Programming Techniques

Microsoft® C/C++

Microsoft® C/C++

Version 7.0

Programming Techniques

For MS-DOS® and Windows™ Operating Systems

Microsoft Corporation

Document No. LN24775-1191

10 9 8 7 6 5 4 3 2 1

Contents Overview

Contents

Appendix

Figures and Tables

Figures

Tables

Introduction

Programming Techniques describes how to take advantage of the special features of Microsoft C/C++. The topics covered by this manual include language extensions, special-purpose library functions, and the interaction between programming strategies and compiler options.

This manual is not a reference for the tools included with Microsoft C/C++. If you have specific questions about the CodeView debugger, the Programmer's WorkBench (PWB), or any of the command-line utilities, see the *Environment and Tools* manual or Help.

Scope of This Book

Programming Techniques is divided into two parts. Part 1, "Improving Program Performance," helps you write more efficient programs. It provides specific information about optimizing—when and why to use various optimizing options. It describes precompiled headers, which can reduce compilation time during development. Part 1 explains how to compile your program into p-code, a form of code that produces smaller executable files. It also explains memory management options for both C and C++ and when to use them. Chapter 6 describes the inline assembler, a feature that lets you mix assembly language with your C and C++ source code, and Chapter 7 describes the floating-point math packages.

Part 2, "Special Environments," describes graphics capabilities and the QuickWin library. It also shows how to program in mixed languages and offers tips to make your programs more portable. The Microsoft C run-time libraries contain graphics functions for low-level graphics operations, such as drawing lines, rectangles, and circles. The libraries also contain functions for creating presentation graphics, such as pie charts and bar charts. Microsoft C/C++ also includes a library that lets you convert DOS programs with simple input and output requirements into Windows programs.

Document Conventions

Note The term "DOS" refers to both the MS-DOS and IBM Personal Computer DOS operating systems. The name of a specific operating system is used when it is necessary to note features that are unique to the system.

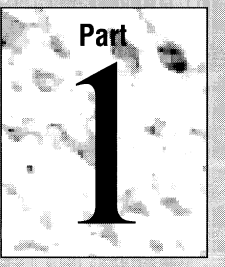
This book uses the following typographic conventions:

Example	Description
STDIO.H	Uppercase letters indicate filenames, segment names, registers, and terms used at the operating-system command level.
char, **_setcolor**, **__far**	Bold type indicates C and C++ keywords, operators, language-specific characters, and library routines. Within discussions of syntax, bold type indicates that the text must be entered exactly as shown. Many functions and constants begin with either a single or double underscore. These are part of the name and are mandatory. For example, to have the **__cplusplus** manifest constant be recognized by the compiler, you must enter the leading double underscore.
expression	Words in italics indicate placeholders for information you must supply, such as a filename.
[[*option*]]	Items inside double square brackets are optional.
#pragma pack {1 | 2}	Braces and a vertical bar indicate a choice among two or more items. You must choose one of these items unless double square brackets ([[ ]]) surround the braces.
`#include <io.h>`	This font is used for examples, user input, program output, and error messages in text.
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.

Improving Program Performance

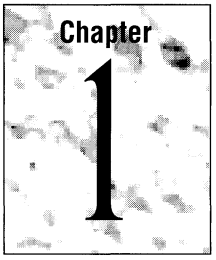


Improving Program Performance

Microsoft C/C++ helps you create the fastest, smallest applications using its sophisticated optimizer and enhanced memory-management capabilities.

Chapter 1 tells you when to use certain optimizations and describes how Microsoft C/C++ generates code that is efficient in execution speed and in size. Chapter 2 describes precompiled headers, a feature that can dramatically reduce compilation time. Chapter 3 explains how to compile your program into p-code, a type of code that produces smaller executable files. Chapters 4 and 5 explain the tools Microsoft C/C++ gives you to allocate and manage program memory in both C and C++, including the **__based** type and the virtual memory manager. For situations where your program requires localized optimization, you can use the inline assembler, described in Chapter 6, to produce the tightest possible code. If your application requires floating-point math computation, you will find Chapter 7 helpful in explaining the options in the Microsoft C/C++ math packages.

Optimizing Your Programs



The Microsoft C/C++ compiler translates C or C++ source statements into machine-executable instructions. In addition, the compiler rewrites or "optimizes" parts of your program to make it more efficient in ways that are not apparent at the source level.

The compiler performs three general types of optimization:

- It modifies or moves sections of code so that fewer instructions are used, or so that the instructions used make more efficient use of the processor.
- It moves code and combines operations to maximize use of registers because operations on data stored in processor registers are far faster than the same operations on data stored in memory.
- It eliminates sections of code that are redundant or unused.

This chapter explains the various ways you can control how the Microsoft C/C++ compiler optimizes your code.

1.1 Controlling Optimization from PWB

The Programmer's WorkBench (PWB) is an integrated development environment for editing, building, and debugging applications written in Microsoft C or C++. For more information on the PWB, see the *Environment and Tools* manual.

There are two ways to compile from inside the Programmer's WorkBench:

- Debug compile. In a default debug compile, the compiler performs no optimizations at all.
- Release compile. In a default release compile, the compiler performs most optimizations.

To select the optimizations the compiler performs in either a debug compile or a release compile, pull down the Options menu and choose the Language Options

submenu. From that menu, open either the C or C++ Compiler Options dialog box. From that dialog box, you can specify either a debug or a release compile, and you can open the Optimizations dialog box to select individual optimizations.

The optimizations in each of the Compiler Options dialog boxes correspond to a command-line option to CL. (In fact, the PWB constructs a command line from your input and passes it to CL.)

Note In this chapter, optimization options are discussed in terms of the effect of the optimization, the command-line option to invoke the optimization, and pragmas that control the optimization. All of these optimizations can be controlled at the compilation-unit (file) level using the Compiler Options dialog boxes.

1.2 Controlling Optimization from the Command Line

Controlling optimization from the command line requires that you determine which optimizations you need for your application. You then specify those optimizations using command-line options that begin with /O (and in some cases /G).

If there is any conflict between options, the compiler uses the last option specified on the command line. The command line

```
CL /Oa /Ol /Ot TEST.C
```

compiles the program TEST.C. It specifies that the compiler can

- Optimize on the assumption that you are doing no aliasing (/Oa)
- Perform loop optimization (/Ol)
- Perform other general speed-enhancing optimizations (/Ot)

The preceding command line can also be written

```
CL /Oalt TEST.C
```

1.3 Controlling Optimization with Pragmas

Occasionally you will need to exercise a fine level of control over compiler optimizations. Command-line options allow you to control optimization over an entire compilation unit (file). In addition, Microsoft C/C++ supports several pragmas that allow you to exercise such control on a per-function basis.

The pragmas that control optimization are described in this chapter under the type of optimization they affect.

You can control each of the following optimization parameters on a function-by-function basis using the **optimize** pragma:

- Behavior of code with respect to aliasing (**a** and **w**)
- Inlining of function calls (**b0**, **b1**, or **b2**)
- Reduction of local common subexpressions (**c**)
- Reduction of global common subexpressions (**g**)
- Global register allocation (**e**)
- Loop optimization (**l**)
- Maximization of optimizations (**x**)
- Aggressiveness of optimizations (**z**)
- Disabling of unsafe optimizations (**n**)
- Achieving consistent floating-point results (**p**)
- Use of a single exit point for each function (**r**)
- Optimizing for smaller code size or for faster execution speed (**s** or **t**)

There is also an option for compiling your program into p-code (/Oq), and options that apply only when p-code is enabled (/Of, /Of–, /Ov, and /Ov–). See Chapter 3, "Reducing Program Size with P-Code," for information on these options.

Any optimization or combination of options can be enabled or disabled using the **optimize** pragma. For example, if you have one function that uses aliases heavily, you need to inhibit optimizations that could cause problems with aliases. You do not, however, want to inhibit these optimizations for code that does not do aliasing. To do this, use the **optimize** pragma as follows:

```
/* Function(s) that do not do aliasing. */
.
.
.
#pragma optimize( "a", off )
/* Function(s) that do aliasing. */
.
.
.
#pragma optimize( "a", on )
/* More function(s) that do not do aliasing. */
```

The parameters to the **optimize** pragma can be combined in a string to enable or disable multiple options at once. For example,

```
#pragma optimize( "lge", off )
```

disables loop optimization, global common subexpression optimization, and global register allocation.

1.4 Default Optimization

Many optimizations are not explicitly disabled by any command-line option except /Od (disable optimizations) or /f (fast compile). These optimizations are small in scope and are almost always helpful. They include

- Short-range common subexpression elimination
- Dead-store elimination
- Constant propagation

Common Subexpression Elimination

In common subexpression elimination, the compiler finds code containing repeated subexpressions and produces modified code in which the subexpressions are evaluated only once. Subexpression elimination is usually done with temporary variables as shown in the following example:

```
a = b + c * d;
x = c * d / y;
```

The preceding two lines contain the common subexpression `c * d`. This code can be modified to evaluate `c * d` only once; the result is placed in a temporary variable (usually a register):

```
tmp = c * d;
a = b + tmp;
x = tmp / y;
```

Dead-Store Elimination

Dead-store elimination is an extension of common subexpression elimination. Variables that contain the same value in a short piece of code can be combined into a single temporary variable.

In the following code fragment, the compiler detects that the expression `func( x )` is equivalent to `func( a + b )`:

```
x = a + b;
x = func( x );
```

Thus, the compiler can rewrite the code as follows:

```
x = func( a + b );
```

Constant Propagation

When doing constant propagation, the compiler analyzes variable assignments and determines if they can be changed to constant assignments. In the following example, the variable `i` must have a value of `7` when it is assigned to `j`:

```
i = 7;
j = i;
```

Instead of assigning `i` to `j`, the constant `7` can be assigned to `j`:

```
i = 7;
j = 7;
```

While you could make any of these changes in the source file, doing so might reduce the readability of the program. In many cases, optimizations not only increase the efficiency of the program but allow you to write more readable code without any actual efficiency loss.

Remove optimization before using a symbolic debugger.

In some cases, you might want to disable even the default optimizations. Because optimizations may rearrange code in the object file, it can become difficult to recognize parts of your code during debugging. It is usually best to remove all optimization before using a symbolic debugger. You can remove all optimization with the /Od (disable optimizations) option or the /f (fast compile) option.

You can disable all optimizations for a function by including the statement `#pragma optimize( "", off )`. To restore optimization to its former state, use the statement `#pragma optimize( "", on )`.

