present.

The standard memory-model options (/AT, /AS, /AM, /AC, /AL, and /AH) can be specified in the /A*string* form. As an example of how to construct memory models, the standard memory-model options are listed following with their /A*string* equivalents:

Standard	Custom equivalent
/AT	/Asnd
/AS	/Asnd
/AM	/Alnd
/AC	/Asfd
/AL	/Alfd
/AH	/Alhd

For example, you might want to create a huge-compact model that allows huge data items but only one code segment. The option for specifying this model is /Ashd.

Note The tiny model is identical to the small model except that it causes the linker to search for CRTCOM.LIB. The executable file generated when you specify tiny model is a .COM file rather than an .EXE file.

Setting a Size for Code Pointers

Within a custom memory model, you choose whether code pointers are short or long:

Option	Size
/As*xx*	Short (near) code pointers
/Al*xx*	Long (far) code pointers

The /As (short) option tells the compiler to generate near 16-bit pointers and addresses for all functions. This is the default for tiny-, small-, and compact-model programs.

The /Al (long) option means that far 32-bit pointers and addresses are used to address all functions. Far pointers are the default for medium-, large-, and huge-model programs.

Setting a Size for Data Pointers

Data pointers can be near, far, or huge:

Option	Size
/A*x*n*x*	Near data pointers
/A*x*f*x*	Far data pointers
/A*x*h*x*	Huge data pointers

The /An (near) option tells the compiler to use 16-bit pointers and addresses for all data. This is the default for tiny-, small-, and medium-model programs.

The /Af (far) option specifies that all data pointers and addresses are 32 bits. This is the default for compact- and large-model programs.

The /Ah (huge) option specifies that all data pointers and addresses are far (32-bit) and that arrays are permitted to extend beyond a 64K segment. This is the default for huge-model programs.

With far data pointers, no single data item can be larger than a segment (64K). This is because address arithmetic is performed only on 16 bits (the offset portion) of the address. When huge data pointers are used, individual data items can be larger than a segment (64K) because address arithmetic is performed on both the segment and the offset.

Setting Up Segments

Within a customized model, you can choose to make the stack segment (SS) equal the data segment (DS), in which case they overlap:

Option	Effect
/A*xx*d	SS == DS.
/A[[*xx*]]u	SS != DS; DS is reloaded on function entry.
/A[[*xx*]]w	SS != DS; DS is not reloaded on function entry.

Segment Setup Option (/Ad)

The option /Ad tells the compiler that the segment addresses stored in the SS and DS registers are equal. The stack segment and the default data segment are combined into a single segment. This is the default for all standard-model programs. In small- and medium-model programs, the stack plus all data must occupy less than 64K; thus, any data item is accessed with only a 16-bit offset from the segment address in the SS and DS registers.

In compact-, large-, and huge-model programs, all data is placed in the default data segment up to a threshold set with the /Gt option. The address of this segment is stored in the DS and SS registers. All pointers to data, including pointers to local data (the stack), are full 32-bit addresses. This is important to remember when passing pointers as arguments in multiple-segment programs. Although you may have more than 64K of total data in these models, no more than 64K of data can occupy the default segment. The /Gt, /Gx, and /ND options control allocation of items in the default data segment if a program exceeds this limit.

Segment Setup Option (/Au)

The option /Au tells the compiler that the stack segment does not necessarily coincide with the data segment. In addition, it adds the **__loadds** attribute to all functions within a module, forcing the compiler to generate code to load the DS register with the correct value prior to entering the function body. Combine the /ND option with /Au to name data segments other than the default. When /Au is combined with /ND, the address in the DS register is saved upon entry to each function, and the new DS value for the module in which the function is defined is loaded into the register. The previous DS value is restored on exit from the function. Therefore, only one data segment is accessible at any given time. Using the /ND option, you can combine these segments into a single segment.

If a standard memory-model option precedes it on the command line, the /Au option can be specified without any letters indicating data pointer or code pointer sizes. The program uses a standard memory model, but different segments are set up for the stack and data segments.

For related information, see the following section, "Segment Setup Option (/Aw)."

Segment Setup Option (/Aw)

The option /Aw, like /Au, causes the compiler to assume that the stack segment is separate from the data segment. The compiler does not automatically load the DS register at each function entry point. The /Aw option is useful in creating applications that interface with an operating system or with a program running at the operating-system level. The operating system or the program running with the operating system actually receives the data intended for the application program and places that data in a segment; then the operating system or program must load the DS register with the segment address for the application program.

As with the /Au option, the /Aw option can be specified without data pointer and code pointer letters if a standard memory-model option precedes it on the command line. In such a case, the program uses the specified memory model just as with /Au but the DS register is not reloaded at each function entry point.

Even though /Au and /Aw indicate that the stack may be in a separate segment, the stack's size is still fixed at the default size unless this is overridden with the /F compiler option or the /STACK linker option.

Use caution when writing DLLs with /Aw.

The /Aw option is useful for writing dynamic-link libraries (DLLs) for Windows, but exercise caution when using it. Declare all entry points to the dynamic-link library as **__loadds** to force DS to be loaded on entry to the function (exactly as with the /Au option). This adds a costly operation to each function that acts as an entry point, but not to any of the functions that are private to the DLL. This is more efficient than using the /Au option, because most of the DLL's functions do not have to perform redundant loads of the DS register. For example:

```
void __export __loadds __far __pascal LibFunc( void )
{
    .
    .
    .
    HelperFunc();
}

void HelperFunc( void )
{
    .
    .
    .
}
```

The library entry point, `LibFunc`, is declared as **__loadds** to force the DS register to be loaded on entry. The function `HelperFunc`, which is private to the dynamic-link library, is declared as a normal C function. Because it cannot be called from outside of the module, `HelperFunc` does not need to reload DS.

Note Combined use of the /GD option and the **__export** keyword produces the most efficient DLL code for ensuring correct initialization of DS on entry to a function. You should use /GD and **__export** rather than either /Au or /Aw and **__loadds**.

If you choose one of the options specifying that the stack segment is not equal to the data segment (SS != DS), you cannot pass the address of frame variables as arguments to functions that take near pointers. That is, in tiny, small, and medium models, you cannot pass the address of a local variable (which is allocated on the stack) as an argument, because the receiving function assumes the pointer is relative to the data segment. However, the receiving function can solve this problem by declaring the pointer to be the following:

```
based(__segname("_STACK"))
```

Another solution is to cast the pointer to a far pointer in both locations as follows:

```
/* Call func with an explicit cast to far */
func( (char far *)frame_var );
.
.
.
void func( char far *formal_var )
```

Library Support for Customized Memory Models

Most C and C++ programs make function calls to the routines in the C run-time library. When you write mixed-model programs, you are responsible for determining which library (if any) is suitable for your program and for ensuring that the appropriate library is linked. Table 2.3 shows the libraries from which to extract the startup routine for each customized memory model.

Table 2.3 Startup Routines for Customized Memory Models

Memory-model option	From library
/Asn*x*; /AS plus /A*x*	SLIBC*f*.LIB
/Asf*x*; /Ash*x*; /AC plus /A*x*	CLIBC*f*.LIB
/Aln*x*; /AM plus /A*x*	MLIBC*f*.LIB
/Alf*x*; /Alh*x*; /AL plus /A*x*; /AH plus /A*x*	LLIBC*f*.LIB

The /A*x* option represents either /Au or /Aw. In the library names, *f* is either E (emulator library), 7 (8087/80287 library), or A (alternate math library).

Placement of Data in the Compact, Large, and Huge Memory Models

In a memory model permitting multiple data segments, a global data item may be allocated in either the default data segment or in a far data segment. The data item's location and the way it is referenced depend on whether it is declared with a defining declaration or a referencing declaration (for more information, see Chapter 3, "Declarations and Types," in the *C Language Reference*).

Defining Declarations

Defining declarations include initialized data items and data items declared **static**, which are initialized to zero by default. The compiler can allocate space for data items in this category. These data items are placed in the default data segment unless their size exceeds a certain threshold. This threshold is specified by the /Gt option, whose syntax is:

/Gt[[*number*]]

The /Gt option causes all initialized data items whose size is greater than *number* bytes to be allocated to a new data segment. When *number* is specified, it must follow the /Gt option immediately, with no intervening spaces. When *number* is omitted, the default threshold value is 256. When the /Gt option is omitted, the default threshold value is 32,767.

This option is useful with programs that have more than 64K of initialized static and global data in small data items. Without this option, your program fills the default data segment and cannot be linked. The /Gt option does not apply to items declared with the **__near** or **__far** keyword.

Referencing Declarations

Referencing declarations include data items declared **extern** and uninitialized, nonstatic data items. The compiler cannot allocate space for data items in this category because it lacks information found in the other modules. When all the modules in the program are linked together, the linker can examine all references to these data items and determine where they are placed.

In the compact, large, and huge memory models, the compiler by default assumes that the linker places data items in this category in the default data segment. All references to such data items are done with near addressing. This improves the efficiency of your application.

Note If you reference a data item with near addressing but declare it with **__far** in the module in which it is declared, your program produces unpredictable results.

This default near addressing is useful for writing compact-, large-, and huge-model applications for Windows. If you want simultaneously to run multiple instances of your application for Windows, you cannot use far addressing with your global data.

Unsized arrays are treated as far.

The /Gx option affects how a data item is referenced only if the data's location is not otherwise specified. If an uninitialized or **extern** data item is declared with **__near** or **__far**, it is referenced as specified. If a data item is larger than the threshold specified by the /Gt option, it is referenced with far addressing. Unsized arrays are treated as far because they might be larger than the threshold. You must explicitly declare an unsized array with **__near** if you want it referenced with near addressing.

The /Gx option does not affect pointers. Pointers remain far by default, and the dynamic allocation functions still return far pointers.

Naming Modules and Segments

Using the /NM, /NT, and /ND options, you can name the module, the code segment, and the data segment. The following list summarizes the options' syntax:

Option	Effect
/NM *modulename*	Names the module
/NT *textsegment*	Names the code segment
/ND *datasegment*	Names the data segment

"Module" is another name for an object file created by the compiler from a single source file. Every module has a name. The compiler uses this name in error messages if problems are encountered during processing. The module name is usually the same as the source-file name. You can change this name using the /NM (name the module) option. The new *modulename* can include any combination of letters and digits. The space between /NM and *modulename* is optional.

Every module has at least two segments: a code segment (sometimes called the text segment) containing the program instructions and a data segment containing the program data.

The compiler usually creates the code and data segment names. The default names depend on the memory model chosen for the program. For example, in small-model programs, the code segment is named **_TEXT** and the data segment is named **_DATA**.

Table 2.4 summarizes the naming conventions for code and data segments.

Table 2.4 Segment-Naming Conventions

Model	Code	Data	Module
Tiny	**_TEXT**	**_DATA**	—
Small	**_TEXT**	**_DATA**	—
Medium	*module*_**TEXT**	**_DATA**	*filename*
Compact	**_TEXT**	**_DATA**	*filename*
Large	*module*_**TEXT**	**_DATA**	*filename*
Huge	*module*_**TEXT**	**_DATA**	*filename*

In memory models that contain multiple data segments (compact, large, and huge), **_DATA** is the name of the default data segment. Other data segments have unique private names. You can override the default names with the options /NT (name the text, or code, segment) and /ND (name the data segment).

The /ND option is commonly used to create and compile modules that contain data only. Such modules can be accessed from other parts of the program by declaring their variables as external.

If you change the name of the default data segment with /ND, your program must load the DS register with the segment selector of your named data segment before it accesses it. You must therefore compile your program either with the /A*string* form of the memory-model option and the /Au option for the segment setup, or with the /A option for a standard memory model followed by /Au. For example:

```
CL /AS /Au /ND DATA1 PROG1.C
```

The /Au option forces the compiler to generate code to load DS with the correct data-segment value on entry to the code.

All modules whose data segments have the same name have these segments combined into a single segment named `DATA1` at link time.

The functions in the small data model run-time libraries that rely on the default data segment being named **_DATA** fail if you use the /ND option to rename the default data segment. This restriction affects tiny-, small-, and medium-model programs.

Specifying Code Segments

Using the **alloc_text** pragma, you can name the segment in which particular functions are allocated. It has the following syntax:

#pragma alloc_text (*textsegment*, *function1* [[, *function2*]]...)

If you use overlays or swapping techniques to handle large programs, you can use **alloc_text** to tune the contents of their code (text) segments for maximum efficiency. The **alloc_text** pragma must appear before the definitions of any of the specified functions and after the declarations of these functions. Functions referenced in an **alloc_text** pragma should be defined in the same module as the pragma. If this is not done and an undefined function is later compiled into a different code segment, the error may not be caught.

Another way to specify the segment in which a function resides is to use based addressing for functions. You can also use based addressing to specify the segment in which a data item resides.

2.5 Using Based Pointers and Data

Visual C++ provides the keyword **__based** to give you greater control over memory management in a segmented architecture. You can use **__based** to control the placement of data or functions within segments and to get more efficient pointer operations.

This section explains how to use based pointers and based data allocation. The use of based functions is explained in Section 2.6, "Using Based Addressing for Functions," page 59.

Based Pointers

Based pointers combine the advantages of near and far pointers. Based pointers are 2 bytes in size, like near pointers, but their range is not limited to the default data segment. Like far pointers, they can refer to any available memory location. Based pointers provide a more efficient way to represent addresses outside the default data segment by exploiting the commonality among multiple pointers.

This is possible because a based pointer contains only the offset portion of an address. To use such a pointer, you must define a "base" for it. A base consists of the segment portion of an address and is stored separately from the pointer itself. If many based pointers refer to locations within the same segment, they can all share

the same base. The offset and segment values are combined whenever a based pointer is used to access a memory location.

By comparison, every far pointer contains both an offset and a segment value, which can result in wasted space if many far pointers refer to locations within one segment. Near pointers contain only an offset, but because they always use the DS register for their segment value, they are restricted to addressing the default data segment.

Using based instead of far pointers makes your program smaller.

The use of based pointers instead of far pointers makes your program smaller by saving two bytes for each pointer that shares a base with another. Under certain conditions, based pointers can also be faster than far pointers. If your program has many based pointers that are all based on the same segment and if those pointers are used consecutively, the compiler does not need to load a new segment value each time a pointer is used. If you enable full optimizations in such circumstances, based pointers can be almost as fast as near pointers.

Define a pointer's base using the **__based** keyword, followed by a base expression in parentheses, where you might otherwise place **__near**, **__far**, or **__huge**. For example:

```
void __near np;
void __based(base) bp;
```

There are several types of base that you can specify for a based pointer:

- A fixed base
- A variable base
- The **__self** keyword
- The **void** keyword

These types of base are described in the following sections.

Pointers with a Fixed Base

Pointers based on a fixed segment are restricted to accessing locations in a single segment. This segment is specified when the based pointers are declared. You can make assignments to the based pointers themselves, which changes the offset portion of the address. Making assignments in this way causes the pointers to refer to different locations within the segment. However, you cannot change the base that the based pointers use.

There are two ways to specify a fixed base for based pointers: by using a named segment or by using the segment in which a variable is stored.

Using a Named Segment

You can specify a named segment as the base for your pointers by using the **__segname** keyword and a string literal. For example, the following example declares a pointer based in the default code segment:

```
void __based(__segname("_CODE")) *bp;
```

The pointer bp can address any location in the default code segment. There are four segments accessible through predefined strings:

Segment	Definition
_CODE	Current code segment
_CONST	Constant segment
_DATA	Default data segment
_STACK	Stack segment

The following example declares a pointer based in the default data segment:

```
char __based(__segname("_DATA")) *bp;
```

This is equivalent to a near pointer.

You can also specify user-defined segments, as long as the segment is allocated somewhere else in the program. For example:

```
char __based(__segname("MYSEG")) *bp;
```

You can define MYSEG with an assembly-language file or by allocating data in a named segment. For more information, see "Data Stored in a Named Segment," on page 57.

Using the Segment of a Variable

You can also base your pointers on the segment in which another variable is stored. Specify this type of base by casting the address of a variable to the **__segment** data type, as follows:

```
int i;
void __based((__segment)&i) *bp;
```

This declaration allows bp to access any location in the same segment in which i is stored. If i is declared as **__near** or if the program is compiled in tiny, small, or medium model, this is equivalent to declaring bp as a near pointer.

Pointers with a Variable Base

Pointers with a variable base can access any available memory locations. When you make assignments to the based pointers themselves, you change the offset

portion of the address, which allows you to refer to various locations within one particular segment. You can also make assignments to the base itself. The compiler uses the updated value of the base whenever one of these based pointers is used. In this way, changing a single base value effectively changes the locations referenced by all the based pointers using that base.

There are three ways to specify a variable base for based pointers: by using the segment value of another pointer, by using a variable of type **__segment**, or by using another pointer.

Using the Segment Value of Another Pointer

You can give a based pointer the segment of another pointer as its base value. To do this, cast a pointer to the **__segment** data type, as follows:

```
char __near *np;
char __far *fp;
void __based((__segment)np) *bnp;
void __based((__segment)fp) *bfp;
```

Notice that this syntax is similar to that used to base a pointer on the segment in which a variable is stored. The difference is that you cannot change where a variable is allocated, but you can change the value of a pointer.

Because `np` is a near pointer, it uses the DS register as its segment value. Accordingly, `bnp` uses DS as its base and is equivalent to a near pointer.

Because `fp` is a far pointer, it contains a segment value, and `bfp` uses that segment as its base. If you change the segment portion of `fp`, `bfp` refers to a location in the new segment. (Remember that far pointer arithmetic is performed only on the offset portion, so incrementing `fp` won't affect the base of `bfp`. However, if you make an assignment to `fp` that changes its segment, the base of `bfp` is similarly modified.)

Using a Segment Variable

In addition to using a cast to the **__segment** data type, you can define variables of type **__segment**. You can then base your pointers on such a segment variable, as follows:

```
__segment videomem;              /* Define a segment variable.   */
char __based(videomem) *vidptr;

videomem = 0xB800;               /* Use video memory as segment. */
                                 /* Move to row 10, column 40.   */
vidptr = (char __based(videomem) *)(2 * ((80 * 9) + 39));
*vidptr = 'A';                   /* Write an A there.            */
```

In this example, `videomem` is a segment variable that contains the segment in which video memory resides. Because `vidptr` is based on `videomem`, any value assigned to `vidptr` is interpreted as an offset into video memory. A cast is used in

the assignment to `vidptr` to prevent a compiler warning. If `videomem` is assigned a new value, `vidptr` acts as an offset from that new value and evaluates to an entirely different address.

You cannot base a pointer on a constant that is cast to the **__segment** type, as in the following example:

```
unsigned vidptr __based((__segment)0xB800) *vidptr;   /* Error. */
```

You must use a segment variable that is defined separately.

Pointers based on a segment variable are especially useful in conjunction with based heaps. Using Visual C++, you can define a special heap that resides in a segment. You can use such a based heap to allocate objects dynamically, just as you do with a traditional heap. These dynamically allocated objects can all be referenced with pointers based on that segment.

The following program demonstrates the creation of a based heap:

```
/* Compile in small model */

#include <malloc.h>
#include <stdio.h>
#include <string.h>

__segment segvar;
char __based(segvar) *b_string;

void main()
{
    if( (segvar = _bheapseg( 1000 )) != _NULLSEG )
    {
        if( (b_string = _bmalloc( segvar, 20 )) != _NULLOFF )
        {
            _fstrcpy( (char __far *)b_string, (char __far *)"This is a test.\n" );
            printf( "%Fs", (char __far *)b_string );
            printf( "Size = %d\n", sizeof b_string );  /* Always 2 */
            _bfree( segvar, b_string );
        }
        else
            puts( "bmalloc failed" );
        _bfreeseg( segvar );
    }
    else
        puts( "_bheapseg failed" );
}
```

First, the program calls the library function **_bheapseg** and requests 1000 bytes in a new based heap:

```
if( (segvar = _bheapseg( 1000 )) != _NULLSEG )
```

If it cannot allocate the amount of memory requested, **_bheapseg** returns **_NULLSEG** (null segment). Otherwise, the function returns the valid address of a segment, which is assigned to `segvar`.

Next, the program calls **_bmalloc** and requests 20 bytes of memory from the based heap. The variable `segvar` is passed to identify the based heap that **_bmalloc** should use. Just as **malloc** returns a pointer to a block of memory, **_bmalloc** returns an offset to a block of memory. This offset is assigned to the based pointer `b_string`:

```
if( (b_string = _bmalloc( segvar, 20 )) != _NULLOFF )
```

The value **_NULLOFF** means "null offset" and indicates the failure of **_bmalloc**. If the allocation succeeds, the program continues with this code:

```
_fstrcpy( (char __far *)b_string, (char __far *)"This is a test.\n" );
printf( "%Fs", (char __far *)b_string );
printf( "Size = %d\n", sizeof b_string );  /* Always 2 */
```

The standard **strcpy** function won't work because this is a small-model program that expects all pointers to be near. The **_fstrcpy** function accepts far pointers, and it is possible to cast a based pointer to a far pointer. Then the string and its size are printed.

Finally, the block of memory and the based heap are freed:

```
_bfree( segvar, b_string );
_bfreeseg( segvar );
```

The run-time library provides a complete set of memory-management functions that work with based heaps.

Using Another Pointer

You can also base your pointers on the complete address of another pointer, instead of using only the segment portion of its address. In this case, a based pointer acts as an offset from the pointer itself, instead of simply sharing the segment with that pointer. For example:

```
int *ip;
int __based(ip) *bp;
```

Whenever `bp` is used, the compiler adds together the offset of `ip` and the offset stored in `bp` and uses the segment of `ip` to find the address.

The following example illustrates pointers based on a pointer:

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <string.h>

int *ip;              /* int pointer */
int __based(ip) *bp;  /* Based on ip */
char __based(ip) *cp;

void main()
{
    int *mem1, *mem2;

    bp = (int __based(ip) *)0;    /* bp equals *(ip+0) */
    cp = (char __based(ip) *)2;   /* cp equals *(ip+2) */

    if( (mem1 = (int *)malloc( 100 )) != NULL )
        if( (mem2 = (int *)malloc( 100 )) != NULL )
        {
            ip = mem1;  /* ip points to mem1 */
            *bp = 5;
            strcpy( (char *)cp, "String stored in mem1." );

            ip = mem2;  /* ip now points to mem2 */
            *bp = 12345;
            strcpy( (char *)cp, "String stored in mem2." );

            ip = mem1;  /* Point to mem1, */
            /* which still holds previous values */
            printf( "%s  *bp= %i\n", (char *)cp, *bp );

            ip = mem2;  /* Point to mem2 */
            /* Display the values there */
            printf( "%s  *bp= %i\n", (char *)cp, *bp );

            free( mem2 );
            free( mem1 );
        }
        else puts( "Second malloc failed." );
    else puts( "First malloc failed." );
}
```

Two calls to **malloc** provide two sections of memory, whose addresses are stored in the variables `mem1` and `mem2`. When `ip` is assigned one of these addresses (`mem1`), the pointers based on `ip` point somewhere within that piece of memory. When `ip` is assigned the address in `mem2`, the effective addresses of `bp` and `cp` also change.

Note Pointers based on pointers are the only form of based pointers that can be used in a 32-bit program. They are the only type of based pointer that can be used in a flat (that is, nonsegmented) address space.

If you have a group of pointers that all refer to locations within a buffer of memory, you can define them as offsets from a pointer that references the start of the buffer. If you relocate that buffer, you can update the entire group of pointers by modifying just the pointer that acts as their base. If you write the buffer to disk, you can also write the based pointers to disk. Once you reload the buffer into memory, you can make the based pointers valid again by updating their base.

Pointers Based on the __self Keyword

You can base a pointer on the segment that the pointer itself is stored in. This is done by using the **__self** keyword, cast to the **__segment** type. Consider the following example:

```
typedef struct node NODE;

struct node
{
    int name;
    NODE __based((__segment)__self) *left;
    NODE __based((__segment)__self) *right;
};
```

This example declares a structure named `NODE` for use in a binary tree. Each node in the tree contains pointers to its two child nodes. These pointers are self-based, so they refer to locations within the segment in which the node itself is stored. This is possible only when an entire tree can fit in a single segment. Based pointers provide an advantage over far pointers in such a data structure by reducing the size of each node by 4 bytes.

You may want to build a tree out of nodes that contain self-based pointers. Do not use **malloc** to allocate the nodes, because it may return memory in different

segments. Instead, use a based heap along with pointers based on a segment variable. The following example assumes the type declaration previously given.

```
void main()
{
    __segment segvar;
    NODE __based(segvar) *nodeptr;

    /* Ignore error checking for this example. */
    segvar = _bheapseg( 30000 );
    nodeptr = _bmalloc( segvar, sizeof(NODE) );
    nodeptr->left = _bmalloc( segvar, sizeof(NODE) );
    nodeptr->right = _bmalloc( segvar, sizeof(NODE) );
    nodeptr->name = 1;
    nodeptr->left->name = 2;
    nodeptr->right->name = 3;
}
```

This program first allocates a based heap of 30,000 bytes and uses `segvar` to store the heap's segment. Then the program allocates `NODE` objects from that based heap, so all the nodes in the tree reside in the segment specified by `segvar`. Note that `nodeptr` is based on `segvar`, instead of being self-based. A self-based pointer declared as a local variable in a function uses the SS register as its base, which may not be in the same segment as `segvar`.

Pointers Based on the void Keyword

The final way to declare a based pointer is to base it on **void**. Such a pointer is not based on any particular segment. It is an offset that can be combined with any segment to form a full address. You can combine a segment value and a void-based pointer using the "base operator," which consists of a colon and a greater-than symbol (**:>**):

segment:>*offset*

Such an expression denotes a complete address and can be dereferenced with the indirection operator (*). You can use the base operator only with pointers based on **void**, not with other types of based pointers.

The segment value can be a variable of type **__segment**, or it can be an integer cast to type **__segment**. For example:

```
__segment videomem = 0xB800;   /* Use video memory as segment.  */
char __based(void) *offptr;

                               /* Set offset to row 10, col 40. */
offptr = (char __based(void) *)(2 * ((80 * 9) + 39));
*(videomem:>offptr) = 'A';     /* Write an A there.             */
offptr += 2;                   /* Move to col 41.               */
*(((__segment)0xB800):>offptr) = 'A'; /* Repeat.                 */
```

The pointer `offptr` can be used with any segment variable. If you have many segments organized in the same way, you can use one void-based pointer to access the same relative location in each of them.

Based Data Allocation

The section "Using a Segment Variable," on page 51, describes dynamic allocation of based data using the run-time library functions. Using Visual C++, you can also statically declare data that is based in a specified segment.

There are three ways to specify that data is declared in a particular segment: by specifying a named segment, by using a segment variable, and by using the address of another variable.

Data Stored in a Named Segment

You can specify a named segment that a variable is to be stored in by using the **__segname** keyword and a string literal. Note that the syntax for this is the same as that used to base a pointer on a named segment. For example:

```
/* Compile in small model */

#include <stdio.h>
#include <malloc.h>

char __based(__segname("_CODE")) mystring[] = "A code-based string.\n";
int __based(__segname("_CODE")) myint = 12345;

void main()
{
    printf( "%Fs %d", (char __far *)mystring, myint );
}
```

The variable `mystring` is declared as an array of characters based in the code segment. The variable `myint` is an integer that is also based in the code segment.

Note that the small-model version of **printf** treats `mystring` as a near pointer. The **F** in the format specifier **%Fs** forces the function to treat `mystring` as a far pointer, and the cast to **char __far*** coerces the address to 4 bytes.

One reason for placing data in your code segment is that you are using the small memory model and your default data segment is full. Rather than move up to the compact memory model, which makes all data pointers far, you can move some data into the code segment, if you have room there.

You can also name your own segments. For example:

```
char __based(__segname("MYSEGMENT")) otherstring[] = "Another based string.\n";
```

This declaration creates a new segment called `MYSEGMENT` and places the string there. You can reference data in that segment using far pointers or pointers based on that named segment.

If the segment named ends in "_TEXT," the compiler marks that segment as a code segment, making it a read-only segment.

Note You cannot store data in the **_STACK** segment. Of the four predefined segments, you can store data in only the **_CODE**, **_DATA**, and **_CONST** segments.

Data Based on a Segment Variable

You can also declare data that is based on a segment variable. Data declared this way is stored at a location determined at run time. This is useful if you want to make some variables relocatable. When you move the block of memory containing the variables, you can simply assign a new value to the segment variable. Using this technique, you can access the variables by name, rather than by using pointers.

The following example demonstrates how to declare data based on a segment variable:

```
/* FILE1.C */
char __far c;
__segment segvar;

main()
{
    segvar = (__segment)&c;
    foo();
    .
    .
    .
```

```
    /* Relocate segment; assign new value to segvar. */
    .
    .
    .
    foo();
}

/* FILE2.C */
extern __segment segvar;
extern char __based(segvar) c;

foo()
{
    c = 1; /* Can refer to c, no matter where it is. */
}
```

The compiler uses the segment value stored in `segvar` whenever `c` is accessed in FILE2.C.

Data Based on the Address of Another Variable

You can also allocate data in the same segment as another based variable. To do this, cast the address of the variable to the **__segment** data type. Note that this is the same syntax used to base a pointer on the segment in which a variable is stored. For example:

```
int __based(__segname("MYSEGMENT")) myint1;
int __based((__segment)&myvar1) myint2;
```

The variable whose segment is being used must itself be based on a named segment.

2.6 Using Based Addressing for Functions

With Visual C++, you can declare functions as based, so you can specify the code segment the functions reside in. Grouping functions into segments allows you to use near functions safely and to improve performance when you swap overlays to disk.

You can declare a function with both the **__near** and **__based** keywords or with both the **__far** and **__based** keywords, even though such declarations are illegal for data. This is because the meaning of the **__near** and **__far** keywords for functions differs slightly from their meaning for data. Near functions can reside anywhere in memory, but you can call them only from functions in the same code segment. Far functions can also reside anywhere in memory, and you can call them from functions in other code segments. Thus, you can use the **__near** and **__far** keywords to describe a function's calling convention and use a **__based** expression to specify its location in memory.

The segment in which functions reside is usually determined by the memory model of your program. In the tiny, small, and compact models, all functions are stored in a single code segment. In the medium, large, and huge models, functions are stored in multiple code segments; there is a separate segment for each source file.

Placing functions correctly can reduce swapping.

The location of functions in segments becomes important when tuning large programs that use overlays. By placing the functions that most frequently call one another within the same segment, you can reduce swapping. The location of functions is also important when using near functions in a program that has multiple segments. A function might try to call a near function that resides in another segment, causing a run-time error.

To prevent this problem, you can declare functions as based to ensure that they are stored in the same segment. For example:

```
/* FILE 1 - compiled under large model. */

void __based(__segname("MYSEG")) farfunc()    /* Far by default. */
{
    nearfunc();
}

/* FILE 2 - compiled under large model. */

void __near __based(__segname("MYSEG")) nearfunc()
{
    /* ... */
}
```

If these two functions are not declared as based in the same segment, they are placed in separate segments because they're declared in separate files. In that situation, this program suffers a link-time or run-time error because `farfunc` cannot perform a near call to `nearfunc` when `nearfunc` is in another segment. However, because both functions are based in the `MYSEG` segment, the program links and runs correctly.

Functions can be based only on a segment constant; unlike data, they cannot be based on segment variables, nor can they be based on pointers, **void**, or the **__self** segment. The **__near** or **__far** keyword can appear before or after the based expression.

Based addressing for functions replaces the **alloc_text** pragma as a method of controlling the placement of functions. If both a based expression and an **alloc_text** pragma specify a segment for a function to be placed in, the based expression takes precedence.

2.7 Using the Virtual Memory Manager

Virtual memory is a facility for accessing storage beyond the 640K of memory available to MS-DOS. Visual C++ provides a virtual memory manager through a set of functions in the run-time library. This memory manager uses expanded memory (EMS), extended memory (XMS), and disk storage to simulate a heap of nearly unlimited size. By using this virtual heap, your program can access those three memory resources through a single interface and acquire far more memory than is available from the traditional **malloc** family of functions.

Note that the virtual memory functions are available only for 16-bit MS-DOS programs. Programs for Windows and 32-bit programs do not need to use these functions. P-code programs cannot use the virtual memory manager.

The virtual memory manager works by copying blocks of virtual memory into MS-DOS memory when they're in use and swapping them out to auxiliary storage when they're not. In general, a program that uses the virtual memory manager must perform the following steps:

- Initialize the virtual memory manager, by calling the **_vheapinit** function
- Allocate virtual memory blocks as needed, by calling the **_vmalloc** function
- Load or lock virtual memory blocks into the MS-DOS address space in order to access their contents, by using the **_vload** or **_vlock** function
- Unlock virtual memory blocks when they're not being accessed, by calling the **_vunlock** function
- Free virtual memory blocks when they're no longer needed, by calling the **_vfree** function
- Terminate the virtual memory manager, by calling the **_vheapterm** function

The following sections describe these steps in more detail.

Initializing the Virtual Memory Manager

You initialize the virtual memory manager by calling **_vheapinit** and passing it three arguments:

- The minimum amount of MS-DOS memory that must be available for the virtual memory manager to be installed (in 16-byte paragraphs)
- The maximum amount of MS-DOS memory that it can use (in paragraphs)
- Flags indicating which types of auxiliary storage it can use to hold swapped-out blocks

The virtual memory manager may round up the minimum value you specify. If after rounding the minimum amount of memory is not available, the virtual memory manager is not installed. The virtual memory manager needs several kilobytes in order to function effectively.

If you want the virtual memory manager to use as much MS-DOS memory as it can, specify **_VM_ALLDOS** as the second argument. You should not specify this if your program is performing tasks that require a lot of free memory, such as spawning a process.

To specify the types of auxiliary storage that the virtual memory manager can use, use the **_VM_EMS**, **_VM_XMS**, or **_VM_DISK** flag. One or more of these flags can be specified if they are joined by the bitwise-OR operator (|). To use all three types, specify **_VM_ALLSWAP**. If not all forms of storage are available when the program runs, the virtual memory manager uses what is available.

A typical call to **_vheapinit** looks like this:

```
if ( !_vheapinit( 0, _VM_ALLDOS, _VM_ALLSWAP ) )
{
    /* Initialization failed - perform error handling */
}
else
    /* Continue with normal program execution */
```

This call to **_vheapinit** specifies that the virtual memory manager should attempt to install itself no matter how little memory is available, though the attempt may fail if insufficient memory is available. This call also specifies that the virtual memory manager should use as much memory as is available and that it should use all forms of auxiliary storage.

When your program is done using virtual memory, it must call the **_vheapterm** function to terminate the virtual memory manager.

Note If your program ends without calling **_vheapterm**, various system memory resources may not be available to subsequent programs.

You can initialize and terminate the virtual memory manager as many times as you want within your program.

Virtual Memory Handles

When you allocate a block of virtual memory, **_vmalloc** does not return a pointer the way **malloc** does. Instead, **_vmalloc** returns a "handle," which is a value of type **_vmhnd_t** that uniquely identifies the block of virtual memory. You cannot use such a handle to access memory directly, nor can you perform address

arithmetic on a handle. You can only pass a handle to other virtual memory functions.

In order to access the contents of a virtual memory block, you must either load it or lock it into MS-DOS memory.

Loading Blocks

The **_vload** function takes a handle and copies the associated block of virtual memory into MS-DOS memory. The function returns a far pointer to the location at which the block of memory is loaded. You use this pointer to read or modify the contents of the block.

The **_vload** function keeps the contents of the block in MS-DOS memory only temporarily. The next time you call any function of the virtual memory manager, a loaded block may be swapped out to auxiliary storage, making the pointer returned by **_vload** invalid. Accordingly, you should access the contents of a loaded block only until the next call to the virtual memory manager.

Dirty Blocks vs. Clean Blocks

When you load a block of virtual memory with **_vload**, you must specify either the flag **_VM_CLEAN** or **_VM_DIRTY**, indicating that the block is either "clean" or "dirty." If your program reads the block of memory but does not modify its contents, the block is clean. If your program modifies the block of memory, the block is dirty. The specified flag tells the virtual memory manager what to do when it needs the region of MS-DOS memory that the loaded block occupies. If a block is clean, the virtual memory manager is free to overwrite it the next time it has to load a new block of memory. If a loaded block is dirty, the virtual memory manager must write out its contents to auxiliary storage before it loads a new block.

Every block of virtual memory that you allocate must be flagged as dirty at least once, if only to initialize its contents. If the block is treated as read-only from that point forward, it can be flagged as clean during subsequent loads. Otherwise, it must be flagged as dirty each time the program modifies it.

Note that when a dirty block is saved, its contents are retained only until the block is freed or the virtual memory manager is terminated. If you want to save the block's contents beyond that point, you must load the block into MS-DOS memory and explicitly copy its contents to a permanent disk file.

Locking and Unlocking Blocks

To retain access to a block for an arbitrarily long period of time, use the **_vlock** function. Like **_vload**, **_vlock** takes a handle, copies the associated block of virtual memory into MS-DOS memory, and returns a far pointer to it. However, **_vlock**

locks a block of memory so that it remains in MS-DOS memory even if you make subsequent calls to the virtual memory manager. A locked block remains in MS-DOS memory until it is unlocked with **_vunlock**. You can lock a block multiple times; the block is not swapped out until you have unlocked it an equal number of times. The number of locks currently held on a virtual memory block can be determined by calling the **_vlockcnt** function.

You must also specify a clean or dirty flag when you unlock a locked block of virtual memory with **_vunlock**. With this function, you specify the flag after you have accessed the block instead of before, as was the case with **_vload**. For a block that has been locked more than once, different clean or dirty flags can be specified for the **_vunlock** calls. If **_VM_DIRTY** is specified with any of the **_vunlock** calls, the block is treated as dirty.

You can lock a block that has already been loaded into MS-DOS memory. If you do so, the virtual memory manager may relocate the block within MS-DOS memory, so you should use the pointer returned by **_vlock** rather than the one previously returned by **_vload**.

Both **_vload** and **_vlock** return **NULL** if they are unable to load or lock a block of virtual memory. Always test return values for these functions before using them as pointers.

Keep as few blocks locked as possible.

Having a large number of blocks locked at any one time can interfere with the virtual memory manager's ability to swap blocks in and out of MS-DOS memory. Therefore, you should always keep as few blocks locked as possible.

Techniques for Using Virtual Memory

Virtual memory can be used as a replacement for MS-DOS memory in data structures. For example, you can build a linked list that resides in virtual memory; such a linked list could contain far more nodes than an ordinary linked list.

The declaration for the node type of such a linked list might look as follows:

```
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <vmemory.h>
#include <string.h>

typedef struct node NODE;
```

```
struct node
{
    int key;
    char data[100];
    _vmhnd_t next;
};

/* Globals */
_vmhnd_t vhead = _VM_NULL;    /* First element in list  */
_vmhnd_t vlast = _VM_NULL;    /* Last element in list   */
```

Each NODE structure contains a **_vmhnd_t** field rather than a pointer to connect it to the succeeding node.

You can use these NODE structures the same way you use the nodes of an ordinary linked list, except that you must load each node into MS-DOS memory before you access its contents. For example, the following procedure adds a new node to a linked list:

```
int add( NODE new_node )
{
    _vmhnd_t vtemp;
    NODE __far *temp;
    NODE __far *last;

    if ( (vtemp = _vmalloc( sizeof( NODE ) )) == _VM_NULL )
        return 0;     /* Could not allocate virtual memory. */

    if ( (temp = (NODE __far *)_vload( vtemp, _VM_DIRTY )) == NULL )
    {
        _vfree( vtemp );    /* Free the block that was just allocated */
        return 0;           /*   but could not be loaded.             */
    }

    temp->key = new_node.key;    /* Copy in new data.      */
    strncpy( temp->data, new_node.data, 100 );
    if ( vhead == _VM_NULL )     /* No nodes in list yet  */
    {
        vhead = vtemp;
        vlast = vhead;
    }
```

```
    else          /* Add to end of list. */
    {
        last = (NODE __far *)_vload( vlast, _VM_DIRTY );
        last->next = vtemp;
        vlast = vtemp;
    }
    return 1;     /* Node successfully added. */
}
```

The `add` function always uses two variables when manipulating a node: a handle and a pointer. When creating the node to be added, the function uses a handle to allocate and load the block of memory and then uses a pointer to write the new data into the node. Similarly, when attaching the node to the end of the list, the function uses a handle to load the last node and then uses a pointer to modify its `next` field. Note that `temp` and `last` are explicitly declared as far pointers. This is because no matter what model the program is compiled under, **_vload** returns far pointers. Also note that the **_VM_DIRTY** flag is specified in both calls to **_vload**, because in both cases the function is modifying the block of memory.

The `find` function has similar features:

```
NODE *find( int search_key )
{
    _vmhnd_t vcurr;
    NODE __far *curr;
    NODE *temp;

    if ( vhead == _VM_NULL )
        printf( "list empty \n" );
    vcurr = vhead;
    while ( vcurr != _VM_NULL )
    {
        if ( (curr = (NODE __far *)_vload( vcurr, _VM_CLEAN )) == NULL )
            return NULL;   /* Could not load block. */

        if ( curr->key == search_key )
        {
            if ( (temp = (NODE *)malloc( sizeof( NODE ) )) == NULL )
                return NULL;    /* Could not allocate memory. */

            temp->key = curr->key;   /* Copy data from node. */
            strncpy( temp->data, curr->data, 100 );
            return temp;
        }
        else
            vcurr = curr->next;
    }
    return NULL;
}
```

As with the `add` function, the `find` function uses both a handle and a pointer to access nodes. The function traverses the linked list, comparing each node's key with the key being searched for. To examine a node, the function uses a handle to load it into MS-DOS memory and then uses a pointer to access the `key` field. Note that this time the **_VM_CLEAN** flag is specified in the call to **_vload**, because the function is only reading the block of memory, not writing to it.

Other standard operations on a linked list, such as deleting or modifying a node, can be performed by making a minor modification to the usual implementations: You must load a block before you can access its contents. If you need to access a block many times, you should probably lock it. Or if you need to have more than one block in memory at once (if, for instance, you're comparing blocks' contents), you should lock one or more of them.

Other data structures traditionally implemented with pointers, such as binary trees, can also take advantage of virtual memory if you use handles in addition to pointers. Another possible technique is to maintain an array of handles, each of which refers to a large buffer of virtual memory. Your program can switch among these buffers, keeping just one or two of them in MS-DOS memory at any given time.

Only a portion of virtual memory is accessible at any one time.

When writing a program to use virtual memory, or converting an existing program to use it, you should remember virtual memory's limitations. Although virtual memory allows a program to store a large amount of data, only a small portion of it is immediately accessible at any one time. Your program may read and write a large amount of data at all times, which requires the virtual memory manager to perform a lot of swapping. In such a case, your program's performance is not as efficient as the performance of a program that accesses only a small amount of data at a time.

Another way to make your program use extended memory, expanded memory, and disk storage is to break it up into overlays. For information on creating overlays, see Chapter 3, "Creating Overlaid MS-DOS Programs," in *Command-Line Utilities User's Guide*.

CHAPTER 3

Managing Memory for 16-Bit C++ Programs

Chapter 2, "Managing Memory for 16-Bit C Programs," describes how you can most efficiently use memory to optimize your C program. This chapter describes how to manage memory in your C++ program.

This chapter explains:

- How memory models apply to classes
- How to control the addressing of dynamically created objects
- How to control the placement of member functions using the **__based** keyword

You should be familiar with the C++ language before reading this chapter (for information on the C++ language, see the *C++ Tutorial* and the *C++ Language Reference*). You should also have read the material in Chapter 2, "Managing Memory for 16-Bit C Programs."

The material covered in this chapter is relevant only to 16-bit programs.

3.1 Memory Models for Classes

To understand how memory models apply to classes, it's necessary to know how objects are represented in memory.

Member functions are stored once for the entire class.

Each object contains its own copy of the data members defined for its class (except for static members). However, an object does not contain its own copy of the code for the member functions. Member functions are stored only once for the entire class.

When you call a member function of a particular object, the address of that object is passed to that function as a hidden argument. For example, the C++ statement

```
myWindow.resize();
```

is analogous to the C statement

```
resize( &myWindow );
```

The member function implicitly uses the address to access the object's data members. That address is available from inside the member functions as the **this** pointer.

The Ambient Memory Model

Microsoft Visual C++ assigns a memory model to every class, known as the "ambient" memory model for that class. The ambient model of a class affects several characteristics of the objects of that class:

- The address space in which an object resides
- The address mode of a pointer or a reference to an object
- The address mode of the **this** pointer used in member functions of that class

The default ambient model for all classes is the model you specify for data at compilation. In the tiny, small, or medium memory models, all objects reside in the default data segment and all pointers and references to objects are near. In the compact, large, and huge memory models, objects can reside in any segment and all pointers and references to objects are far.

You can declare a particular class to have an ambient model other than the default by specifying **__near** or **__far** before the class name. For example, the following declaration specifies `Node` as a far class:

```
class __far Node
{
public:
    Node();
    void print();
    ~Node();
private:
   // ...
};
```

Objects of class `Node` can be stored in any data segment, no matter what memory model the program is compiled with. Pointers and references to `Node` objects are automatically far, and a far address is passed whenever a member function is called. For example,

```
Node  head;          // Far allocation
Node *pNode;         // Far pointer

head.print();        // Far address passed to print()
pNode = &head;       // Far address taken
```

If you explicitly specify an ambient model for a class, it must match that of its base class. If a class has multiple base classes, all of them must have the same ambient model. If you don't explicitly specify the **__near** or **__far** keyword in a class's declaration, the class inherits the ambient model of its base class or classes. If the class has no base class or classes, it uses the default model implied by the memory model for the entire module.

Overriding the Ambient Memory Model

You can override a class's ambient model when you declare an individual object or a pointer to an object. Place one of the addressing keywords before the identifier:

```
Node __near myNode;
Node __near *npNode;

myNode.print();    // Near address converted to far when print() called
npNode = &myNode;  // Near address taken
```

In this example, `myNode` is an object of type `Node` stored in the default data segment. Taking the address of `myNode` produces a near pointer. Whenever a member function is invoked for `myNode`, the object's near address is converted to the far address that the function expects.

You can use the **__near**, **__far**, or **__huge** keyword when you declare an object. You can also use a **__based** expression, as long as you base the object on a segment constant or a segment variable. You cannot base data on a pointer, **void**, or the **__self** segment.

When you override the class's ambient model for an individual object, the address of that object has a different addressing mode from that expected by the class's member functions. In such cases, the compiler attempts to perform a type conversion on the address of the object.

In the previous example, the address of `myNode` is automatically converted from a near address to a far address before it is passed to the `Node` constructor. The same thing happens when `print()` is called.

However, consider the following class definition:

```
class __near RecArray
{
public:
    RecArray( int size );
    void printNames();
    ~RecArray();
private:
    // ...
};
```

In this case, the declaration and statement

```
RecArray __far bigArray( 5000 );      // Error: constructor
                                      //   expects near address

bigArray.printNames();                // Error: printNames()
                                      //   expects near address
```

result in type conversion errors, because the compiler cannot convert the far address of `bigArray` into the near address expected by the constructor and the `printNames()` member function.

Overloading the this Pointer

If the standard conversions do not allow your objects to be passed to member functions, you must overload the member functions on the addressing mode of the **this** pointer. To specify the addressing mode for a member function, place the **__near**, **__far**, or **__huge** keyword after its parameter list. For example,

```
class __near RecArray
{
public:
    RecArray();
    RecArray() __far;
    void printNames();
    void printNames() __far;
    ~RecArray();
    ~RecArray() __far;
private:
    // ...
};
```

Now when you declare a far `RecArray` object, the far constructor is called. Similarly, if you call the `printNames()` member function for that far object, the far `printNames()` function is invoked.

The keyword (**__near**, **__far**, or **__huge**) following the function name describes the addressing mode of the **this** pointer within the member function. When you call a member function for an object, if the standard conversions can match your call statement to more than one function, the function with the best match is selected. You cannot declare a member function as having a based **this** pointer.

It is not required that you overload all the member functions on the **this** pointer. You only need to overload member functions that are called for objects that don't have the addressing mode specified for the class.

Specifying the Addressing Mode of Return Objects

When you specify the return type of a function, you usually specify an addressing mode only if the function returns a pointer. You don't specify an addressing mode if the function returns a built-in type. For example,

```
char __far *func1();      // Return a far pointer to a char
__far char func2();       // Error: meaningless
```

In the declaration of `func1()`, the **__far** keyword modifies the pointer being returned. In the declaration of `func2()`, the **__far** keyword modifies the character being returned, which is illegal.

However, if a function returns an object, you can specify an addressing mode. With C++, you can invoke member functions for the temporary object returned by a function, even if you don't assign the object to another variable. For example,

```
RecArray makeArray( FILE *handle );  // Function returning
                                     //   an object

main()
{
    makeArray( currFile ).printNames();  // printNames() invoked
                                         //   for temporary object
}
```

The `RecArray` object returned by `makeArray()` is not assigned to another object and thus cannot be referenced after that line of code. It is used only to call the `printNames()` member functions.

You might need to specify the addressing mode of the object returned, if the member function you call accepts only one addressing mode. Consider the following declaration:

```
class __far RecArray
{
public:
    RecArray();
    void printNames();
    void printAll() __near;
    ~RecArray();
private:
    // ...
};
```

The member function `printNames()` can be called for far `RecArray` objects and for near `RecArray` objects through type conversion. However, the member function `printAll()` can be called only for near `RecArray` objects. Given the previous declaration of `makeArray()`, the statement

```
makeArray( currFile ).printAll();
```

is an error, because the compiler creates a far temporary `RecArray` object, and its address cannot be converted to the near address that `printAll()` expects.

To specify the addressing mode of a function's return type, place the **class**, **struct**, or **union** keyword and the addressing mode keyword (**__near**, **__far**, or **__huge**) before the return type, as follows:

```
class __near RecArray makeArray( FILE *handle );
```

This specifies that `makeArray()` returns a near `RecArray` object. This declaration lets you call `printAll()` for the return object. Note that this syntax can be used only for functions that return an instance of a user-defined type, not for functions that return built-in types.

Virtual Table Pointers

If a class uses virtual functions, the compiler builds an array of function pointers for that class. This array is known as a virtual function table, or a "v-table." Every object of such a class contains a hidden member called a "v-table pointer." When

you call a virtual function for an object, your program uses that object's v-table pointer to find the v-table and then looks in the v-table to find the address of the function that must be called.

Similarly, if a class inherits from a virtual base class the compiler builds an array containing the offsets of the virtual bases. This array is called the virtual base displacement table, or "v-base table." Every object of such a class contains a hidden member called a "v-base table pointer." When you access one of an object's data members that was defined by the virtual base, your program uses that object's v-base table pointer to find the v-base table and then looks in the table to find the offset of the virtual base.

V-table pointers' addressing mode is determined by the class memory model.

The addressing mode of these virtual table pointers is determined by the memory model of the class. The virtual tables of a near class are stored in the default data segment, and objects of that class have near virtual table pointers. The virtual tables of a far class are stored in an anonymous far segment in the TEXT group. Objects of far classes have far virtual table pointers. These characteristics cannot be overriden by specifying a memory model keyword in the declaration of an individual object.

You can use the /NV option to specify the name of the segment in which the virtual tables are stored. For near classes, the segment specified must be one of the segments in DGROUP. For far classes, any segment can be specified.

3.2 The Free Store

The free store in C++ corresponds to the heap in C; it provides the memory for objects created at run time. In Visual C++, the operators **new** and **delete** have been overloaded so you can allocate and deallocate near, far, and huge objects and objects based on a segment variable. These operators are similar to the **malloc** and **free** functions in C.

The new Operator

Visual C++ has four versions of the **new** operator, which allocate objects in the near, far, huge, and based address spaces. The **new** operator is the only operator or function that can be overloaded on its return type; the only overloading allowed for the return type is on the addressing mode.

By default, the return type of the **new** operator depends on the memory model under which the program was compiled. For example, in the small and medium memory models, **new** returns objects in the near address space.

If you explicitly specify an ambient model for a class, the **new** operator uses that address space when allocating objects of that class. For example,

```
class __far Node     // Class is far
{
};

Node *pN;

main()
{
    pN = new Node;  // Far allocation, even if program is
                    //   compiled with a memory model that
                    //   uses near data
}
```

You can override both the program's memory model and the class's ambient model by explicitly specifying the address space of the object being allocated. To do so, place the **__near**, **__far**, **__huge**, or **__based** keyword after the name of the type. You must use a segment variable with the **__based** keyword. All the standard conversions between pointers apply, as described in the *C++ Language Reference*. The following example shows how you can use the various forms of **new**:

```
class Node
{
};

Node *pN;                   // Depends on default memory model
Node __near *npN;
Node __far *fpN;
Node __huge *hpN;
__segment segvar;
Node __based(segvar) *bpN;

main()
{
    pN =  new Node;                   // Depends on default memory model
    npN = new __near Node;
    fpN = new __far Node;
    hpN = new __huge Node;
    segvar = _bheapseg( 1000 );
    bpN = new __based(segvar) Node;

    fpN = new __near Node;            // Convert near to far
    fpN = new __based(segvar) Node;   // Convert based to far
    npN = new __far Node;             // Error: cannot convert
                                      //   from far to near
}
```

You can write your own version of the **new** operator if you want to use a customized memory allocation scheme (for instance, one that provides zero-initialized storage or one that's optimized for your program's pattern of memory usage). The **new** operator that you define must have the same return type and arguments as the one you want to replace. To do this, you must use one of the following prototypes:

```
void __near *operator new( size_t size );
void __far *operator new( size_t size );
void __huge *operator new( unsigned long elems, size_t size );
void __based(void) *operator new( __segment segvar, size_t size );
```

The argument of type **size_t** is automatically set to the size of the object being allocated. You can also define class-specific versions of any of these forms of **new**.

The **new** operator for huge objects behaves as an array allocator does.

The **new** operator for huge objects behaves as an array allocator does, even if only one object is being allocated. It receives two arguments: the number of elements being allocated and the size of each element. If the total size of the array is larger than 128K, the element size must be a power of two.

The **new** operator for based objects has an additional argument, which is a segment variable. This argument receives the value of the segment used in the allocation expression.

When you redefine the **new** operator, you can make your version of the operator accept additional arguments, known as "placement arguments." These arguments must appear last when you declare **new**'s argument list, but they must appear before the type name and in parentheses when you call **new**. For example, to define a **new** operator for based objects that takes a short integer as a placement argument, you use the following prototype:

```
void __based(void) *operator new( __segment segvar, size_t size,
                                  short place );
```

This prototype permits expressions such as:

```
bpN = new __based(segvar) (112) Node;
```

The placement argument receives the value 112 as a short integer.

All of the default **new** operators have the **__cdecl** calling convention.

The delete Operator

The **delete** operator is overloaded to accept pointers that are near, far, huge, or based. When you delete an object, the addressing mode of the pointer determines which **delete** operator is invoked. Thus, the following example invokes four different **delete** operators:

```
npN = new Node __near;
fpN = new Node __far;
hpN = new Node __huge;
bpN = new Node __based(segvar);

delete npN;                    // Invokes near delete
delete fpN;                    // Invokes far delete
delete hpN;                    // Invokes huge delete
delete bpN;                    // Invokes based delete
```

The addressing mode of the pointer does not necessarily indicate the address space of the object. For example,

```
Node __far *fpN;

fpN = new Node __near;  // Type conversion: near to far

delete fpN;             // Error: far delete invoked for near object
```

In this example, the compiler chooses the inappropriate **delete** operator for the pointer, which results in a run-time error. To prevent this problem, you must explicitly cast the pointer to the desired addressing mode:

```
delete (Node __near *)fpN;
```

You must always ensure that the **delete** operator invoked corresponds to the **new** operator used to allocate the object.

Just as with the **new** operator, you can write your own version of the **delete** operator to implement a customized memory-allocation scheme. If you want to implement different behavior for the different versions of the **delete** operator, you must use one of the following prototypes:

```
void operator delete( void __near *nptr );
void operator delete( void __far *fptr );
void operator delete( void __huge *hptr );
void operator delete( __segment segvar, void __based(void) *bptr );
```

You can also define class-specific versions of any of these forms of **delete**. When defining a class-specific version, you can specify an optional final argument of type

size_t. If present, the argument is automatically set to the size of the object being deleted. You cannot define two versions of **delete** that are distinguished only by the **size_t** argument; that is, you cannot overload the **delete** operator for a given addressing mode within class scope. However, you can define versions of **delete** that have the same addressing mode but different scopes—that is, one with global scope and one with class scope.

All the default versions of **delete** have the **__cdecl** calling convention.

The _set_new_handler Function

Using Visual C++, you can specify what actions should be taken when the free store is exhausted. You do this by defining an error-handling function and passing it to the **_set_new_handler** function, defined in the include file NEW.H. Whenever the **new** operator supplied by the compiler cannot allocate the memory requested, it checks to see if an error handler has been installed. If an error handler is defined, **new** calls it; otherwise, **new** simply returns zero. You can write a simple error handler that prints an error message, performs some cleanup tasks, and then exits the program, or you can write a more sophisticated error handler that attempts to recover memory so that **new** can retry the allocation.

The error handler you write must take the same arguments as the **new** function that invokes it. For the near or far free stores, the error handler must take one argument of type **size_t**, indicating the amount of memory requested, and return an integer. For the huge free store, the error handler must take an argument of type **unsigned long**, indicating the number of elements being allocated, and one of type **size_t**, indicating the size of each element. An error handler for the based free store must take an additional argument of type **__segment**, indicating the segment.

All error handlers require the **__cdecl** calling convention.

The error handler should return a zero if it is unable to recover the amount of memory requested. Otherwise, it should return a nonzero value. The **__cdecl** calling convention is required for all error handlers.

The following examples are sample prototypes for error handlers:

```
int my_near_handler( size_t size );
int my_far_handler( size_t size );
int my_huge_handler( unsigned long elems, size_t size );
int my_based_handler( __segment segvar, size_t size );
```

The **_set_new_handler** function maps onto either the **_set_nnew_handler** or **_set_fnew_handler** function, depending on the program's memory model. You can also call these functions explicitly, or you can call the corresponding functions for the huge and based free stores. All of these functions return a pointer to the previously installed error handler, or a **NULL** if no handler was installed.

The following are prototypes for the functions that install the various error handlers. The types _PNH, _PNHH, and _PNHB are **typedef**s for pointers to the error-handling functions.

```
_PNH __cdecl _set_nnew_handler( _PNH handler );
_PNH __cdecl _set_fnew_handler( _PNH handler );
_PNHH __cdecl _set_hnew_handler( _PNHH handler );
_PNHB __cdecl _set_bnew_handler( _PNHB handler );
```

If the error handler returns a nonzero value, the **new** operator supplied by the compiler tries the allocation again. If the allocation fails again, **new** calls the error handler again. This continues until the error handler returns zero or until the allocation succeeds.

In multiprocess, multithreaded environments, separate error handlers exist for each process and thread. No handlers are preinstalled when a process begins. When a thread starts, it gets copies of its parent's handlers for all free stores.

3.3 Based Addressing for Member Functions

Just as you can declare ordinary functions as based, you can also declare member functions as based. This is useful if you declare virtual functions as **__near**, which requires that they be called from within the same segment they reside in. Note that a base class's definition of a function and a derived class's definition may reside in separate files.

For example, consider the following program:

```
// FILE 1 - Compiled under large model
class Shape
{
    // ...
    virtual void __near redraw();
}

void __near Shape::redraw()
{
    // ...
}

void __far test( Shape *currShape )
{
    currShape->redraw();              // Invoke virtual function
}
```

In this example, you can safely declare `redraw` as a near function, because it is in the same file as the function that calls it, and the compiler places it in the same segment. However, a derived class may be declared in another file, as follows:

```
// FILE 2 - Compiled under large model
class Circle : public Shape
{
    // ...
    void __near redraw();
}

void __near Circle::redraw()
{
    // ...
}
```

Because this program is compiled under large model, each file defines a separate code segment. The `test` function cannot perform a near call to a `redraw` function in another segment. As a result, the `test` function may either succeed or fail, depending on whether its argument is an instance of a base class or an instance of a derived class. For example,

```
Shape my_shape;
Circle my_circle;

test( &my_shape );   // Okay - Shape::redraw in the same
                     //   segment as the test function
test( &my_circle );  // Error - Circle::redraw in different segment
```

The second function call causes a run-time error.

The easiest way to avoid this problem is to declare all your virtual functions as **__far**, so they can be called from any segment. However, if you want to declare your virtual functions as **__near**, you can avoid problems by declaring the virtual function to be based in the same segment as `test`:

```
virtual void __near __based(__segname("MYSEG")) redraw();

void __far __based(__segname("MYSEG")) test( Shape *curr_shape );
```

The **__based** expression remains in effect through all subsequent redefinitions of `redraw`. Derived classes can define their own versions of `redraw`, and they are stored in the same segment as the base class's version.

If redefinitions of a virtual function are declared as being based in a different segment, the compiler issues a warning.

CHAPTER 4

Using the 16-Bit Inline Assembler

This chapter explains how to use the Microsoft Visual C++ 16-bit inline assembler. Assembly language serves many purposes, such as improving program speed, reducing memory needs, and controlling hardware. Using the inline assembler, you can embed assembly-language instructions directly in your C and C++ source programs without extra assembly and link steps. The inline assembler is built into the compiler—you don't need a separate assembler such as the Microsoft Macro Assembler (MASM). For more information on the interaction between C and assembly language, see Chapter 8, "Programming with Mixed Languages."

4.1 Advantages of Inline Assembly

Because the inline assembler doesn't require separate assembly and link steps, it is more convenient than a separate assembler. Inline assembly code can use any C variable or function name that is in scope, so it is easy to integrate it with your program's C code. Because the assembly code can be mixed in line with C or C++ statements, it can do tasks that are cumbersome or impossible in C or C++.

The uses of inline assembly include:

- Writing functions in assembly language
- Spot-optimizing speed-critical sections of code
- Calling MS-DOS and BIOS routines with the **INT** instruction
- Creating terminate-and-stay-resident (TSR) code or handler routines that require knowledge of processor states

Inline assembly is a special-purpose tool. If you plan to port an application to different machines, you'll probably want to place machine-specific code in a separate module. Because the inline assembler doesn't support all of MASM's macro and data directives, you may find it more convenient to use MASM for such modules.

4.2 The __asm Keyword

The **__asm** keyword invokes the inline assembler and can appear wherever a C or C++ statement is legal. It cannot appear by itself. It must be followed by an assembly instruction, a group of instructions enclosed in braces, or, at the very least, an empty pair of braces. The term "**__asm** block" here refers to any instruction or group of instructions, whether or not in braces.

Following is a simple **__asm** block enclosed in braces. The code prints the "beep" character, ASCII (American Standard Code for Information Interchange) 7.

```
__asm
{
    mov ah, 2
    mov dl, 7
    int 21h
}
```

Alternatively, you can put **__asm** in front of each assembly instruction:

```
__asm mov ah, 2
__asm mov dl, 7
__asm int 21h
```

Because the **__asm** keyword is a statement separator, you can also put assembly instructions on the same line:

```
__asm mov ah, 2   __asm mov dl, 7   __asm int 21h
```

Braces can prevent ambiguity and needless repetition.

All three examples generate the same code, but the first style (enclosing the **__asm** block in braces) has some advantages. The braces clearly separate assembly code from C or C++ code and avoid needless repetition of the **__asm** keyword. Braces can also prevent ambiguities. If you want to put a C or C++ statement on the same line as an **__asm** block, you must enclose the block in braces. Without the braces, the compiler cannot tell where assembly code stops and C or C++ statements begin. Finally, because the text in braces has the same format as ordinary MASM text, you can easily cut and paste text from existing MASM source files.

The braces enclosing an **__asm** block don't affect variable scope, as do braces in C and C++. You can also nest **__asm** blocks; nesting does not affect variable scope.

4.3 Using Assembly Language in __asm Blocks

The inline assembler has much in common with other assemblers. For example, it accepts most expressions that are legal in MASM (for more information, see "Expressions," following). This section describes the use of assembly-language features in **__asm** blocks.

Instruction Set

The 16-bit inline assembler supports the full instruction set of the Intel 80286 and 80287 processors. To use 80286 or 80287 instructions, compile with the /G2 option.

Expressions

Inline assembly code can use most MASM expressions, which are combinations of operands and operators that evaluate to a single value or address. In an inline assembler expression, you cannot use more than one user-defined symbol per operand. Expressions such as the following are not supported:

```
__asm
{
    label1: pushf
    label2: iret
    mov     ax, (offset label2) - (offset label1)
}
```

Data Directives and Operators

Although an **__asm** block can reference C or C++ data types and objects, it cannot define data objects with MASM directives or operators. Specifically, you cannot use the definition directives **DB**, **DW**, **DD**, **DQ**, **DT**, and **DF**, or the operators **DUP** and **THIS**. MASM structures and records are also unavailable. The inline assembler doesn't accept the directives **STRUC**, **RECORD**, **WIDTH**, or **MASK**.

EVEN and ALIGN Directives

Although the inline assembler doesn't support most MASM directives, it does support **EVEN** and **ALIGN**. These directives put **NOP** (no operation) instructions in the assembly code as needed to align labels to specific boundaries. This makes instruction-fetch operations more efficient for some processors (not including 8-bit processors such as the Intel 8088).

Macros

The inline assembler is not a macro assembler. You cannot use MASM macro directives (**MACRO**, **REPT**, **IRC**, **IRP**, and **ENDM**) or macro operators (**<>**, **!**, **&**, **%**, and **.TYPE**). An **__asm** block can use C preprocessor directives, however. For more information, see "Using C and C++ in __asm Blocks," on page 87.

Segment References

You must refer to segments by register rather than by name (the segment name **_TEXT** is invalid, for instance). Segment overrides must use the register explicitly, as in `es:[bx]`.

Type and Variable Sizes

The **LENGTH**, **SIZE**, and **TYPE** operators have a limited meaning in inline assembly. They cannot be used at all with the **DUP** operator (because you cannot define data with MASM directives or operators). But you can use them to find the size of C or C++ variables or types:

- The **LENGTH** operator can return the number of elements in an array. It returns 1 for nonarray variables.
- The **SIZE** operator can return the size of a C or C++ variable. A variable's size is the product of its **LENGTH** and **TYPE**.
- The **TYPE** operator can return the size of a C or C++ type or variable. If the variable is an array, **TYPE** returns the size of a single element of the array.

For example, if your program has an eight-element **int** array,

```
int arr[8];
```

the following C and assembly expressions yield the size of `arr` and its elements:

__asm	C	Size
`LENGTH arr`	`sizeof(arr)/sizeof(arr[0])`	8
`SIZE arr`	`sizeof (arr)`	16
`TYPE arr`	`sizeof(arr[0])`	2

Comments

Instructions in an **__asm** block can use assembly-language comments:

```
__asm mov ax, offset buff ; Load address of buff
```

Because C macros expand into a single logical line, avoid using assembly-language comments in macros (see "Defining __asm Blocks as C Macros" on page 96). An **__asm** block can also contain C-style comments, as noted following.

The _emit Pseudoinstruction

The **_emit** pseudoinstruction is similar to the **DB** directive of MASM. Using it, you can define a single immediate byte at the current location in the current text segment. However, **_emit** can define only one byte at a time, and it can only define bytes in the text segment. It uses the same syntax as the **INT** instruction.

One use for **_emit** is to define 80386-specific and 80486-specific instructions, which the inline assembler does not support. The following example, for instance, defines the 80386 **CWDE** instruction:

```
/* Assumes 16-bit mode */
#define cwde __asm _emit 0x66 __asm _emit 0x98
 .
 .
 .
__asm {
    cwde
    }
```

Debugging and Listings

Programs containing inline assembly code can be debugged with the Microsoft CodeView debugger, assuming you compile with the /Zi option.

Inline assembly code can be debugged with CodeView.

Within CodeView, you can set breakpoints on both C or C++ and assembly-language lines. If you enable mixed assembly and source mode, you can display both the source and disassembled form of the assembly code.

Note that putting multiple assembly instructions or source language statements on one line can hamper debugging with CodeView. Using the CodeView debugger in source mode, you can set breakpoints on a single line but not on individual statements on the same line. The same principle applies to an **__asm** block defined as a C macro, which expands to a single logical line.

If you create a mixed source and assembly listing with the /Fc compiler option, the listing contains both the source and assembly forms of each assembly-language line. Macros are not expanded in listings, but they are expanded during compilation.

For more information, see the *Command-Line Utilities User's Guide*.

4.4 Using C or C++ in __asm Blocks

Because inline assembly instructions can be mixed with C or C++ statements, they can refer to C or C++ variables by name and use many other elements of those languages.

An **__asm** block can use the following language elements:

- Symbols, including labels and variable and function names
- Constants, including symbolic constants and **enum** members
- Macros and preprocessor directives
- Comments (both `/* */` and `//`)
- Type names (wherever a MASM type is legal)
- **typedef** names, generally used with operators such as **PTR** and **TYPE** or to specify structure or union members

Within an **__asm** block, you can specify integer constants with either C notation or assembler radix notation (0x100 and 100h are equivalent, for example). Using this feature, you can define (using **#define**) a constant in C and then use it in both C or C++ and assembly portions of the program. You can also specify constants in octal by preceding them with a 0. For example, 0777 specifies an octal constant.

Using Operators

An **__asm** block cannot use C or C++ specific operators, such as the << operator. However, operators shared by C and MASM, such as the * operator, are interpreted as assembly-language operators. For instance, outside an **__asm** block, square brackets ([ ]) are interpreted as enclosing array subscripts, which C automatically scales to the size of an element in the array. Inside an **__asm** block, they are seen as the MASM index operator, which yields an unscaled byte offset from any data object or label (not just an array). The following code illustrates the difference:

```
int array[10];

__asm mov array[6], bx ;  Store BX at array+6 (not scaled)

array[6] = 0;          /* Store 0 at array+12 (scaled) */
```

The first reference to `array` is not scaled, but the second is. Note that you can use the **TYPE** operator to achieve scaling based on a constant. For example, the following statements are equivalent:

```
__asm mov array[6 * TYPE int], 0 ; Store 0 at array + 12

array[6] = 0;                    /* Store 0 at array + 12 */
```

Using C or C++ Symbols

An **__asm** block can refer to any C or C++ symbol in scope where the block appears. (C and C++ symbols are variable names, function names, and labels—that is, names that aren't symbolic constants or **enum** members. You cannot call C++ member functions.)

A few restrictions apply to the use of C and C++ symbols:

- Each assembly-language statement can contain only one C or C++ symbol. Multiple symbols can appear in the same assembly instruction only with **LENGTH**, **TYPE**, and **SIZE** expressions. You can also use two symbols if one is a register variable.
- Functions referenced in an **__asm** block must be declared (prototyped) earlier in the program. Otherwise, the compiler cannot distinguish between function names and labels in the **__asm** block.

- An **__asm** block cannot use any C or C++ symbols with the same spelling as MASM reserved words (regardless of case). MASM reserved words include instruction names such as **PUSH** and register names such as SI.
- Structure and union tags are not recognized in **__asm** blocks.

Accessing C or C++ Data

A great convenience of inline assembly is the ability to refer to C or C++ variables by name. An **__asm** block can refer to any symbols, including variable names, that are in scope where the block appears. For instance, if the C variable `var` is in scope, the instruction

```
__asm mov ax, var
```

stores the value of `var` in AX.

If a class, structure, or union member has a unique name, an **__asm** block can refer to it using only the member name, without specifying the variable or **typedef** name before the period (.) operator. If the member name is not unique, however, you must place a variable or **typedef** name immediately before the period (.) operator. For example, the following structure types share `same_name` as their member name:

```
struct first_type
{
    char *weasel;
    int same_name;
};

struct second_type
{
    int wonton;
    long same_name;
};
```

If you declare variables with the types

```
struct first_type hal;
struct second_type oat;
```

all references to the member `same_name` must use the variable name, because `same_name` is not unique. But the member `weasel` has a unique name, so you can refer to it using only its member name:

```
__asm
{
    mov bx, OFFSET hal
    mov cx, [bx]hal.same_name ; Must use 'hal'
    mov si, [bx].weasel       ; Can omit 'hal'
}
```

Note that omitting the variable name is merely a coding convenience. The same assembly instructions are generated whether or not the variable name is present.

You can access data members in C++ without regard to access restrictions. However, you cannot call member functions.

Writing Functions

If you write a function with inline assembly code, it's easy to pass arguments to the function and return a value from it. The following examples compare a function first written for a separate assembler and then rewritten for the inline assembler. The function, called `power2`, receives two parameters, multiplying the first parameter by two to the power of the second parameter. Written for a separate assembler, the function might look like this:

```
; POWER.ASM
; Compute the power of an integer
;
        PUBLIC _power2
_TEXT SEGMENT WORD PUBLIC 'CODE'
_power2 PROC

        push bp          ; Save BP
        mov bp, sp       ; Move SP into BP so we can refer
                         ;   to arguments on the stack
        mov ax, [bp+4]   ; Get first argument
        mov cx, [bp+6]   ; Get second argument
        shl ax, cl       ; AX = AX * ( 2 ^ CL )
        pop bp           ; Restore BP
        ret              ; Return with sum in AX

_power2 ENDP
_TEXT   ENDS
        END
```

Function arguments are usually passed on the stack.

Because it's written for a separate assembler, the function requires a separate source file and assembly and link steps. C and C++ function arguments are usually passed on the stack, so this version of the `power2` function accesses its arguments by their positions on the stack. (Note that the **MODEL** directive, available in MASM and some other assemblers, also makes it possible to access stack arguments and local stack variables by name.)

The POWER2.C program following writes the `power2` function with inline assembly code:

```
/* POWER2.C */
#include <stdio.h>

int power2( int num, int power );

void main( void )
{
    printf( "3 times 2 to the power of 5 is %d\n", \
            power2( 3, 5) );
}

int power2( int num, int power )
{
    __asm
    {
        mov ax, num     ; Get first argument
        mov cx, power   ; Get second argument
        shl ax, cl      ; AX = AX * ( 2 to the power of CL )
    }
    /* Return with result in AX */
}
```

The inline version of the `power2` function refers to its arguments by name and appears in the same source file as the rest of the program. This version also requires fewer assembly instructions. Because C automatically preserves BP, the **__asm** block doesn't need to do so. It can also dispense with the **RET** instruction, because the C part of the function performs the return.

Because the inline version of `power2` doesn't execute a C **return** statement, it causes a harmless warning if you compile at warning levels two or higher:

```
warning C4035: 'power2' : no return value
```

The function does return a value, but the compiler cannot tell that in the absence of a **return** statement. Simply ignore the warning in this context.

4.5 Using and Preserving Registers

In general, you should not assume that a register has a given value when an **__asm** block begins. An **__asm** block inherits whatever register values happen to result from the normal flow of control.

If you use the **__fastcall** calling convention, the compiler passes function arguments in registers instead of on the stack. This can create problems in functions with **__asm** blocks, because a function has no way to tell which parameter is in which register. If the function happens to receive a parameter in AX and immediately stores something else in AX, the original parameter is lost.

Don't use the **__fastcall** calling convention for functions with **__asm** blocks.

To avoid such register conflicts, don't use the **__fastcall** convention for functions that contain an **__asm** block. If you specify the **__fastcall** convention globally with the /Gr compiler option, declare every function containing an **__asm** block with the attribute **__cdecl** or **__pascal**. (The **__cdecl** attribute tells the compiler to use the C calling convention for that function. The **__pascal** attribute tells the compiler to use the FORTRAN/Pascal convention, which is the default for C++ functions.) If you are not compiling with /Gr, avoid declaring the function with **__fastcall**.

As you may have noticed in the POWER2.C example in "Writing Functions," on page 90, the `power2` function doesn't preserve the value in the AX register. When you write a function in assembly language, you don't need to preserve the AX, BX, CX, DX, ES, and flags registers. However, you should preserve any other registers you use (DI, SI, DS, SS, SP, and BP).

Note If your inline assembly code changes the direction flag using the STD or CLD instructions, you must restore the flag to its original value.

Functions return small values in the AX and DX registers.

The POWER2.C example in "Writing Functions," on page 90, also shows that functions return values in registers. This is true for return values that are 4 bytes or smaller (except for structures), whether the function is written in assembly language or in C or C++.

If the return value is short (a **char**, **int**, or **near** pointer), it is stored in AX. The POWER2.C example returned a value by terminating with the desired value in AX.

If the return value is long, store the high word in DX and the low word in AX. To return a longer value (such as a floating-point value), store the value in memory and return a pointer to the value (in AX if **near** or in DX:AX if **far**).

Assembly instructions that appear inline with C or C++ statements are free to alter the AX, BX, CX, and DX registers. C and C++ don't expect these registers to be maintained between statements, so you don't need to preserve them. The same is true of the SI and DI registers, with some exceptions (see "Optimizing," on page 97). You should preserve the SP and BP registers unless you have some reason to change them—to switch stacks, for example.

4.6 Using Floating-Point Instructions

You can use most of the 80387 floating-point instructions in an **__asm** block. However, the 16-bit floating-point emulator libraries, *x*LIBCE.LIB, do not support 14 80387-specific floating-point instructions. Use of these instructions in inline assembler code causes an error on any machine that does not have a coprocessor. These instructions fall into two categories: unemulated 80*x*87 instructions and those specific to the 80387.

Unemulated 80x87 Instructions

If a program was linked with any of the 16-bit floating-point emulator libraries (*x*LIBCE.LIB) and contains the **fldenv**, **fstenv**, **fsave**, **frstor**, **fbld**, **fbstp**, or **fnop** instruction, it fails with the following run-time error if it is run on a machine that does not have a coprocessor:

```
"run-time error M6107 : MATH - floating-point error : unemulated"
```

Instructions Specific to the 80387

The 16-bit compiler does not accept the **fsin**, **fcos**, **fsincos**, **fucom**, **fucomp**, **fucompp**, or **fprem1** 80387 coprocessor instruction.

4.7 Jumping to Labels

Like an ordinary C or C++ label, a label in an **__asm** block has scope throughout the function in which it is defined (not only in the block). Both assembly instructions and **goto** statements can jump to labels inside or outside the **__asm** block.

Labels in **__asm** blocks have function scope and are not case sensitive.

Labels defined in **__asm** blocks are not case sensitive; both **goto** statements and assembly instructions can refer to those labels without regard to case. C and C++ labels are case sensitive only when used by **goto** statements. Assembly instructions can jump to a C or C++ label without regard to case.

The following do-nothing code shows all the permutations:

```
void func( void )
{
    goto C_Dest;  /* Legal: correct case   */
    goto c_dest;  /* Error: incorrect case */

    goto A_Dest;  /* Legal: correct case   */
    goto a_dest;  /* Legal: incorrect case */
```

```
    __asm
    {
        jmp C_Dest ; Legal: correct case
        jmp c_dest ; Legal: incorrect case

        jmp A_Dest ; Legal: correct case
        jmp a_dest ; Legal: incorrect case

        a_dest:    ; __asm label
    }

    C_Dest:       /* C label */
    return;
}
```

Don't use C library function names as labels in **__asm** blocks. For instance, you might be tempted to use `exit` as a label, as follows:

```
; BAD TECHNIQUE: using library function name as label
jmp exit
    .
    .
    .
exit:
    ; More __asm code follows
```

Because **exit** is the name of a C library function, this code might cause a jump to the **exit** function instead of to the desired location.

As in MASM programs, the dollar symbol ($) serves as the current location counter. It is a label for the instruction currently being assembled. In **__asm** blocks, its main use is to make long conditional jumps:

```
jne $+5 ; next instruction is 5 bytes long
jmp farlabel
; $+5
    .
    .
    .
farlabel:
```

4.8 Calling C Functions

An **__asm** block can call C functions, including C library routines. The following example calls the **printf** library routine:

```
#include <stdio.h>

char format[] = "%s %s\n";
char hello[] = "Hello";
char world[] = "world";

void main( void )
{
    __asm
    {
        mov  ax, offset world
        push ax
        mov  ax, offset hello
        push ax
        mov  ax, offset format
        push ax
        call printf
    }
}
```

Because function arguments are passed on the stack, you simply push the needed arguments—string pointers, in the previous example—before calling the function. The arguments are pushed in reverse order, so they come off the stack in the desired order. To emulate the C statement

```
printf( format, hello, world );
```

the example pushes pointers to `world`, `hello`, and `format`, in that order, then calls **printf**.

4.9 Calling C++ Functions

An **__asm** block can call only global C++ functions that are not overloaded. If you call an overloaded global C++ function or a C++ member function, the compiler issues an error.

You can also call any functions declared with **extern "C"** linkage. This allows an **__asm** block within a C++ program to call the C library functions, because all the standard header files declare the library functions to have **extern "C"** linkage.

4.10 Defining __asm Blocks as C Macros

C macros offer a convenient way to insert assembly code into your source code, but they demand extra care because a macro expands into a single logical line. To create trouble-free macros, follow these rules:

- Enclose the **__asm** block in braces.
- Put the **__asm** keyword in front of each assembly instruction.
- Use old-style C comments (`/* comment */`) instead of assembly-style comments (`; comment`) or single-line C comments (`// comment`).

To illustrate, the following example defines a simple macro:

```
#define BEEP __asm      \
/* Beep sound */        \
{                       \
    __asm mov ah, 2     \
    __asm mov dl, 7     \
    __asm int 21h       \
}
```

At first glance, the last three **__asm** keywords seem superfluous. They are needed, however, because the macro expands into a single line:

```
__asm /* Beep sound */ { __asm mov ah, 2 __asm mov dl, 7 __asm int 21h }
```

The third and fourth **__asm** keywords are needed as statement separators. The only statement separators recognized in **__asm** blocks are the newline character and **__asm** keyword. Because a block defined as a macro is one logical line, you must separate each instruction from the previous with **__asm**.

The braces are essential as well. If you omit them, the compiler can be confused by C or C++ statements on the same line to the right of the macro invocation. Without the closing brace, the compiler cannot tell where assembly code stops, and it sees C or C++ statements after the **__asm** block as assembly instructions.

Use C comments in **__asm** blocks written as macros.

Assembly-style comments that start with a semicolon (;) continue to the end of the line. This causes problems in macros because the compiler ignores everything after the comment, all the way to the end of the logical line. The same is true of single-line C or C++ comments (`// comment`). To prevent errors, use old-style C comments (`/* comment */`) in **__asm** blocks defined as macros.

An **__asm** block written as a C macro can take arguments. Unlike an ordinary C macro, however, an **__asm** macro cannot return a value. So you cannot use such macros in C or C++ expressions.

Be careful not to invoke macros of this type indiscriminately. For instance, invoking an assembly-language macro in a function declared with the **__fastcall** convention may cause unexpected results. (For more information, see "Using and Preserving Registers," on page 92.)

You can convert MASM macros to C macros.

Note that some MASM-style macros can be written as C macros. Following is a MASM macro that sets the video page to the value specified in the `page` argument:

```
setpage   MACRO page
          mov ah, 5
          mov al, page
          int 10h
          ENDM
```

The following code defines `setpage` as a C macro:

```
#define setpage( page ) __asm     \
    {                             \
        __asm mov ah, 5           \
        __asm mov al, page        \
        __asm int 10h             \
}
```

Both macros do the same job.

4.11 Optimizing

The presence of an **__asm** block in a function affects optimization in several ways. First, the compiler doesn't try to optimize the **__asm** block itself. What you write in assembly language is exactly what you get.

Second, the presence of an **__asm** block affects register variable storage. Under normal circumstances (unless you suppress optimization with the /Od option), the compiler automatically stores variables in registers. This is not done, however, in any function that contains an **__asm** block. To get register variable storage in such a function, you must request it with the **register** keyword.

Because the compiler stores register variables in the SI and DI registers, these registers represent variables in functions that request register storage. The first eligible variable is stored in SI and the second in DI. Preserve SI and DI in such functions unless you want to change the register variables.

Keep in mind that the name of a variable declared with **register** translates directly into a register reference (assuming a register is available for such use). For example, if you declare

```
register int sample;
```

and the variable `sample` happens to be stored in SI, the **__asm** instruction

```
__asm mov ax, sample
```

is equivalent to

```
__asm mov ax, si
```

If you declare a variable with **register** and the compiler cannot store the variable in a register, the compiler issues a warning to that effect at compile time. You must remove the **register** declaration from that variable to get rid of the warning.

Register variables are the exception to the general rule that an assembly-language statement can contain no more than one C or C++ symbol. If one of the symbols is a register variable, for example,

```
register int v1;
int v2;
```

then an instruction can use two C or C++ symbols, as in

```
mov v1, v2
```

Finally, the presence of inline assembly code inhibits the following optimizations for the entire function in which the code appears:

- Loop (/Ol)
- Global register allocation (/Oe)
- Global optimizations and common subexpressions (/Og)

These optimizations are suppressed no matter which compiler options you use.

CHAPTER 5

Controlling Floating-Point Math Operations

This chapter describes how to control the way your Microsoft Visual C++ programs perform floating-point math operations. It describes the math packages that you can include in the C run-time libraries when you run the Setup program, then discusses the options you can specify on the CL command line to choose the appropriate library for linking and controlling floating-point instructions.

This chapter also explains how to override floating-point options by changing libraries at link time, and how to control use of the Intel math coprocessor (80*x*87) using the NO87 environment variable.

5.1 Declaring Floating-Point Types

Visual C++ supports three floating-point types that conform to the Institute of Electrical and Electronics Engineers, Inc. (IEEE) standard 754 format:

- Type **float**, a 32-bit floating-point quantity
- Type **double**, a 64-bit floating-point quantity
- Type **long double**, an 80-bit floating-point quantity (not supported in the alternate math package)

You can declare variables as any of these types. You can also declare functions that return any of these types.

Declaring Variables as Floating-Point Types

You can declare variables as **float**, **double**, or **long double**, depending on the needs of your application. The principal differences between the three types are the significance they can represent, the storage they require, and their range. Table 5.1 shows the relationship between significance and storage requirements.

Table 5.1 Floating-Point Types

Type	Significant digits	Number of bytes
float	6–7	4
double	15–16	8
long double	19	10

Floating-point variables are represented by a mantissa, which contains the value of the number, and an exponent, which contains the order of magnitude of the number.

Table 5.2 shows the number of bits allocated to the mantissa and the exponent for each floating-point type. The most-significant bit of any **float**, **double**, or **long double** is always the sign bit. If it is 1, the number is considered negative; otherwise, it is considered a positive number.

Table 5.2 Lengths of Exponents and Mantissas

Type	Exponent length	Mantissa length
float	8 bits	23 bits
double	11 bits	52 bits
long double	15 bits	64 bits

Because exponents are stored in an unsigned form, the exponent is biased by half its possible value. For type **float**, the bias is 127; for type **double**, it is 1023; for type **long double**, it is 16,383. You can compute the actual exponent value by subtracting the bias value from the exponent value.

The mantissa is stored as a binary fraction greater than or equal to one and less than two. For types **float** and **double**, there is an implied leading 1 in the mantissa in the most-significant bit position, so the mantissas are actually 24 and 53 bits long, respectively, even though the most-significant bit is never stored in memory.

Instead of the storage method just described, the floating-point package can store binary floating-point numbers as denormalized numbers. Denormalized numbers are nonzero floating-point numbers with reserved exponent values in which the most-significant bit of the mantissa is 0. By using the denormalized format, the range of a floating-point number can be extended at the cost of precision. You cannot control whether a floating-point number is represented in normalized or denormalized form; the floating-point package determines the representation. The floating-point packages never use denormalized form unless the exponent becomes less than the minimum that can be represented in a normalized form.

Table 5.3 shows the minimum and maximum value you can store in variables of each floating-point type. The values listed in this table apply only to normalized

floating-point numbers; denormalized floating-point numbers have a smaller minimum value. Note that numbers retained in 80*x*87 registers are always represented in 80-bit normalized form; numbers can only be represented in denormalized form when stored in 32- or 64-bit floating-point variables (variables of type **float** and type **long**).

Table 5.3 Range of Floating-Point Types

Type	Minimum value	Maximum value
float	1.175494351 E – 38	3.402823466 E + 38
double	2.2250738585072014 E – 308	1.7976931348623158 E + 308
long double	3.362103143112093503 E – 4932	1.189731495357231765 E + 4932

If precision is less of a concern than storage, consider using type **float** for floating-point variables. Conversely, if precision is the most important criterion, use type **long double**.

Visual C++ observes type-widening rules.

Floating-point variables can be promoted to a type of greater significance (for example, from type **float** to type **double**). Promotion often occurs when you perform arithmetic on floating-point variables. This arithmetic is always done in as high a degree of precision as the variable with the highest degree of precision. For example, consider the following type declarations:

```
float f_short;
double f_long;
long double f_longer;

f_short = f_short * f_long;
```

In the preceding example, the variable `f_short` is promoted to type **double** and multiplied by `f_long`; then the result is rounded to type **float** before being assigned to `f_short`.

In the following example (which uses the declarations from the preceding example), the arithmetic is done in **float** (32-bit) precision on the variables; the result is then promoted to type **long double**.

```
f_longer = f_short * f_short;
```

Declaring Functions That Return Floating-Point Types

You can declare functions that return the floating-point types **float**, **double**, and **long double**. Functions that return types **float** or **double** do not place their return values in registers; they place their return values in a global location called the floating-point accumulator (**__fac**). Functions that return the type **long double**

place their return values on the numeric data processor (NDP) stack, a simulated stack made up of registers in the math coprocessor.

In 32-bit programs, all functions that return floating-point values place their return values on the NDP stack. In addition, all functions that use the **__fastcall** calling convention and return floating-point values place their return values on the NDP stack.

You can write reentrant functions that return floating-point types.

Using the current thread's private stack to return values allows you to write reentrant functions by eliminating possible contention between threads for the floating-point accumulator.

Note You do not need to use the **__pascal** keyword with functions that return the type **long double**. There is no contention between threads for the NDP stack, because the operating system saves the values of the coprocessor's registers for each thread.

5.2 Run-Time Library Support of Type long double

Of the math packages offered by the Microsoft C and C++ compilers, only the emulator package and the math coprocessor package support the **long double** type; the alternate math package does not support it. In the math packages that support **long double**, each of the normal floating-point math functions has a special version that supports type **long double**. These functions have the same name as the functions that support type **float** and type **double**, except that they end with **l**. For example, the function that returns the absolute value of a variable of type **float** or type **double** is **fabs**. The **long double** equivalent function is **_fabsl**. The two exceptions to this rule are the **_atold** and **_strtold** functions.

5.3 Summary of Math Packages

The Microsoft C and C++ compilers offer a choice of the following three math packages for handling floating-point operations:

- Emulator (default)
- Math coprocessor (a library that supports the Intel 80*x*87 family of math coprocessors)
- Alternate math

When you install Visual C++, you can choose a Setup option to build combined libraries that include the floating-point math library that you choose. Any programs linked with that library use the math package included in the library; you must use

the appropriate CL option to make sure that the library you want is used at link time.

The following descriptions of these math packages are designed to help you choose the appropriate math option for your needs when you build a library using Setup. For more information about Setup and about building combined libraries, see the *Visual Workbench User's Guide*.

For simplicity, the names of libraries are noted in the form *m*LIBC*f*.LIB, where *m* is the model designator and *f* is the floating-point math package designator.

Emulator Package

Programs created using the emulator math package automatically detect and use an 80*x*87 numeric coprocessor if one is installed. If no coprocessor is installed, these 80*x*87 instructions are carried out in software. The emulator package is the default math package; Setup uses it if you do not explicitly choose another package. Also, the emulator math option is the option selected by default by the compiler if no other floating-point math option is specified.

Use the emulator math package to maximize accuracy on systems without math coprocessors or if your program will be run on some systems with coprocessors and some systems without coprocessors.

The emulator package performs basic operations to the same degree of accuracy as a math coprocessor. However, the emulator routines used for transcendental math functions (such as **sin**, **cos**, and **tan**) differ slightly from the corresponding functions performed on a coprocessor. This difference can cause a slight discrepancy (usually within 2 bits) between the results of these operations when performed with the software emulation and the results when performed with a math coprocessor.

When you use the emulator package, some floating-point exceptions are masked.

When you use a math coprocessor or the emulator floating-point math package, interrupt-enable, precision, underflow, and denormalized-operand exceptions are masked by default. The remaining floating-point exceptions are unmasked. For more information about 80*x*87 floating-point exceptions, see the discussion of the **_control87** function in the *Run-Time Library Reference*.

Math Coprocessor Package

The math coprocessor package uses the 80*x*87 math coprocessor exclusively for floating-point calculations. If you use the math coprocessor package, the machine on which your application is to run must have an 80*x*87 coprocessor to perform floating-point operations. This package gives you the fastest, smallest programs possible for handling floating-point math.

Alternate Math Package

The alternate math package gives you the smallest and fastest programs possible without a coprocessor. However, the program results are not as accurate as results given by the emulator package. In addition, the alternate math package does not support the **long double** type.

The alternate math package uses the same format as the IEEE standard-format numbers with less precision and less thorough error checking. The alternate math package does not support infinities, NANs ("not a number"), and denormal numbers.

5.4 Selecting Floating-Point Options (/FP)

You can select a floating-point library and the method of accessing floating-point routines by specifying command-line options to CL. You can choose from the emulator, alternate, and math coprocessor libraries. You can also access the floating-point routines by issuing a function call (or calls) or by generating inline 80*x*87 instructions to execute the floating-point operation. The inline math coprocessor package (selected with the /FPi87 option) generates the smallest and the fastest code because the compiler generates true 80*x*87 coprocessor instructions. However, if you cannot depend on the target computer having a coprocessor, you must use either the emulator or alternate math option.

To specify floating-point options on the CL command line, you must specify an option from the list in Table 5.4. You specify these options to CL starting with the floating-point option string /FP.

Based on the floating-point option and the memory-model option you choose, the compiler embeds a library name in the object file that it creates. This library is then considered the default library; that is, the linker searches in the standard places for a library with that name. If it finds a library with that name, the linker uses the library to resolve external references in the object file being linked. Otherwise, it displays a message indicating that it could not find the library.

This mechanism allows the linker to automatically link object files with the appropriate library. However, you can link with a different library in some cases. For more information about linking with different libraries, see Table 5.4, following, and Section 5.5, "Library Considerations for Floating-Point Options," on page 108.

Table 5.4 summarizes the floating-point options and their effects. These options are described in detail in the following sections.

Table 5.4 Summary of Floating-Point Options

CL option	Effect	Coprocessor	Libraries selected
/FPi (specify emulator)	Default. /FPi produces code larger than /FPi87, but it can work without a coprocessor; using it is the most efficient way to get maximum precision without a coprocessor.	Uses coprocessor if one is present[1]	*m*LIBCE.LIB[2]
/FPi87 (specify coprocessor)	Smallest and fastest option available with a coprocessor.	Requires coprocessor	*m*LIBCE.LIB[2]
/FPc (specify emulator calls)	Slower than /FPi, but makes possible use of the alternate math library at link time.	Uses coprocessor if one is present[1]	*m*LIBCE.LIB[2, 3]
/FPc87 (specify 80*x*87 calls)	Slower than /FPi87, but makes possible use of the alternate math library at link time.	Requires coprocessor unless the library is changed at link time[5]	*m*LIBC7.LIB[3, 4]
/FPa (specify alternate math package)	Fastest and smallest option available without a coprocessor, but sacrifices some accuracy for speed.	Ignores coprocessor	*m*LIBCA.LIB[2, 4]

1 Use of the coprocessor can be suppressed by setting the NO87 environment variable.

2 Can be linked explicitly with *m*LIBC7.LIB at link time.

3 Can be linked explicitly with *m*LIBCA.LIB at link time.

4 Can be linked explicitly with *m*LIBCE.LIB at link time.

5 Use of the coprocessor can be suppressed by setting NO87 if you change to the emulator library at link time.

Optimizations such as constant propagation and constant subexpression elimination can cause some expressions to be evaluated at compile time. Such evaluations always use IEEE format and are unaffected by the floating-point option you choose. For more information about optimizing, see Chapter 6, "Optimizing 16-Bit Programs," in the *Command-Line Utilities User's Guide*.

The /FPi (Specify Emulator) Option

The /FPi option generates inline instructions for an 80*x*87 coprocessor and places the name of the emulator library (*m*LIBCE.LIB) in the object file. At link time, you can specify the math coprocessor library (*m*LIBC7.LIB) instead. If you do not choose a floating-point option, the compiler uses the /FPi option by default.

The /FPi option is useful if you cannot be sure that an 80*x*87 coprocessor is available on the target computer. Programs compiled using this option work as described following:

- If a coprocessor is present at run time, the program uses the coprocessor.
- If no coprocessor is present, the program uses the emulator. In this case, the /FPi option offers the most efficient way to get maximum precision in floating-point results.

When you use the /FPi option, the linker does not generate inline 80*x*87 instructions. Instead, it generates software interrupts to library code, which then fixes up the interrupts to use either the emulator or the coprocessor, depending on whether a coprocessor is present. If you want true inline 80*x*87 instructions, use the "specify coprocessor" (/FPi87) option.

The /FPi87 (Specify Coprocessor) Option

The /FPi87 option instructs the compiler to place 80*x*87 coprocessor instructions in your code for many math operations. It also causes the name of a math coprocessor library (*m*LIBC7.LIB) to be embedded in the object file.

If you use the /FPi87 option and link with the library *m*LIBC7.LIB, an 80*x*87 coprocessor must be present at run time, or the program fails and the following error message is displayed:

```
run-time error R6002
- floating point not loaded
```

Compiling with the /FPi87 option results in the smallest, fastest programs possible for handling floating-point results.

The /FPc (Specify Emulator Calls) Option

The /FPc option generates floating-point calls to the emulator library and places the names of an emulator library (*m*LIBCE.LIB) in the object file. At link time, you can specify a math coprocessor library (*m*LIBC7.LIB) or an alternate math library (*m*LIBCA.LIB) instead. Thus, /FPc gives you more flexibility in the libraries you can use for linking than the "specify emulator" (/FPi) option.

Using the /FPc option is also recommended in the following cases:

- If you compile modules that perform floating-point operations and plan to include these modules in a library
- If you compile modules that you want to link with libraries other than the libraries provided with Visual C++

You cannot link with an alternate math library if your program uses the intrinsic forms of floating-point library routines (that is, if you have compiled the program with the /Oi or /Ox option or have specified math functions in an **intrinsic** pragma).

The /FPc87 (Specify 80x87 Calls) Option

The /FPc87 option generates function calls to routines in the math coprocessor library (*m*LIBC7.LIB) that issue the corresponding 80*x*87 instructions. As with the "specify coprocessor" (/FPi87) option, at link time you can choose to link with an emulator library (*m*LIBCE.LIB). However, /FPc87 offers more flexibility in choosing libraries, because you can change your mind and link with the appropriate alternate math library as well (*m*LIBCA.LIB).

The disadvantages of using the /FPc87 option as opposed to the /FPi87 option are as follows:

- Your executable file's size is larger because a call requires more instructions than a true coprocessor instruction.
- Your program does not execute as fast because you must issue a function call for each floating-point operation.

You cannot link with an alternate math library if your program uses the intrinsic forms of floating-point library routines (that is, if you have compiled the program with the /Oi or /Ox option or have specified math functions in an **intrinsic** pragma).

You must have a math coprocessor installed to run programs compiled with the /FPc87 option and linked with a math coprocessor library. Otherwise, the program fails and the following error message is displayed:

```
run-time error R6002
- floating point not loaded
```

Note Certain optimizations are not performed when you use the calls to math coprocessor option. This can reduce the efficiency of your code; also, because arithmetic of different precision can result, there may be slight differences in your results.

The /FPa (Specify Alternate Math Package) Option

The /FPa option generates floating-point calls and selects the alternate math library for the appropriate memory model (*m*LIBCA.LIB). Calls to this library provide the fastest and smallest option for code intended to run on a machine without an 80*x*87 coprocessor. With this option, you can choose an emulator library (*m*LIBCE.LIB) or a math coprocessor library (*m*LIBC7.LIB) at link time.

You cannot link with an alternate math library if your program uses the intrinsic forms of floating-point library routines (that is, if you have compiled the program with the /Oi or /Ox option or have specified math functions in an **intrinsic** pragma).

5.5 Library Considerations for Floating-Point Options

You may want to use libraries in addition to the default library for the floating-point option you have chosen in your compile options. For example, you may want to create your own libraries (or other collections of subprograms in object-file form), then link these libraries at a later time with object files that you have compiled using different options.

The following sections describe these cases and ways to handle them. Although the discussion assumes that you are putting your object files into libraries, the same considerations apply if you are simply using individual object files.

Using One Standard Library for Linking

You must use only one standard C run-time library when you link. To specify the library from the LINK command line, use the /NODEFAULTLIBRARYSEARCH (/NOD) option and then type the name of the combined library file you want to use in the *link-libinfo* field of the CL command line. Doing so overrides the library names embedded in the object files.

Inline Instructions or Calls

When deciding on a floating-point option, you should decide whether you want to use inline instructions. If you do, compile with the "specify coprocessor" (/FPi87) or "specify emulator" (/FPi) option. Otherwise, compile for floating-point function calls using the "specify 80*x*87 calls" (/FPc87), "specify emulator calls" (/FPc), or "specify alternate math package" (/FPa) option.

If you choose to use inline instructions for your precompiled object files, you cannot link with an alternate math library (*m*LIBCA.LIB). However, inline instructions achieve the best performance from your programs on machines that have an 80*x*87 coprocessor installed.

If you choose to use calls, your programs are slower but at link time you can switch to any standard C run-time library (that is, any library created by the Setup program) that supports the memory model you have chosen.

5.6 Compatibility Between Floating-Point Options

Each time you compile a source file, you can specify a floating-point option. When you link two or more source files to produce an executable program file, you must ensure that floating-point operations are handled consistently and that the environment is set up properly to allow the linker to find the required library.

If you are building libraries of C or C++ routines that contain floating-point operations, the "specify emulator calls" (/FPc) option provides the most flexibility.

The examples that follow illustrate how you can link your program with a library other than the default. The floating-point option and the substitute library are compatible.

The following example compiles the program `CALC.C` with the "specify medium memory model" (/AM) option. Because no floating-point option is specified, the "specify emulator" (/FPi) option is used. The /FPi option generates 80*x*87 instructions and specifies the emulator library MLIBCE.LIB in the object file. The /LINK field specifies the /NODEFAULTLIBRARYSEARCH (/NOD) option and the names of the medium-model math coprocessor library. Specifying the math coprocessor library forces the program to use an 80*x*87 coprocessor; the program fails if a coprocessor is not present.

```
CL /AM CALC.C /link MLIBC7 /NOD
```

The following example compiles `CALC.C` using the small (default) memory model and the "specify alternate math package" (/FPa) option. The /LINK field specifies the /NOD option and the library SLIBCE.LIB. Specifying the emulator library causes all floating-point calls to refer to the emulator library instead of the alternate math library.

```
CL /FPa CALC.C /link SLIBCE /NOD
```

The following example compiles `CALC.C` with the "specify 80*x*87 calls" (/FPc87) option, which places the library name SLIBC7.LIB in the object file. The /LINK field overrides this default-library specification by giving the /NOD option and the name of the small-model alternate math library (SLIBCA.LIB).