1.5 Customizing Your Optimizations

The default optimizations are sufficient for many applications, but you may want to tune your programs according to criteria not known to the compiler. The optimization options offer you a way of providing the compiler specific goals for optimizing your code.

Choosing Speed or Size (/Ot and /Os)

In addition to the default optimizations, the Microsoft C/C++ compiler also automatically uses the /Ot option, which optimizes for speed. The /Ot option enables optimizations that increase speed but may also increase size. If you would rather optimize for program size, use the /Os option. The /Os option enables optimizations that decrease program size but may also decrease program speed.

To optimize for speed or size on a per-function basis, use the **optimize** pragma with the **t** option. The on setting instructs the compiler to optimize for speed; the

off setting instructs the compiler to optimize for compactness of code. For example,

```
#pragma optimize( "t", off )     /* Optimize for smallest
                                    code. */
.
.
.
#pragma optimize( "t", on )      /* Optimize for fastest
                                    code. */
```

The /Os option is implied when you use the /Oq option (p-code generation).

Generating Intrinsic Functions (/Oi)

In place of some normal function calls, the C/C++ compiler can insert "intrinsic functions," which operate more quickly. Every time a function is called, a set of instructions must be executed to store parameters and to create space for local variables. When the function returns, more code must be executed to release space used by local variables and parameters and to return values to the calling routine. These instructions take time to execute. In the context of an average-sized function, the additional code is minimal, but if the function is only a line or two, the additional code can comprise almost half of the function's compiled code.

One way to avoid this type of code expansion is to avoid such short functions, especially in often-used sections of code where speed is critical. But many library functions contain only a line or two of code. The compiler provides two forms of certain library functions. One form is a standard C function, which requires the overhead of a function call. The other form is a set of instructions that performs the same action as the function without issuing a function call. This second form is called an intrinsic function. Intrinsic functions are always faster than their function-call equivalents and can provide significant optimizations at the object-code level.

For example, the function **strcpy** might be written as follows:

```
int strcpy(char * dest, char * source)
{
    while( *dest++ = *source++ );
}
```

The compiler contains an intrinsic form of **strcpy**. If you instruct the compiler to generate intrinsic functions, any call to **strcpy** will be replaced with this intrinsic form.

Note While the preceding example is written in C for clarity, most of the library functions use assembly language to take full advantage of the 80*x*86 instruction set. Intrinsic functions are not simply library functions defined as macros.

Compiling with the /Oi option causes the compiler to use the intrinsic forms of the following functions:

abs	**labs**	**memset**	**strcat**
_disable	**_lrotl**	**_outp**	**strcmp**
_enable	**_lrotr**	**_outpw**	**strcpy**
_inp	**memcmp**	**_rotl**	**strlen**
_inpw	**memcpy**	**_rotr**	**_strset**

While the following floating-point functions do not have true intrinsic forms, they do have versions that pass arguments directly to the floating-point chip instead of pushing them on the normal argument stack:

acos	**fmod**	**_acosl**	**_fmodl**
asin	**log**	**_asinl**	**_logl**
atan	**log10**	**_atanl**	**_log10l**
atan2	**pow**	**_atan2l**	**_powl**
ceil	**sin**	**_ceill**	**_sinl**
cos	**sinh**	**_cosl**	**_sinhl**
cosh	**sqrt**	**_coshl**	**_sqrtl**
exp	**tan**	**_expl**	**_tanl**
floor	**tanh**	**_floorl**	**_tanhl**

Warning! The compiler performs optimizations assuming math intrinsics have no side effects. This assumption is true except if you have written your own **_matherr** function and that function alters global variables. If you have written a **_matherr** function to handle floating-point errors, and your function has side effects, use the **function** pragma to instruct the compiler not to generate intrinsic code for math functions.

If you want the compiler to generate intrinsic functions for only a subset of the functions listed above, use the **intrinsic** pragma rather than the /Oi option. The **intrinsic** pragma has the following format:

#pragma intrinsic(*function1*, ...)

If you want to have intrinsic functions generated for most of the functions above and function calls for only a few, compile with the /Oi option and force function use with the **function** pragma. The **function** pragma has the following format:

#pragma function(*function1*, ...)

The following code illustrates the use of the **intrinsic** pragma:

```
#pragma intrinsic(abs)

void main( void )
{
    int i, j;

    i = big_routine_1();
    j = abs( i );
    big_routine_2( j );
}
```

Generating intrinsic functions for this program causes the call to **abs** to be replaced with assembly-language code that takes the absolute value of a number. The program will execute more quickly because the function-calling overhead is no longer required when **abs** is called.

In the previous example, the overall speed increase is small because there is only a single call to **abs**. In the following example, where the call to **abs** is in a loop and there are many calls, you can save a significant amount of execution time by generating intrinsic functions.

```
#pragma intrinsic( abs )
void main( void )
{
int i, j, x;

    for( j = 0; j < 1000; j++ )
    {
    for( i = 0; i < 1000; i++ )
    {
        x += abs( i - j );
    }
   }
   printf( "The value of x is %d\n", x );
}
```

The following is a list of restrictions on using the intrinsic forms of function calls:

- Do not use the intrinsic forms of the floating-point math functions with the alternate math libraries (*m*LIBCA*y*.LIB).
- The /Oi option is not available if you use the /Oq option (p-code generation).
- If you use the /Ox (maximum optimization) option, you are enabling the /Oi (generate intrinsic functions) option. Be careful that your use of /Ox does not conflict with the points listed previously.

Inlining Function Calls (/Ob0, /Ob1, and /Ob2)

Inlining is similar to the use of intrinsic functions, except that it is not restricted to a specific set of library functions. Inlining allows the compiler to insert a copy of a function in each place it is called. This removes the overhead of calling a function (described in the previous section), but having multiple copies of a function can make your program larger.

You can explicitly mark a function as a candidate for inlining by declaring it with the **__inline** keyword, or in C++, the **inline** keyword. Any C++ member functions that are defined within the class declaration are implicitly considered inline functions.

Inlining is performed at the discretion of the compiler. If a function more than a few lines long is declared as an inline function, the compiler ignores the **__inline** keyword. The /Ob*n* option controls the degree to which the compiler performs inlining.

The /Ob0 option disables all inlining, even for functions explicitly declared as inline functions. This is the default when /Od is specified.

The /Ob1 option expands all functions declared as inline, at the compiler's discretion. This is the default when /Od is not specified.

The /Ob2 option expands all functions declared as inline, at the compiler's discretion, and any other functions that the compiler considers suitable for inlining.

Assuming No Aliasing (/Oa and /Ow)

The /Oa and /Ow options control the assumptions the compiler makes regarding "aliasing" when it performs optimizations. These options can significantly improve the performance of your program, but there are a few situations in which these options are not appropriate.

"Aliasing" occurs when more than one name is used to refer to a single memory location. For example,

```
char    c;
char    *cptr;

cptr = &c;      /* Take the address of c */
           /* c now has two names: c and *cptr */
c = 1;          /* Use first name  */
*cptr = 2;      /* Use second name */
```

The expression `*cptr` is an alias for `c`, because it is another name for that variable. If you refer to the variable by both names, you are using aliasing.

One optimization technique that the compiler performs is to store frequently used variables in registers because accessing a register takes less time than accessing a memory location. If the compiler detects the use of aliasing, it does not place the variable in a register because modifications through an alias could lead to inconsistent values being used for that variable. For example, consider the code fragment above. If the compiler placed `c` in a register, and if you modified `*cptr`, the compiler would have inconsistent values for `c`: one stored in a register, and another stored in a memory location. To avoid this problem, the compiler does not place `c` in a register.

The compiler can detect simple cases of aliasing like the one described above. However, the compiler cannot identify all possible forms of aliasing. By default, the compiler assumes that your program may be using aliasing that it cannot detect, just to be on the safe side. This means that the compiler assumes that any time you modify a memory location through a pointer, you might also be modifying the value of one of the following:

- Any global variable
- Any local variable whose address has been taken
- The memory location referenced by any other pointer

This assumption limits the amount of optimization the compiler can perform.

The /Oa option tells the compiler that your program does not perform any aliasing (other than the very simple forms that the compiler can detect). This allows the compiler to optimize your code more fully. However, if you specify this option when compiling a program that does perform aliasing, the compiler may produce incorrect code.

Here is an example of a program that performs aliasing that the compiler cannot detect. This program generates incorrect results when compiled under /Oa:

```
/* OATEST.C
 * Fails when compiled with \Oa.
 * Passes when compiled with default optimization.
 */
#include <stdio.h>
char buf[10];        /* Global array */

char *return_buf()
{
    return buf;
}

void main()
{
    char *first,
     *second;
```

```
    first = buf;
    second = return_buf();
    *first = 2;
    *second = 3;
    if( *first == 3 )
    printf( "Pass\n" );
    else
    printf( "Fail\n" );
}
```

In this example, both `*first` and `*second` refer to the same memory location. This location is assigned two different values, one through each of its names. If this program is compiled without the /Oa option, the compiler assumes that the reference to `*second` could refer to the same memory location as does `*first`, and that `*first` might be modified by the statement `*second = 3`. If, however, you do specify the /Oa option, the compiler assumes that `*first` and `*second` refer to different memory locations. The compiler then assumes `*first` retains the value of 2, so it skips the **if** statement and goes straight to the **else** clause, which prints "Fail."

The reason the compiler cannot detect the aliasing in the previous example is that the compiler examines code in only one function at a time when it performs optimization. The compiler doesn't examine the code in `return_buf` when it is compiling the function `main`. Consequently, the compiler has no indication that `*first` and `*second` refer to the same variable. If you were to explicitly set `second` to the value of `first` within the function `main`, the compiler would detect the aliasing.

The compiler can detect aliasing only when a pointer is explicitly set to the address of a variable. Aside from that case, when /Oa is in effect, the compiler assumes that a variable's value is changed by operations to only that variable itself, and not by operations to any other variable. The compiler does assume that a variable's value may be changed when the variable's address is passed to a function. This also applies if the address is cast to an integer when the function is called. For example,

```
void func1( char *cptr );  /* Prototype for func1 */
void func2( int i );       /* Prototype for func2 */

void main()
{
    char c;
    int j;

    c = 'a';
    func1( &c );        /* Compiler assumes c may be modified */

    func2( (int)&c );   /* Compiler assumes c may be modified */
```

```
    j = (int)&c;
    func2( j );          /* Dangerous: compiler doesn't assume */
            /*    that c may be modified          */
}
```

Note that the compiler does not assume that `func2` can change the value of `c` when a separate integer variable is used to pass `c`'s address.

The /Oa option means "assume no aliasing." The /Ow option means "assume no aliasing, except between functions."

Another option that controls aliasing assumptions is the /Ow option, which tells the compiler that you are performing aliasing between functions. When you specify the /Ow option, the compiler assumes that calling any function may have side effects, instead of only those functions that take pointers as parameters. That is, calling any function may modify the value of any global variable, or any local variable whose address has been taken, or the memory location referenced by any pointer. (This option is useful in Windows programming, because certain Windows functions may cause the contents of handles to be modified.) Consequently, after a function call the compiler reloads the value of variables stored in registers, and the compiler does not perform certain optimizations (such as common subexpression elimination or dead-store elimination) across function calls.

Here is a program that performs aliasing between functions. This program generates incorrect results when compiled with /Oa, but generates correct results when compiled with /Ow:

```
/* OWTEST.C
 * fails when compiled with -Oa
 * passes when compiled with -Ow
 */
#include <stdio.h>
#include <malloc.h>
typedef struct list
{
    struct list *next,
        *back;
    int val;
} LIST;

LIST *glob_plist;

LIST *setup( size_t size )
{
    glob_plist = malloc( size );
    return glob_plist;
}

void ow_func()
{
    glob_plist->val = 22;
}
```

```
void main()
{
    LIST *plist;

    plist = setup( sizeof( LIST ) );
    plist->val = 23;
    ow_func();          /* Function modifies plist->val */
    if( plist->val == 22 )
    {
    printf( "Pass\n" );
    exit( 0 );
    }
    else
    {
    printf( "Fail\n" );
    exit( 1 );
    }
}
```

In this example, both `plist->val` and `glob_plist->val` refer to the same memory location, though each appears in a different function. If you compile this program with /Oa, the compiler assumes that the value of `plist->val` is not changed by the call to `ow_func`. As a result, the compiler assumes that `plist->val` still equals 23 in the next statement, so it skips the **if** statement and goes straight to the **else** clause, which prints "Fail." If you compile this program with /Ow, the compiler assumes that `ow_func` may change the value of `plist->val`. The compiler performs the **if** statement, so the program prints "Pass."

Like the /Oa option, the /Ow option assumes that code within a single function does not perform aliasing; the only difference between the two is in their assumptions regarding function calls. The /Ow option is "weaker" than the /Oa option, because it tells the compiler that you may perform a certain type of aliasing, while the /Oa option tells the compiler that you are performing none. As a result, the /Ow option causes the compiler to perform less optimization than the /Oa option (but more than if neither option were specified).

Take the following steps to see if the /Oa or /Ow options are appropriate for your program:

1. When developing your program, compile your program without the /Oa option. (To make it easier to debug your program with a symbolic debugger, you should compile your program without any optimizations, using /Od.)
2. Once you're satisfied that your program executes correctly, compile the program with /Oa and any other optimizations you want.
3. If the program no longer executes correctly, your program is performing aliasing that the compiler cannot detect. Aliasing bugs most frequently show up as corruption of data, where global or local variables are being assigned seemingly random values. If you can locate the functions in which aliasing is occurring, you can use pragmas to turn off the /Oa option for those particular functions.

4. If you cannot find the functions in which aliasing is occurring, replace the /Oa option with /Ow and recompile your program.
5. If the program still does not execute correctly, and you cannot find the functions in which aliasing is occurring, remove both /Oa and /Ow from your compile options.

If you are looking for instances of aliasing in your program, look for the following situations:

- When a variable, particularly a global variable, is referenced through both the variable itself and a pointer to that variable
- When multiple pointers are used to reference the same memory location(s)

In the preceding list, the term "reference" means read or write; that is, whether a variable is on the left-hand side of an assignment statement or the right-hand side, you are still referring to it. In addition, any function calls that use a variable as a parameter are references to that variable.

Note that the compiler assumes the value of variables declared as **volatile** may change at any time. As a result, the compiler does not perform any optimization on such variables.

Performing Loop Optimizations (/Ol)

The /Ol option enables a set of optimizations involving loops. Because loops involve sections of code that are executed repeatedly, they are targets for optimization. These optimizations all involve moving or rewriting code so that it executes faster.

Loop optimization can be turned on with the /Ol option or with the **loop_opt** pragma. The following line enables loop optimization for all subsequent functions:

```
#pragma loop_opt( on )
```

The following line turns it off:

```
#pragma loop_opt( off )
```

The /Ol option removes invariant code.

An optimal loop contains only expressions whose values change through each execution of the loop. Any subexpression whose value is constant should be evaluated before the body of the loop is executed. Unfortunately, these subexpressions are not always readily apparent. The optimizer can remove many of these expressions from the body of a loop at compile time. This example illustrates invariant code in a loop:

```
i = -100;
while( i < 0 )
{
    i += x + y;
}
```

In the preceding example, the expression `x + y` does not change in the loop body. Loop optimization removes this subexpression from the body of the loop so that it is only executed once, not every time the loop body is executed. The optimizer will change the code to the following fragment:

```
i = -100;
t = x + y;
while( i < 0 )
{
    i += t;
}
```

Loop optimization is much more effective when the compiler can assume no aliasing. While you can use loop optimization without the /Oa or /Ow option, use /Oa to ensure that the most options possible are used.

Here is a code fragment that could have an aliasing problem:

```
i = -100;
while( i < 0 )
{
    i += x + y;
    *p = i;
}
```

If you do not specify the /Oa option, the compiler must assume that either `x` or `y` could be modified by the assignment to `*p`. Therefore, the compiler cannot assume the subexpression x + y is constant for each loop iteration. If you specify that you are not doing any aliasing (with the /Oa option), the compiler assumes that modifying `*p` cannot affect either `x` or `y`, and that the subexpression is indeed constant and can be removed from the loop, as in the previous example.

Note All loop optimizations specified by the /Ol option or the **loop_opt** pragma are safe optimizations. To enable aggressive loop optimizations, you must use the enable aggressive optimizations (/Oz) option. While the optimizations enabled by the combination of /Ol and /Oz are not safe for all cases, they will work properly for most programs.

Calling the **setjmp** or **longjmp** functions when loop optimization is in effect can cause the compiler to generate incorrect code. Use the **loop_opt** pragma or the **optimize** pragma with the **g** option to disable this optimization in functions that call **setjmp** and **longjmp**.

Disabling Unsafe Loop Optimizations (/On)

The disable unsafe loop optimizations (/On) option is obsolete and is only retained for compatibility with existing makefiles. Loop optimizations are, by default, safe optimizations. The /On option is the default and has the opposite effect of the /Oz (enable aggressive optimizations) option.

Enabling Aggressive Optimizations (/Oz)

The compiler can perform extremely aggressive optimizations. These optimizations produce high code quality both in terms of speed and size. Certain programs, however, cannot be optimized with the technologies enabled by the /Oz option. For these programs, you should not specify this option; you can still use all other optimization options.

Because the optimization strategies enabled by the /Oz option are so aggressive, they are not part of the maximum optimization (/Ox) option.

Examples of the effects of the /Oz option are

- Loop optimization (/Ol). Loop optimization enables a technology that anticipates program flow and tries to remove invariant expressions from loops. When you specify the enable aggressive optimizations option (/Oz), the compiler removes invariant expressions even when it might cause an error. Errors with the enable aggressive optimizations option occur most often when an invariant expression that can cause an exception is protected by an **if** statement. The invariant expression is hoisted out of the loop body, causing it to be evaluated prior to the evaluation of the **if** statement that was designed to protect it. Here are two examples that illustrate this problem:

```
for( i = 0; i  100; ++i )
   if( float_val != 0.0F )
   /* Protect against divide-by-zero. */
       float_result = pi / float_val;

while( condition )
   if( ptr_val != NULL )
     /* Protect pointer dereference. */
       char_var = *ptr_val;
```

- Global register allocation (/Oe). The enable aggressive optimizations option enables some register allocation strategies that can cause invalid segment selectors to be placed in registers. Although this problem is benign in DOS, it causes protection faults in Windows.

Note You can instruct the compiler to enable aggressive optimizations on a function-by-function basis by using the **optimize** pragma with the **z** option.

Enabling Function-Level Linking (/Gy)

The /Gy option enables linking on a function-by-function basis. When function-level linking is enabled, the linker removes unreferenced functions from the executable file, making your program significantly smaller. See Chapter 13, "CL Command Reference," of the *Environment and Tools* manual for more information.

C++ member functions are always compiled with function-level linking enabled. Use the /Gy option for nonmember functions in C++ and all functions in C.

Removing Stack Probes (/Gs)

Every time a function is called, the stack provides space for all parameters and local variables declared in that function. A short assembly function that checks for a stack overflow condition is then called. Stack overflows are usually caused either by infinite loops or by runaway recursive routines. Such errors can also be caused by extremely large parameters or local variables.

Stack probes can be important during program development. Stack-overflow errors alert you to problems in your code. When the program has been tested, however, stack checking often becomes unnecessary. The compiler allows you to remove stack-checking code with either the /Gs option or the **check_stack** pragma. Eliminating stack probes produces programs that are smaller and that run more quickly.

Enabling Global Register Allocation (/Oe)

The global register allocation option (/Oe) instructs the compiler to analyze your program and allocate CPU registers as efficiently as possible. Without the global register allocation option, the compiler uses the CPU's registers for several purposes:

- Holding temporary copies of variables
- Holding variables declared with the **register** keyword
- Passing parameters to functions declared with the **__fastcall** keyword (or functions in programs compiled with the /Gr command-line option)

Variables in registers are sometimes placed back in memory.

When you enable global register allocation, the compiler ignores the **register** keyword and allocates register storage to variables (and possibly to common subexpressions). The compiler allocates register storage to variables or subexpressions according to frequency of use. Because of the limited number of physical registers, variables held in registers are sometimes placed back in memory to free the register for another use.

Here is a C program example that demonstrates how the compiler might rewrite your code to accomplish this:

```
/* Original program */

func()
{
    int i, j;
    char *pc;

    for( i = 0; i < 1000; ++i )
    {
    j = i / 3;
    *pc++ = (char)i;
    }

    for( j = 0, -pc; j < 1000;
    ++j, -pc )
    *pc-;
}

/* Example of how the compiler might optimize the
 * code to move i and j in and out of registers */

func()
{
    int i, j;
    char *pc;

    {
    register int i; /* i is in a register for this block. */
    for( i = 0; i < 1000; ++i )
    {
        j = i / 3;
        *pc++ = (char)i;
    }
    }

    {
    register int j; /* j is in a register for this block. */
    for( j = 0, -pc; j < 1000;
        ++j, -pc )
        *pc-;
    }
}
```

In the preceding example, there are blocks (enclosed in curly braces) whose only purpose is to delimit the span of code across which variables should remain in registers.