```
CL /FPc87 CALC.C /link SLIBCA.LIB/NOD
```

5.7 Using the NO87 Environment Variable

Programs compiled using either the /FPc or the /FPi option automatically use an 80*x*87 coprocessor at run time if one is installed. You can override this and force the use of the software emulator by setting an environment variable named NO87.

Use the NO87 environment variable to suppress use of the 80*x*87 coprocessor at run time.

If NO87 is set to any value when the program is executed, use of the coprocessor is suppressed. The value of the NO87 setting is printed on the standard output as a message. The message is printed if a coprocessor is present and suppressed, or if no coprocessor is present.

You can set an environment variable by using the SET command from the command line. For example,

```
SET NO87=Use of coprocessor suppressed
```

This command causes the message `Use of coprocessor suppressed` to appear when a program that uses an emulator library is executed. If you don't want a message to be printed, set NO87 equal to one or more spaces. A blank string for NO87 causes a blank line to be printed.

Note that only the presence or absence of the NO87 definition is important in suppressing use of the coprocessor. The actual value of the NO87 setting is used only for printing the message.

The NO87 variable takes effect with any program linked with an emulator library (*m*LIBCE.LIB). It has no effect on programs linked with math coprocessor libraries (*m*LIBC7.LIB) or on programs linked with alternate math libraries (*m*LIBCA.LIB).

5.8 Incompatibility Issues

The exception handler in the libraries for 80*x*87 floating-point calculations (*m*LIBCE.LIB and *m*LIBC7.LIB) is designed to work without modification on the IBM PC family of computers and on closely compatible computers, including the WANG personal computer, the AT&T 6300, and the Olivetti personal computers. Also, the libraries need not be modified for the Texas Instruments Professional Computer, even though it is not compatible. Any machine that uses nonmaskable interrupts (NMI) for 80*x*87 exceptions runs with the unmodified libraries. If your

computer is not one of these, and if you are unsure whether it is completely compatible, you may need to modify the math coprocessor libraries.

All Microsoft languages that support 80*x*87 coprocessors intercept 80*x*87 exceptions in order to produce accurate results and properly detect error conditions. To make the libraries work correctly on incompatible machines, you can modify the libraries. To make this easier, an assembly-language source file, EMOEM.ASM, is included with Visual C++. Any machine that sends the 80*x*87 exception to an 8259 priority interrupt controller can be supported by a simple table change to the EMOEM.ASM module. The source file contains further instructions about how to modify EMOEM.ASM, patch libraries, and executable files.

PART 2

Special Environments

CHAPTER 6

Programming for Windows

The following chapter discusses several issues of interest if you are developing programs for 16-bit versions of Windows. Topics covered are:

- Optimizing protected-mode prolog and epilog code for Windows
- Specifying program starting execution points
- Writing exit procedure routines for Windows

6.1 Optimizing Protected-Mode Prolog and Epilog Code for Windows

Functions written for Windows have two sequences of code, called prolog and epilog, added to their entry and exit points, respectively. This code sets the DS register to the address of the data segment for the function's associated application or dynamic-link library (DLL).

The following two sections discuss three compiler options—/GA, /GD, and /GE*string*. These options generate optimized prolog/epilog code for protected-mode applications and generate both optimized prolog/epilog code and a linker EXPDEF record for protected-mode DLLs. Among other optimizations, these options eliminate the **inc** BP and **dec** BP instructions required by real-mode functions for Windows.

Using /GA and /GD to Optimize Prolog/Epilog Code

For protected-mode applications, use the /GA option to generate the correct prolog/epilog code for all far functions explicitly marked as **__export**. For protected-mode DLLs, use the /GD option to generate the correct prolog/epilog code and to create a linker EXPDEF record for all far functions explicitly marked as **__export**. Using either /GA or /GD generates more efficient code than using either the /GW or /Gw "generate real-mode prolog/epilog code for Windows" option.

The results of using either option depend on the absence or presence of the **__export** keyword. In Microsoft Visual C++, **__export** has two different actions:

- For DLLs (using /GD), **__export** provides the name of a function to the linker.
- For both applications and DLLs (using both /GA and /GD), **__export** generates optimized prolog/epilog code.

Table 6.1 shows the number of bytes and instructions saved for each function call when you use /GA or /GD instead of /GW or /Gw.

Table 6.1 Byte and Instruction Savings with the /GA or /GD Option

Option combination	Bytes saved	Instructions saved
/GA or /GD without **__export**	10	7
/GA with **__export**	4	3
/GD with **__export**	4	0

As shown in Table 6.1, use of /GA or /GD without **__export** saves the most bytes and instructions, thus generating the smallest possible prolog/epilog code for functions. Selective use of **__export** increases the prolog/epilog size of functions that you determine require exporting.

For further savings, use /G2 to generate 80286 code.

The code generated in all four cases is smaller than that generated by /Gw or /GW. In all cases, you can also use the /G2 option to generate protected-mode 80286 code; this saves an additional four bytes.

Both /GA and /GD define the _WINDOWS preprocessor symbol, and /GD defines the _WINDLL preprocessor symbol. The /GA option requires use of both *m*LIBC*f*W.LIB and the Windows applications programming interface (API) library. The /GD option requires use of both *m*DLLC*f*W.LIB and the Windows API library. You cannot use the /Gw, /GW, or /Gq (generate simplified real-mode prolog/epilog code for Windows) option with either /GA or /GD.

Using /GE*string* to Modify the Default Behavior of /GA and /GD

Using the /GE*string* option, you can fine-tune the behavior of the /GA and /GD options. The /GE*string* option can be used only in conjunction with /GA and /GD. For protected-mode programs, the *string* argument is one or more letters, with no intervening spaces, from the following list:

String	Optimizing procedure
f	Creates prolog/epilog code for all far functions even if they are not marked as **__export**.
a	Loads DS from AX.

String	**Optimizing procedure**
d	Loads DS from DGROUP—the default behavior for /GD.
s	Loads DS from SS—the default behavior for /GA. Use this string only if SS=DS.
e	Forces emission of linker EXPDEF records for all functions marked as **__export**.

The linker refers by name to every function that is explicitly or implicitly marked as **__export**, unless the function is given an ordinal number in a definition (.DEF) file's **EXPORTS** section. Thus marking a DLL function as **__export** (or forcing it to be **__export** by using the /GEf option) has two disadvantages.

One disadvantage is that doing so slows function calls. Functions referred to by name are loaded into the resident name table, whereas functions referred to by ordinal number reside on disk in the nonresident table. Large resident name tables slow function calls because finding the name of a given function requires a linear string search through the table.

The second disadvantage is that, because a DLL can contain both functions meant for public use and those meant for private use within the DLL, forcing all functions into the resident name table exposes the private functions to potential misuse.

Conflicts Between __fastcall and Prolog/Epilog Code

Prolog/epilog code for Windows and the **__fastcall** calling convention can conflict in their use of the AX register. As a result, the following combinations of options and keywords generate errors:

- **__fastcall**, **__far**, /Gw
- **__fastcall**, **__far**, **__export**, /GA
- **__fastcall**, **__far**, **__export**, /GD
- **__fastcall**, **__far**, /GA, /GEf
- **__fastcall**, **__far**, /GD, /GEf
- **__fastcall**, **__far**, **__export**, /GA, /GEf
- **__fastcall**, **__far**, **__export**, /GD, /GEf

6.2 Specifying Program Starting Execution Points

When you use the libraries for Windows that are provided with Visual C++, your program's starting execution point can be either the **main**, **WinMain**, or **LibMain** function. The starting execution point depends on the library used. The following three sections discuss how the run-time system's startup code reacts to the presence or absence of a **main**, **WinMain**, or **LibMain** function.

Executable Files for Windows Version 3.x

If your program is an executable file for Windows version 3.*x*, the run-time system's startup code looks first for a function named **main** to treat as the entry point. If a **main** function does not exist, then the startup code attempts to use the **WinMain** function. If neither of these functions exist, the run-time system displays the following error message:

```
"R6021 no main function"
```

Dynamic-Link Libraries for Windows Version 3.x

If your program is a DLL for Windows version 3.*x*, the run-time system's startup code looks first for a function named **main** to treat as the entry point. If a **main** function does not exist, then the startup code attempts to use **LibMain**. Unless you link in one of the NOCRT libraries, such as SNOCRTW.LIB, a **main** or **LibMain** function is not required and no error is reported.

Windows Version 3.x and the NOCRT Libraries

If a program you created for Windows version 3.*x* does not use the run-time system's startup code, you must link it with one of the NOCRT libraries, such as SNOCRTW.LIB. In such cases, your program must have either a **WinMain** (if it is an executable file) or a **LibMain** (if it is a DLL). If neither of these functions exist, the run-time system displays an error message.

6.3 DLL Initialization Code for Windows

Because the startup functionality previously provided by LIBENTRY.OBJ (and LIBENTRY.ASM) is now provided by the standard run-time libraries that support DLLs for Windows, do not link with LIBENTRY.OBJ.

Compiling a DLL with Microsoft C version 6.0 required that the first object module linked be LIBENTRY.OBJ or the equivalent. Microsoft C/C++ version 7.0 and Visual C++ automatically provide the library initialization code that used to be provided by LIBENTRY.OBJ. If you provide your own initialization entry object, makefiles used to build DLLs in Microsoft C version 6.0 must be changed to eliminate linking in LIBENTRY.OBJ. If you don't use the run-time library functions, you can still use the correct initialization code. Do so without the run-time library overhead by linking with an *x*NOCRTDW.LIB library, where *x* is S, M, C, or L (small, medium, compact, or large model).

6.4 Termination Routines for Windows

The C run-time libraries that support DLLs for Windows (such as SDLLCEW.LIB) contain both startup and termination code. Beginning with Microsoft C/C++ version 7.0, these libraries include the Windows exit procedure (**WEP**), a termination routine. You can also optionally call your own termination code from the run-time library's **WEP** as needed.

The WEP Routine

Visual C++ automatically links the C run-time **WEP** routine into DLLs that are written for Windows. In addition to terminating a program, **WEP** performs several cleanup functions when Windows unloads a DLL from memory. For example, it releases memory and calls **atexit** routines.

Note that the run-time libraries with names containing NOCRT (such as SNOCRTDW.LIB) do not have a **WEP** routine. Use these libraries when creating DLLs that do not use any C run-time services or routines.

Using a **WEP** requires that you follow some important rules and restrictions when building a DLL for Windows.

For your program to make use of a **WEP**, the routine must remain resident in memory as long as the associated DLL remains in memory. To ensure that it does, you must include the following statements in the DLL's definition file (.DEF) file:

```
SEGMENTS 'WEP_TEXT' FIXED PRELOAD
EXPORTS    WEP @1 RESIDENTNAME
```

The **SEGMENTS** declaration tells Windows to load the WEP_TEXT segment into a fixed memory location as it loads the DLL into memory. **RESIDENTNAME** tells Windows to keep the **WEP** routine's name resident in memory so that the **WEP** can always be called—even when there is little available memory. The **WEP** routine's code is very small. Making the WEP_TEXT segment resident in memory should not adversely affect the performance of Windows.

If you use any of the run-time DLL libraries, such as SDLLCEW.LIB, you must include this information in the definition file.

Writing Your Own WEP Routine

You can, optionally, provide your own termination routine to perform additional DLL termination processing. Note that the C run-time library **WEP** termination source code is provided with the startup source files (WEP.ASM) so that sophisticated users can alter or refer to it as needed. If you write a termination routine, it

must fit the following prototype. Unlike the **WEP** routine for Windows, the name of your routine must begin with a leading underscore:

```
int __far __pascal _WEP( int <parameter> );
```

The value returned by your **_WEP** routine, if any, is passed through to Windows as if it were the **WEP** termination value for Windows. In most cases, this should be 1 to signify success.

If you have an existing **WEP** routine, you can use it with Visual C++ by adding a leading underscore (_) to the name. If a procedure named **_WEP** exists at link time, the C run time system automatically calls it at DLL termination time.

Note that the **_WEP** routine must be compiled as **extern "C"** when used in a C++ program:

```
Extern "C" int __far __pascal _WEP( int <parameter> );
```

DLL termination code should avoid the following:

- Deep stack usage. In some cases, DLL termination code can be called on a stack that has insufficient space remaining. This stack overflow produces unpredictable results.
- Operating system requests. Due to the potential for insufficient stack space, avoid calls to the operating system.
- File input and output. Files are owned by processes (executable files), not DLLs. When DLL termination code is called, processes have already stopped and files are already closed. This is the reason that the run-time library DLL termination code does not attempt to do a final **_flushall** routine as it does in other environments.

Most DLL termination problems occur due to memory constraints (for example, insufficient memory to load the DLL initialization code or to swap in DLL termination code). For cases in which memory contention rarely occurs, DLLs run even if you do not follow the previous restrictions.

For more information on DLL termination in Windows, refer to the Windows Software Development Kit documentation.

CHAPTER 7

QuickWin Programs

QuickWin is a run-time library that helps you turn programs for MS-DOS into simple Windows-hosted applications. This chapter explains the user interface and programming features provided by QuickWin, and it explains how to use them to build your own QuickWin applications.

Using QuickWin, many programs written for MS-DOS can be compiled with the Visual Workbench to run in a text window. In this release of QuickWin, a graphics library component has been added that enables a QuickWin program to call graphics functions and display graphics output in a QuickWin window.

A QuickWin text window behaves the same way as the MS-DOS character-mode display. You can write to a QuickWin text window and receive input through it with C run-time library input and output routines, such as **printf** and **scanf**, or with C++ iostream facilities, such as **cout** and **cin**.

A QuickWin graphics window behaves the same way as the MS-DOS graphics-mode display and is controlled by calls to functions very similar to those found in the MS-DOS GRAPHICS.LIB library.

Note If your MS-DOS program uses console or serial port input/output (I/O) functions or if it spawns processes, you cannot link it with the QuickWin library. Console I/O functions are the functions prototyped in the CONIO.H header file, such as **_kbhit**.

QuickWin makes it easy to add a Windows-like look to MS-DOS programs.

Using QuickWin, you can run and debug your MS-DOS programs in the Visual Workbench without having a detailed knowledge of programming for Windows. Note that QuickWin offers only a portion of Windows capability: Because you cannot call Windows application programming interface (API) functions from your QuickWin program, you cannot write a complete Windows-hosted application using QuickWin. You can, however, add a Windows-like flavor to your applications.

QuickWin is also useful for programmers experienced with Windows. When you have a simple MS-DOS program that you'd like to see in a window without completely overhauling the application, use QuickWin.

Additionally, QuickWin applications have access to all the Windows address space and can share data with other Windows-hosted applications.

7.1 Capabilities of QuickWin Graphics

The QuickWin graphics library, a component of the QuickWin library, helps you turn MS-DOS graphics programs into simple applications for Windows. "MS-DOS graphics programs" refers to programs that use GRAPHICS.LIB routines. Using QuickWin, graphics programs written for MS-DOS can be compiled with the Visual Workbench to run in a window.

The QuickWin library supports all routines in GRAPHICS.LIB. Most QuickWin graphics applications behave as do the graphics applications written for MS-DOS. QuickWin graphics routines that work differently from MS-DOS graphics routines are discussed in Section 7.9, "Differences Between MS-DOS Graphics and QuickWin Graphics," on page 147.

The logical graphics screen is a graphical output screen that emulates system graphics modes.

The QuickWin graphics library creates a multiple document interface (MDI) child window for the graphics program and treats it like an MS-DOS graphics screen. MDI applications can contain multiple child windows within the application's client area. The MDI graphics window is referred to as the "logical graphics screen." This logical graphics screen is a graphical output screen that emulates the current system graphics video card. Note that a logical graphics screen may not be entirely visible in some video modes. When the entire logical graphics screen is not visible, scroll bars are provided.

Your application can display graphics in the graphics child window by calling the graphics routines in the QuickWin library. QuickWin programs that call these routines have the same capabilities as other QuickWin applications, along with the additional ability to open and manipulate graphics child windows.

You can add a Windows-like flavor to your graphics applications if you use the enhanced QuickWin graphics features. With enhanced QuickWin graphics features, you can:

- Add multiple graphics child windows.
- Control which graphics child window is the active window.
- Close any graphics child window.
- Get keyboard input from a graphics child window.

7.2 Two Ways to Use QuickWin

You can use QuickWin in two ways. You can link your existing MS-DOS application with the QuickWin library to create a standard QuickWin application, or you can alter your source code to take advantage of the enhanced capabilities and functions in the QuickWin library.

Standard QuickWin Programs

The simplest way to use QuickWin is to link your MS-DOS application with the QuickWin library without altering your source code. Your program then has the standard QuickWin user interface features described in Section 7.4, "The QuickWin User Interface," on page 125. Your standard QuickWin program:

- Runs with the Windows operating system, in a window.
- Can be minimized or maximized, as can any Windows-based application. (Minimized child windows appear as icons in the lower part of the client window; maximized windows fill the screen.)
- Provides a standard QuickWin menu bar.
- Takes advantage of the Clipboard by providing Copy and Paste commands.
- Provides Help for the QuickWin features.
- Takes advantage of the protected-mode memory management capabilities of Windows.

Enhanced QuickWin Programs

You can also use QuickWin to take advantage of more Windows features (although not features provided by the functions in the Windows API). To use these enhanced features, you must alter your source code. You can use QuickWin to:

- Add multiple child windows (also called document windows).
- Control the size and placement of child windows, including whether they are tiled or cascaded. (Cascaded windows overlap; tiled windows are arranged so that all windows are fully visible, with no overlap.)
- Control the size of a text window's buffer, determining how much of the window's text is stored (and can be scrolled through even when it is not all visible).
- Control which child window is the foremost window. The foremost window is said to have the "input focus," which means that keyboard input is directed to this window.

- Add an About dialog box customized with your text.
- Simulate mouse clicks in some of the QuickWin menus.
- Yield processing time to other applications running with Windows.
- Add custom application and document icons to your program.

Adding these enhanced features requires the use of additional run-time functions, available only to the QuickWin library. These functions are listed in the next section.

QuickWin Functions

This section lists the functions specific to the QuickWin library. For further descriptions of these functions, consult Help.

The following functions are specific to QuickWin programs:

Function	Description
_wabout	Sets the string that appears in the About dialog box
_fwopen	Opens a new text window stream
_wopen	Opens a text window handle (works for text windows only)
_wclose	Closes a text window handle (works for text windows only)
_wgetexit	Gets the application's exit behavior
_wgetfocus	Gets a window's current frame focus
_wgetscreenbuf	Gets a window's current screen buffer size
_wgetsize	Gets a window's current size and position on the screen
_wmenuclick	Chooses a menu item
_wsetexit	Sets the application's exit behavior
_wsetfocus	Sets a window's frame focus
_wsetscreenbuf	Sets a window's screen buffer size
_wsetsize	Sets a window's size and position on the screen (works for text windows only)
_wyield	Yields processor control to Windows for queue servicing

The following functions are specific to the QuickWin graphics library, a component of the QuickWin library. They are not included in the MS-DOS GRAPHICS.LIB

library. For further descriptions of these and the GRAPHICS.LIB functions, consult Help. These functions can be used only for graphics child windows:

Function	Description
_inchar	Reads a single character from the keyboard
_wgopen	Opens a new graphics child window
_wgclose	Closes an existing graphics child window
_wggetactive	Returns the handle of the active graphics child window
_wgsetactive	Makes a graphics child window the active window

7.3 Comparison of QuickWin and Windows

QuickWin provides a rich set of Windows features, but it does not provide the total capability of Windows. With QuickWin, you cannot:

- Call Windows API functions.
- Detect and respond to mouse clicks in a window.
- Display and use your own menus, controls, and dialog boxes.
- Add your own customized Help information to QuickWin Help.
- Run your program in real mode.

7.4 The QuickWin User Interface

When a QuickWin program runs, it displays an MDI Windows-style client window (also called an application window) titled with the program's name. The window has the standard controls for applications that run with Windows, including a Control-menu box, a window border with corners for resizing the window, and buttons for minimizing and maximizing the window. The client window also has a menu bar at the top and a status bar at the bottom. The menu bar provides menus common to all QuickWin applications; the status bar provides status information to the user.

Within the client window is a child window titled "Stdin/Stdout/Stderr," which displays the C or C++ input/output streams. This child window also has controls and can have one or more scroll bars. QuickWin text windows are text only; text is black on white. If a call to a graphics function is made, a graphics child window with a default title of "Graphic1" appears. All subsequent graphics output is

directed to this window. Figure 7.1 shows the QuickWin user interface; note that the default title of the graphics child window has been changed from "Graphic1" to "SMILEY."

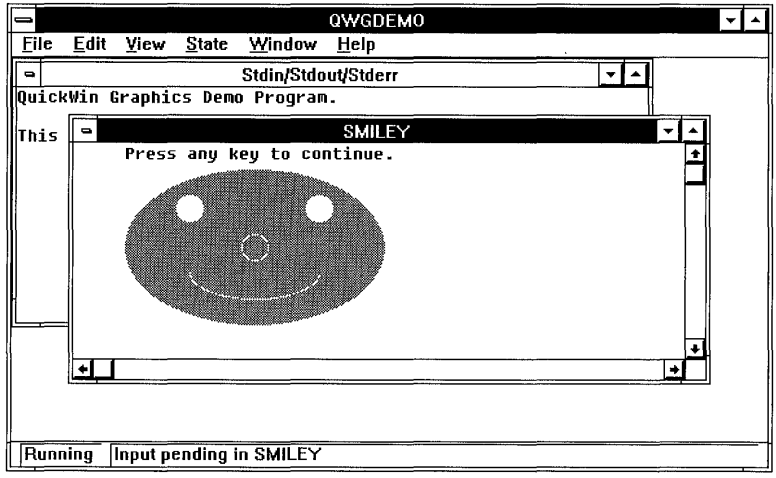


Figure 7.1 QuickWin User Interface

QuickWin programs that take advantage of the enhanced features can display more than one child window. Such programs can also control the size and position of windows and which window is the foremost window, among other capabilities.

QuickWin Menus

When you run a QuickWin program, the client window always contains the standard QuickWin menu bar. The menu bar contains File, Edit, View, State, Window, and Help menus.

File Menu

The QuickWin File menu has one command, Exit, which ends the program, closing all windows. By default, any of your program's windows that still exist remain on the screen. You can control this behavior by calling the **_wsetexit** function. For more information about using **_wsetexit**, see "Keeping Windows on the Screen," on page 144.

Edit Menu

The QuickWin Edit menu has commands for selecting, copying, and pasting text within or between windows or between applications. These commands are Mark,

Paste, Copy Tabs, Copy, and Select All. If a graphics child window is the foremost window, only the Copy command is available.

The Mark command puts the window that has the input focus in Mark mode, ready for the user to select text for copying to the Clipboard. The string "Mark – " is prefixed to the window title.

Text can be selected with the keyboard or the mouse. To select with the keyboard, first choose the Mark command. Then use the arrow keys to move the cursor from the upper-left corner of the window to any corner of the desired text area. To select, hold down the SHIFT key and press an arrow key. The selected text is highlighted. To select with the mouse, click in the window and drag a rectangle outlining the selection. For mouse selection, choosing Mark is optional. If the mouse is used, the string "Select – " is prefixed to the window title instead of "Mark – ".

Beginning a selection always pauses the program, at which time a check mark appears by the Pause command on the State menu, the program does not accept input, and processing time is yielded to other applications running with Windows. To resume processing, choose Resume from the State menu, choose Copy or Copy Tabs from the Edit menu, or click in the window with the mouse. A check mark appears by the Resume command on the State menu, the program accepts input, and the selection highlighting is removed.

When text has been selected, use the Copy or Copy Tabs command to copy the selected text onto the Clipboard.

The Copy Tabs command copies the selected text onto the Clipboard in CF_TEXT format: Its characters are taken from the ANSI character set, each line ends with a carriage return and linefeed, and a null character terminates the block of text. Before the text is placed onto the Clipboard, all sequences of blanks except leading blanks are converted to single tabs. This command is useful for pasting data into applications such as Microsoft Excel, which uses tabs to delineate input data items.

The Copy command is like Copy Tabs, except that no tab conversion is performed. If a graphics window has the focus, this is the only command available on the Edit menu. In this case, the Copy command copies the entire logical graphics screen onto the Clipboard as a bitmap and the user can paste the copied graphics output into any Windows-hosted application.

The Select All command selects and highlights all text in the foremost window. Using Select All is equivalent to selecting all the text in a window with the mouse. The window title is prefixed with "Select – ".

The Paste command takes the most recently copied block of text from the Clipboard and places it in the program's paste buffer. The text must be in CF_TEXT format. Read calls to any window in the program are satisfied from this buffer until it is empty; subsequent input comes from the standard input stream. The status bar displays the line "Paste Input Pending" when there is text in the paste buffer.

View Menu

The QuickWin View menu contains two commands, Size To Fit and Full Screen, that control how the logical graphics screen is displayed. Note that both these commands are unavailable if a text child window has the focus. The selected menu command has a check mark in front of its name. In Full Screen mode, the user interface is not visible; therefore, you cannot see the menus.

The Size To Fit command stretches or shrinks the size of the selected logical graphics screen to fit the client area of the graphics child window.

Note When Size to Fit is chosen in the View menu, the resulting graphics may appear somewhat distorted.

The Full Screen command stretches or shrinks the selected logical graphics screen to fit the entire screen. Once in Full Screen mode, the user can return to Windows and restore the application to its previous mode by clicking the mouse once or by pressing ESC.

State Menu

The QuickWin State menu has commands for pausing and resuming the program. The Pause command temporarily suspends the program. While the program is paused, other Windows-hosted applications can run without competition for resources from the QuickWin program. The Resume command lets the program resume execution and removes any highlighting of selected text in a text child window. The selected command, Pause or Resume, has a check mark in front of its name.

The State menu exists to allow pausing for text selection and for yielding time to other applications for Windows, such as a calendar or calculator. You do not have to pause to give one of your program's windows in the background the focus or to perform other operations within your program.

Window Menu

The QuickWin Window menu has commands for arranging windows, selecting the window with the input focus, clearing the paste buffer, and showing or hiding the status bar. In addition, the lower portion of the menu lists all open child windows. Figure 7.2 shows the Window menu as it appears in the example program QWGDEMO.CPP, which is described later in this chapter. The Window menu contains the following commands:

- The Cascade command arranges the program's document windows in an overlapped fashion.
- The Tile command arranges the program's document windows so they are all visible at once.

- The Arrange Icons command organizes any child windows that have been minimized to icons. The icons are arranged evenly along the bottom of the client window.
- The Input command activates the window with pending input. This command is enabled only when a graphics child window has pending input (upon a call to **_inchar**). In addition, the status bar displays a message when a window has input pending.
- The Clear Paste command clears the paste buffer. If a graphics child window has the focus, the Clear Paste command is removed from the Window menu.
- The Status Bar command turns the status bar display on and off. A check mark appears next to this command when the status bar is visible and disappears when it is not.
- The lower portion of the Window menu lists all open child windows for the QuickWin application. A check mark appears in front of the name of the foremost child window. You can give another window focus by choosing its name from the menu.

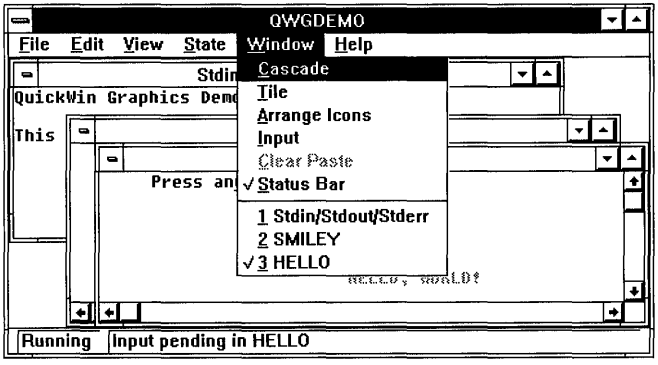


Figure 7.2 Window Menu in QWGDEMO.CPP

Help Menu

The QuickWin Help menu has commands for displaying Windows-based Help for the QuickWin interface. (Note that you cannot augment this Help information with program-specific information.) The Index command displays an index of Help for QuickWin; the Using Help command displays information about using Help. The About command displays a dialog box with information about your QuickWin application. By default, the text describes QuickWin itself, but you can customize the dialog box (for more information, see "About Dialog Box," on page 132).

Other QuickWin Features

This section describes other features of the QuickWin user interface, using the following terms:

- The horizontal direction is represented by the "x-axis."
- The vertical direction is represented by the "y-axis."
- The "origin" (point 0,0) is the upper-left corner of your screen. The x- and y-axes start at the origin. You can change the origin in some coordinate systems.

Arrow Keys

The arrow keys move the logical graphics screen one pixel in the opposite direction of the key pressed, even when no portion of the graphics image extends beyond the window's boundaries. For example, if the RIGHT ARROW key is pressed, the logical graphics screen "moves" one pixel to the left.

If a text child window has the focus, the arrow keys also move the text in the opposite direction of the key pressed, if the text child window contains text that extends beyond its boundaries.

Page Up and Page Down

If a text child window contains text that extends beyond the horizontal boundaries of the child window, the PAGE UP and PAGE DOWN keys move the text along the y-axis in the opposite direction of the key pressed.

If a graphics child window has the input focus, the PAGE UP and PAGE DOWN keys move the logical graphics screen along the y-axis according to the graphics child window's client size. For example, when you set the video mode to **_VRES16COLOR**, QuickWin creates a bitmap of size 640×480 pixels in 16 colors. If the client size of graphics child windows is 100×80, the section of the bitmap that you see in normal mode is the rectangular area with the viewport coordinates (0,0) and (99,79). Pressing the PAGE DOWN key moves the logical graphics screen up by 80 pixels; you see the portion of the bitmap that is the rectangular area with the coordinates (0,80) and (99,159).

Pressing CTRL+PAGE UP or CTRL+PAGE DOWN moves the logical graphics screen along the x-axis.

Home and End

If a text child window has the input focus, the HOME key causes the beginning of the text output to be displayed and the END key causes the end of the text output to be displayed. This is equivalent to moving the scroll box to the top or to the bottom of the scroll bar, respectively.

If a graphics child window has the input focus, HOME and END display the left side and the right side, respectively, of the logical graphics screen without moving the screen along the y-axis. Pressing CTRL+HOME displays the upper-left corner of the graphics output; pressing CTRL+END displays the opposite corner.

Printing QuickWin Graphics Output

To print output from QuickWin graphics applications, first copy the output to the Clipboard. When you copy the bitmap to the Clipboard, you can paste it into any application that supports graphics printing. Follow these steps to print QuickWin graphics output:

1. Give the input focus to the graphics window by choosing its name from the Window menu, or by clicking the mouse once inside the window's border.
2. From the Edit menu, choose Copy to copy the window's contents to the Clipboard.
3. Paste the bitmap into any Windows-based application that supports graphics printing. Consult the application's documentation on how to paste and print bitmaps.

7.5 Overview of the Enhanced Capabilities of QuickWin

Many C and C++ programs require no changes to be compiled as QuickWin applications. However, you have the option of giving your program more of a Windows-like look and greater flexibility using features described in this section. Details about using these features and calling QuickWin library functions are covered in Section 7.8, "Writing Enhanced QuickWin Programs," on page 135.

Note The MS-DOS GRAPHICS.LIB library does not support the enhanced QuickWin graphics features.

About Dialog Box

You can customize the About dialog box.

In QuickWin, an About dialog box identifies your program by name and supplies a copyright notice. This dialog box appears when the user chooses the About command from the QuickWin Help menu. By default, QuickWin displays information about QuickWin itself, but you can customize the dialog box by specifying a text string to display. Use the **_wabout** function to set the About text. Figure 7.3 shows the About dialog box as it appears in the example program QWGDEMO.CPP, which is described later in this chapter.

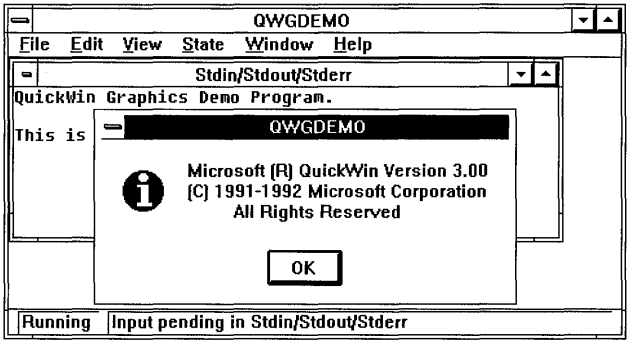


Figure 7.3 About Dialog Box in QWGDEMO.CPP

Multiple Child Windows

By default, QuickWin displays a client window with a menu bar and one text child window, titled "Stdin/Stdout/Stderr." The default input/output streams use this window. However, you have the option of opening additional text or graphics child windows. Use the **_fwopen** or **_wopen** function to open new text child windows. Use the **_wgopen** function to open new graphics child windows. These functions are described in "Opening Child Windows," on page 137. If your program reads or writes multiple files, you can use document windows to display those files on the screen.

Active Window

When you open multiple child windows, the foremost window is said to have the input focus and be the "active" window. For text child windows, use the **_wgetfocus** function to determine which window has the focus. Use the **_wsetfocus** function to make a particular window the foremost window, giving it the focus. These routines are useful for bringing a hidden or partially obscured window to the foreground.

For graphics child windows, use the **_wggetactive** function to determine which window is active. Use the **_wgsetactive** function to make a particular graphics child window the active window. Although keyboard input cannot be directed to an active graphics child window, the window can continue to receive input from the application even if it does not have the focus.

The QWGDEMO.CPP program found in the MSVC\SAMPLES\QWGDEMO directory (or in the directory where you placed your sample source files) demonstrates use of these functions.

Program Control of Menus

Users of a program that runs with Windows can choose commands from the menu bar either with the mouse or with the keyboard. Your program can also choose some of these commands for its own purposes, without user intervention. Although your program cannot add menus of its own to the menu bar, it can have some control over the default QuickWin menus by simulating a mouse click on a given menu item, as if a user had chosen the menu command with the mouse.

The menu commands you can activate in this way are limited to the Tile, Cascade, Arrange Icons, and Status Bar commands on the Window menu. Simulating menu clicks is especially useful if you want your program's document windows to appear initially in certain positions on the screen. For example, you might want them either tiled or cascaded. Use the **_wmenuclick** function to have a program activate a menu command. This feature is useful for setting up the initial configuration of the windows and the status bar in your program and for reconfiguring them as conditions change.

Program Control of Windows

In your program, you can also directly control the size and position of child windows and the amount of text they retain for scrolling, and you can control how your program behaves when an **exit** function is called.

Use the **_wgetsize** and **_wsetsize** functions to determine or to reset a window's current size and position. The **_wsetsize** function can only be used to reset a text window's size and position. It cannot be used to resize graphics windows. Use **_wgetscreenbuf** to get the size of a window's text buffer (the amount of text it can retain and scroll through). Use **_wsetscreenbuf** to set the size of a window's text buffer so it can retain more or less text. For example, you can read a text file and write it into a window with a buffer appropriately sized with **_wsetscreenbuf** so that users can scroll through the entire contents of the file.

Use the **_wsetexit** and **_wgetexit** functions to specify whether your program's windows remain on the screen after the program calls an **exit** function. Your program can behave in three possible ways at exit time:

- It can leave all windows on the screen by default.
- It can leave no windows on the screen.
- It can allow the user to choose whether to leave windows on screen using a dialog box.

Use **_wgetexit** to get the current exit behavior setting. Use **_wsetexit** to set the desired exit behavior. For more information about these functions, see "Keeping Windows on the Screen," on page 144.

7.6 Building QuickWin Programs

Many MS-DOS programs can be built as QuickWin programs simply by building them as "QuickWin.EXE" project types using the Project command on the Options menu within the Visual Workbench. An MS-DOS program can generally become a QuickWin program as long as it doesn't make calls to console I/O functions (those functions prototyped in CONIO.H, such as **_kbhit**) and doesn't spawn processes. When testing your QuickWin programs, remember that QuickWin programs do not run in real mode.

The Visual Workbench simplifies the process of building an MS-DOS program as a QuickWin program. For a detailed description of how to build a QuickWin application, see Chapter 3, "Building a Sample QuickWin Program," in the *Visual Workbench User's Guide*.

7.7 Running QuickWin Programs

This section explains how to run QuickWin programs from within the Visual Workbench, from within Windows, and from the MS-DOS command line.

From Within the Visual Workbench

After successfully building your QuickWin program, choose Go from the Run menu or press the F5 key.

With the Windows Run Command

In the Windows Program Manager, choose Run from the File menu. Type the program name, prefixed with a path if needed. Then choose the OK button.

From the Windows File Manager

In the Windows File Manager, double-click the name of the program's .EXE file.

From an Icon in Windows

In Windows, choose New from the File menu and select the Add Program Item option to add your program to a group (a collection of applications in the Windows Program Manager window). This adds an icon for your program to the Program Manager window; you can double-click the icon with the mouse to run the program.

From the Command Line

Type `WIN` (not case sensitive) followed by the program name. If the program is not in the current directory or in a directory specified by the PATH environment variable, specify a path. For example:

```
C:WIN C:\PROGRAMS\HELLO
```

This command starts Windows and then runs the HELLO program, located in the PROGRAMS directory on drive C. If you are already in an MS-DOS session running with Windows, you cannot start the HELLO program in this manner.

Note QuickWin programs that contain the enhanced features cannot be run as MS-DOS programs. They can be run only with Windows, in standard or 386 enhanced mode.

7.8 Writing Enhanced QuickWin Programs

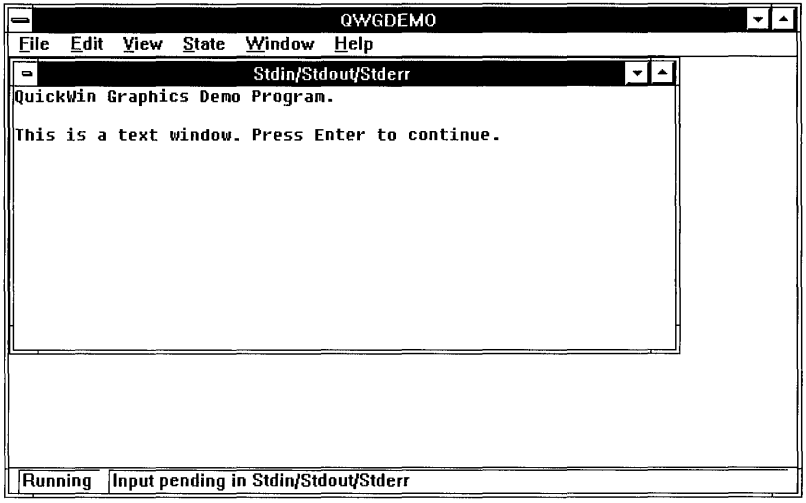
This section explains how to program with the enhanced features of QuickWin to improve the appearance of the QuickWin application's child windows and to further control the behavior of your programs.

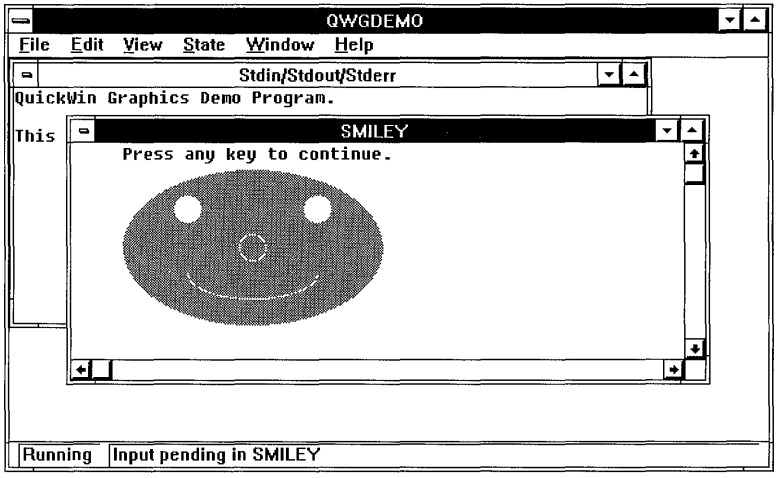
QuickWin Sample Programs

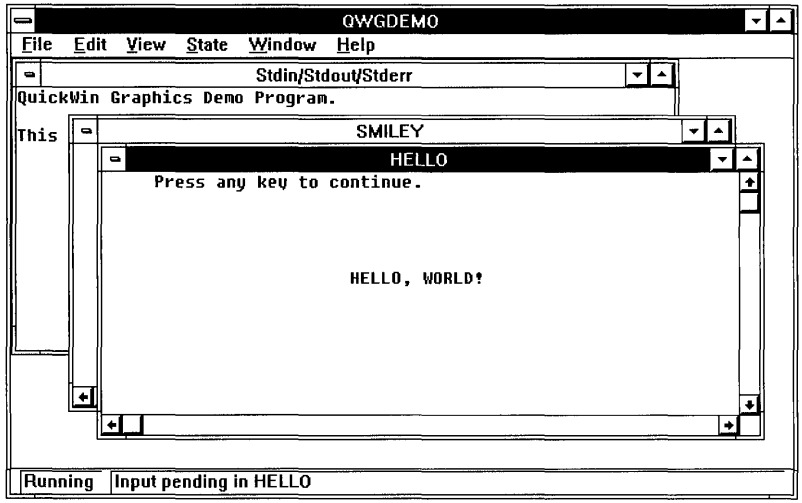
The program QWGDEMO.CPP, found in the \MSVC\SAMPLES\QWGDEMO directory (or in the directory where you placed your sample source files) can be compiled as an enhanced QuickWin program. It demonstrates the enhanced QuickWin features.

Note QWGDEMO.CPP cannot be run as an MS-DOS program. Because it contains QuickWin enhancements, it can be run only in Windows in standard or 386 enhanced mode.

The three illustrations following show the output of QWGDEMO.CPP.







Customizing the About Dialog Box

Use the **_wabout** function to specify the text in your program's About dialog box. This text appears in a dialog box when the user chooses the About command from the QuickWin Help menu. For example, QWGDEMO.CPP uses the following line of code to specify text:

```
result = _wabout( "QuickWin Demo" );
```

Pass the function a pointer to a null-terminated string. The function returns an **int**. Note that the About dialog box always displays an OK button and default information about QuickWin, even if you don't call the **_wabout** function. If you call **_wabout**, the OK button is still displayed, along with the specified string.

The function returns 0 if successful, or a nonzero value if not.

Opening Child Windows

In your QuickWin program, you may want to open new child windows in which to display your program's data. You can have up to 20 graphics child windows opened. You can also have up to 20 text windows opened for a maximum total of 40 text child windows and graphics child windows.

Each graphics window can be in either a text mode or a graphics mode. There are several text modes and graphics modes from which to choose. For a list of available modes, see the description of the **_setvideomode** function in the *Run-Time Library Reference*. The graphics windows are initially set to **_TEXTC80** mode; you can use **_setvideomode** to change the mode. The following table shows the differences

between sending text output to QuickWin text windows, QuickWin graphics windows in graphics mode, and QuickWin graphics child windows in text mode.

Window type	To write "Hello"	Graphics call allowed
Text window	`printf( "Hello\n" );`	None
Graphics window in text mode	`_outtext( "Hello" );`	Text calls only
Graphics window in graphics mode	`_outgtext( "Hello" );` `_outtext( "Hello" );`	Any supported calls (including text calls)

The sample program, QWGDEMO.CPP, uses all three types of windows.

Depending on what you want to do, you can use one of three QuickWin functions, **_wopen**, **_fwopen**, or **_wgopen**, to open new text or graphics child windows.

Opening Text Windows

The **_wopen** function is a low-level routine that returns a file handle, which you can use for text window I/O or to call several other QuickWin functions, such as **_wsetsize**, **_wsetfocus**, and **_wsetscreenbuf**. You can perform I/O in this kind of window with C run-time library functions such as **_write** and **_read**. The use of these functions is explained later in this section.

To write to a text window or read from it as a stream, you need a file pointer of type **FILE ***. The **_fwopen** function is a high-level routine that returns a file pointer you can pass to C run-time library I/O routines, such as **fprintf** and **fscanf**, which require a stream argument.

Note If you open text windows with **_fwopen**, you can use the **_fileno** macro to obtain a file handle for use with QuickWin and other routines that require a handle argument. Do not use such a handle with the **_wclose** function, however.

The _wopeninfo and _wsizeinfo Structures

Both **_wopen** and **_fwopen** require arguments of type **_wopeninfo** and **_wsizeinfo**. These are defined as C structures in the IO.H file for Windows. The **_wopeninfo** structure is declared as follows:

```
struct _wopeninfo
    {
    unsigned int _version;
    const char __far * _title;
    long _wbufsize;
    };
```

The **_version** field contains the Windows version number. Use the constant **_QWINVER**, declared in IO.H. The **_title** field holds a null-terminated string; this

is the title of your window. The **_wbufsize** field contains the size of the window screen buffer (in bytes). The default size is 2048.

The **_wsizeinfo** struct is declared as:

```
struct _wsizeinfo
    {
    unsigned int _version;    /* Use _QWINVER */
    unsigned int _type;       /* Size for window */
    unsigned int _x;          /* Upper-left x-coordinate */
    unsigned int _y;          /* Upper-left y-coordinate */
    unsigned int _h;          /* Height of window */
    unsigned int _w;          /* Width of window */
    };
```

The **_version** field contains the Windows version number. Use the constant **_QWINVER**. For use in opening text windows, the **_type** field specifies the size of the window as one of the following constants:

_WINSIZEMIN
Minimizes the text window

_WINSIZEMAX
Maximizes the text window

_WINSIZECHAR
Uses the listed coordinates in the **_x**, **_y**, **_h**, and **_w** fields for the text window size

If you specify a **_type** field of **_WINSIZEMIN** or **_WINSIZEMAX**, you can leave the **_x**, **_y**, **_h**, and **_w** fields empty.

To open a text window, first declare variables of the **_wopeninfo** and **_wsizeinfo** types and fill in their fields. Then call either **_wopen** or **_fwopen**.

If you pass **NULL** for either the **_wsizeinfo** or **_wopeninfo** argument, the **_fwopen** function uses default values. The **_wopen** function works similarly, except that the **_wopeninfo** argument cannot be **NULL**. You must pass a pointer to a **_wopeninfo** structure.

The **_wopen** function returns an integer file handle to the new text window if successful, or –1 if not. The **_fwopen** function returns a stream pointer to the new text window if successful, or **NULL** if not.

Opening Graphics Windows

Use the **_wgopen** function to open a graphics child window (in text mode or graphics mode). Once you've created a new graphics child window, you need to make it the active window for graphics-routine calls to be directed to it. Each graphics child window is identified by a number called its handle. The

_wggetactive function returns the handle of the active window. You can activate a different QuickWin graphics child window with the **_wgsetactive** function. To do this, you should save the handle of each graphics child window you open with **_wgopen** and then pass the handle of the window you want to make active to the **_wgsetactive** function.

The following example opens a new graphics child window:

```
/* Open a graphics child window and make it the active window */
handle = _wgopen( name );
status = _wgsetactive( handle );
```

Note If an application's child windows are maximized when a call to **_wgopen** is made, the child windows are restored before the new window is opened.

Use **_inchar** to accept input in a graphics child window. The **_inchar** function reads a single character from the keyboard and returns the ASCII value of that character without any buffering. The program QWGDEMO.CPP uses **_inchar** to pause the program until the user presses a key to continue without having to give the input focus to the text child window.

Reading from and Writing to Text Child Windows

Reading from or writing to a QuickWin text window resembles reading from or writing to a file. QuickWin windows behave as I/O streams. You can pass the file pointer obtained from the **_fwopen** function as the stream argument to C run-time library I/O functions.

For example, this code demonstrates writing a text prompt to a window and reading a response from the user:

```
FILE * fp;            /* Declare a file pointer */
struct _wopeninfo wo; /* Declare a variable of type _wopeninfo */
.
.
.
fp = _fwopen( &wo, NULL, "w+" );        /* Open a window */
fprintf( fp, "Enter a filename: \n" );  /* Write to the window */
rewind( fp );                           /* Reset the stream */
fscanf( fp, "%s", &scan );              /* Read from the window */
```

Note Each time you switch from reading to writing or from writing to reading, call the **rewind** function to reset the stream.

For examples of using document windows for input and output, see QWGDEMO.CPP.

Resizing and Positioning Text Child Windows

To resize or reposition a text window, use the **_wsetsize** function. (For information about the **_wsizeinfo** structure, see "Opening Child Windows," on page 137.) You can also examine the current size and position of a text window by calling the **_wgetsize** function.

Note The **_wgetsize** and **_wsetsize** functions cannot be used to resize or reposition graphics child windows.

A child window cannot be larger than its client window.

Both resizing functions require a file handle argument and an argument of type **_wsizeinfo**. The **_wgetsize** function also requires an **int** argument specifying the "request type." The request type can have one of two values: **_WINCURRREQ**, which returns the current size of the window, or **_WINMAXREQ**, which returns the maximum size to which the window can grow (the child window cannot exceed the current size of the client window). You can also query the size of the client (application) window. Pass the manifest constant **_WINFRAMEHAND** as the window handle to **_wgetsize**, which returns information about the client window.

The **_type** field of the **_wsizeinfo** structure can have one of four values: **_WINSIZEMIN**, for a minimized window; **_WINSIZEMAX**, for a maximized window; **_WINSIZERESTORE**, to restore a minimized window to its previous size; or **_WINSIZECHAR**, using which you can specify (in the remaining fields of the **_wsizeinfo** structure) the coordinates of the text window's upper-left corner and the window's height and width in characters.

To illustrate, the following code maximizes a child window:

```
FILE * fp;                  /* File handle to window */
struct _wsizeinfo ws;       /* Size structure variable */
ws._version = _QWINVER;     /* Version value */
ws._type = _WINSIZEMAX;     /* Maximize window */
.
.
.
/* Set the window size */
result = _wsetsize( _fileno( fp ), &ws );
```

The **_wsetsize** and **_wgetsize** functions return 0 if successful, or –1 if not. The **_wgetsize** function also fills in the **_wsizeinfo** structure if successful. You can then extract the size information from the structure.

For additional examples, see QWGDEMO.CPP.

Setting the Amount of Scrollable Text

By default, the screen buffer associated with each QuickWin text window can store 2048 characters. If this amount exceeds the display capacity of the window, QuickWin puts scroll bars on the window so the user can scroll through the window's contents.

The maximum buffer size for a new window can be set by specifying the size in the **_wbufsize** field of the **_wopeninfo** structure that you pass to the **_fwopen** function.

You can also limit the maximum buffer size at any other time with the **_wsetscreenbuf** function. This function takes two arguments: a file handle to the window and the desired upper limit on buffer size. The *bufsiz* argument can be a number or one of the following constants: **_WINBUFDEF**, which uses the default window screen buffer size, or **_WINBUFINF**, which places no limit on the buffer size. Unless you use **_WINBUFINF**, only the most recent characters are stored, up to the buffer's capacity. In any case, the buffer is always allocated dynamically, so that it fits its contents.

To illustrate, the following code resizes a window's buffer to store 16,384 bytes:

```
#define BUFSIZE 16384
result = _wsetscreenbuf( _fileno( fp ), BUFSIZE );
```

You can also use the **_wgetscreenbuf** function to examine the current size of a window's screen buffer.

The **_wsetscreenbuf** function returns 0 if successful, or –1 if not. The **_wgetscreenbuf** function returns the current buffer size (in bytes) or **_WINBUFINF** if successful, or –1 if not.

For further examples, see QWGDEMO.CPP.

Giving Focus to a Text Child Window

When the user selects a text window with the mouse or the keyboard, the selected window is highlighted and appears in front of all other windows if windows are cascaded or is simply highlighted if windows are tiled. The selected window has input focus and is also called the foremost window. To give a text window the focus, and bring it to the front if windows are cascaded, call the **_wsetfocus** function.

For example, before writing to one of several cascaded windows, you can bring the target window to the front with **_wsetfocus** and then write to it, as shown by the following code:

```
/* Check result, then write to the window */
result = _wsetfocus( _fileno( fp ) );
```

You can also learn whether a child window has the focus by calling the **_wgetfocus** function. The **_wgetfocus** function returns an integer handle to the window with the focus if successful, or –1 if not.

For further examples, see QWGDEMO.CPP.

Closing a Child Window

Once you finish using a text window, you usually close it. For windows opened with the **_wopen** function, you can call the QuickWin **_wclose** function. For windows opened with **_fwopen**, you can call the C run-time library **fclose** or **_fcloseall** function. For graphics child windows opened with **_wgopen**, you can call the **_wgclose** function.

The **_wclose** function takes a second argument to specify whether the text window should "persist" (remain on the screen) after closing. The *persist* parameter can have one of the following values: **_WINNOPERSIST**, which erases the window, or **_WINPERSIST**, which leaves the window on the screen. A "persistent" window of this kind no longer responds to I/O calls, but you can select and copy text from it, scroll through its text, and continue to use its menus. To illustrate, you might write a file to a window, then allow the user to examine the file's contents after the window is closed to further writing. For more information about how your windows behave at exit time, see the following section, "Keeping Windows on the Screen."

If you leave the window on the screen, you can later send another **_wclose** call to the same file handle to remove the window.

The following code demonstrates closing a window without leaving it on the screen:

```
result = _wclose( wfh, _WINNOPERSIST );
```

Once you finish using a graphics child window, you can close it with **_wgclose**. All data in the window is lost when you close it. When a graphics call is first made, QuickWin creates a default graphics window named "Graphic1." The QWGDEMO.CPP program calls **_wgclose** to close the "Graphic1" graphics window and free all memory associated with it. The following code demonstrates closing "Graphic1" in QWGDEMO.CPP:

```
/* Close the default "Graphic1" window */
handle = _wggetactive();
status = _wgclose( handle );
/* Open a new graphics child window named "Smiley" */
/* in which to draw a smiley face */
handle = _wgopen( name );
```

Keeping Windows on the Screen

Sometimes it is useful to leave your program's windows on the screen after the program terminates. This allows the user to inspect their contents, use the scroll bars, use the menus, and copy or paste text in the windows.

As described previously, you can use **_wclose** for text windows to control whether your program's windows remain on the screen. QuickWin also gives you additional control over the behavior of your windows when the program calls the **exit** function.

By default, your windows remain on the screen. You can call the **_wsetexit** function in your program to alter that default. You can specify that windows remain on the screen (as in the default), that windows not remain on the screen, or that the user may choose whether windows remain on the screen. If you specify user choice, a dialog box appears with this message:

```
Program terminated with exit code n; Exit window?
```

If the user responds "No" to this dialog box, the program quits without closing windows. The user can examine window contents or select and copy text onto the Clipboard, but further input or output is disabled. Exiting with the Exit command does not open a dialog box.

Call **_wsetexit** at any time to specify the state of your windows upon exit. If the **exit** function is subsequently called, the behavior is based on the value you set. You can pass one of the following manifest constants to **_wsetexit**:

_WINEXITPROMPT
Prompts the user with a dialog box; the user can specify the behavior. This constant's numeric value is 1.

_WINEXITNOPERSIST
Windows do not remain on the screen. This constant's numeric value is 2.

_WINEXITPERSIST
Windows remain on the screen; this is the default value. This constant's numeric value is 3.

The **_wsetexit** function returns 0 if successful, or –1 if not.

Call **_wgetexit** to learn what the current exit setting is. The function returns the numeric value of the current setting (one of the previous values) if successful, or –1 if not.

The following code demonstrates the use of **_wsetexit** and **_wgetexit** to determine the current exit setting and then to reset it:

```
nExit = _wgetexit();
if( nExit == _WINEXITPERSIST )
    _wsetexit( _WINEXITNOPERSIST );
```

Simulating Mouse Clicks in the Menu Bar

Your program can activate a limited subset of menu commands using the **_wmenuclick** function. The commands you can choose are limited to a subset of the Window menu commands as represented by the following constants:

_WINTILE
Tile the windows

_WINCASCADE
Cascade the windows

_WINARRANGE
Arrange any document icons at the bottom of the application window

_WINSTATBAR
Show or hide the status bar

The following code demonstrates using the **_wmenuclick** function to display the status bar:

```
result = _wmenuclick( _WINSTATBAR );
```

The **_wmenuclick** function returns 0 if successful, or –1 if not.

For further examples, see QWGDEMO.CPP.

Yielding Time to Other Applications

If your QuickWin program runs concurrently with other applications for Windows, it should yield processing time to the other applications so they can service their message queues. QuickWin attempts to yield to other applications at appropriate times, but there may be cases where your program should make additional calls to the **_wyield** function.

QuickWin takes care of message processing in Windows for you.

If Windows appears sluggish when your program runs, insert additional **_wyield** calls. In particular, you may want to make **_wyield** calls during lengthy processing loops. This allows the user to select menu commands or switch to another application without having to wait for your program to finish processing.

Note QuickWin programs do not require the standard message loop for Windows.

The **_wyield** function returns void.

Using Custom Icons

The QuickWin library provides default icons for your application and its child windows. Windows displays these icons when the user minimizes the application's client window or its child windows. You can create your own icons and add them to your executable file, and Windows displays them instead of the default icons.

To add icons to your QuickWin program, follow these steps:

1. Create the icon (.ICO) files using the App Studio image editor. For information on using the image editor, see Chapter 7, "Using the Graphics Editor," in the *App Studio User's Guide*.
2. Create a resource script (.RC file) with the contents

   ```
   FRAMEICON   ICON frame.ico
   CHILDICON   ICON child.ico
   GRAPHICICON ICON graphic.ico
   ```

   where `frame.ico` and `child.ico` are the names of the files containing the frame and child icons and `graphic.ico` is the name of the file containing the graphics window icon. The icon resources must have the resource names **FRAMEICON**, **CHILDICON**, and **GRAPHICICON**.
3. Using the Visual Workbench, set a project (if you have not done so already). This is necessary to tell the compiler that you are including a user-defined resource script (.RC file). Then rebuild the project. The icon resources are compiled and added to your executable file. For a detailed description of how to build a QuickWin application, see Chapter 3, "Building a Sample QuickWin Program," in the *Visual Workbench User's Guide*.

Providing Help

A Help file, MSCXX.HLP, is provided with Visual C++. The file contains information on the QuickWin user interface. It must be stored in the same directory as your QuickWin application or in a directory specified in the PATH environment variable.

A user views Help by:

- Choosing Index from the Help menu.
- Highlighting any command on a QuickWin menu and pressing F1.

For information on moving between screens in the Help file, a user chooses Using Help from the Help menu.

Note The QuickWin Help is limited to information about the QuickWin user interface. You cannot add your own context-sensitive help to a QuickWin program. However, if you plan to release a QuickWin application you have developed, you may ship QWIN.HLP along with your executable file(s). This enables your users to obtain Help on any of the QuickWin menus while they are using your application.

7.9 Differences Between MS-DOS Graphics and QuickWin Graphics

This section discusses the ways in which QuickWin graphics routines behave differently from the same MS-DOS functions in GRAPHICS.LIB. The "Routines" entry in each section identifies functions that differ between the two libraries.

Internal Error System

Routines All graphics routines in the QuickWin library

In addition to the normal error messages generated by GRAPHICS.LIB, QuickWin also generates run-time error messages that help you evaluate other problems. If an error occurs during execution, your QuickWin application displays one of the following error message boxes:

- The "QuickWin Error" message box, which displays the error number, the OK button, and either the "Out of Memory" message or the "Internal Error—unexpected error" message. Choose the OK button to terminate current operation; your application does not terminate when an error occurs.

  For example, if you try to copy a logical graphics screen without enough memory, the error message appears. Choose the OK button to terminate the Copy command, then close some applications and try again.

- The "QuickWin Fatal Error" message box, which contains either the "Out of Memory" message or the "Internal Error—unexpected error" message. Your application terminates when a fatal error occurs. The most common reason for a fatal error is insufficient memory during a critical section of operations. For example, a fatal error occurs when no memory is available on startup.

Using Function Keys

Any MS-DOS graphics applications that use function keys for input do not work correctly under QuickWin graphics. Windows traps these keys before the QuickWin application can use them.

Setting the Line-Style Mask

Routines **_setlinestyle**

The mask (line style) used for line drawing in GRAPHICS.LIB has no direct equivalent in QuickWin graphics. QuickWin uses a Windows pen style that matches the mask you specify. The following mapping between line style and Windows pen style is maintained:

Line style	Pen style
0xFFFF	**PS_SOLID**
0xEEEE	**PS_DASH**
0xECEC	**PS_DASHDOT**
0xECCC	**PS_DASHDOTDOT**
0xAAAA	**PS_DOT**
0x0000	**PS_NULL**

For a complete description of Windows pen styles, see the online Windows API documentation.

Setting the Fill Mask

Routines **_setfillmask**

QuickWin graphics fills a shape with the current fill pattern when you specify the **_GFILLINTERIOR** constant. Each bit in the fill pattern whose value is 1 represents a pixel set to the current color; each bit whose value is 0 represents a pixel set to the background color. In GRAPHICS.LIB, the 0-value bit leaves the pixel unchanged.

Checking Graphics Errors with _grstatus

Routines All graphics routines in the QuickWin library

The **_grstatus** function returns the status of the most recently used graphics routine. The return values to **_grstatus** set by QuickWin graphics routines differ from those set by MS-DOS (GRAPHICS.LIB) graphics routines in several ways:

- Any QuickWin graphics routine can set **_grstatus** to **_GRERROR** when there is no active window.
- Because QuickWin graphics applications support all video modes on all monitors, **_grstatus** does not return **_GMODENOTSUPPORTED** for the **_setvideomode** and **_setvideomoderows** functions, as it does in GRAPHICS.LIB.

- The warning codes **_GRCLIPPED** and **_GRNOOUTPUT** are not set at any time in QuickWin.

Registering Fonts

Routines **_registerfonts**

The **_registerfonts** function registers all fonts installed in Windows regardless of what argument is supplied. The *filename* argument is not used but must still be specified. The following example registers all fonts installed in Windows:

```
status = _registerfonts( 'dummy string' );
```

Displaying Character-Based Text

Routines **_outgtext**, **_outtext**, **_settextcolor**, **_settextcursor**, **_wrapon**

These routines send text to the screen in both graphics and text modes.

All standard input and output (using **scanf** and **printf** statements) goes to the default QuickWin text child window. A QuickWin text child window must have the input focus before it can accept standard input. Use **_outtext** and **_outgtext** to send text strings to QuickWin graphics child windows. Note that writing characters from the extended ASCII character set using **_outtext** and **_outgtext** only works with GRAPHICS.LIB.

Calling **_outtext** in either text or graphics mode while a graphics window is in Size To Fit mode displays the text at normal size, as if the graphics window is not in Size To Fit mode. Any redrawing of the window causes the text to be scaled appropriately.

The **_settextcolor** function sets the current text color (attribute). The text does not blink when the color attribute is set in the range of 16–31, as in GRAPHICS.LIB. The colors in the range of 16–31 are the same as those in the range 0–15.

Note that when using GRAPHICS.LIB, if an out-of-range text color is requested in a call to the **_settextcolor** routine the text color is set to the maximum nonblinking color index. When using QuickWin graphics, the text color is left unchanged. Also, unlike the GRAPHICS.LIB **_settextcolor** function (which can set index values greater than 31 for monitors capable of displaying 256 colors), the QuickWin graphics **_settextcolor** function cannot set the index to a value greater than 31.

On systems with Enhanced Graphics Adapters (EGAs), text color 7 (white) appears the same as text color 8 (gray).

On systems with monochrome graphics adapters, calling **_settextcolor** with the following text color indices sets the text color to white: 7, 11, 13, 14, 15, 23, 27,

29, 30, and 31. When using GRAPHICS.LIB, any color index other than black sets the text color to white.

The **_settextcursor** function in QuickWin can set the cursor shape only to a full block cursor or to no cursor. Underline and double underline cursor shapes are not available in QuickWin.

The **_wrapon** function controls whether text output with the **_outtext** function wraps to a new line or is simply truncated when the text output reaches the edge of the defined text window. In QuickWin, the **_wrapon** function returns the previous value of the *option* argument if successful; otherwise, it returns –1 if an internal error occurs.

Selecting Display Options

The routines in this section determine the graphics environment characteristics, establish operational modes for text or graphics, and control the cursor.

Manipulating Screen Page Routines

Routines **_getactivepage**, **_getvisualpage**, **_setactivepage**, **_setvisualpage**

The **_getactivepage** and **_getvisualpage** functions in QuickWin always return a page number of 0. In GRAPHICS.LIB, they return the number of the active video page.

The **_setactivepage** and **_setvisualpage** functions in QuickWin accept only 0 for the *page* argument. In GRAPHICS.LIB, the *page* argument selects the active video page.

Setting Text Output

Routines **_setgtextvector**, **_settextrows**, **_setvideomoderows**

In QuickWin graphics, only left-to-right horizontal text positioning is allowed. You can set the graphics text vector argument only to (1,0). Because this is the default setting, you do not need to call **_setgtextvector**. If you specify anything besides (1,0) to **_setgtextvector**, **_grstatus** returns **_GRERROR**. You can also write a function to display text in a vertical direction (for example, to label the vertical axis of a graph).

In QuickWin graphics text modes, the number of text rows you set using **_settextrows** and **_setvideomoderows** can range from 1 through 256. In QuickWin, the number of rows of text set is not limited by video hardware restrictions, as it is in GRAPHICS.LIB. The size of the text font, though, is always the same.

All graphics modes have one fixed number of text rows that cannot be altered. The number of text rows does not correspond to the number of rows of text that the graphics modes have under GRAPHICS.LIB. Use the return value from **_setvideomode** or the **numtextrows** element in a **_videoconfig** structure to determine how many rows of text are available for a particular graphics mode.

Setting the Video Mode

Routines **_getvideoconfig**, **_setvideomode**

You can set the video mode to any mode in QuickWin graphics whether or not it is supported by your hardware. The QuickWin library creates a bitmap with the same size as the mode requested, but the color and pixel characteristics remain the same as those of the hardware mode running with Windows.

For example, if you are running on a Video Graphics Array (VGA) adapter, setting the video mode to Color Graphics Adapter (CGA) gives you a 320×200 pixel bitmap. This is the same size as a CGA screen, but you receive 16 colors and a pixel size identical to a VGA pixel. If you set the mode to **_SRES256COLOR** on an EGA adapter, QuickWin gives you an 800×600 bitmap (which is the size of a super VGA screen), but only the 16 colors of the EGA adapter. When you choose a graphics mode with a resolution higher than that supported by a particular computer, you need to scroll through the bitmap to see those parts not shown on the screen.

Setting the *mode* argument in **_setvideomode** to **_MAXCOLORMODE** or **_MAXRESMODE** gives you a bitmap correctly sized for the graphics mode running with Windows.

The **_getvideoconfig** function returns **NULL** for no active graphics child window; otherwise, it returns the pointer to the **_videoconfig** structure. The values for the **adapter**, **monitor**, and **memory** elements of the **_videoconfig** structure are always set to 0.

Setting Palettes

Routines **_remapallpalette**, **_remappalette**, **_selectpalette**, **_setbkcolor**

Remapping of colors on computers capable of displaying 20 colors (or fewer) is not allowed in QuickWin. On these computers, the **_remapallpalette**, **_remappalette**, and **_setbkcolor** (in graphics mode) functions return –1 and set **_grstatus** to **_GRERROR**. On computers that can display more than 20 colors, the number of colors that can be remapped is $n-20$ (where n is the number of colors that the computer can display). Similarly, setting a graphics color with **_setcolor** to an index of greater than 235 makes the current color black.

Selecting a palette is not allowed in QuickWin, regardless of the number of colors the computer can display. The **_selectpalette** function returns 0 and sets **_grstatus** to **_GRERROR** in QuickWin.

Drawing Lines

Routines **_lineto**, **_lineto_w**

With QuickWin graphics, the final pixel specified by a **_lineto** call is not drawn. For instance, assume the current position is (1,1). The call