Note You can enable or disable global register allocation on a function-by-function basis using the **optimize** pragma with the **e** option.

Calling the **setjmp** or **longjmp** functions when global register optimization is in effect can cause the compiler to generate incorrect code. Use the **optimize** pragma with the **e** option to disable this optimization in functions that call **setjmp** and **longjmp**.

Enabling Common Subexpression Optimization (/Oc and /Og)

When you use option /Og (enable global common subexpression optimizations), the compiler searches entire functions for common subexpressions. Option /Oc (default common subexpression optimization) examines only short sections of code for common subexpressions. You can disable default common subexpression optimization with the /Od option. For more information about common subexpression optimization, see "Default Optimization" on page 8.

This option is not available if you use the /Oq option (p-code generation).

Note You can enable or disable block-scope common subexpression optimization on a function-by-function basis using the **optimize** pragma with the **c** option. You can enable or disable global common subexpression optimization on a function-by-function basis using the **optimize** pragma with the **g** option.

Calling the **setjmp** or **longjmp** functions when global commmon subexpression optimization is in effect can cause the compiler to generate incorrect code. Use the **optimize** pragma with the **g** option to disable this optimization in functions that call **setjmp** and **longjmp**.

Achieving Consistent Floating-Point Results (/Op)

Floating-point numbers stored in memory use either 32, 64, or 80 bits, depending on whether they are of type **float**, type **double**, or type **long double**. The 80*x*87 family of coprocessors uses 80-bit registers for all operations. If a value of type **float** or type **double** is kept in these registers through a number of operations, it will be more accurate than if that value is moved to and from memory between operations.

Because of the difference in precision between memory and register representation of a floating-point number, a value stored in memory is not always equal to the same value in the 80*x*87 register.

The difference in precision primarily affects strict equality or strict inequality tests (== and !=); however, relational tests of magnitude (>, >=, <=, and <) can behave erroneously if the coprocessor is able to maintain significant digits that memory variables cannot.

You can avoid the difference in precision by using the /Op option. This option forces floating-point values to be written to memory between floating-point operations. While storing these values to memory reduces the precision of floating-point expressions, it also ensures that these expressions will produce consistent results regardless of the rest of the code.

You can change the handling of floating-point results on a function-by-function basis using the **optimize** pragma with the **p** option.

Note Using the /Op option suppresses other optimizations because the floating-point registers are not available for storage of intermediate results. Because you suppress these optimizations, code compiled with the /Op option executes more slowly than code compiled without this option.

Generating Code for a Specific Processor (/G0, /G1, /G2, /G3, /G4)

The compiler generates 8086 object code (/G0) unless you take special steps. Because the newer processors (the 80186, 80188, 80286, 80386, and 80486) are backward-compatible with the 8086 instruction set, using this instruction set ensures compatibility with all 80*x*86-based computers. While you gain compatibility across the entire family of 80*x*86 processors, you lose the advantage of some of the more powerful instructions in the newer processors.

If you know your program will only be running on an 80186, 80188, 80286, 80386, or 80486 processor, you can cause the compiler to generate instructions specific to these processors. These instructions increase the speed of your program, but you lose compatibility with machines that use older processors in the 80*x*86 family. Table 1.1 lists the options for processor-specific code generation:

Table 1.1 Processor Compatibility

Command-Line Option	Compatible Processors
/G0	8088, 8086, 80188, 80186, 80286, 80386, 80486
/G1	80188, 80186, 80286, 80386, 80486
/G2	80286, 80386, 80486
/G3	80386, 80486
/G4	80486

The /G3 and /G4 options are only available when compiling a 32-bit program.

Optimizing for Maximum Efficiency (/Ox)

The /Ox option combines a number of different optimizations:

- Enable global register allocation (/Oe)

- Enable global common subexpression optimization (/Og)
- Enable block-scoped common subexpression optimization (/Oc)
- Generate intrinsic functions (/Oi)
- Perform maximum inlining (/Ob2)
- Perform loop optimizations (/Ol)
- Optimize for speed (/Ot)
- Remove stack probes (/Gs)

Use /Ozax /Gr to get the fastest program.

The /Ox option does not include several optimizations that can improve code efficiency: /Oa (assume no aliasing), /Oz (enable aggressive optimizations), and /Gr (use fastcall calling convention). Before enabling these optimizations, you should read the sections that describe the /Oa and /Oz options and the fastcall calling convention to determine if they are appropriate for your application. See the next section for linker options that can make your program faster.

Use /Ose /Gsy to get the smallest program.

If you are more concerned with executable file size than execution time, use the /Ose and /Gsy options. The /Oa option can reduce the size of your program further, but you should use it only if it is appropriate for your application. If your program will be run only on an 80286 or higher processor, use the /G2 option to produce smaller code by using the advanced instruction set. If your program will be run only on machines with an 80*x*87 coprocessor, use the /FPi87 option to produce smaller code for floating-point calculations. See the next section for linker options that can make your program smaller.

1.6 Linker (LINK) Options That Control Optimization

Most code optimization is performed before the object file is produced. There are four optimizations that the linker can perform to speed program execution and reduce the disk space used by an executable file.

Enabling Far Call Optimization (/FARCALLTRANSLATION)

You can call a function two ways. In a far call, the function is called using both the segment and the offset of the function. This allows a program to call a routine outside a 64K segment. In a near call, both the calling statement and the function must be located in the same segment. Only the offset is used to access the function; the segment address is implicit. You can only use near calls to routines located in the same segment.

Because of the architecture of the processor, near function calls execute faster than far calls. The decision to declare functions as near or far is often made when selecting a memory model. As it is difficult to determine where the linker will place a

given function in memory, it is impractical for the programmer to choose the way a function is called.

The /FARCALLTRANSLATION option enables far call optimization. When you use this option, any function calls within the same segment as the function being called are converted to near calls. This optimization has no effect if you have selected the tiny, small, or compact model, because all calls are already near calls.

The abbreviation for the /FARCALLTRANSLATION option is /F.

How /FARCALLTRANSLATION Affects Your Code

The linker can perform a form of post-optimization (an optimization that occurs after most of the actual code generation is complete) that translates far calls into near calls when possible. This optimization allows a given function to be called with both near and far calls in the same program. To perform this translation, the linker takes a section of object code such as

```
CALL    FAR    _func
```

where `func` is defined in the current segment, and replaces it with the following code:

```
PUSH    CS
CALL    NEAR    _func
NOP
```

This substitution works because the linker has inserted `PUSH CS` to place a far return address on the stack.

Use /FARCALLTRANSLATION with /PACKCODE.

The /FARCALLTRANSLATION option is most effective when used in conjunction with the /PACKCODE option discussed in "Packing Code (/PACKCODE)" on this page. Using the /PACKCODE option causes far calls that were intersegment to become intrasegment calls. The /FARCALLTRANSLATION feature can then take advantage of the new grouping to translate all intrasegment far calls into near calls.

Packing Code (/PACKCODE)

The /PACKCODE linker option groups neighboring code segments together. When used with the /F option, the /PACKCODE option greatly increases the number of near calls that can be made to a function. This option can be followed with a limit (expressed in bytes) at which to stop packing and to begin a new group. Here is the syntax for the /PACKCODE option:

/PACKCODE:*number*

where *number* is an optional hexadecimal, octal, or decimal number that specifies the limit for packing. The radix (octal, decimal, or hexadecimal) is specified just as you would specify it to a C or C++ program.

Radix	Rules for Specification
Octal	Specify the octal number with a leading 0. You can only use the digits 0 through 7 in an octal number. For example, 07777.
Decimal	Specify the decimal number without a leading 0. For example, 65530.
Hexadecimal	Specify the hexadecimal number with a leading 0x. For example, 0x3FFF.

If you omit the packing limit, the linker supplies a default value of 65, 530.

The abbreviation for the /PACKCODE option is /PACKC.

Packing Data (/PACKDATA)

The /PACKDATA option is analogous to the /PACKCODE option, except that it groups together neighboring data segments instead of code segments. By grouping data segments, you reduce the overhead needed to use them. Here is the syntax for the /PACKDATA option:

/PACKDATA:*number*

where *number* is an optional hexadecimal, octal, or decimal number that specifies the limit for packing. The radix (hexadecimal, octal, or decimal) is specified just as you would specify it to a C or C++ program. For more information on specifying hexadecimal, octal, or decimal numbers, see "Packing Code (/PACKCODE)" on page 26.

If the packing limit is omitted, the linker supplies a default value of 65,535 (0xFFFF).

The abbreviation for the /PACKDATA option is /PACKD.

Packing the Executable File (/EXEPACK)

The executable file created by the compiler often contains sequences of repeated bytes. You can remove these repeated sequences with the /EXEPACK option. This decreases the size of the resulting executable file as well as program load time.

Note Because the /EXEPACK option removes debug information from the executable file, you should not use it with the /CODEVIEW option.

Removing Unreferenced Functions (/PACKFUNCTIONS)

The /PACKFUNCTIONS option is on by default. This option removes unreferenced functions from the executable file if they were compiled with /Gy (enable function-level linking). This reduces the size of your program significantly.

If you want to keep unreferenced functions in your executable (for example, for debugging purposes), you can turn off the /PACKFUNCTIONS option by specifying /NOPACKFUNCTIONS. See Chapter 14, "Linking Object Files with LINK," of the *Environment and Tools* manual for more information.

1.7 Optimizing in Different Environments

The environment in which you plan to use a program can have a bearing on the types of optimizations that you should use.

Optimizing in DOS

You need not take special precautions for programs written under DOS unless you are writing a terminate-and-stay-resident (TSR) program. If an interrupt-driven routine could modify a memory location in a program, you should declare that variable **volatile**.

Optimizing in Microsoft Windows™

Microsoft Windows™ can move segments dynamically. As a result of dynamic heap compaction, the contents of handles can be modified. The /Ow option specifies that your program does not perform any aliasing except between functions, so the compiler does not perform optimizations across function calls. See "Assuming No Aliasing (/Oa and /Ow)" on page 13 for more information.

The /GA and /GD options optimize the entry/exit code for protected-mode Windows applications and DLLs, respectively. For more information on these options, see Chapter 13, "CL Command Reference," of the *Environment and Tools* manual.

1.8 Choosing Function-Calling Conventions

In Microsoft C/C++, functions can call other functions using three different conventions. Note that, while no calling convention has been defined as "standard," most C/C++ compilers use conventions similar to those described here. The C calling convention requires the most object code to set up, but it is the only calling

convention that supports functions with variable-length argument lists. The FORTRAN/Pascal calling convention is more compact, but does not allow for variable-length argument lists. C++ uses the FORTRAN/Pascal calling convention, except for functions declared with variable-length argument lists, which implicitly use the C calling convention. The **__fastcall**, or register calling convention is the fastest of the three calling conventions, but it does not support variable-length argument lists or mixed-language program interfaces.

The C Calling Convention (/Gd)

Because C allows functions to have a variable number of parameters, parameters must be pushed onto the stack from right to left. (If parameters were pushed from left to right, it would be difficult for the compiler to determine which parameter was first.) If you do not specify command-line options that modify the function-calling convention, the C calling convention is used; otherwise, the **__cdecl** keyword must be used before any function using the C calling convention.

If, for example, you use the /Gr (register calling convention) option when you compile, and the function `add_two` must have the C calling convention, declare `add_two` as follows:

```
int __cdecl add_two( int x, int y );
```

The FORTRAN/Pascal Calling Convention (/Gc)

The FORTRAN/Pascal calling convention is used for C++ functions and any C functions declared with either the **__fortran** or **__pascal** keywords. (The two keywords currently produce identical results.) Parameters to these functions are always pushed on the stack from left to right. While any C function can be declared with the FORTRAN/Pascal convention, it is used primarily for prototypes to Pascal or FORTRAN routines called from within C programs. This calling convention can also produce smaller, faster programs.

The /Gc option (generate Pascal-style function calls) can be used to make all functions in a file observe the FORTRAN/Pascal calling convention.

Note that C run-time library routines must still be called using C calling conventions. Because these routines are declared using the **__cdecl** keyword header files, you must include the appropriate header files in any program using run-time library routines.

Functions with variable-length parameter lists (such as **printf**) cannot use the FORTRAN/Pascal calling convention.

The Register Calling Convention (/Gr)

You can decrease execution time if parameters to functions are passed in registers rather than on the stack. Compiling with the /Gr command-line option enables the register calling convention for an entire file. The **__fastcall** keyword enables the register calling convention on a function-by-function basis.

The register calling convention will produce the most speed benefits in programs that spend a significant amount of time performing function calls, such as recursive programs or programs that call functions from within loops. You should compile small programs with the /Gr option and use **__fastcall** on only selected functions in large programs.

Because the 80*x*86 processor has a limited number of registers, only the first three parameters are allocated to registers; the rest are passed using the FORTRAN/Pascal calling convention.

Note The compiler allocates different registers for variables declared as **register** and for passing arguments using the register calling convention. This calling convention will not conflict with any register variables that you may have declared.

Exercise caution when using the register calling convention for any function written in inline assembly language. Your use of registers in assembly-language could conflict with the compiler's use of registers for storing parameters.

The __fastcall Calling Convention

This section describes the details of the **__fastcall** calling convention. The information is for the use of assembly-language programmers who are interested in using either the inline assembler or the Microsoft Macro Assembler (MASM) to write functions declared as **__fastcall**. Functions declared as **__fastcall** accept arguments in registers rather than on the stack; functions declared as **__cdecl** or **__pascal** accept parameters only on the stack.

Note The register usage documented here may change in future releases of the compiler.

Argument-Passing Convention

The **__fastcall** calling convention is a "strongly typed" register calling convention. This typing allows the compiler to generate better code by passing arguments in registers that correspond to the data type you are passing. Because the compiler chooses registers depending on the type of the argument and not in a strict linear order, the calling program and called function must agree on the types of the arguments in order to communicate data correctly.

For each type of argument there is a list of register candidates. The arguments are allocated to registers or, if no suitable register remains unused, are pushed onto the stack left-to-right. Each argument is put in the first register candidate that does not already contain an argument. Table 1.3 shows the basic types and the register candidate list for each.

Table 1.3 Register Candidates

Type	Register Candidates
character	AL, DL, BL
unsigned character	AL, DL, BL
integer	AX, DX, BX
unsigned integer	AX, DX, BX
long integer	DX:AX
unsigned long integer	DX:AX
near pointer	BX, AX, DX
far or huge pointer	passed on the stack

All far and huge pointers are pushed on the stack, as are all structures, unions, and floating-point types.

Return Value Convention

The **__fastcall** return value convention is based on the size of the return value, except with floating-point types. All floating point types are returned on the top of the NDP stack. For more information about the NDP stack and returning floating-point values, see Chapter 7, "Controlling Floating-Point Math Operations." The following list shows how values 4 bytes or smaller, including unions and structures, are returned from a **__fastcall** function.

Size	Return Convention
1 Byte	AL Register
2 Bytes	AX Register
4 Bytes	DX, AX Registers (for pointers, the segment is returned in DX, the offset in AX; for long integers, the most-significant byte is returned in DX, least-significant byte in AX)

Note that the protocol for returning values 4 bytes or smaller is the same as for functions declared as **__cdecl**. To return structures and unions larger than 4 bytes, the calling program passes a hidden parameter as the last item pushed. This parameter is a near pointer, implicitly SS-relative, to a buffer in which the value is to be returned. A far pointer to SS:*hidden-param* must be returned in DX:AX. This is the same convention for returning structures as **__pascal.**

Stack Adjustment Convention

Unlike functions declared as __**cdecl**, functions declared as __**fastcall** must pop the arguments off the stack. The calling program does not adjust the stack after function return.

Register Preservation Requirement

All functions must preserve the DS, BP, SI, and DI registers. Your __**fastcall** function can modify the values in AX, BX, CX, DX, and ES.

Function-Naming Convention

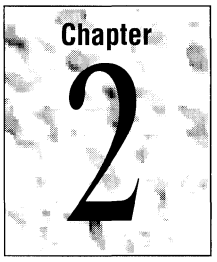
The public name put into the object file for a function declared as __**fastcall** is the name given by the user with a leading "at sign" (@). No case translation is performed on the function name. The function declaration

```
int __fastcall FCFunc( void );
```

causes the compiler to place the public symbol `@FCFunc` in your object file at every location `FCFunc` is referenced in your program.

If you do not declare the function as __**fastcall** in your C or C++ program, the compiler assumes the default calling convention. The default for C is usually the C calling convention but can be changed by the /Gc (FORTRAN/Pascal Calling Convention), /Gr (Register Calling Convention), or /Gd (C Calling Convention) options. The default for C++ is the FORTRAN/Pascal calling convention. If the linker gives you an unresolved external reference, you may have failed to declare an external __**fastcall** function properly. For more information about calling conventions, see Chapter 11, "Programming with Mixed Languages."

Using Precompiled Header Files



Microsoft C/C++ provides the option to precompile header files. Precompilation, especially when used in conjunction with the fast compile (/f) option, can dramatically reduce compile time for header files that are frequently compiled without modification. This process saves the state of a compilation (including CodeView information) at a point after header files have been processed. In later compilations, the compiler simply restores the saved compilation state from a precompiled header (.PCH) file, rather than recompiling the unchanged header files.

2.1 When to Use Precompiled Headers

Precompiled headers are useful during the development cycle to reduce compilation time, especially if:

- You are changing the body of your source files more frequently than the header files, or the header files comprise a significant portion of the code for that module.
- Your program is comprised of multiple modules, all of which use a standard set of include files. In this case, all include files can be precompiled into one precompiled header (if all these modules use the same compilation options).

The first compilation—the one that creates the precompiled header file—takes a bit longer than a normal compilation. Subsequent compilations can proceed more quickly by including the precompilation of the header files.

Precompiled headers work for C and C++ programs.

Precompiled headers work for 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 You can have only one precompilation per source file. The compiler saves or restores its state at a specified point, then continues as it normally would. You use precompiled versions of multiple header files with the same source file.

2.2 Creating and Using Precompiled Headers

You create a new precompiled header (.PCH) file by compiling with the /Yc ("create") option. This option lets you specify the name of the .PCH file and choose the place at which to save the compilation state. You can also control these factors using the the /Fp ("precompiled header filename") option and the optional **hdrstop** pragma.

You use a precompiled header by compiling with the /Yu ("use") option, which tells the compiler to restore the compilation state from a given precompiled header. Like the /Yc option, this option offers alternate ways to specify filenames, and it can make use of the **hdrstop** pragma.

The following sections describe the precompiled header options and the **hdrstop** pragma in more detail.

2.3 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 36, gives you extra control over the behavior of these options.

Create Precompiled Header Option (/Yc)

The "create precompiled header" option (/Yc) 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 creates a precompiled header consisting of the state of the compilation up to and including the preprocessing of an include file with that name that is specified in your base source 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.

Precompiled Header Filename Option (/Fp)

The /Fp option gives you extra control over the name of the .PCH file. Use it to specify a .PCH filename that is different from the name of the associated include file or source file. For example, if you want to create a precompiled header file for a debugging version of your program, you can specify a command such as:

```
CL /DDEBUG /YcPROG.H /FpDPROG PROG.CPP
```

This command creates a precompilation of all header files up to and including PROG.H and stores it in a file called DPROG.PCH. If you need a release version in parallel, you simply change the compilation command to:

```
CL /YcPROG.H /FpRPROG PROG.CPP
```

This command creates a separate precompilation of the header files up to and including PROG.H and stores it in RPROG.PCH.

You can also specify the /Fp option when using (/Yu) a precompiled header.

Use Precompiled Header Option (/Yu)

The "use precompiled header" option (/Yu) instructs the compiler to restore its state from a precompilation using a precompiled header file. The syntax of this option is:

/Yu[[*filename*]]

Using /Yu with a Filename

If a filename is specified, it must correspond to one of the header files included in the source file using an **#include** preprocessor directive. The compiler skips to the specified **#include** directive, then restores its state from the precompiled header file.

The precompiled header file has the same base name as the specified include file, and a .PCH extension. 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 restores its state 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 restores the state of the compiler from the precompiled header file specified in that 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.

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 compilation options). If the filename does not contain a path specification, the precompiled header file is assumed to be in the current directory.

Note that 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 )
```

The **hdrstop** pragma is ignored unless either the /Yu or /Yc compiler option is specified without a filename.

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 in any headers.

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
```

The precompilation facility saves the state of the current compilation.

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 the precompilation facility saves the state of the current compilation, not an arbitrary symbol set for a given header file. The correspondence is between source files and precompiled header files, not between header files and precompiled header files.