```
status = _lineto (4,4)
```

sets only pixels (1,1), (2,2), and (3,3). The new current position after the **_lineto** call is (4,4); however, that pixel is not set by the **_lineto** call.

Calling Rectangle Functions with a Fill Mask

Routines **_rectangle**, **_rectangle_w**

Suppose you make a call to one of the rectangle functions with the fill flag set to **_gfillinterior** and you use a fill mask other than the default solid mask. In such a case, the rectangle produced is 1 pixel smaller than the dimensions specified in the call. For example, a call that requests a filled rectangle using a fill mask that is not solid that extends from (10,10) to (20,20) produces a rectangle that extends from (10,10) to (19,19).

Drawing Graphics Outside a Viewport

Routines **_arc** functions, **_ellipse** functions, **_lineto** functions, **_outgtext**, **_pie** functions, **_polygon** functions, **_rectangle** functions, **_setpixel** functions

If you try to draw a graphical element that is entirely outside a viewport, GRAPHICS.LIB returns 0 from the graphics call. However, QuickWin returns 1, as if the element had been successfully drawn.

Drawing Lines and Rectangles on Monochrome Adapters

Routines **_lineto** functions, **_polygon** functions, **_rectangle** functions

The following applies only to computers with monochrome graphics adapters. If you draw a line or rectangle that cannot be completely contained in the current clipping region of the graphics child window, a call to the functions listed in "Routines" causes incomplete results.

CHAPTER 8

Programming with Mixed Languages

There are times when your C or C++ programs need to call programs written in other languages or when programs written in other languages need to call your C or C++ functions. This is called mixed-language programming. For example, when a particular subprogram is available commercially in a language other than C or C++ or when algorithms are described more naturally in a different language, you'll probably want to use more than one language.

This chapter describes the elements of mixed-language programming—how to make calls from programs written in one language to routines written in another. (In this chapter, "routine" refers to any function, procedure, or subroutine that can be called from another module.) Unless otherwise stated, the information about C programming also applies to C++ programming.

8.1 Making Mixed-Language Calls

Mixed-language programming always involves a call to a function, procedure, or subroutine. For example, a Basic main module may need to execute a specific task that you want to program separately. Instead of calling a Basic subprogram, however, you decide to call a C function.

Mixed-language calls involve calling functions in separate modules. Instead of compiling all of your source modules with the same compiler, you use different compilers. In the instance mentioned previously, you compile the main-module source file with the Basic compiler, compile another source file (written in C) with the C compiler, and then link the two object files. Figure 8.1 illustrates how the syntax of a mixed-language call works, using this instance as its example.

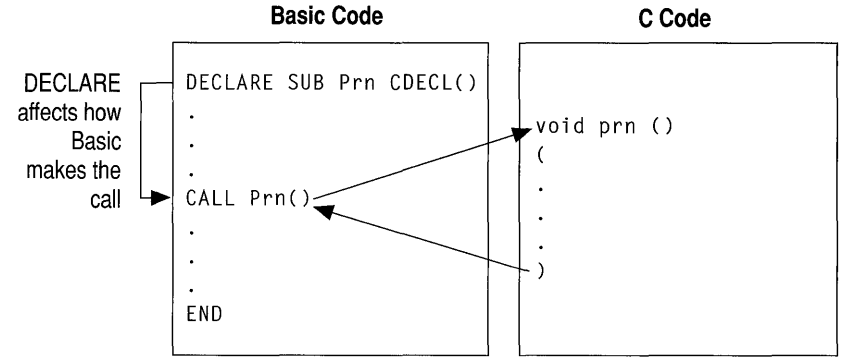


Figure 8.1 Mixed-Language Call

In Figure 8.1, the Basic call to C is `CALL Prn`, similar to a call to a Basic subprogram. There are two differences between this mixed-language call and a call between two Basic modules:

- The subprogram `Prn` is implemented in C, using standard C syntax.
- The implementation of the call in Basic is affected by the **DECLARE** statement, which uses the **CDECL** keyword to create compatibility with C. The **DECLARE** statement is an example of a mixed-language "interface" statement. (For more information on the **DECLARE** statement, see your Basic reference documentation.) These interface statements override default naming and calling conventions. Each language provides its own form of interface.

You can make mixed-language calls to routines regardless of whether they have return values. Table 8.1 shows the correspondence between calls to routines in different languages.

Table 8.1 Language Equivalents for Routine Calls

Language	Call with return value	Call with no return value
Assembly language	Procedure	Procedure
Basic	FUNCTION procedure	Subprogram
C and C++	function	(**void**) function
FORTRAN	FUNCTION	SUBROUTINE
Pascal	Function	Procedure

For example, a C module can make a subprogram call to a FORTRAN subroutine. You can prototype a FORTRAN subroutine as a function with a **void** type.

Note Basic **DEF FN** functions and **GOSUB** subroutines cannot be called from another language.

8.2 Language Convention Requirements

To mix languages, the calling program must observe the same conventions as the called program. The conventions described in this section govern the following:

- How compilers treat identifiers, including function and variable names (naming conventions)
- How the subprogram call is implemented (calling conventions)
- How parameters are passed (parameter-passing conventions)

Naming Convention Requirement

Both the calling program and the called subprogram must agree on the names of identifiers. Identifiers can refer to subprograms (functions, procedures, and subroutines) or to variables that have a public or global scope. Each language alters the names of identifiers.

The term "naming convention" refers to the way a compiler alters the name of a routine before placing it in an object file. Languages may alter the identifier names differently. You can choose from several naming conventions to ensure that the names in the calling program agree with those in the called program. If the names of called routines are stored differently in each object file, the linker is not be able to find a match. It instead reports unresolved external references.

Microsoft compilers place machine code into object files; they also place the names of all publicly accessed routines and variables in object files. Thus, the linker can compare the name of a routine called in one module with the name of a routine defined in another module and recognize a match. Names are stored in the ASCII character set.

Some languages translate names to uppercase.

Basic, FORTRAN, and Pascal use similar naming conventions. They translate each letter to uppercase. Basic type declaration characters (**%**, **&**, **!**, **#**, **$**) are dropped.

Each language recognizes a different number of characters. FORTRAN recognizes the first 31 characters of any name (unless identifier names are truncated), Pascal the first 8, and Basic the first 40. If a name is longer than the language recognizes, the additional characters are simply not placed in the object file.

Note Versions of Microsoft FORTRAN previous to 5.0 truncated identifiers to 6 characters. As of version 5.0, FORTRAN retains up to 31 characters of significance unless you use the /4Yt option.

C and C++ are case-sensitive languages.

Neither the C nor the C++ compiler translates any letters to uppercase.

The C compiler inserts a leading underscore (_) in front of the name of each routine. The C compiler recognizes the first 31 characters of a name (32 including the underscore). You can change the number of characters it recognizes with the /H option; for more information, see Chapter 1, "CL Command Reference," in the *Command-Line Utilities User's Guide.*

The C++ compiler decorates identifier names to retain type information through the linking process. The C++ compiler recognizes the first 247 characters of a name.

Differences in naming conventions are dealt with automatically by mixed-language keywords, as long as you follow two rules:

- If you use any FORTRAN routines that were compiled with the /4Yt command-line option or with the **$TRUNCATE** metacommand enabled, make all names 6 characters or less. Make all names 6 characters or less when using FORTRAN routines compiled with versions of the FORTRAN compiler prior to 5.0.
- Do not use the /NOIGNORECASE linker option (which causes the linker to treat identifiers in a case-sensitive manner). With C or C++ modules, this means that you must be careful not to rely upon differences between uppercase and lowercase letters when programming.

  CL automatically uses the /NOIGNORECASE option when linking. To solve the problems created by this behavior, either link separately with the LINK utility or use all lowercase letters in your C or C++ function names and public variables (global variables that are not declared as static).

Note If you use the command-line option /Gc (use the Pascal/FORTRAN calling convention) when you compile, or if you declare a function or variable with the **__pascal** keyword, the compiler translates your identifiers to uppercase.

Figure 8.2 illustrates a complete mixed-language development example, showing how naming conventions enter into the process.

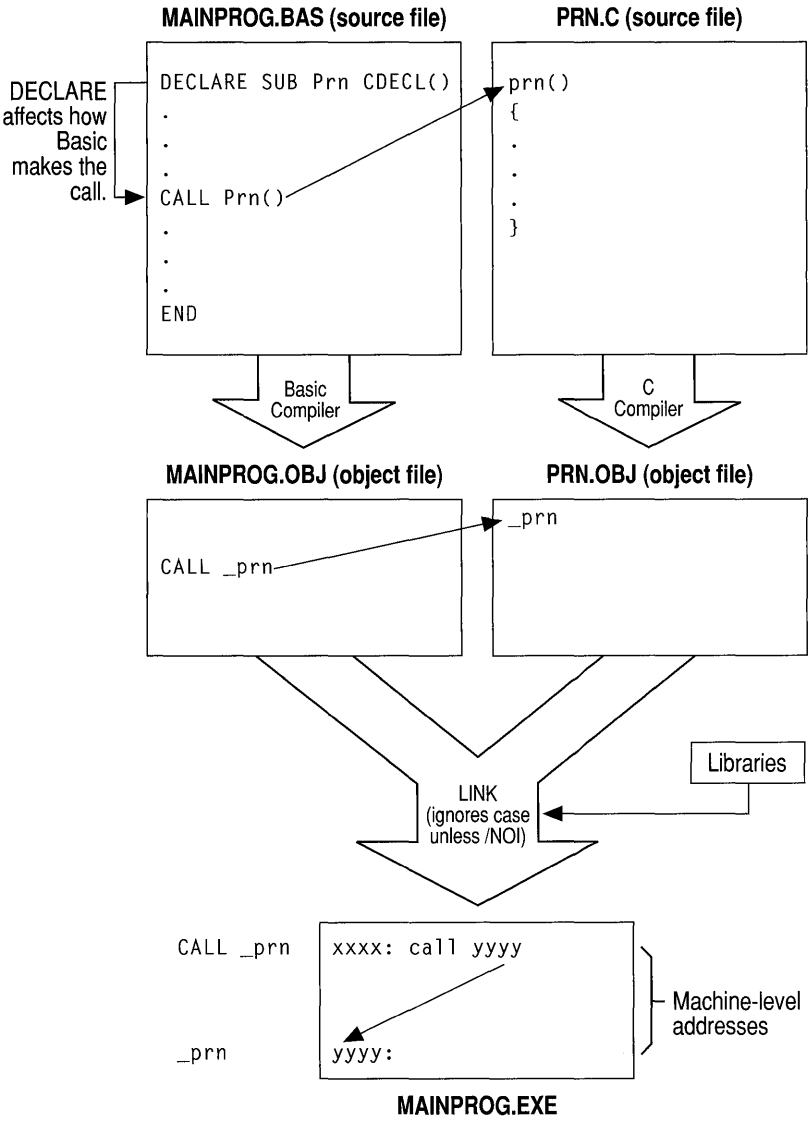


Figure 8.2 Naming Convention

In Figure 8.2, note that the Basic compiler inserts a leading underscore in front of `Prn` as it places the name into the object file, because the **CDECL** keyword directs the Basic compiler to use the C naming convention. Basic also converts all letters to lowercase when this keyword is used. (Converting letters to lowercase is not part of the C naming convention; however, it is consistent with the programming style of many C programs.)

Calling Convention Requirement

The term "calling convention" refers to the way a language implements a call. The choice of calling convention affects the machine instructions that a compiler generates to execute (and return from) a function, procedure, or subroutine call.

It is crucial that the two routines concerned—the routine issuing a call and the routine being called—use the same protocol. Otherwise, the processor may receive inconsistent instructions, causing the program to behave incorrectly.

The use of a calling convention affects programming in three ways:

- The calling routine uses a calling convention to determine the order in which to pass arguments (parameters) to another routine. This convention can be specified in a mixed-language interface statement or declaration.
- The called routine uses a calling convention to determine the order in which to receive the parameters passed to it. In most languages, this convention can be specified in the routine's heading. Basic, however, always uses its own convention to receive parameters.
- Both the calling routine and the called routine must agree on which of them is responsible for adjusting the stack in order to remove parameters.

In other words, each call to a routine uses a certain calling convention; each routine heading specifies or assumes some calling convention. The two conventions must be compatible. With all languages except Basic, it is possible to change the calling convention at the point of the call or at the declaration of the called routine. Usually, however, it is easier to adopt the convention of the called routine. For example, a C function generally uses its own convention to call another C function and uses the Pascal convention to call Pascal.

C++, Basic, FORTRAN, and Pascal use the same standard calling convention. C uses a different convention.

Effects of Calling Conventions

Calling conventions dictate three things:

- The way parameters are communicated from one routine to another. (In Microsoft mixed-language programming, parameters or pointers to the parameters are passed on the stack.)
- The order in which parameters are passed from one routine to another.
- The part of the program responsible for adjusting the stack.

Some languages pass parameters in a different order than C.

The C++, Basic, FORTRAN, and Pascal calling conventions push parameters onto the stack in the order in which they appear in the source code. For example, the Basic statement

```
CALL Calc( A, B )
```

pushes argument A onto the stack before it pushes B. These conventions also specify that the stack is adjusted by the called routine just before returning control to the caller.

The C calling convention pushes parameters onto the stack in the reverse order from their appearance in the source code. For example, the C function call

```
calc( a, b );
```

pushes b onto the stack before it pushes a. In contrast with the other high-level languages, the C calling convention specifies that a calling routine always adjusts the stack immediately after the called routine returns control.

The Basic, FORTRAN, and Pascal conventions produce slightly less object code. However, the C convention makes calling with a variable number of parameters possible. (Because the first parameter is always the last one pushed, it is always on the top of the stack; therefore, it has the same address relative to the frame pointer, regardless of how many parameters are actually passed.) If a C++ function is declared to accept a variable number of parameters, the function automatically uses the C calling convention.

Note The C **__fastcall** keyword, which specifies that parameters are to be passed in registers, is incompatible with programs written in other languages. Avoid using **__fastcall** or the /Gr (use the register calling convention) command-line option for C or C++ functions that you intend to make public to Basic, FORTRAN, or Pascal programs.

Parameter-Passing Requirement

Your programs must agree on the calling convention and the naming convention; they must also agree on the order in which they pass parameters. It is important that your routines send parameters in the same way to ensure proper data transmission and correct program results.

Microsoft compilers support three methods for passing a parameter:

- Near reference, which passes a variable's near (offset) address. This address is expressed as an offset from the default data segment.

  The near reference method gives the called routine direct access to the variable itself. Any change the routine makes to the parameter changes the variable in the calling routine.

- Far reference, which passes a variable's far (segmented) address.

  The far reference method is similar to passing by near reference, except that a longer address is passed. This method is slower than passing by near reference but is necessary when you pass data that is outside the default data segment. (This is an issue in Basic or Pascal only if you have specifically requested far memory.)

- Passing by value, which passes only the variable's value, not its address.

  The value method, the called routine knows the value of the parameter but has no access to the original variable. Changes to a value passed by a parameter have no effect on the value of the parameter in the calling routine.

These different parameter-passing methods mean that you must consider the following when programming with mixed languages:

- You need to make sure that the called routine and the calling routine use the same method for passing each parameter (argument). In most cases, you will need to check the parameter-passing defaults used by each language and possibly make adjustments. Each language has keywords or language features you can use to change parameter-passing methods.
- You may want to choose a specific parameter-passing method rather than using the defaults of any language.

Table 8.2 summarizes the parameter-passing defaults for each language.

Table 8.2 Default Methods by Which Parameters Are Passed

Language	By near reference	By far reference	By value
Basic	All	—	—
C and C++	Near arrays	Far arrays	All data except arrays
FORTRAN	All (medium model)	All (large model)	With attributes[1]
Pascal	**VAR, CONST**	**VARS, CONSTS**	Other parameters

1 When a Pascal or C attribute is applied to a FORTRAN routine, passing by value becomes the default.

8.3 Compiling and Linking

After you have written your source files and decided on a naming convention, a calling convention, and a parameter-passing convention, you are ready to compile and link individual modules.

Compiling with Correct Memory Models

With Basic, FORTRAN, and Pascal, no special options are required to compile source files that are part of a mixed-language program.

With C or C++, not all memory models are compatible with other languages.

Basic, FORTRAN, and Pascal use only far (segmented) code addresses. Therefore, you must use one of two techniques with C or C++ programs that call one of these languages: Compile the C or C++ modules in medium, large, or huge model (using the /AM, /AL, or /AH command-line option), because these models also use far code addresses, or apply the **__far** keyword to the definitions of C or C++ functions you make public. If you use command-line options to specify the medium, large, or huge model, all your function calls become far by default. This means you don't have to declare your functions explicitly with the **__far** keyword.

Choice of memory model affects the default data pointer size in C, C++, and FORTRAN, although this default can be overridden with the **__near** and **__far** keywords. With C, C++, and FORTRAN, choice of memory model also affects whether data objects are located in the default data segment; if a data object is not located in the default data segment, it cannot be passed by near reference.

For more information about code and data address sizes in C and C++, refer to Chapter 2, "Managing Memory for 16-Bit C Programs," and Chapter 3, "Managing Memory for 16-Bit C++ Programs."

Linking with Language Libraries

In most cases, you can easily link modules compiled with different languages. You can do any of the following to ensure that all required libraries link in the correct order:

- Put all language libraries in the same directory as the source files.
- List directories containing all needed libraries in the LIB environment variable.
- Let the linker prompt you for libraries.

In each of the previous cases, the linker finds libraries in the order that it requires them. If you enter the library names on the command line, make sure you enter them in an order that allows the linker to resolve your program's external references.

Here are some points to observe when specifying libraries on the command line:

- If you are using FORTRAN to write one of your modules, you need to link with the /NOD (/NODEFAULTLIBRARYSEARCH) LINK option and explicitly specify all the libraries you need on the LINK command line. You can also specify these libraries with an automatic-response file (or batch file), but you cannot use a default-library search.
- If your program uses both FORTRAN and C, specify the library for the most recent version of each first. In addition, make sure that you choose a C-compatible library when you install FORTRAN.
- If you are listing Basic libraries on the LINK command line, specify those libraries first.

The following example shows how to link two modules, `mod1` and `mod2`, with a user library, GRAFX, the C run-time library, LLIBCE, and the FORTRAN run-time library, LLIBFORE:

```
LINK /NOD mod1 mod2,,,GRAFX+LLIBCE+LLIBFORE
```

8.4 C Calls to High-Level Languages

Just as you can call C routines written using Microsoft Visual C++ from other Microsoft languages, you can call routines written in Microsoft FORTRAN and other Microsoft languages from C. With FORTRAN, Pascal, and C, freestanding routines can be written with no restriction. When calling Basic routines, however, you must write the main program in Basic; any subprograms are free to call one another, whether they are written in C or Basic.

For information about how to pass particular kinds of data, see "Handling Data in Mixed-Language Programming" on page 182.

Executing a Mixed-Language Call

The C interface to other languages uses standard C prototypes, with the **__fortran** or **__pascal** keyword. Using either of these keywords causes the routine to be called with the FORTRAN/Pascal naming and calling convention. (The FORTRAN/Pascal convention also works for Basic.) Here are the recommended steps for executing a mixed-language call from C:

1. Write a prototype for each mixed-language routine called. The prototype should declare the routine **extern** for the purpose of program documentation.

   Instead of using the **__fortran** or **__pascal** keyword, you can simply compile with the "use the Pascal/FORTRAN calling convention" option (/Gc). The /Gc option causes all functions in the module to use the FORTRAN/Pascal naming and calling conventions, except where you apply the **__cdecl** keyword.

2. Pass the values of variables or pointers to variables. You can obtain a pointer to a variable with the address-of (**&**) operator.

   In C, array names are always passed as pointers to the first element of the array; they are always passed by reference.

   The prototype you declare for your function ensures that you are passing the correct length address (that is, near or far).

3. Issue a function call in your program as though you were calling a C function.

4. Always compile the C module in either medium, large, or huge model, or use the **__far** keyword in your function prototype. This ensures that a far (intersegment) call is made to the routine.

Using the __fortran or __pascal Keyword

There are two rules of syntax that apply when you use the **__fortran** or **__pascal** keyword:

- The **__fortran** and **__pascal** keywords modify only the item immediately to their right.
- The **__near** and **__far** keywords can be used with the **__fortran** and **__pascal** keywords in prototypes. The sequences **__fortran __far** and **__far __fortran** are equivalent.

The keywords **__pascal** and **__fortran** have the same effect on the program; using one or the other makes no difference except for internal program documentation. Use **__fortran** to declare a FORTRAN routine, **__pascal** to declare a Pascal routine, and either keyword to declare a Basic routine.

The following example declares `func` to be a Basic, Pascal, or FORTRAN function taking two **short** parameters and returning a **short** value.

```
short __pascal func( short sarg1, short sarg2 );
```

The following example declares `func` to be pointer to a Basic, Pascal, or FORTRAN routine that takes a **long** parameter and returns no value. The keyword **void** is appropriate when the called routine is a Basic subprogram, Pascal procedure, or FORTRAN subroutine, because it indicates that the function returns no value.

```
void ( __fortran * func )( long larg );
```

The following example declares `func` to be a **__near** Basic, Pascal, or FORTRAN routine. The routine receives a **double** parameter by reference (because it expects a pointer to a **double**) and returns a **short** value.

```
short __near __pascal func( __near double * darg );
```

The following example is equivalent to the preceding example (**__pascal __near** is equivalent to **__near __pascal**).

```
short __pascal __near func( __near double * darg );
```

You can make C adopt the conventions of other languages.

When you call a Basic subprogram, you must use the FORTRAN/Pascal conventions to make the call. When you call FORTRAN or Pascal, however, you have a choice. You can make C adopt the conventions described in the previous section, or you can make the FORTRAN or Pascal routine adopt the C conventions.

To make a FORTRAN or Pascal routine adopt the C conventions, put the **C** attribute in the heading of the routine's definition. The following example shows the syntax for the **C** attribute in a FORTRAN subroutine-definition heading:

```
SUBROUTINE FFROMC [C] (N)
INTEGER*2 N
```

The following example shows the syntax for the **C** attribute in a Pascal procedure-definition heading:

```
PROCEDURE Pfromc( n : INTEGER ) [C];
```

To make a C function adopt the FORTRAN/Pascal conventions, declare the function as **__fortran** or **__pascal**. For example,

```
void __pascal CfromP( int n );
```

8.5 C Calls to Basic

No Basic routine can be executed unless the main program is in Basic, because a Basic routine requires the environment to be initialized in a way that is unique to Basic. No other language performs this special initialization.

However, your program can start up in Basic, call a C function that does most of the work of the program, and then call Basic subprograms and function procedures as needed. Figure 8.3 illustrates how to do this.

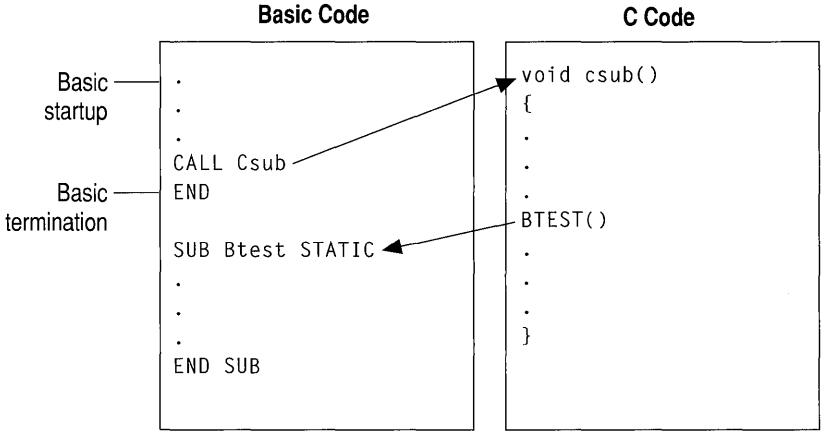


Figure 8.3 C Call to Basic

Follow these rules when you call Basic from C:

1. Start up in a Basic main module. You use the **DECLARE** statement to provide an interface to the C module.
2. In the C module, write a prototype for the Basic routine and include type information for parameters. Use either the **__fortran** or **__pascal** keyword to modify the routine itself.
3. Make sure that all data is passed as near pointers. Basic can pass data in a variety of ways but is unable to receive data in any form other than near reference. With near pointers, the program assumes that the data is in the default data segment. If you want to pass data that is not in the default data segment, copy the data to a variable in the default data segment.
4. Compile the C module in medium or large model to ensure far (intersegment) calls.

The following example demonstrates a Basic program that calls a C function. The C function then calls a Basic function that returns twice the number passed to it and a Basic subprogram that prints two numbers.

```
' Basic source
'
' The main program is in Basic because of Basic's startup
' requirements. The Basic main program calls the C function
' Cprog.
'
' Cprog calls the Basic subroutine Dbl.
'
DEFINT A-Z
DECLARE SUB Cprog CDECL()
CALL Cprog
END
'
FUNCTION Dbl(N) STATIC
    Dbl = N*2
END FUNCTION
'
SUB Printnum(A,B) STATIC
    PRINT "The first number is ";A
    PRINT "The second number is ";B
END SUB
```

```
/* C source; compile in medium or large model */

int __fortran dbl( int __near * N );
void __fortran printnum( int __near * A, int __near * B );

void cprog()
{
    int a = 5;
    int b = 6;

    printf( "%d times 2 is %d\n", a, dbl( &a ) );
    printnum( &a, &b );
}
```

In the previous example, note that the addresses of `a` and `b` are passed, because Basic expects to receive addresses for parameters. This is important because C passes parameters by value unless you use the address-of (**&**) operator to obtain the address or are passing an array. Also note that the function prototype for `printnum` declares the parameters as near pointers. The prototype causes the variables to be passed by near reference. If `a` or `b` is declared as **__far**, the C compiler issues a warning that you are converting a far pointer to a near pointer and that a segment was lost in the conversion.

Calling and naming conventions are resolved by the **CDECL** keyword in the Basic declaration of `Cprog`, and by the **__fortran** keyword in the C declaration of `dbl` and `printnum`.

Basic can invoke one of your functions as part of the termination procedure.

Versions of Microsoft QuickBasic™ later than 4.0 provide a "user entry point," **B_OnExit**, which can be called directly from C. You can use the **B_OnExit** function to make sure you have performed an orderly termination. The following example shows how to use **B_OnExit**.

```
#include <malloc.h>     /* For declaration of _fmalloc */
#include <stdlib.h>     /* For declaration of onexit_t */

/* The prototype for B_OnExit declares it as a function
 * returning type onexit_t that takes one parameter. The
 * parameter is a far pointer to a function that returns
 * no value.
 */
extern onexit_t __pascal __far B_OnExit( onexit_t );
void TermProc( void );

int * p_IntArray;

void InitProc( void )
{
    /* Allocate far space for 20-integer array */
    p_IntArray = (int *)_fmalloc( 20 * sizeof( int ) );
    /* Log termination routine (TermProc) with Basic. */
    B_OnExit( TermProc );
}

void TermProc( void )
{
    free( p_IntArray );     /* Release far space allocated */
}                           /*   previously by InitProc.   */
```

8.6 C Calls to FORTRAN

This section shows two examples of C-FORTRAN programs. There are two types of subprogram calls to FORTRAN routines: calls to subroutines and calls to functions. Functions return a value; subroutines do not. The examples in this section illustrate how to handle the difference between function and subroutine calls.

Calling a FORTRAN Subroutine from C

The following example demonstrates a C main module calling a FORTRAN subroutine, **MAXPARAM**. This subroutine adjusts the lower of two arguments to be equal to the higher argument.

```
/* C source file - calls FORTRAN subroutine
 * Compile in medium or large model
 */

extern void __fortran maxparam( int __near * I, int __near * J );

/* Declare as void, because there is no return value.
 * FORTRAN keyword causes C to use FORTRAN/Pascal
 * calling and naming conventions.
 * Two integer parameters are passed by near reference.
 */

main()
{
    int a = 5;
    int b = 7;

    printf( "a = %d, b = %d", a, b );
    maxparam( &a, &b );
    printf( "a = %d, b = %d", a, b );
}
```

```
C   FORTRAN source file, subroutine MAXPARAM
C
$NOTRUNCATE

       SUBROUTINE MAXPARAM (I, J)
       INTEGER*2 I [NEAR]
       INTEGER*2 J [NEAR]
C
C   I and J received by near reference,
C   because of NEAR attribute
C
       IF (I .GT. J) THEN
          J = I
       ELSE
          I = J
       ENDIF
       END
```

In the previous example, the C program adopts the naming convention and calling convention of the FORTRAN subroutine. The two programs must agree on whether

parameters are to be passed by reference or by value. The following keywords affect how the two programs interface:

- The **__fortran** keyword directs C to call `maxparam` with the FORTRAN/Pascal naming convention (as `MAXPARAM`); **__fortran** also directs C to call `maxparam` with the FORTRAN/Pascal calling convention.
- Because the FORTRAN subroutine `MAXPARAM` may alter the value of either parameter, both parameters must be passed by reference. In this case, near reference was chosen; this method is specified in C by the use of near pointers, and in FORTRAN by applying the **NEAR** keyword to the parameter declarations.

  Far reference can be specified by using far pointers in C. In that case, you don't declare the FORTRAN subroutine `MAXPARAM` with the **NEAR** keyword. If you compile the FORTRAN program in medium model, declare `MAXPARAM` using the **FAR** keyword.

Calling a FORTRAN Function from C

The following example demonstrates a C main module calling the FORTRAN function `fact`. This function returns the factorial of an integer value.

```
/* C source file - calls FORTRAN function.
 * Compile in medium or large model.
 */

int __fortran fact( int N );

/* FORTRAN keyword causes C to use FORTRAN/Pascal
 * calling and naming conventions.
 * Integer parameter passed by value.
 */

main()
{
    int x = 3;
    int y = 4;

    printf( "The factorial of x   is %4d", fact( x ) );
    printf( "The factorial of y   is %4d", fact( y ) );
    printf( "The factorial of x+y is %4d", fact( x + y ) );
}
```

```
C   FORTRAN source file - factorial function
C
$NOTRUNCATE
       INTEGER*2 FUNCTION FACT (N)
       INTEGER*2 N [VALUE]
C
C   N is received by value, because of VALUE attribute
C
         INTEGER*2 I
         FACT = 1
         DO 100 I = 1, N
            FACT = FACT * I
100      CONTINUE
         RETURN
         END
```

In the previous example, the C program adopts the naming convention and calling convention of the FORTRAN subroutine. Both programs must agree on whether parameters are passed by reference or by value. Note that the C program passes the parameters by value rather than by reference. Passing parameters by value is the default for C. To accept parameters passed by value, the keyword **VALUE** is used in the declaration of `N` in the FORTRAN function. The **__fortran** keyword directs C to call `fact` with the FORTRAN/Pascal naming convention (as `FACT`); **__fortran** also directs C to call `fact` with the FORTRAN/Pascal calling convention.

When passing a parameter that should not be changed, pass the parameter by value. Passing by value is the default method in C and is specified in FORTRAN by applying the **VALUE** attribute to the parameter declaration.

8.7 C Calls to Pascal

This section shows two examples of C-Pascal programs. There are two types of subprogram calls to Pascal routines: calls to procedures and calls to functions. Functions return a value; procedures do not. The examples in this section illustrate how to handle the difference between function and procedure calls.

Calling a Pascal Procedure from C

The following example demonstrates a C main module calling a Pascal procedure, `maxparam`. This procedure adjusts the lower of two arguments to be equal to the higher argument.

```
/* C source file - calls Pascal procedure.
 * Compile in medium or large model.
 */

void __pascal maxparam( int __near * a, int __near * b );

/* Declare as void, because there is no return value.
 * The __pascal keyword causes C to use FORTRAN/Pascal
 * calling and naming conventions.
 * Two integer parameters are passed by near reference.
 */

main()
{
    int a = 5;
    int b = 7;

    printf( "a = %d, b = %d", a, b );
    maxparam( &a, &b );
    printf( "a = %d, b = %d", a, b );
}

(* Pascal source code - Maxparam procedure. *)

MODULE Psub;
PROCEDURE Maxparam( VAR a:INTEGER; VAR b:INTEGER );

(* Two integer parameters are received by near reference.
 * Near reference is specified with the VAR keyword.
 *)
BEGIN
    IF a > b THEN
        b := a
    ELSE
        a := b
    END;
END.
```

In the previous example, the C program adopts the Pascal naming convention and calling convention. Both programs must agree on whether parameters are passed by reference or by value; the following keywords affect the conventions:

- The **__pascal** keyword directs C to call `Maxparam` with the FORTRAN/Pascal naming convention (as `MAXPARAM`); **__pascal** also directs C to call `Maxparam` with the FORTRAN/Pascal calling convention.
- Because the procedure `Maxparam` can alter the value of either parameter, both parameters must be passed by reference. In this case, near reference is used; this method is specified in C by the use of near pointers, and in Pascal with the **VAR** keyword.

  Far reference can be specified by using far pointers in C. To specify far reference in Pascal, use the **VARS** keyword instead of **VAR**.

Calling a Pascal Function from C

The following example demonstrates a C main module calling a Pascal function, `fact`. This function returns the factorial of an integer value.