Note If neither the compilation option nor the **hdrstop** pragma specifies a filename, the base name of the source file is used.

A base file can contain as many as two **hdrstop** pragmas. You might do this when compiling with both /Yu and /Yc in order to build one .PCH file from another. In

such a case, the first **hdrstop** is associated with a use (/Yu) and the second with the creation (/Yc). If you're building one .PCH file from another, take care to avoid unwanted conflicts between .PCH filenames.

Including Debugging Information (/Yd)

The /Yd "debugging information" option allows you to override the default placement of CodeView information in object files. It is used with precompiled headers created with the /Zi option.

By default, any CodeView debugging information (symbols and type information) for a precompiled header is placed in the object file for which the precompiled header is created, rather than in the .PCH file itself. When you create other object files using this precompiled header, the new object files do not replicate the same debugging information. Instead, they simply cross-reference the debugging information contained in the first object file. By cross-referencing this information, rather than replicating it in multiple object files, you can save disk space and speed up the build process.

For example, say that you have two base files, F.CPP and G.CPP, each of which contains 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 /Zi 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 /Zi G.CPP
```

The object file G.OBJ does not include the debugging information for the precompiled header, but simply cross-references that information in the F.OBJ file. Note that you must link with the F.OBJ file if you choose to link with the /CODEVIEW option.

If your precompiled header was not compiled with /Zi, you can still use it in later compilations using /Zi. However, the debugging information will be placed in the current object file, and local symbols for functions defined in the precompiled header will not be available to CodeView.

The default behavior described above can be undesirable if you are distributing a debugging library. The /Yd option lets you override the default, in order to place complete debugging information in every object file. The syntax for this option is:

/Yd

When you create a precompiled header using /Yd, the .PCH file itself contains the debugging information. When you use a precompiled header using /Yd, the debugging information is replicated in the resulting object file.

2.4 Consistency Rules

When you use a precompiled header, the compiler assumes the same compilation environment that was in effect when you created the .PCH file, unless you specify otherwise. You should take care to specify a consistent environment (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

The following table lists compiler options that might trigger an inconsistency warning when using a precompiled header:

Table 2.1 Compilation Option Consistency

Option	Name	Rule
/A*X* or /A*xxx*	Memory model selection	Must be the same between the compilation that created the precompiled header and the current compilation. If these options differ, a warning message results.
/D	Define constant	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 these change.
/E or /EP	Send preprocessed output to standard output	Precompiled headers do not work with the /E or /EP options.

Table 2.1 *(continued)*

Option	Name	Rule
/Fr or /FR	Generate 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 cross-referenced by other files in the same manner as CodeView information; see "Including Debugging Information (/Yd)," on page 38. Unlike CodeView information, you cannot override the placement of Source Browser information.
/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.
/Zi	Generate CodeView information	If this option is in effect when the precompiled header is created, subsequent compilations that use the precompilation can use that CodeView 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 CodeView.

Note The precompiled header facility is not intended for use with files that are not C or C++ programs. Precompiled headers are not guaranteed to work with text files.

Include Path Consistency

A precompiled header 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

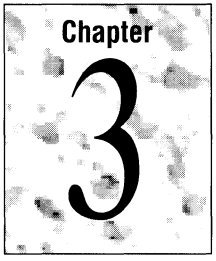
Pragmas processed during the compilation of a precompiled header normally affect the file in which the precompiled header is subsequently used. The following pragmas do not affect the remainder of the compilation:

comment	**message**	**skip**	**title**
linesize	**page**	**subtitle**	
listing	**pagesize**		

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	**data_seg**	**native_caller**	**same_seg**
auto_inline	**function**	**pack**	**segment**
code_seg	**intrinsic**		

Reducing Program Size with P-Code



Most compilers translate programs into machine code that your computer can execute directly. With Microsoft C/C++, you can compile your program into an alternate form of code called "p-code." P-code produces much smaller programs than machine code, but your machine cannot execute them directly. Instead, programs compiled into p-code are executed by a run-time interpreter, which is a small program incorporated into your executable (.EXE) file. As a result, p-code programs are slower than machine-code programs. They are meant to be used when size is more important than speed. You can also compile selected portions of your program into p-code and compile the rest into machine code to get a better combination of size and speed.

This chapter describes:

- How to compile your programs into p-code.
- How p-code works.
- How to optimize your p-code program.
- How to fine-tune p-code optimization.

For information about debugging code programs, see Chapter 10, "Special Topics," in the *Environment and Tools* manual.

3.1 Compiling Your Program into P-Code

You can take advantage of p-code without making any changes to your source code. You need only specify p-code as the target code when you compile your program from within the Programmer's Workbench or from the command line.

Selecting P-Code from Within the Programmer's WorkBench

You optimize with p-code from within the Programmer's WorkBench (PWB) integrated environment by using one of PWB's predefined project templates.

From the Options Menu, choose Project Templates, and then Set Project Template. In the Set Project Template dialog box, make sure that C or C++ is selected in the Run-Time Support list. Then move to the Project Template list and select either DOS P-Code EXE or Windows P-Code EXE.

The following section describes the optimizations implied when you choose to build a p-code executable.

Selecting P-Code from the Command Line

Specify /Oq as the first option on the command line. For example, to make MYPROG.C into the smallest executable file possible, type:

```
CL /Oq MYPROG.C
```

This command compiles MYPROG.C into p-code and, besides invoking LINK, invokes the Make P-Code (MPC) utility, which is required to produce a p-code executable file.

On average, a program compiled into p-code is about 40 percent smaller than a program compiled into machine code, excluding the size of the p-code interpreter. (For more information about the MPC utility, see "Controlling the P-Code Build Process" on page 53).

Compiling with the /Oq option or selecting p-code from within the Programmer's WorkBench implies the following optimizations:

- Turn P-Code Quoting On (/Of)
- Enable Frame Sorting (/Ov)
- Enable Post-Code Generation Optimization (/Oo)

If you want to compile only selected portions of your program into p-code, you can do so by making changes to your source code or by using the relevant compiler options. These procedures are described in "Fine-Tuning Your P-Code Program," on page 50.

In addition, if you want greater control over the compilation and linking process, you can execute MPC individually, instead of having CL invoke it.

3.2 The P-Code Model

Because p-code is designed to produce the smallest code possible, it uses several size-reduction techniques not found in ordinary assembly language. You don't have to understand how p-code works in order to save space in your programs, but

if you want to debug a program at the p-code level using the CodeView debugger, you need to understand the techniques p-code uses.

You should be familiar with the concepts of assembly-language programming before reading this section.

The P-Code Stack Machine

While machine language consists of instructions for the microprocessor in your computer, p-code consists of instructions for an imaginary processor that is simulated by the run-time interpreter. This imaginary processor is known as a "stack machine," because it uses a stack for almost all of its operations. In contrast, the microprocessor in your computer uses its registers for most operations, and uses its stack primarily for function calls.

The stack holds the operands used by the instructions. In assembly language, you usually specify a source and a destination for each instruction, indicating where the operands reside and where to place any result. For example:

```
ADD  AX, BX
```

With a stack machine, you usually don't need to specify a source or a destination. Each instruction pops its operands off the stack and pushes its result back onto the stack. For example, the p-code instruction

```
AddW
```

implies the following operations (using C-style pseudocode):

```
w2 = pop();           // get first operand from stack
w1 = pop();           // get second operand from stack
push(w1 + w2);        // place result on stack
```

Omitting the source and destination saves space and helps make p-code as compact as it is. Some p-code instructions do specify a source and destination if they are modifying the value of variables, which are not stored on the stack. However, most instructions use the stack for at least one of their arguments.

The stack can store items of different data types, including bytes, words, and longs. However, floating-point types (float, double, and long double) are stored on a separate stack, called the "coprocessor stack."

The stack replaces the need for the general purpose registers AX through DX. However, the DS, SS, CS, IP, SP, and BP registers can still be accessed by the stack machine.

In addition, there are two pseudoregisters, available only in p-code:

- The PQ register, used during "quoting" (see "Reducing Duplicate Code with Quoting" on this page)
- The temporary register, whose high and low words are accessible as TH and TL, respectively

Reducing Duplicate Code with Quoting

"Quoting" is a method of decreasing code size by avoiding duplication of code. It is similar to using function calls: your program contains only one instance of the code for a function, even though it may be used many times. The program calls the function instead of repeating the function code.

Quoting extends this technique to code that isn't defined as a function. The compiler examines the code that it generates, looking for places where a sequence of instructions is repeated. If it finds such repetitions, it replaces all but one of the occurrences with the equivalent of a function call, and makes the remaining occurrence the equivalent of the function body. Quoting differs from function calls in that there are no arguments and no return value; quoting changes only the path of execution. The low space overhead for quoting makes it worth using for sequences only a few bytes long.

Quoting is implemented by two instructions: the **QUOTE** instruction indicates the function call, and the **EQUOTE** instruction indicates the end of the function body. No label marks the beginning of the function body.

A **QUOTE** instruction takes a one- or two-byte offset as an argument. When a **QUOTE** is executed, it saves the address of the next instruction as a return address and performs a jump to the specified offset. The return address is not pushed onto the stack; it is stored in the PQ register instead. When an **EQUOTE** instruction is executed, it checks whether PQ contains an address. If not, **EQUOTE** does nothing; if it does, **EQUOTE** performs a jump back to that address. This allows the quoted section of code to be executed in two ways: in sequence with the preceding and following code or as a quote call.

The compiler does not nest quotes; that is, a quoted section of code itself cannot contain quote instructions. However, a quoted section may contain a procedure call, and the procedure call may execute a different section of quoted code. This is possible because the stack frame belonging to each p-code procedure has its own PQ register.

Quoting enabled (/Of) is the compiler default. However, quoting makes compiled p-code difficult to read and debug. Therefore, you should use the disable-quoting option (/Of–) during program development. See "Fine-Tuning Your P-Code Program" on page 50.

Entry Points to P-Code Functions

If your program contains both functions compiled into p-code and functions compiled into machine code, it is possible that a machine code function will call a p-code function. When this happens, your program must stop executing machine code and turn control over to the p-code interpreter, which then begins executing the p-code.

To enable this transition from machine code to p-code, a p-code function normally contains a "native entry point" at its beginning, which consists of a few machine code instructions. For example,

```
myfunc:                          // native entry point
    MOV  AX,DS                   // possible Windows preamble
    NOP
    CALL __PcodeCallFC
    DB __index