```
/* C source file - calls Pascal function.
 * Compile in medium or large model.
 */

int __pascal fact(int n);

/* PASCAL keyword causes C to use FORTRAN/Pascal
 * calling and naming conventions.
 * Integer parameter passed by value.
 */

main()
{
    int x = 3;
    int y = 4;

    printf( "The factorial of x   is %4d", fact( x ) );
    printf( "The factorial of y   is %4d", fact( y ) );
    printf( "The factorial of x+y is %4d", fact( x + y ) );
}

(* Pascal source code - factorial function. *)
MODULE Pfun;
FUNCTION Fact (n : INTEGER) : INTEGER;
```

```
(* Integer parameters received by value, the Pascal default. *)

    BEGIN
        Fact := 1;
        WHILE n > 0 DO
            BEGIN
                Fact := Fact * n;
                n := n - 1;          (* Parameter n modified. *)
            END;
    END;
END.
```

In the previous example, the C program adopts the Pascal naming convention and calling convention. Both programs must agree on whether parameters are passed by reference or by value. The **__pascal** keyword directs C to call `fact` with the FORTRAN/Pascal naming convention (as `FACT`); **__pascal** also directs C to call `fact` with the FORTRAN/Pascal calling convention.

The Pascal function `fact` should receive a parameter by value. Otherwise, the Pascal function corrupts the parameter's value in the calling module. Passing by value is the default method for both C and Pascal.

8.8 C Calls to Assembly Language

In Visual C++, you can write assembly-language programs either by using the inline assembler or by creating a stand-alone module using the Microsoft Macro Assembler (MASM). If you use the inline assembler, you do not need to take any special precautions other than those outlined in Chapter 4, "Using the 16-Bit Inline Assembler." This section explains the techniques for interfacing your assembly-language routines with your C program.

When deciding whether to use the inline assembler or MASM, there are several considerations. Here are some advantages MASM provides over the inline assembler:

- MASM supports declaration of data in MASM format; inline assembly does not.
- MASM has a more powerful macro capability than does inline assembly.
- Modules written for MASM can be interfaced more easily with modules written in more than one Microsoft high-level language.
- MASM assembles large assembly-language programs more quickly than the inline assembler.
- MASM supports assembly-language code written prior to the existence of the inline assembler.
- MASM error messages and warnings are more complete than those of the inline assembler.

The inline assembler is far more efficient for some assembly-language programming tasks. Here are some of the benefits of the inline assembler:

- You can do spot optimizations by including short sections of assembly-language code in your C programs with the inline assembler.
- Code written in inline assembler does not necessarily incur the overhead of a function call; code assembled using MASM always does.
- You can include inline assembly code in your C source files; code written for MASM must be in a separate file.

Writing the Assembly-Language Procedure

You must write your assembly-language procedure so that it uses the same calling conventions and naming conventions as your C program. If you follow these conventions, you can write recursive procedures (procedures that call themselves), and you can use the CodeView debugger to locate errors in the code.

Note This section discusses only the simplified segment directives provided with MASM version 5.0 or later. If you are using a version prior to 5.0, you have to specify complete **SEGMENT** directives.

The standard assembly-language interface method consists of the following steps:

1. Set up the procedure
2. Enter the procedure
3. Allocate local data (optional)
4. Preserve register values
5. Access parameters
6. Return a value (optional)
7. Exit the procedure

The next sections describe each of these steps in detail.

Setting Up the Procedure

The linker cannot combine a assembly-language procedure with a C program unless you define compatible segments and declare the procedure properly. Perform the following steps to set up the procedure:

- Use the **.MODEL** directive at the beginning of the source file; this directive automatically causes the appropriate kind of returns to be generated (**NEAR** for tiny, small, or compact model, **FAR** for medium, large, or huge model).

  If you are using a version of MASM prior to 5.0, declare the procedure **NEAR** for small or compact model, **FAR** for medium, large, or huge model.
- Use the simplified segment directives **.CODE** and **.DATA** to declare the code and data segments.

  If you are using a version of MASM prior to 5.0, declare the segments using the **SEGMENT**, **GROUP**, and **ASSUME** directives. These directives are described in the *Microsoft Macro Assembler Reference*.
- Use the **PUBLIC** directive to declare the procedure label public. This declaration makes the procedure visible to other modules. Also declare any data you want to make public as **PUBLIC**.
- Use the **EXTRN** directive to declare any global data or procedures accessed by the routine as external. The safest way to use **EXTRN** is to place the directive outside any segment definition; however, place near data inside the data segment.
- Observe the C naming convention; precede all procedure names and global data names with an underscore.

Entering the Procedure

When you enter the procedure, in most cases you set up a "stack frame." This allows you to access parameters passed on the stack and to allocate local data on the stack. You do not need to set up the stack frame if your procedure accepts no arguments and does not use the stack.

To set up the stack frame in a 16-bit program, issue these instructions:

```
push    bp
mov     bp,sp
```

To set up the stack frame in a 32-bit program, issue these instructions:

```
push    ebp
mov     ebp,esp
```

This sequence establishes BP as the frame pointer. You cannot use SP for this purpose because it is not an index or base register. Also, the value of SP may change as more data is pushed onto the stack. However, the value of the base register BP remains constant for the life of the procedure unless your program changes it, so each parameter can be addressed as an offset from BP.

The previous instruction sequence preserves the value of BP, because it is needed in the calling procedure as soon as your assembly-language procedure returns. The value in SP is transferred to BP to establish a stack frame on entry to the procedure.

Allocating Local Data

Your assembly-language procedure can use the same technique for allocating temporary storage for local data that is used by high-level languages. To set up local data space, decrease the contents of SP just after setting up the stack frame. (To ensure correct execution, always increase or decrease SP by an even number.) Decreasing SP reserves space on the stack for local data. You must restore the space at the end of the procedure as follows:

```
push    bp
mov     bp,sp
sub     sp,space
```

In the previous example, `space` is the total size in bytes of the local data you want to allocate. Local variables are then accessed as fixed negative displacements from BP.

In the following example, the entry sequence establishes a stack frame and allocates temporary local storage for two words (4 bytes) of data. Later in the example, the program accesses the local storage, initializing both words to 0.

```
push    bp          ; Save old stack frame.
mov     bp,sp       ; Set up new stack frame.
sub     sp,4        ; Allocate 4 bytes of local storage.
.
.
.
mov     WORD PTR [bp-2],0
mov     WORD PTR [bp-4],0
```

Note that local variables are also called dynamic, stack, or automatic variables.

Preserving Register Values

A procedure called from C should preserve the values of SI, DI, SS, and DS (in addition to BP, which is already saved). You should push any register value that your procedure modifies onto the stack after setting up the stack frame and

allocating local storage, but prior to entering the main body of the procedure. Registers that your procedure does not alter need not be preserved.

Warning Routines that your assembly-language procedure calls must not alter the SI, DI, SS, DS, or BP registers. If they do, and you have not preserved the registers, they can corrupt the calling program's register variables, segment registers, and stack frame, causing program failure. If your procedure modifies the direction flag using the **STD** or **CLD** instructions, you must preserve the flags register.

The following example shows an entry sequence that sets up a stack frame, allocates 4 bytes of local data space on the stack, then preserves the SI, DI, and flags registers.

```
push    bp          ; Save caller's stack frame.
mov     bp,sp       ; Establish new stack frame.
sub     sp,4        ; Allocate local data space.
push    si          ; Save SI and DI registers.
push    di
pushf               ; Save the flags register.
...
```

In the preceding example, you must exit the procedure with the following code:

```
popf                ; Restore the flags register.
pop     di          ; Restore the old value in the DI
                    ;   register.
pop     si          ; Restore the old value in the SI
                    ;   register.
mov     sp,bp       ; Restore the stack pointer.
pop     bp          ; Restore the frame pointer.
ret                 ; Return to the calling routine.
```

If you do not issue the preceding instructions in the order shown, you place incorrect data in registers. Follow these rules when restoring the calling program's registers, stack pointer, and frame pointer:

- Pop all registers that you preserve in the reverse order from which they were pushed onto the stack. For example, in the preceding code SI and DI are pushed and DI and SI are popped.
- Restore the stack pointer by transferring the value of BP into SP before restoring the value of the frame pointer.
- Always restore the frame pointer last.

Accessing Parameters

Once you have established the frame pointer, allocated local storage (if required), and pushed any registers that need to be preserved, you can write the main body of the procedure. Figure 8.4 shows how functions that observe the C calling convention use the stack frame.

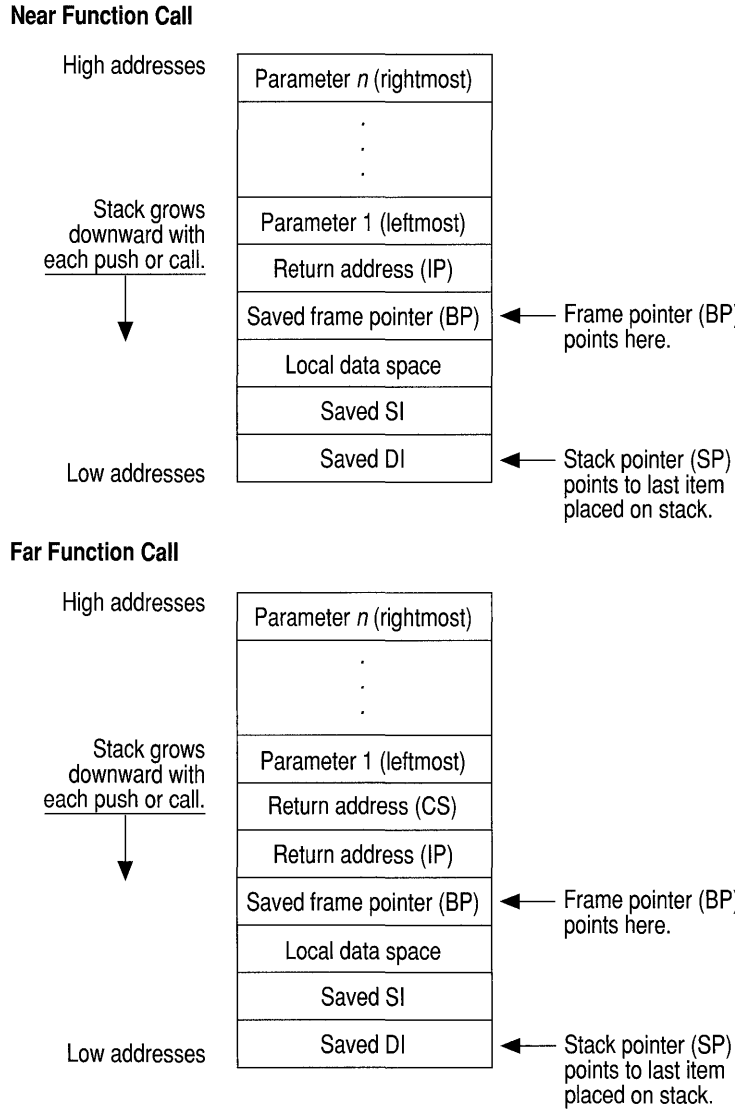


Figure 8.4 C Stack Frame

The stack frame for the assembly-language procedure shown in Figure 8.4 is established by the following:

1. The calling program pushes each of the parameters onto the stack, after which SP points to the last parameter pushed.
2. The calling program issues a **CALL** instruction, which causes the return address (the place in the calling program to which control ultimately returns) to be placed on the stack. This address can be either 2 bytes long (for near calls) or 4 bytes long (for far calls). SP now points to this address.
3. The first instruction of the called procedure saves the old value of BP with the instruction `push bp`. SP now points to the saved copy of BP.
4. BP is used to hold the current value of SP, with the instruction `mov bp,sp`. BP therefore now points to the old value of BP (saved on the stack).
5. While BP remains constant throughout the procedure, SP is often decreased to provide room on the stack for local data or saved registers.

In general, the displacement from BP for a parameter *x* is equal to the size of the return address plus two plus the total size of parameters between *x* and BP.

To calculate the size of parameters between *x* and BP, you must start with the rightmost parameter because C pushes parameters from right to left. For example, consider a **FAR** procedure that has one argument of type **int** (2 bytes). The displacement of the parameter is

```
Argument's displacement = size of far return address + 2
                        = 4 + 2
                        = 6
```

The argument can thus be loaded into BP with the following instruction:

```
mov     bx,[bp+6]
```

After you determine the displacement of each parameter, you can use **EQU** directives or structures to refer to the parameter with a single identifier name in your assembly source code. For example, you can use a more readable name to reference the parameter at `bp+6` if you put the following statement at the beginning of the assembly source file:

```
Arg1    EQU     [bp+6]
```

You can then refer to the first parameter in your source as `Arg1` in any instruction. Use of this feature is optional.

For far (segmented) addresses, Visual C++ pushes the segment address before pushing the offset address. When pushing arguments larger than 2 bytes, high-order words are always pushed before low-order words and parameters larger than 2 bytes are stored on the stack in the order most significant to least significant. This

standard for pushing segment addresses before pushing offset addresses facilitates the use of the assembly-language instructions **LDS** (load data segment) and **LES** (load extra segment).

Returning a Value

Your assembly-language procedure can return a value to a C calling program. All return values of 4 bytes or less are passed in registers. Far pointers to return values larger than 4 bytes are returned in the DX and AX registers. The DX register contains the segment address; the AX register contains the offset relative to the segment contained in DX.

Table 8.3 shows the register conventions for returning simple data types to a C program.

Table 8.3 Register Conventions for Simple Return Values

Data type	Returned in
char	AL
int, short, __near *	AX
long, __far *	High-order portion (or segment address) in DX, low-order portion (or offset address) in AX

Your procedures can return structures.

To return a structure from a procedure that uses the C calling convention, you must copy the structure to a global variable, then return a pointer to that variable in the AX register (or in DX:AX, if you compiled in compact, large, or huge model).

Procedures that use the FORTRAN/Pascal calling convention return structures similarly, with the following exceptions:

- The calling program allocates space for the return value on the stack.
- The calling program passes a pointer to the location where the return value is to be placed in a hidden parameter.
- Instead of copying your structure into a global data item, you copy it into the location pointed to by the hidden parameter.
- You must still return the pointer to that location in the AX register (or in DX:AX for far data models).

You can return floating-point values from your procedures.

Procedures that use the C calling convention and return type **float** or type **double** must always copy their return values into the global variable **__fac**. To return floating-point values from procedures declared with the FORTRAN/Pascal calling convention, you must return the result on the stack, just as you do a structure.

To return a value of type **long double**, you must place the value on the numeric data processor (NDP, or 80*x*87) stack using the **FLD** instruction. The C run-time math

routines guarantee that the only value on the NDP stack is a return value; your routines must observe the same rule.

Exiting the Procedure

Before you exit your assembly-language procedure, you must perform several steps to restore the calling program's environment. Whether you perform some of these steps depends on which actions you took in allocating space for local variables and preserving registers.

You must follow these steps (if appropriate to your procedure) in the order shown:

1. If you saved any of the registers SS, DS, SI, or DI, they must be popped off the stack in an order reverse to that in which they were saved. If you pop these registers in any other order, your program will behave incorrectly.
2. If you allocated local data space at the beginning of the procedure, you must restore SP with the instruction `mov sp,bp`.
3. Restore BP with the instruction `pop bp`. This step is always necessary.
4. Return to the calling program by issuing the **ret** instruction.

The following example shows the simplest possible entry and exit sequence. In the entry sequence, no registers are saved and no local data space is allocated.

```
push   bp
mov    bp,sp   ; Set up the new stack frame.
.
.
.
pop    bp      ; Restore the caller's stack frame.
ret
```

The following example shows an entry and exit sequence for a procedure that saves SI and DI and allocates local data space on the stack.

```
push   bp
mov    bp,sp   ; Establish local stack frame.
sub    sp,4    ; Allocate space for local data.
push   si      ; Preserve the SI and DI registers.
push   di
.
.
.
pop    di      ; Pop saved registers.
pop    si
mov    sp,bp   ; Free local data space.
pop    bp      ; Restore old stack frame.
ret
```

8.9 C++ Calls to High-Level Languages

In C++, you can specify a linkage specification to permit communication between a C++ module and modules written in other languages. Visual C++ supports only the "C" linkage specification.

You declare a linkage specification as follows:

```
extern "C"
{
    void prn();
}
```

This example declares `prn` to be a function with C linkage. Calls to that function are made using the C calling convention.

To call functions written in languages other than C, declare the function as you would in C and use a "C" linkage specification. For example, to call the Pascal function `fact`, declare it as follows:

```
extern "C" {    int __pascal fact( int n ); }
```

This example declares `fact` to be a function with the Pascal calling convention.

If you want a C++ function to be called from other languages, you must declare it with the **extern "C"** linkage specification. Such a function can be called from another language in the same way a C function is called. You cannot declare a member function with a linkage specification. You can specify a linkage specification for only one instance of an overloaded function; all other instances of an overloaded function have C++ linkage.

For more information on the **extern "C"** linkage specification, see the *C++ Language Reference*.

8.10 Handling Data in Mixed-Language Programming

This section contains detailed information about naming and calling conventions in a mixed-language program. It also describes how various languages represent strings, numerical data, arrays, and logical data.

Default Naming and Calling Conventions

Each language has its own default naming and calling conventions, as shown in Table 8.4.

Table 8.4 Default Naming, Calling, and Parameter-Passing Conventions

Language	Calling convention	Naming convention	Parameter-passing convention
Basic	FORTRAN/Pascal	Case insensitive	Near reference
C	C	Case sensitive	Value (scalar variables), reference (arrays and pointers)
C++	FORTRAN/Pascal	Case sensitive	Value (scalar variables), reference (arrays and pointers)
FORTRAN	FORTRAN/Pascal	Case insensitive	Reference
Pascal	FORTRAN/Pascal	Case insensitive	Value

Basic Conventions

When you call Basic routines from C, you must pass all arguments by near reference (near pointer). You can modify the conventions observed by Basic routines that interface with C functions by using the **DECLARE**, **BYVAL**, **SEG**, and **CALLS** keywords. For more information on these keywords, see your Basic reference documentation.

FORTRAN Conventions

You can modify the conventions observed by FORTRAN routines that call C functions by using the **INTERFACE**, **VALUE**, **PASCAL**, and **C** keywords. For more information about the use of these keywords, see your FORTRAN reference documentation.

Pascal Conventions

You can modify the conventions observed by Pascal routines that interface with C functions by using the **VAR**, **CONST**, **ADR**, **VARS**, **CONSTS**, **ADRS**, and **C** keywords. For more information about the use of these keywords, see your Pascal reference documentation.

Numeric Data Representation

Table 8.5 shows how to declare numeric variables of similar type in different languages.

Table 8.5 Equivalent Numeric Data Types

Basic	C and C++	FORTRAN	Pascal
x%	**short**	**INTEGER*2**	**INTEGER2**
INTEGER	**int**	—	**INTEGER** (default)
—	**unsigned short**[1]	—	**WORD**
—	**unsigned**	—	—
x&	**long**	**INTEGER*4**	**INTEGER4**
LONG	—	**INTEGER** (default)	—
—	**unsigned long**[1]	—	—
x!	**float**	**REAL*4**	**REAL4**
x (default)	—	**REAL**	**REAL** (default)
SINGLE	—	—	—
x#	**double**	**REAL*8**	**REAL8**
DOUBLE	—	**DOUBLE PRECISION**	—
—	**long double**	—	—
—	**unsigned char**	**CHARACTER*1**[2]	**CHAR**

1 Types **unsigned short** and **unsigned long** are not supported by Basic or FORTRAN. Type **unsigned long** is not supported by Pascal. A signed integral type can be substituted, but the maximum range is less.

2 The FORTRAN type **CHARACTER*1** is not the same as **LOGICAL**.

The FORTRAN types **COMPLEX*8** and **COMPLEX*16** are not implemented in C but can be represented with structures. The FORTRAN types **LOGICAL*2** and **LOGICAL*4** are also not implemented in C. **LOGICAL*2** is stored as a 1-byte Boolean indicator followed by an unused byte; **LOGICAL*4** is stored as a 1-byte Boolean indicator followed by three unused bytes.

Strings

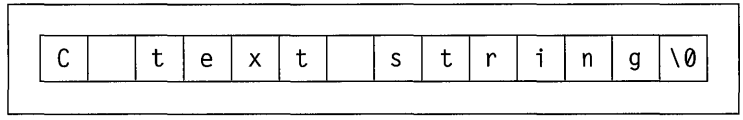
Each language implements strings differently. This section describes the ways that strings are implemented in Microsoft languages.

C and C++ String Format

C and C++ store strings as arrays of bytes and use a null character (`'\0'`) as an end-of-string delimiter. For example, consider the following string:

```
char c_string[] = "C text string";
```

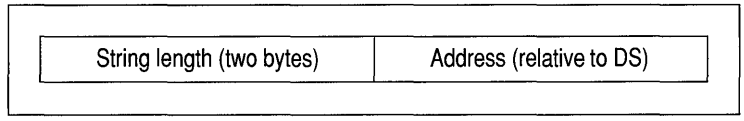
This string is represented in memory as shown following.



Because `c_string` is an array like any other, C and C++ pass it by reference in function calls. Note that this does not apply to string classes written in C++.

Basic String Format

Basic stores strings as 4-byte descriptors pointing to the actual string data. The format of the descriptor is as shown following.



The first field of the string descriptor contains an integer indicating the length (in bytes) of the string. The second field contains the address of the string in the default data segment.

Do not attempt to alter the length of Basic strings, because they are managed by Basic string-space management routines. You cannot count on a particular string remaining at a given offset during the execution of a Basic program because the Basic string-space management routines allocate strings to different areas of memory depending on program requirements.

The format of the string at DS:*Address* is a simple array of characters. The string is exactly the length indicated in the descriptor.

To pass a Basic string to C, append a null character.

Because C needs the null character to delimit the end of the string, you should append `chr$( 0 )` to your Basic string before passing it to your C function. For example,

```
A$ = "I am a BASIC string"
A$ = A$ + chr$( 0 )

CALL CFunc( SADD(A$) )
```

Note that the Basic call is made by near reference using the **SADD** keyword.

Use a string descriptor to pass a C string to Basic.

To pass a C string to Basic, create a structure for the string descriptor. For example,

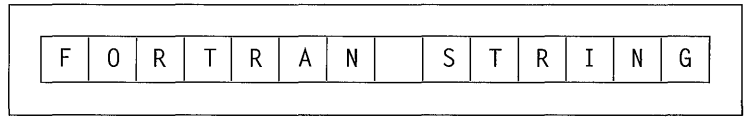
```
char c_string[] = "C String Data";

struct tagBASICStringDes
{
    char *   sd_addr;
    int      sd_len;
}
str_des;

str_des.sd_addr = c_string;
str_des.sd_len = strlen( c_string );

BASICFunction( &str_des );
```

FORTRAN String Format

FORTRAN stores strings as a series of bytes at a fixed location in memory. There is no delimiter at the end of the string. Consider the string declared as follows:

```
STR = 'FORTRAN STRING'
```

The string is stored in memory as shown following.



FORTRAN passes strings by reference, as it does all other data.

Note FORTRAN's variable length strings cannot be used in mixed-language programming because the temporary variable used to communicate string length is not accessible to other languages.

To pass a C string to FORTRAN (or Pascal), pass the variable by reference as you usually would. In your FORTRAN or Pascal routine, you must specify the length of the string; strings that are passed as arguments from one language to another must be of fixed length.

Pascal String Format

Pascal represents strings as fixed-length arrays of **CHAR** or as strings with a length byte followed by the string data.

To pass a fixed-length string to C, append a null character.

To pass a fixed-length string to a C function, use the concatenation operator (*) to append a null character. Then pass the string to the C function by reference (by declaring the string as **CONST**, **CONSTS**, **VAR**, or **VARS**). For example,

```
PROGRAM PasStr( input, output );
type
    stype15 = string(15);  (* fixed-length *)
var
    str : stype15;

PROCEDURE PasStrToC( VAR s1 : stype15 ) [C]; EXTERN;

BEGIN
    str := 'Pass this to C' * chr( 0 );
    PasStrToC( str );
END.
```

A more flexible way to pass Pascal strings to C functions is to declare them as type **ADRMEM** or **ADSMEM**, then pass the address of the string. For example,

```
PROCEDURE PasStrToC( s1adr : ADRMEM ) [C]; EXTERN;
```

Then you can call the C function with this code:

```
PasStrToC( ADR str );
```

Using this method, you can pass strings of different lengths to C functions.

Note The Pascal type **LSTRING** is not compatible with C; you can pass a string declared as **LSTRING** by first assigning it to another variable of type **STRING**, then passing that variable.

Whenever you pass a variable of type **STRING** or type **LSTRING** by value, Pascal pushes the whole string onto the stack and passes the length of the string as another parameter. C cannot access strings passed in this manner.

Passing a string from a C function to a Pascal function or procedure is identical to passing a string from a C function to a FORTRAN routine. The only provision you must make is to specify the length of the string to your Pascal function.

Arrays

When you use an array in a program written in a single language, the method for array handling is consistent. When you mix languages, you need to be aware of the differences between array-handling techniques in various languages.

Unlike most Microsoft languages, Basic keeps an array descriptor, which is similar to the Basic string descriptor discussed in "Strings," on page 184. This array descriptor is necessary because Basic handles memory allocation for arrays dynamically (at run time). Dynamic allocation requires Basic to shift arrays in memory.

To pass a Basic array to a C function, use the **VARPTR** and **VARSEG** keywords.

The **VARPTR** and **VARSEG** keywords obtain the address of the first element of the array and its segment, respectively. The following example shows how to call a C function with a near reference and a far reference to an array:

```
DIM ARRAY%( 20 )
DECLARE CNearArray CDECL( BYVAL Addr AS INTEGER )
DECLARE CFarArray CDECL( BYVAL Addr AS INTEGER, BYVAL Seg AS INTEGER )
.
.
.
CALL CNearArray( VARPTR( ARRAY%(0) ) )
CALL CFarArray( VARPTR( ARRAY%(0) ), VARSEG( ARRAY%(0) ) )
```

The C functions receiving ARRAY can be declared as follows:

```
__cdecl CNearArray( int * array );
__cdecl CFarArray( int far * array );
```

The routine that receives the array must not make a call back to Basic. If it does, the location of the array data can change and the address that was passed to the routine becomes meaningless.

If you only need to pass one member of the array from Basic to your C function, you can pass it by value as follows:

```
CALL CFunc( ARRAY%(8) )
```

Array Declaration and Indexing

Each language varies in the way that arrays are declared and indexed. Array indexing is a source-level consideration and involves no transformation of data. There are two differences in the way elements are indexed by each language:

- The value of the lower array bound is different among Microsoft languages.

  By default, FORTRAN indexes the first element of an array as 1. Basic and C index it as 0. In Pascal, you can begin indexing at any integer value. In recent versions of Basic and FORTRAN, you also have the option of specifying lower bounds at any integer value.
- Some languages vary subscripts in row-major order; others vary subscripts in column-major order.

The differences in how subscripts are varied only affect arrays with more than one dimension. With row-major order (used by C and Pascal), the rightmost dimension changes first. With column-major order (used by FORTRAN, and Basic by default), the leftmost dimension changes first. Thus, in C, the first four elements of an array declared as `X[3][3]` are

```
X[0][0]    X[0][1]    X[0][2]    X[1][0]
```

In FORTRAN, the four elements are

```
X(1,1)     X(2,1)     X(3,1)     X(1,2)
```

The preceding C and FORTRAN arrays illustrate the difference between row-major and column-major order and also the difference in the assumed lower bound between C and FORTRAN. Table 8.6 shows equivalences for array declarations in each language. In this table, *r* is the number of elements of the row dimension (which changes most slowly), and *c* is the number of elements of the column dimension (which changes most quickly).

Table 8.6 Equivalent Array Declarations

Language	Array declaration	Notes
Basic	**DIM** *x*(*r*–1, *c*–1)	With default lower bound of 0
C	*type x*[*r*][*c*] **struct** { *type x*[*r*][*c*]; } *x*	When passed by reference When passed by value
FORTRAN	*type x*(*c*, *r*)	With default lower bound of 1
Pascal	*x* : **ARRAY** [*a*..*a*+*r*–1, *b*..*b*+*c*–1] **OF** *type*	

The order of indexing extends to any number of dimensions you declare. For example, the C declaration

```
int arr1[2][10][15][20];
```

is equivalent to the FORTRAN declaration

```
INTEGER*2 ARR1( 20, 15, 10, 2 )
```

The constants used in a C array declaration represent dimensions, not upper bounds as they do in other languages. Therefore, the last element in the C array declared as `int arr[5][5]` is `arr[4][4]`, not `arr[5][5]`.

Structures, Records, and User-Defined Types

The C **struct** type, the Basic user-defined type, the FORTRAN record (defined with the **STRUCTURE** keyword), and the Pascal **record** type are equivalent. Therefore, these data types can be passed between C, FORTRAN, Pascal, and Basic.

These types can be affected by the storage method. By default, C, FORTRAN, and Pascal use word alignment for types shorter than one word (type **char** and **unsigned char**). This storage method specifies that occasional bytes can be inserted as padding so that word and double-word objects start on an even boundary. (In addition, all nested structures and records start on a word boundary.)

If you are passing a structure or record across a mixed-language interface, your calling routine and called routine must agree on the storage method and parameter-passing convention. Otherwise, data is not interpreted correctly.

Because Pascal, FORTRAN, and C use the same storage method for structures and records, you can interchange data between routines without taking any special precautions unless you modify the storage method. Make sure the storage methods agree before interchanging data between C, FORTRAN, and Pascal.

Basic packs user-defined types, so your C function must also pack structures (using the /Zp command-line option or the **pack** pragma) to agree.

The C++ **class** type has the same layout as the corresponding C **struct** type, unless the class defines virtual functions or has base classes. Classes that lack those features can be passed in the same way as C structures.

You can pass structures as parameters by value or by reference. Both the calling program and the called program must agree on the parameter-passing convention. For more information about the language you are using, see "Parameter-Passing Requirement," on page 159.

External Data

External data refers to data that is both static and public; that is, the data is stored in a set place in memory as opposed to being allocated on the stack and the data is visible to other modules.

External data can be defined in C, C++, Pascal, and assembly language. Note that a data definition is distinct from an external declaration. A data definition causes a compiler to create a data object; an external declaration informs a compiler that the object is to be found in another module. FORTRAN can only define external data in COMMON blocks. (For more information about sharing external data with FORTRAN programs, see "Common Blocks," on page 192.)

There are three requirements for programs that share external data between languages:

- One of the modules must define the data.

  You can define a static data object in a C module by defining a data object outside all functions. (If you use the **static** keyword in C, however, the data object is not made public.)

  You can make a C++ data object visible to other languages by declaring it with the **extern "C"** linkage specification. However, you cannot use any C++ specific features of such data items. For example, you cannot call any member functions for an object declared **extern "C"**.

- The other modules that access the data must declare the data as external.

  In C, you can declare data as external by using an **extern** declaration, similar to the **extern** declaration for functions. In FORTRAN and Pascal, you can declare data as external by adding the **EXTERN** attribute to the data declaration.

- Resolve naming-convention differences.

  In C, you can adopt the FORTRAN/Pascal naming convention by applying **__fortran** or **__pascal** to the data declaration. In C++, you can adopt the C naming convention by using the **extern "C"** linkage specification, and you can adopt the FORTRAN/Pascal naming convention by adding the **__fortran** or **__pascal** keywords. In FORTRAN and Pascal, you can adopt the C naming convention by applying the **C** attribute to the data declaration.

Pointers and Address Variables

Rather than passing data directly, you may want to pass the address of a piece of data. Passing the address amounts to passing the data by reference. In some cases, such as in Basic arrays, there is no other way to pass a data item as a parameter.

C and C++ programs always pass array variables by address. All other types are passed by value unless you use the address-of (**&**) operator to obtain the address.

The Pascal **ADR** and **ADS** types are equivalent to near and far pointers, respectively, in C and C++. You can pass **ADR** and **ADS** variables as **ADRMEM** or **ADSMEM**. Basic and FORTRAN do not have formal address types. However, they do provide ways for storing and passing addresses.

Basic programs can access a variable's segment address with the **VARSEG** function and its offset address with the **VARPTR** function. The values returned by these intrinsic functions should then be passed or stored as ordinary integer variables. If you pass them to another language, pass them by value. Otherwise you are attempting to pass the address of the address, rather than the address itself.

To pass a near address, pass only the offset; if you need to pass a far address, you may have to pass the segment and the offset separately. Pass the segment address first, unless **CDECL** is in effect.

FORTRAN programs can determine near and far addresses with the **LOC** and **LOCFAR** functions. Store the result of the **LOC** function as **INTEGER*2** and the result of the **LOCFAR** function as **INTEGER*4**.

As with Basic, if you pass the result of **LOC** or **LOCFAR** to another language, be sure to pass by value.

Common Blocks

You can pass individual members of a FORTRAN or Basic common block in an argument list, just as you can any data item. However, you can also give a different language module access to the entire common block at once.

C or C++ modules can reference the items of a common block by first declaring a structure with fields that correspond to the common-block variables. Having defined a structure with the appropriate fields, the C or C++ module must then connect with the common block itself. The next two sections present methods for gaining access to common blocks.

Passing the Address of a Common Block

To pass the address of a common block, simply pass the address of the first variable in the block. (In other words, pass the first variable by reference.) The receiving C or C++ module should expect to receive a structure by reference.

In the following example, the C function `initcb` receives the address of the variable `N`, which it considers to be a pointer to a structure with three fields:

```
C       FORTRAN SOURCE CODE
C
        COMMON /CBLOCK/N, X, Y
        INTEGER*2 N
        REAL*8    X, Y
.
.
.
        CALL INITCB( N )

/* C source code */

/* Explicitly set structure packing to word alignment. */
#pragma pack( 2 )
```

```
struct block_type
{
    int    n;
    double x;
    double y;
};

initcb( struct block_type * block_hed )
{
    block_hed->n = 1;
    block_hed->x = 10.0;
    block_hed->y = 20.0;
}
```

Accessing Common Blocks Directly

You can access FORTRAN common blocks directly by defining a structure with the appropriate fields and then using the methods described in "External Data," on page 190. Here is an example of accessing common blocks directly:

```
struct block_type
{
    int    n;
    double x;
    double y;
};

extern struct block_type fortran cblock;
```

You cannot access common blocks directly using Basic common blocks.

Note that the technique of accessing common blocks directly works with FORTRAN common blocks, but not with Basic common blocks. If your C or C++ module must work with both FORTRAN and Basic common blocks, pass the address of the common block as a parameter to the function.

Using a Varying Number of Parameters

Some C functions (for example **printf**) accept a variable number of parameters. To call such a function from another language, you need to suppress the type-checking that usually forces a call to be made with a fixed number of parameters. In Basic, you can remove this type-checking by omitting a parameter list from the **DECLARE** statement. In FORTRAN or Pascal, you can call routines with a variable number of parameters by including the **VARYING** attribute in your interface to the routine, along with the **C** attribute. You must use the **C** attribute because a variable number of parameters is feasible only with the C calling convention. In C++, functions that accept a variable number of parameters automatically use the C calling convention.

CHAPTER 9

Writing Portable C Programs

Because C compilers exist on a variety of computers, some C applications developed for one computer system can be ported to other systems. However, some aspects of language behavior depend on how a particular C compiler is implemented and how a specific computer operates. Therefore, when designing a program to be ported to another system, it is important that you examine programming assumptions.

This chapter describes programming assumptions that can affect writing portable programs.

The American National Standards Institute Standard for the C Language (the ANSI standard) details every instance where language behavior is defined by the implementation. For a summary of implementation-defined behavior for Microsoft Visual C++, see Appendix B, "Implementation-Defined Behavior," in the *C Language Reference*.

9.1 Assumptions About Hardware

To make C programs portable, you must examine two aspects of your code: hardware assumptions and compiler dependency. This section deals with hardware assumptions. Section 9.2, "Assumptions About the Compiler," on page 210, deals with compiler dependency.

Size of Basic Types

In C, the size of basic types (**char**, **signed int**, **unsigned int**, **float**, **double**, and **long double**) is implementation-defined, so relying on a particular data type to be a given size reduces the portability of a program.

Don't make assumptions about the size of data types.

Because the size of basic types is left to the implementation, do not make assumptions about the size or alignment of data types within aggregate types. Use only the **sizeof** operator to determine the size or amount of storage required for a variable or a type.

Following are some rules governing the size of data types.

Type char

Type **char** is the smallest of the basic types, but it must be large enough to hold any of the characters in the implementation's basic character set. Usually, variables of type **char** are 1 byte.

Type int and Type short int

Type **int** often corresponds to the register size of the target machine. Type **short int** may be less than or equal to the size of type **int**. Both **int** and **short** are greater than or equal to the size of type **char** but less than or equal to the size of type **long**.

If you assume that type **int** is a certain size, your code may not be portable because:

- An **int** can be defined as a 16-bit (2-byte) or a 32-bit quantity.
- An **int** is not always large enough to hold array indices. For large arrays, you must use **unsigned int**; for extremely large arrays, use **long** or **unsigned long**. To be certain your code is portable, define your array indices as type **size_t**. You may not know, before porting your code, the maximum value to expect an array index of type **int** to hold. The file LIMITS.H contains manifest constants, listed following, for the maximum and minimum values of each basic integral type:

Constant	Value
CHAR_BIT	Number of bits in a variable of type **char**
CHAR_MIN	Minimum value a variable of type **char** can hold
CHAR_MAX	Maximum value a variable of type **char** can hold
SCHAR_MIN	Minimum value a variable of type **signed char** can hold
SCHAR_MAX	Maximum value a variable of type **signed char** can hold
UCHAR_MAX	Maximum value a variable of type **unsigned char** can hold
SHRT_MIN	Minimum value a variable of type **short** can hold
SHRT_MAX	Maximum value a variable of type **short** can hold
USHRT_MAX	Maximum value a variable of type **unsigned short** can hold
INT_MIN	Minimum value a variable of type **int** can hold
INT_MAX	Maximum value a variable of type **int** can hold
UINT_MAX	Maximum value a variable of type **unsigned int** can hold
LONG_MIN	Minimum value a variable of type **long** can hold
LONG_MAX	Maximum value a variable of type **long** can hold
ULONG_MAX	Maximum value a variable of type **unsigned long** can hold

Type float, Type double, and Type long double

Type **float** is the smallest of the basic floating-point types. Type **double** is usually larger than type **float**, and type **long double** is usually the largest of the floating-point types. You can make the following portability assumptions about floating-point types:

- Any value that can be represented as type **float** can be represented as type **double** (type **float** is a subset of type **double**).
- Any value that can be represented as type **double** can be represented as type **long double** (type **double** is a subset of type **long double**).

The file FLOAT.H contains manifest constants, listed following, for the maximum and minimum values of each basic floating-point type:

Constant	Value
DBL_DIG	Number of decimal digits of precision a variable of type **double** can hold
DBL_MAX	Maximum value a variable of type **double** can hold
DBL_MAX_10_EXP	Maximum value (base 10) the exponent of a variable of type **double** can hold
DBL_MAX_EXP	Maximum value (base 2) the exponent of a variable of type **double** can hold
DBL_MIN	Minimum positive value a variable of type **double** can hold
DBL_MIN_10_EXP	Minimum value (base 10) the exponent of a variable of type **double** can hold
DBL_MIN_EXP	Minimum value (base 2) the exponent of a variable of type **double** can hold
FLT_DIG	Number of decimal digits of precision a variable of type **float** can hold
FLT_MAX	Maximum value a variable of type **float** can hold
FLT_MAX_10_EXP	Maximum value (base 10) the exponent of a variable of type **float** can hold
FLT_MAX_EXP	Maximum value (base 2) the exponent of a variable of type **float** can hold
FLT_MIN	Minimum positive value a variable of type **float** can hold
FLT_MIN_10_EXP	Minimum value (base 10) the exponent of a variable of type **float** can hold
FLT_MIN_EXP	Minimum value (base 2) the exponent of a variable of type **float** can hold
LDBL_DIG	Number of decimal digits of precision a variable of type **long double** can hold

Constant	Value
LDBL_MAX	Maximum value a variable of type **long double** can hold
LDBL_MAX_10_EXP	Maximum value (base 10) the exponent of a variable of type **long double** can hold
LDBL_MAX_EXP	Maximum value (base 2) the exponent of a variable of type **long double** can hold
LDBL_MIN	Minimum positive value a variable of type **long double** can hold
LDBL_MIN_10_EXP	Minimum value (base 10) the exponent of a variable of type **long double** can hold
LDBL_MIN_EXP	Minimum value (base 2) the exponent of a variable of type **long double** can hold

Visual C++ Type Sizes

Table 9.1 summarizes the size of the basic types in Visual C++.

Table 9.1 Size of Basic Types in Visual C++

Type	Number of bytes
char, unsigned char	1
short, unsigned short	2
int, unsigned int	2 or 4*
near pointer	2 or 4*
long, unsigned long	4
far pointer	4 or 8*
float	4

* These data types have different sizes in 16- and 32-bit environments.

Storage Order and Alignment

The C language does not define any specific layout for the storage of data items relative to one another. The layout for storage of structure elements, or unions within a structure or union, is defined by the implementation.

Some processors require that data longer than 1 byte be aligned to 2-byte or 4-byte boundaries. Other processors, such as the 80*x*86 family, do not have such a restriction. However, the 80*x*86 processors work more efficiently with aligned data.

Structure Order and Alignment

The following example illustrates how alignment can affect your program. In the example, a structure is cast to type **long** because the programmer knew the order in which a particular implementation stored data.

```
/* Nonportable code */
struct time
{
    char hour;      /* 0 < hour < 24   - fits in a char */
    char minute;    /* 0 < minute < 60 - fits in a char */
    char second;    /* 0 < second < 60 - fits in a char */
};

    .
    .
    .
    struct time now, alarm_time;
    .
    .
    .
    if ( *(long *)&now >= *(long *)&alarm_time )
    {
        /* Sound an alarm */
    }
```

The preceding code makes these nonportable assumptions:

- The data for `hour` is stored in a higher order position than `minute` or `second`. Because C does not guarantee storage order or alignment of structures or unions, the code may not be portable to other machines.
- Three variables of type **char** are shorter than or the same length as a variable of type **long**. Thus, the code is not portable according to the rules governing the size of basic types, as described in "Size of Basic Types," on page 195.

If either of these assumptions proves false, the comparison (**if** statement) is invalid.

You can write code that makes no assumptions about storage order.

To make the program in the preceding example portable, you can break the comparison between the two long integers into a component-by-component comparison. This technique is illustrated in the following example:

```
/* Portable code */
struct time
{
    char hour;      /* 0 < hour < 24   - fits in a char */
    char minute;    /* 0 < minute < 60 - fits in a char */
    char second;    /* 0 < second < 60 - fits in a char */
};

    .
    .
    .
    struct time now, alarm_time;
    .
    .
    .
    if ( time_cmp( now, alarm_time ) >= 0 )
    {
        /* Sound an alarm */
    }
    .
    .
    .

int time_cmp( struct time t1, struct time t2 )
{
    if( t1.hour != t2.hour )
        return( t2.hour - t1.hour );
    if( t1.minute != t2.minute )
        return( t2.minute - t1.minute );
    return( t2.second - t1.second );
}
```

Windows and Structure Alignment

Programming for Windows is another situation in which the packing of a structure can affect portability. To save memory, structures in Windows are packed on 1-byte boundaries. This means that any structure used by an application to pass information to Windows must be aligned on 1-byte boundaries. Structures that are not passed to Windows need not be packed on 1-byte boundaries.

When compiling an application for a computer running Windows and using an Intel processor, compile the entire application with the /Zp option so that Windows can successfully use the structures that it needs. You can use the **pack** pragma to mark structures that are not meant for communication with Windows in order to specify packing more suitable to the efficient operation of the processor. For more

information on the /Zp compiler option and the **pack** pragma, see Chapter 1, "CL Command Reference," in the *Command-Line Utilities User's Guide*.

Union Order and Alignment

Programmers use unions most often for two purposes: to store data whose exact type is not known until run time and to access the same data in different ways.

Unions falling into the second category are usually not portable. For example, the following union is not portable:

```
union tag_u
{
    char bytes_in_long[4];
    long a_long;
};
```

The intent of the preceding union is to access the individual bytes of a variable of type **long**. However, the union may not work as intended when ported to other computers because:

- It relies on a constant size for type **long**.
- It may assume byte ordering within a variable of type **long**. (Byte ordering is described in detail in "Byte Order in a Word," following.)

The first problem can be addressed by coding the union as follows:

```
union tag_u
{
    char bytes_in_long[sizeof( long ) / sizeof( char )];
    long a_long;
};
```

Note the use of the **sizeof** operator to determine the size of a data type.

Byte Order in a Word

The order of bytes within an integral type longer than a byte (**short**, **int**, or **long**) can vary among computers. Code that assumes an internal order is not portable, as shown by this example:

```
/* Nonportable structure to access an int in bytes. */
struct tag_int_bytes
{
    char lobyte;
    char hibyte;
};
```

A more portable way to access the individual bytes in a word is to define two macros that rely on the constant **CHAR_BIT**, defined in LIMITS.H:

```
#define LOBYTE(a) (char)((a) & 0xff)
#define HIBYTE(a) (char)((unsigned)(a) >> CHAR_BIT)
```

The **LOBYTE** macro is still not completely portable. It assumes that a **char** is 8 bits long, and it uses the constant `0xff` to mask the high-order eight bits. Because portable programs cannot rely on a given number of bits in a byte, consider the revision following:

```
#define LOBYTE(a) (char)((a) & ((unsigned)~0>>CHAR_BIT))
#define HIBYTE(a) (char)((unsigned)(a) >> CHAR_BIT)
```

The new **LOBYTE** macro performs a bitwise complement on 0; that is, all zero bits are turned into ones. It then takes that unsigned quantity and shifts it right far enough to create a mask of the correct length for the implementation.

The following code assumes that the order of bytes in a word is least-significant first:

```
int c;
.
.
.
fread( &c, sizeof( char ), 1, fp );
```

The code attempts to read one byte as an **int**, without converting it from a **char**. However, the code fails in any implementation where the low-order byte is not the first byte of an **int**. The following solution is more portable. In this example, the data is read into an intermediate variable of type **char** before being assigned to the integer variable.

```
int c;
char ch;
.
.
.
fread( &ch, sizeof( char ), 1, fp );
c = ch;
```

The following example shows how to use the C run-time function **fgetc** to return the value. The **fgetc** function returns type **char**, but the value is promoted to type **int** when it is assigned to a variable of type **int**.

```
int c;
.
.
.
c = fgetc( fp );
```

Visual C++ Specific

Visual C++ usually aligns data types longer than 1 byte to an even-byte address for improved performance. For information about controlling structure packing in Visual C++, see the /Zp compiler option and the **pack** pragma in Chapter 1, "CL Command Reference," in the *Command-Line Utilities User's Guide*.

Reading and Writing Structures

Many C programs read data from disk into structures and write data to disk from structures. The functions that perform disk input/output (I/O) in C require you to specify the number of bytes to be transferred. You should always use the **sizeof** operator to obtain the size of the data to be read or written because differing data type sizes or alignment schemes may alter the size of a given structure. For example,

```
fread( &my_struct, sizeof(my_struct), 1, fp );
```

Visual C++ Specific

When performing disk input and output in Visual C++, structures may be different sizes depending on the structure-packing option you have selected. For information about controlling structure packing in Visual C++, see the /Zp compiler option and the **pack** pragma in Chapter 1, "CL Command Reference," in the *Command-Line Utilities User's Guide*.

Bit Fields in Structures

The Microsoft C compiler implements bit fields. However, many C compilers do not.

Using bit fields, you can access the individual bits within a data item. Although the practice of accessing the bits in a data item is inherently nonportable, you can improve your chances of porting a program that uses bit fields if you make no assumptions about order of assignment or size and alignment of bit fields.

Order of Assignment

The order of assignment of bit fields in memory is left to the implementation, so you cannot rely on a particular entry in a bit field structure to be in a higher order position than another. (This problem is similar to the portability constraint imposed by

alignment of basic data types in structures. The C language does not define any specific layout for the storage of data items relative to one another.) For more information, see "Storage Order and Alignment," on page 198.

Size and Alignment of Bit Fields

The Microsoft C compiler supports bit fields up to the size of the type **long**. Each individual member of the bit field structure can be up to the size of the declared type. Some compilers do not support bit field–structure elements that are longer than type **int**.

The following example defines a bit field, `short_bitfield`, that is shorter than type **int**:

```
struct short_bitfield
{
    unsigned usr_bkup : 1; /* 0  <= usr_bkup <  1 */
    unsigned usr_sec  : 4; /* 9 <= usr_sec < 16 */
};
```

The following example defines a bit field, `long_bitfield`, that has an element longer than type **int** in a 16-bit environment:

```
struct long_bitfield
{
    unsigned long disk_pos : 22;  /* 0 <= disk_pos   < 4,194,304 */
    unsigned long rec_no   : 10;  /* 0 <= rec_no < 1024 */
};
```

The bit field `short_bitfield` is likely to be supported by more implementations than `long_bitfield`.

Visual C++ Specific

The following example introduces another portability issue: alignment of data defined in bit fields.

```
struct long_bitfield
{
    unsigned int day    : 5;  /* 0 <= day   < 32 */
    unsigned int month  : 4;  /* 0 <= month < 16 */
    unsigned int year   : 11; /* 0 <= year  < 2048 */
};
```

In a 16-bit environment, Visual C++ does not allow an element in a structure to cross a word boundary. The first two elements, `day` and `month`, take up 9 bits. The third, `year`, would cross a word boundary if it were to begin right after `month`, so instead it must begin on the next word boundary. Thus, there is a 7-bit gap between

the `month` and `year` elements in the Visual C++ representation of this structure. However, other compilers may not use the same storage techniques.

Note that in a 32-bit environment, all three elements can fit within a single word, so there is no gap between any of the elements in the Visual C++ representation of the structure.

Processor Arithmetic Mode

Two types of arithmetic are common on digital computers: one's-complement arithmetic and two's-complement arithmetic. Some programs assume that all target computers perform two's-complement arithmetic. If you take advantage of the fact that a given operation causes a particular bit pattern to be set on either a one's-complement or two's-complement computer, your program is not portable. For example, two's-complement machines represent the 8-bit integer value −1 as a binary 11111111. A one's-complement machine represents the same decimal value (−1) as 11111110. Some programmers assume that −1 fills a byte or a word with ones and use it to construct a mask template that they later shift. This does not work correctly on one's-complement machines, but the error does not surface until the least-significant bit is used.

In two's-complement arithmetic, there is only one value that represents zero. In one's-complement arithmetic, there is a value for zero and a value for negative zero. Use the C relational operators to handle this anomaly correctly. If you write code that deliberately circumvents the C relational operators, tests for zero or **NULL** may not operate correctly.

Visual C++ Specific

Visual C++ produces code only for the Intel 80*x*86 processors, which perform two's-complement arithmetic.

Pointers

One of the most powerful but potentially dangerous features of the C language is its use of indirect addressing through pointers. Bugs introduced by misusing pointers can be difficult to detect and isolate because the errors often corrupt memory unpredictably.

Casting Pointers

Be sure your assumptions do not make your code nonportable when you cast pointers to different types.

```
/* Nonportable coercion */
char c[4];
long *lp;

lp = (long *)c;
*lp = 0x12345678L;
```

This code is nonportable because using a cast to change an array of **char** to a pointer of type **long** assumes a particular byte-ordering scheme. This is discussed in greater detail in "Byte Order in a Word," on page 201.

Pointer Size

A pointer can be assigned (or cast) to any integer type large enough to hold it, but the size of the integer type depends on the computer and the implementation. (In fact, it can even depend on the memory model.) Therefore, you cannot assume that a pointer is the same size as an integer, that is:

```
sizeof( char * ) == sizeof( int )
```

To determine the size of any unmodified data pointer, use

```
sizeof( void * )
```

This expression returns the size of a generic data pointer.

Pointer Subtraction

Code that assumes that pointer subtraction yields an **int** value is nonportable. Pointer subtraction yields a result of type **ptrdiff_t** (defined in STDDEF.H). Portable code must always use variables of type **ptrdiff_t** for storing the result of pointer subtraction.

The Null Pointer

In most implementations, **NULL** is defined as 0. In Visual C++, it is defined as `((void *)0)`. Because code pointers and data pointers are often different sizes, using 0 for the null pointer for both can lead to nonportability. The difference in size between code pointers and data pointers causes problems for functions that expect pointer arguments longer than an **int**. To avoid these problems, use the null pointer, as defined in the include file STDDEF.H; use prototypes; or explicitly cast **NULL** to the correct data type. Here is a portable way to use the null pointer:

```
/* Portable use of the null pointer */
main()
{
    func1( (char *)NULL );
    func2( (void *(*)())NULL );
}

void func1( char * c )
{
}

void func2( void *(* func)() )
{
}
```

The invocations of `func1` and `func2` explicitly cast **NULL** to the correct size. In the case of `func1`, **NULL** is cast to type **char** *; in the case of `func2`, it is cast to a pointer to a function that returns type **void**.

Visual C++ Specific

Subtraction of pointers to huge arrays that have more than 32,767 elements may yield a **long** result. The **__huge** keyword is implementation defined by Visual C++ and is not portable. Here is how to subtract pointers to huge arrays:

```
char __huge *a;
char __huge *b;
long        d;
.
.
.
d = (long)( a - b );
```

In Visual C++, the memory model selected and the special keywords **__near**, **__far**, and **__huge** can change the size of a pointer. The Microsoft memory models and extended keywords are nonportable, but you should be aware of their effects.

Sizes of generic pointers and default pointer sizes are shown in Tables 9.2 and 9.3, respectively.

Table 9.2 Size of Generic Pointers

Declaration	Name	Size
void __near *	Generic near pointer	16 bits
void __far *	Generic far pointer	32 bits
void __huge *	Generic huge pointer	32 bits

Table 9.3 Default Pointer Sizes in 16-Bit Programs

Memory model	Code pointer size	Data pointer size
Tiny	16 bits	16 bits
Small	16 bits	16 bits
Medium	32 bits	16 bits
Compact	16 bits	32 bits
Large	32 bits	32 bits
Huge	32 bits	32 bits

Address Space

The amount of available memory and the address space on systems varies, depending on many factors outside your control. A program designed with portability in mind should handle insufficient-memory situations. To ensure that your program handles these situations, you should always check the error return from any of the dynamic memory allocation routines, such as **malloc**, **calloc**, **_strdup**, and **realloc**.

These situations occur not only because of a lack of installed memory but also because too many other applications are using memory. For example,

- Installed resident software can cause your program to fail. In MS-DOS, these programs are usually device drivers or terminate-and-stay-resident (TSR) utilities.
- An event or combination of events in a multitasking operating system such as XENIX can cause your program to fail. These failures are complex and difficult to predict. Here is an example: The user has installed a daemon to "pop up" every so often and check the system status. The user is running your application along with enough other, large applications to cause a critical shortage of memory. When the daemon pops up, your program may fail on a memory allocation request.
- An application running with Windows can use a large amount of the global heap and not return it to the free pool. This type of behavior causes Windows to deny a **GlobalAlloc** request.

Character Set

The C language does not define the character set used in an implementation. This means that any programs that assume the character set to be ASCII are nonportable.

The only restrictions on the character set are these:

- No character in the implementation's character set can be larger than the size of type **char**.
- Each character in the set must be represented as a positive value by type **char**, whether it is treated as signed or unsigned. So, in the case of the ASCII character set and an 8-bit **char**, the maximum value is 127 (128 is a negative number when stored in a **char** variable).

Character Classification

The standard C run-time support contains a complete set of character classification macros and functions. These functions are defined in the CTYPE.H file and are guaranteed to be portable:

isalnum	**isdigit**	**isprint**	**isupper**
isalpha	**isgraph**	**ispunct**	**isxdigit**
iscntrl	**islower**	**isspace**	

The following example is not portable to implementations that do not use the ASCII character set:

```
/* Nonportable */
if( c >= 'A' && c <= 'Z' )
    /* Uppercase alphabetic */
```

Instead, consider using this:

```
/* Portable */
if( isalpha(c) && isupper(c) )
    /* Uppercase alphabetic */
```

The first of the previous examples is nonportable, because it assumes that uppercase A is represented by a smaller value than uppercase Z and that no lowercase characters fall between the values of A and Z. The second example is portable, because it uses the character classification functions to perform the tests.

In a portable program, you should not perform any comparison on variables of type **char** except strict equality (==). You cannot assume the character set follows an increasing sequence—that may not be true on a different computer.

Case Translation

Translation of characters from uppercase to lowercase or from lowercase to uppercase is called "case translation." The following example shows a coding technique for case translation not portable to implementations using a non-ASCII character set.

```
#define make_upper(c) ((c)&0xcf)
#define make_lower(c) ((c)|0x20)
```

This code takes advantage of the fact that you can map uppercase to lowercase simply by changing the state of bit six. It is extremely efficient but nonportable. To write portable code, use the case-translation macros **toupper** and **tolower** (defined in CTYPE.H).

9.2 Assumptions About the Compiler

Different compilers translate C source code into object code in different ways. The ANSI standard for the C programming language defines how many of these translations must be done; others are implementation-defined.

This section describes assumptions about how the compiler translates your C code that can make your programs nonportable. For a complete description of how Visual C++ handles implementation-defined operations, see Appendix B, "Implementation-Defined Behavior," in the *C Language Reference*.

Sign Extension

"Sign extension" is the propagation of the sign bit to fill unoccupied space when promoting to a more-significant type or when performing bitwise right-shift operations.

Promotion from Shorter Types

Integral promotions from shorter types occur when you make an assignment, perform arithmetic, perform a comparison, or perform an explicit cast.

The behavior of integral promotion is well defined, except for type **char**. The implementation defines whether type **char** is treated as signed or unsigned. The following is an example of promotion as a result of assignment:

```
char c1 = -3;
int  i1;

i1 = c1;
```

In this example, the expected result of the assignment statement is that `i1` is set to –3. If the implementation defines type **char** as unsigned, however, sign extension does not occur, and `i1` is set to 253 (on a two's-complement machine).

Promotion can also occur as a result of a comparison of different types:

```
char c;

if( c == 0x80 )
    .
    .
    .
```

This comparison never evaluates as true on an implementation that sign-extends **char** types but treats hexadecimal constants as unsigned. Use a character constant of the form **'\x80'** or explicitly cast the constant to type **char** to perform the comparison correctly.

The following comparison, which is an example of promotion as a result of a cast, is also nonportable:

```
char c;
unsigned int u;

if( u == (unsigned)c )
```

There are two problems with this code:

- The **char** type may be treated as signed or unsigned, depending on the implementation.
- If the **char** type is treated as signed, it can be converted to **unsigned** in two ways: The **char** value may first be sign-extended to **int**, then converted to **unsigned**, or the **char** may be converted to **unsigned char**, then sign-extended to **int** length.

It is always safe to compare a **signed int** with a **char** constant because C requires all character constants to be positive.

Variables of type **char** are promoted to type **int** when passed as arguments to a function. This causes sign extension on some computers. Consider the following code:

```
char c = 128;

printf( "%d\n", c );
```

Visual C++ Specific

In Visual C++, you can treat type **char** as signed or unsigned. By default, a **char** is considered signed, but if you change the default **char** type using the /J compiler option, you can treat it as unsigned.

Bitwise Right-Shift Operations

Positive or unsigned integral types (**char**, **short**, **int**, and **long**) yield positive or zero values after a bitwise right-shift (>>) operation. For example,

```
(char)120 >> 4
```

yields 7,

```
(unsigned char)240 >> 8
```

yields 0,

```
(int)500 >> 8
```

yields 1, and

```
(unsigned int)65535 >> 4
```

yields 4095.

Negative-signed integral types yield implementation-defined values after a bitwise right-shift operation. This means that you must know whether you want to do a signed or unsigned shift, then code accordingly.

If you don't know how the implementation performs, you may get unexpected results. For example, `(signed char)0x80 >> 3` yields 0xf0 if the implementation performs sign extension on bitwise right shifts. If the implementation does not perform the sign extension, the result is 0x10.

You can use right shifts to speed up division when the divisor can be represented by powers of two and the dividend is positive. To maintain portability, you should use the division operator.

To perform an unsigned shift, explicitly cast the data to an unsigned type. To perform a shift that extends the sign bit, use the division operator as follows: Divide by 2^n, where n is the number of bits you want to shift.

Length and Case of Identifiers

Some implementations do not support long identifiers. Some allow only 6 characters, while others allow as many as 32. They may report each identifier that exceeds the maximum length or truncate identifiers to a given length. Truncation causes serious problems, especially if you have a number of similarly named variables within the scope of a block of code, such as the following:

```
double acct_receivable_30_days;
double acct_receivable_60_days;
double acct_receivable_90_days;
double current_interest_rate;

acct_receivable_30_days *= current_interest_rate;
```

If your target system retains only six significant characters, you will have to rename all your `acct_receivable` variables.

Case sensitivity also affects portability. C is usually a case-sensitive language. That is, `CalculateInterest` is not considered the same identifier as `calculateinterest`. Some systems are not case sensitive, however, so to write portable code differentiate your identifiers by something other than case.

These problems with identifiers can occur in two locations: the compiler and the linker or loader. Even if the compiler can handle long and case-differentiated identifiers, if the linker or loader cannot you can get duplicate definitions or other unexpected errors.

Visual C++ Specific

The Microsoft C compiler issues the /NOIGNORECASE command to the Microsoft Segmented Executable Linker (LINK), specifically instructing it to consider the case of identifiers.

Register Variables

The number and type of register variables in a function depend on the implementation. You can declare more variables as **register** than the number of physical registers the implementation uses. In such a case, the compiler treats the excess register variables as **automatic**.

Because the types that qualify for **register** class differ among implementations, invalid **register** declarations are treated as **automatic**.

If you declare variables as **register** to optimize performance, declare them in decreasing order of importance to ensure that the compiler allocates a register to the most important variables.

Visual C++ Specific

The compiler ignores **register** declarations if you select the global register allocation optimization. You can select global register allocation as follows:

Environment	Selection
CL command line	Specify either the /Oe or /Ox option.
Pragma	Use the **optimize** pragma with the **e** parameter.

Functions with a Variable Number of Arguments

Functions that accept a variable number of arguments are not portable. Although both the ANSI standard and *The C Programming Language* 2d ed. (Kernighan, Brian W., and Ritchie, Dennis M.; Englewood Cliffs, NJ: Prentice Hall, 1988) specify how to write these functions and how they behave, differences still exist among compiler implementors about how to use variable argument lists.

Many UNIX systems support a standard that differs from the ANSI standard for variable arguments. Although this may change, it currently presents a portability concern.

Using Visual C++ run-time libraries and macros, you can choose whichever version of variable argument support you expect to be most portable for your application.

Evaluation Order

The C language does not guarantee the evaluation order of most expressions. Avoid writing constructs that depend on evaluation within an expression to proceed in a particular manner. For example,

```
i = 0;
func( i++, i++ );
.
.
.
func( int a, int b )
{
```

A compiler could evaluate this code and pass 0 as `a` and 1 as `b`. It could also pass 1 as `a` and 0 as `b` and conform equally with the standards.

The C language does guarantee that an expression is completely evaluated at any given "sequence point." A sequence point is a point in the syntax of the language at which all side effects of an expression or series of expressions have been completed.

These are the sequence points in the C language:

- The semicolon (;) statement separator.
- The call to a function after the arguments have been evaluated.
- The end of the first operand of the logical-AND operator (&&), the logical-OR operator (||), the conditional operator (?), or the comma separator operator (,) when it is used to separate statements or in expressions. The comma separator is not a sequence point when it is used between variables in declaration statements or between parameters in a function invocation.
- The end of a full expression such as an initializer, the expression in an expression statement (for example, any expression inside parentheses), the controlling expression of a **while** or **do** statement, any of the three expressions of a **for** statement, or the expression in a **return** statement.

Function and Macro Arguments with Side Effects

Run-time support functions can be implemented either as functions or as macros. Avoid including expressions with side effects inside function invocations unless you are sure the function is not implemented as a macro. Here is an illustration of how an argument with side effects can cause problems:

```
#define limit_number(x) ((x > 1000) ? 1000 : (x))

a = limit_number( a++ );
```

If `a` is greater than 1000, it is incremented once. If `a` is less than or equal to 1000, it is incremented twice, which is probably not the intended behavior.

A macro can be used safely with an argument that has side effects if it evaluates its parameter only once. You can determine whether a macro is safe only by inspecting the code.

A common example of a run-time support function that is often implemented as a macro is **toupper**. You will find your program's behavior confusing if you use the following code:

```
char c;

c = toupper( getc() );
```

If **toupper** is implemented as a function, `getc` is called only once and its return value is translated to uppercase. However, if **toupper** is implemented as a macro,

`getc` is called once or twice, depending on whether `c` is uppercase or lowercase. Consider the following macro example:

```
#define toupper(c) ( (islower(c)) ? _toupper(c) : (c) )
```

If you include the **toupper** macro in your code, the preprocessor expands it as follows:

```
/* What you wrote */
c = toupper( getc() );

/* Macro expansion */
ch = (islower( (getc()) ) ? _toupper( getc() ) : (getc()) );
```

The expansion of the macro shows that the argument to `toupper` is always called twice: once to determine if the character is lowercase and the next time to perform case translation (if necessary). In the example, this double evaluation calls the `getc` function twice. Because `getc` is a function whose side effect is to read a character from the standard input device, the example requests two characters from standard input.

Environment Differences

Many programs perform some file input/output. When writing these programs for portability, consider the following:

- Do not hard-code filenames or paths. Use constants you define either in a header file or at the beginning of the program.
- Do not assume the use of any particular file system. For example, the UNIX-model, hierarchical file system is prevalent on small computers. On larger systems, the file system often follows a different model.
- Do not assume a particular display size (number of rows and columns).
- Do not assume that display attributes exist. Some environments do not support such attributes as color, underlined text, blinking text, highlighted text, inverse text, protected text, or dimmed text.

9.3 Portability of Data Files

Data files are rarely portable across different CPUs. Structures, unions, and arrays have varying internal layout and alignment requirements on different machines. In addition, byte ordering within words and actual word length may vary.

The best way to achieve data-file portability is to write and read data files as one-dimensional character arrays. This procedure prevents alignment and padding problems if the data are written and read as characters. The only portability problem you are likely to encounter if you follow this course is a conflict in character sets; many computers have character-set conversion utilities.

9.4 Portability Concerns Specific to Visual C++

Visual C++ offers extensions using which you can take advantage of the full capabilities of the computer. These extensions are not portable to other compilers or environments. For a list of Microsoft-specific keywords, see Chapter 1, "Elements of C," in the *C Language Reference*.

The *Run-Time Library Reference* contains compatibility information for every function in the run-time library. Any function or macro that is not marked as ANSI compatible may not be portable to other compilers or computer systems.

9.5 Visual C++ Byte Ordering

Tables 9.4 and 9.5 summarize Visual C++ byte ordering for **short** and **long** types, respectively. In these tables, the least-significant byte of the data item is b0; the next byte is denoted by b1, and so on.

Because byte ordering is machine specific, any program that uses this byte ordering is not portable.

Table 9.4 Byte Ordering for Short Types

CPU	Byte order
8086	b0 b1
80286	b0 b1
80386	b0 b1
80486	b0 b1
PDP-11	b0 b1
VAX-11	b0 b1
M68000	b1 b0
Z8000	b1 b0

Table 9.5 Byte Ordering for Long Types

CPU	Byte order
8086	b0 b1 b2 b3
80286	b0 b1 b2 b3
80386	b0 b1 b2 b3
80486	b0 b1 b2 b3
PDP-11	b2 b3 b0 b1
VAX-11	b0 b1 b2 b3
M68000	b3 b2 b1 b0
Z8000	b3 b2 b1 b0

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z

Break the 640K DOS Barrier with Phar Lap and Microsoft Visual C++!

Now you can use your 16-bit Microsoft® Visual C++™ Professional Edition or your Microsoft Win32™ SDK for Windows NT™ (Preliminary Version) compiler to build multi-megabyte DOS applications! With Phar Lap's award-winning DOS-Extenders, your C or C++ program can access all the memory you need and still run under DOS and the Microsoft Windows™ DOS box.

286|DOS-Extender™ and Microsoft Visual C++ Professional Edition

- Access up to 16 megabytes of memory
- No hassles with overlays or EMS
- Debug with CodeView®
- Programs run on any 80286, 386 or 486 PC

386|DOS-Extender™ and Microsoft Win32 SDK for Windows NT (Preliminary Version)

- Access all available memory — up to 4 gigabytes
- Full 32-bit speed and power
- Workstation-like flat memory model
- Programs run on any 80386 or 486 PC

CALL (617) 661-1510 _ _ _ _ _ (cut or fold here) _ _ _ _ _ **CALL (617) 661-1510**

YES! I am interested in Phar Lap's DOS-Extenders. Please send me:

❑ _____ copies of 286|DOS-Extender SDK @ $495 each = _____

❑ _____ copies of 386|DOS-Extender SDK @ $495 each = _____

❑ more information about ❑ 286|DOS-Extender ❑ 386|DOS-Extender

Mass. residents add 5% sales tax: _____

Shipping: _____

	U.S./Canada	International
For each 286|DOS-Extender SDK, add:	$5	$50
For each 386|DOS-Extender SDK, add:	$10	$70

Total: _____

Shipping Address: *Please provide a street address (no P.O. boxes or postal route numbers).*

Name: _____ Position: _____

Company: _____ Phone: _____

Address: _____ Fax: _____

City: _____ State/Country: _____ Zip: _____

Method of Payment:

❑ Check or money order (U.S. $ from U.S. bank) ❑ C.O.D. (U.S. orders only. Add $4)

❑ P.O. # (Approved U.S. orders only): _____

❑ MasterCard ❑ Visa ❑ American Express Card #: _____

Cardholder's Name: _____ Exp. date: _____ Order date: _____

What the experts say about Phar Lap's DOS-Extenders:

286|DOS-Extender

"With 286|DOS-Extender, Phar Lap has brought protected-mode application development within the easy reach of every Microsoft C/C++ owner."
Ray Duncan, PC Magazine, May 1991

"286|DOS-Extender strikes me as a superb way of utilizing all available memory in your machine. Phar Lap has a well-deserved reputation for technical excellence."
Dave Jewell, Program NOW, Aug. 1991

386|DOS-Extender

"Using 386|DOS-Extender has made AutoCAD® 386 the top-selling AutoCAD version. We highly recommend Phar Lap's products."
Robert Wenig, Autodesk

"Interleaf has long recognized Phar Lap's visionary role in the DOS marketplace. I believe Phar Lap produces the finest 32-bit DOS extender available."
Bill Hawkins, Interleaf

Phar Lap® Software, Inc.
60 Aberdeen Avenue
Cambridge, MA 02138
(617) 661-1510
Fax (617) 876-2972

(cut or fold here)

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS MAIL PERMIT NO. 6874 CAMBRIDGE MA

POSTAGE WILL BE PAID BY ADDRESSEE

PHAR LAP SOFTWARE
60 ABERDEEN AVE
CAMBRIDGE MA 02138-9734

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-639

Microsoft®