_PCODE_myfunc:                   // p-code entry point
    DW   ????
    LdcW1
    .
    .
    .
```

These are the instructions generated when a function called `myfunc()` is compiled into p-code. When `myfunc()` is called from a machine code function, those machine code instructions at the top are executed first. They transfer control to the p-code interpreter, which continues execution with the p-code instructions starting at the label `_PCODE_myfunc`. Note that the label `_PCODE_myfunc` is not actually generated by the compiler. It is specified here for illustration purposes only.

When a p-code function is called by another p-code function, the native entry point is bypassed and execution begins immediately with the first p-code instruction.

Instruction-Naming Convention

The names of p-code instructions have the following general form:

operation[[*mode*]][[*qualifier*]] *data type* [[*operand data type*]]

The *operation* portion of the instruction name indicates the operation performed by the instruction; for example, all instructions that begin with **Cmp** perform a comparison.

The optional *mode* specifies the addressing mode used by the instruction and can have one of the following values:

Mode	Meaning
n	Near
f	Far
no	Near + offset
fo	Far + offset

For example, the **LdifBb** instruction performs an indirect load of a byte using a far pointer. The "f" in the instruction specifies that the pointer is far.

The optional *qualifier* specifies the conventions used by the instruction and can have one of the following values:

Qualifier	Meaning
p	Preserve data on stack after instruction
t	Use temporary register
s	Signed or scaled
u	Unsigned
fc	Far C calling convention
nc	Near C calling convention
fp	Far Pascal calling convention
np	Near Pascal calling convention

For example, the **CmpsL** instruction compares two signed long integers. The "s" in the instruction specifies that the operands are signed.

The *data type* specifies the data type on which the operation is performed. It can have one of the following values:

Type	Meaning
V	Void
B	Byte
Q	Bit field
W	Word
S	Short (word—reserved for 386)
L	Long (double word)
N	Near pointer
A	Near address
F	Far pointer
H	Huge pointer

Type	Meaning
R	Float
D	Double
T	Long double

For example, the **CmpuW** instruction compares unsigned words, and the **CmpuL** instruction compares unsigned longs. An instruction can specify more than one data type, if it produces a result whose data type is different from its operand(s). For example, **CvtBW** converts a signed byte to a signed word, and **MulWWL** multiplies two words to form a long word.

The optional *operand data type* appears in instructions that require an additional operand be specified, rather than taking it from the stack. This field can indicate the size of the operand that follows the instruction, or it may specify the value itself so that no separate operand is needed.

Operand Data Type	Meaning
0–9, m1	operand encoded in instruction, where m1 = –1
b	byte
w	word
l	long

For example, consider a statement using the Jump-on-Not-Equal instruction:

```
JneWb 05
```

This statement pops two words off the stack and compares them. If they are not equal, it performs a jump of length 5. The "b" in the instruction indicates that it requires an explicit one-byte operand.

To save space, many p-code instructions have alternate forms that assume a particular value for an operand. For example,

```
JneW5
```

is a single-byte instruction that conditionally performs a jump of length 5. There can be a number of **JneW***n* instructions, each with a different jump length encoded. This saves the space required by a separate operand.

Not all instructions have encoded operands, and not all possible values are encoded into the instructions that do have them. Only the most common values are encoded; other values must appear as a separate operand with the version of the instruction ending in "b," "w," or "l." Each p-code instruction is described fully in Help.

Note The p-code instruction set may change in future releases of the compiler.

3.3 Fine-Tuning Your P-Code Program

"Compiling Your Program into P-Code" on page 43 describes the simplest way to use p-code in your development process. To gain a better combination of size and speed, however, you can also modify your source code before compilation or specify additional options when compiling your program into p-code.

Mixing P-Code and Machine Language

Since p-code runs slower than machine language, you may want the speed-critical sections of your program to be compiled into machine language. If you use the /Oq option, you can specify what portions of your program are compiled into p-code with the **optimize** pragma for the **q** option. The pragma works on a function-by-function basis, and can only appear outside of function scope. For example,

```
#pragma optimize("q",on)

// Functions compiled into p-code

#pragma optimize("q",off)

// Functions compiled into machine code
```

These pragmas are ignored if the program is not compiled with the /Oq option.

The functions that are the best candidates for p-code compilation are those that interact directly with the user. The execution speed of such functions is limited by the speed of the user of the program, rather than the speed of the computer. Also suitable for p-code are rarely used operations and sections of code associated with error conditions. These functions can usually be converted into p-code without the program appearing noticeably slower to the user.

A profiler can help fine-tune p-code programs.

To further assist you in determining which functions should be compiled into p-code and which should remain in machine language, you can use a "profiler." A profiler is a utility that monitors the execution of your program and lists how much time the program spent in each of its functions. This does not necessarily correspond to the frequency with which the functions are called; for example, a function may be called only once during a program, but it may take up most of the program's execution time.

You should run the profiler on your program before you compile any of it into p-code. The functions that are the most time-consuming should remain in machine language. The functions that take the least of your program's time can be converted into p-code to give you space savings with a minimal reduction in execution speed. As a final step in tuning, the profiler can be run on the mixed p-code and native program.

Removing Native Entry Points

"Entry Points to P-Code Functions" on page 47 describes the entry points that allow p-code functions to be called from functions compiled into machine language. The native entry point of a p-code function is, on average, six bytes long. If you have p-code functions that are called only by other p-code functions, you can omit those entry points and save those bytes by using the /Gn compiler option.

You can also control the removal of native-code entry points from within your source code by using the **native_caller** pragma. The **native_caller** pragma takes an on/off argument and works on a function-by-function basis. To turn off native entry-point code generation for a p-code function, enter

```
#pragma native_caller (off)
```

above the definitions of those p-code functions that are only called from other p-code functions. After the definitions of those functions, turn the **native_caller** pragma back on by entering

```
#pragma native_caller (on)
```

To reset the **native_caller** pragma to the value entered on the command line, include the pragma without a parameter:

```
#pragma native_caller ()
```

When optimizing native entry sequences, bear in mind that all p-code functions that are exported, called via function pointers, or defined as **_loadds** require native entry points. This includes Windows call-backs.

Warning! If you execute a program where a machine-language function calls a p-code function without a native entry point, your program will terminate abnormally. In most cases, the Make P-Code utility (MPC) can detect when a p-code function is missing a required native entry code sequence.

Specifying Entry Tables

With the /Gp compiler option, you can specify the maximum number of entry tables for your program. Like the other options for fine-tuning p-code discussed in this section, the /Gp option must be used in conjunction with the /Oq option.

An entry table is needed for every segment that contains a p-code function or a function called by a p-code function. One entry table can describe up to 256 such functions. If a segment contains more than that, the Make P-Code utility (MPC) creates additional entry tables.

If you do not specify the /Gp option, *n* is assigned the default value of 255. In addition to the space that the actual entry tables take up, there is a four-byte overhead for each possible entry table.

Note The MPC utility is invoked automatically when you specify the /Oq option on the CL command line.

Specify /Gp*n* when you compile your source file. When the MPC utility processes the resulting .EXE file, it creates up to *n* entry tables. MPC returns an error if the program needs more than *n* entry tables.

Turning P-Code Quoting On and Off

Quoting, the p-code optimization technique that eliminates duplicate sections of code, can be controlled with the /Of compiler option. (See "Reducing Duplicate Code with Quoting" on page 46.)

Quoting enabled (/Of) is the default. However, quoting makes compiled p-code difficult to read and debug. Therefore, you should use the disable-quoting option (/Of–) during program development. Then you can turn quoting back on (/Of) to produce a smaller, fully optimized release version of your program.

Controlling Frame Sorting

You can control a p-code size optimization that changes the order in which local variables are allocated on the stack.

The compiler reduces the size of p-code programs by using one-byte opcodes to reference certain local variables. These opcodes can reference a limited number of the local variables in each function.

You can control which variables receive the available opcodes by using the /Ov option as follows:

Option	Description
/Ov	Sorts the local variables by frequency of use (default)
/Ov–	Sorts the local variables in the order they occur (lexical order)

In most cases, the default value for this option is satisfactory. However, if you have a large function that exceeds the capacity of the p-code optimizer, you can rewrite the function so that the most frequently used local variables are at the beginning of the stack frame. Then use the second option above (/Ov–) to ensure the maximum size reduction.

3.4 Controlling the P-Code Build Process

When you compile a program into machine code from the command line, CL performs the compilation and then calls LINK to do the linking. When you use the /Oq option to compile a program into p-code, CL calls one other program in addition to LINK: the Make P-Code utility (MPC).

CL calls the MPC utility after calling LINK. MPC reads the executable (.EXE) file produced by LINK and generates several internal tables needed by the run-time interpreter. Once MPC has added these tables, the executable file is ready to run.

MPC requires a segmented executable file as input, even if DOS is the target. P-code object modules contain special-purpose records that force the generation of a segmented executable.

If you want to separate the compilation phase from the link and post-link phases, specify the /c option in addition to the /Oq option. This option tells CL to stop after the compilation step. You can run LINK and MPC in one step by specifying the /PCODE option for LINK. For example, the command

```
LINK /PCODE MYPROG.OBJ
```

links MYPROG.OBJ and runs MPC on the resulting executable MYPROG.EXE.

Invoking MPC by itself gives more control over the build process.

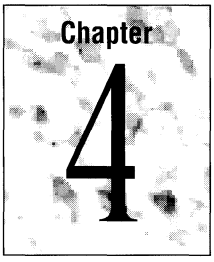
You can also invoke invoke MPC individually. If you don't specify the /PCODE option, LINK performs only the standard linking procedure and does not call any additional programs. However, if the .OBJ file contains p-code, LINK cannot produce a file that can be executed. MPC is required to make a p-code program executable.

Use the MPC program to convert the LINK output into an .EXE file that you can run. Specify a name for MPC's output file using the /Fe option. For example:

```
MPC /Fe MYPROG.EXE MYPROG.PXE
```

This command reads MYPROG.PXE (linker output) and produces a file named MYPROG.EXE. If you don't specify the /Fe option, MPC uses the name of the input file.

Managing Memory in C



When you develop advanced 16-bit applications with Microsoft C/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 will make 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 topics described in this chapter that apply to 32-bit programs are pointers based on a pointer.

4.1 Pointer Sizes

One of the strengths of the C language is that it allows you to use pointers to directly access memory locations.

Every Microsoft 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, or 80386 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 allows you 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 has additional segment registers: FS and GS.)

Near Pointers

If you don't explicitly specify a memory model, Microsoft C/C++ defaults to the small model, which allots up to 64K for the code and another 64K for the data (see Figure 4.1).

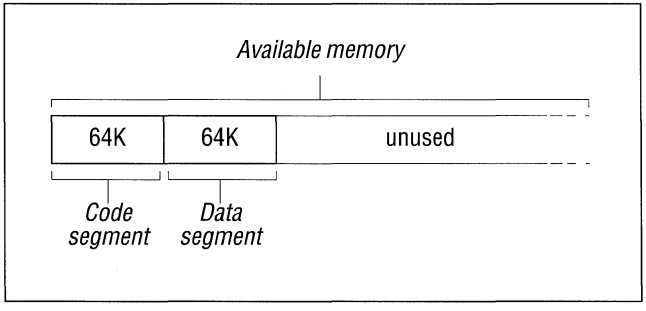


Figure 4.1 Anatomy of a Small-Model Program

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 DOS. The disadvantages of the larger far pointers is 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 Microsoft C/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, Microsoft C/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 this code fragment,

```
int __huge *hp;
int __far *fp;
.
.
.
hp++;
fp++;
```

both `hp` and `fp` are incremented. 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 C/C++ 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. They are explained in more detail in "Using Based Pointers and Data" on page 78.

4.2 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 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**, so they are best for writing code that is portable to other (non-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 C/C++ memory models are shown in Table 4.1.

Table 4.1 Memory Models

	Maximum Total Memory		
Model	**Code**	**Data**	**Data Arrays**
Tiny	<64K	<64K	<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 Microsoft C/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.

The 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.
- The tiny model applies to DOS only; it is not available in Windows.

Although the tiny model imposes the most severe limits on code and data size, it produces the smallest programs. The tiny memory model only offers a load-time speed advantage over the small model; they both produce the fastest programs.

The 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, would be 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 ANSI draft standard for C defines the value returned by **sizeof** to be of type **size_t** (which, in Microsoft C, 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 Microsoft C). Subtracting two huge pointers will yield a **long** value. Microsoft C 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: one is 16 bits wide and the other is 32 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 with different sizes of code and data pointers.

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 example above works correctly in tiny, small, large, and huge models because, in those models, a data pointer is the same size as a code pointer. Under 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, Microsoft C/C++ defaults to the small model, which is adequate for many small to midsized programs.

You can select a memory model from the Programmer's WorkBench (PWB) or from the command line.

Selecting from Within PWB

If you're compiling from the Programmer's WorkBench, pull down the Options menu and open the Language Options submenu. From that menu, open either the C or C++ Compiler Options dialog box, and select a memory model there. Choose one of the standard models or choose Customized and select the options for a customized model.

Selecting from the Command Line

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: Library
/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

4.3 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 could solve the problem by moving up from the small to the compact memory model. Doing so would bump all data pointers from two to four bytes. The .EXE file would grow accordingly. Execution time would slow.

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.

Microsoft C/C++ lets you 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, as well as with a single underscore. Since 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, but

throughout the rest of the program, all addressing would be near. Note that all pointers to elements of an array declared as **__far** must also be declared as **__far**.

Table 4.2 lists the effects of these keywords on data pointers, code pointers, and pointer arithmetic.

Table 4.2 Addressing Declared with Microsoft Keywords

Keyword	Data	Code	Arithmetic
__near	Data reside in default data segment; 16-bit addresses	Functions reside in current code segment; 16-bit addresses	16 bits
__far	Data can be anywhere in memory, not necessarily in the default data segment; 32-bit addresses	Functions can be called from anywhere in memory; 32-bit addresses	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; 32-bit addresses	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; 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 will be incompatible with standard library functions. Watch for these problems that affect pointers:

- A library function that expects a 16-bit pointer as an argument will 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 will return 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. Microsoft C/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 will always be 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 will be 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 will hold 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` will affect 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 will be 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 can be used in combination with the **__near** and **__far** keywords. Based functions are described in "Using Based Addressing for Functions" on page 88. A function can return a based pointer unless it is a pointer based on **__self** (see "Using Based Pointers and Data" on page 78).

The example below 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 example below 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 will convert 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`. Since `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. In this case the program does not work as intended, however, since 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 shown below:

```
/* 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,
                             * no matter which memory
                             * model is used
                             */
}

int test_fun( int __near *ptr1, char __far *ptr2, int a)
{
    printf( "Value of a = %d\n", a );
}
```

4.4 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/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.

The /A*string* option lets you 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 below 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. This model would allow huge data items but only one code segment. The option for specifying this model would be /Ashd.

Note Tiny model is identical to 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.

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*xnx*	Near data pointers
/A*xfx*	Far data pointers
/A*xhx*	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) 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 reloaded on function entry
/A[[*xx*]]w	SS != DS; DS 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, initialized global and static data are placed in the default data segment up to a certain threshold. 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 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 was 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. The /ND option lets you 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.

The /Au option is useful for Microsoft Windows dynamic-link libraries (DLLs), since it forces DS to be loaded on entry to each function. It is also useful for writing extensions to the Programmer's WorkBench. This is a costly operation, however, so consider using the /Aw option.

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 under 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 Windows DLLs with /Aw.

The /Aw option is useful for writing Windows dynamic-link libraries (DLLs), 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 like 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. Since it cannot be called from outside of the module, `HelperFunc` does not need to reload DS.

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 will assume the pointer is

relative to the data segment. However, the receiving function could solve this problem by declaring the pointer to be the following:

```
based(__segname("_STACK"))
```

Another solution would be 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 4.3 shows the libraries from which to extract the startup routine for each customized memory model.

Table 4.3 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 (see Section 3.1 of the *C Language Reference* for more information).

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.

Option	Effect
/Gt[[*number*]]	Sets the threshold

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** keywords.

Referencing Declarations

Referencing declarations include data items declared **extern** and uninitialized, non-static 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.

By default in the compact, large, and huge memory models, the compiler makes no assumptions about where the linker places those data items. All references to those data items are done with far addressing, in case they are placed in a far segment. If you can guarantee that your uninitialized and **extern** data items reside in the default data segment, you can use the /Gx option to have them referenced 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 will produce unpredictable results.

This option is also useful for writing compact-, large-, and huge-model Windows applications. If you want to run multiple instances of your Windows program simultaneously, you cannot use far addressing with your global data. If your global data resides in the program's default data segment, you can use the /Gx option to reference it as near. This allows you to run multiple instances of the program simultaneously.

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 switch, 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

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 C/C++ 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 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 normally 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 4.4 summarizes the naming conventions for code and data segments.

Table 4.4 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 text) and /ND (name data).

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" will 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

The **alloc_text** pragma lets you 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, **alloc_text** allows you 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.

4.5 Using Based Pointers and Data

Microsoft C/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 the next section.

Based Pointers

Based pointers combine the advantages of near and far pointers. Based pointers are two 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 they always use the DS register for their segment value, so 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. See "Data Stored in a Named Segment" on page 86 for more information.

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 that `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` will refer 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` will be 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` were assigned a new value, `vidptr` would act as an offset from that new value and evaluate 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. Microsoft C/C++ lets you define a special heap that resides in a segment. You can use such a based heap to allocate objects dynamically, just as you would 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 1,000 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 four bytes.

If you 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 given above:

```
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 (**:>**). That is,

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'; // do it again
```

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 81 describes dynamic allocation of based data using the run-time library functions. Microsoft C/C++ also allows you to 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** would treat `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 four 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 names ends in _TEXT, the compiler marks that segment as a code segment, making it a read-only segment.

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. This lets you 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.

4.6 Using Based Addressing for Functions

With Microsoft C/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 normally 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 were not declared as based in the same segment, they would be placed in separate segments because they're declared in separate files. In that situation, this program would suffer a link-time or run-time error because `farfunc` cannot perform a near call to `nearfunc` when `nearfunc` is in another segment. However, since both functions are based in the MYSEG segment, the program links and executes 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.

4.7 Using the Virtual Memory Manager

Virtual memory is a facility for accessing storage beyond the 640K of memory available to DOS. Microsoft C/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 only available for 16-bit DOS programs. Windows programs and 32-bit programs do not need to use these functions.

The virtual memory manager works by copying blocks of virtual memory into 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 **_vheapinit**
- Allocate virtual memory blocks as needed by calling **_vmalloc**
- Load or lock virtual memory blocks into the DOS address space in order to access their contents, using **_vload** and **_vlock**
- Unlock virtual memory blocks when they're not being accessed by calling **_vunlock**
- Free virtual memory blocks when they're no longer needed by calling **_vfree**
- Terminate the virtual memory manager by calling **_vheapterm**

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 DOS memory that must be available for the virtual memory manager to be installed (in 16-byte paragraphs), the maximum amount of DOS memory that it can use (in paragraphs), and 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 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 flags **_VM_EMS**, **_VM_XMS**, or **_VM_DISK**. 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 DOS memory.

Loading Blocks

The **_vload** function takes a handle and copies the associated block of virtual memory into 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 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. This flag tells the virtual memory manager what to do when it needs the region of 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 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 DOS memory, and returns a far pointer to it. However, **_vlock** locks a block of memory so that it remains in DOS memory even if you make

subsequent calls to the virtual memory manager. A locked block remains in 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 **_vlockcnt**.

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 DOS memory. If you do so, the virtual memory manager may relocate the block within 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 the return value of these functions before using it as a pointer.

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 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 of 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 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 we just allocated
        return 0;           //     but could not load
    }

    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, 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**, since 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 `search` 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 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**, since 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 their 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 as well as pointers.

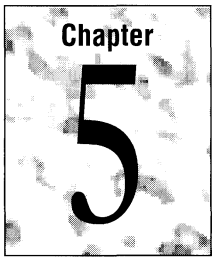
Another possible technique is to maintain an array of handles, each of which refers to a large buffer of virtual memory. Your program can switch among these buffers, keeping just one or two of them in DOS memory at any given time.

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. While 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. If your program reads and writes a large amount of data at all times, it will require the virtual memory manager to perform a lot of swapping, so its performance will not be as efficient as one which accesses only a small amount of data at a time.

Another way to make your program use extended memory, expanded memory, and disk is to break it up into overlays. For information on creating overlays, see Chapter 15, "Creating Overlaid DOS Programs," of the *Environment and Tools* manual.

Managing Memory in C++



Chapter 4, "Managing Memory in C," 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 (see the *C++ Tutorial* and the *C++ Language Reference* manuals for information on the C++ language). You should also have read the material in Chapter 4, "Managing Memory in C."

The material covered in this chapter is relevant only to 16-bit programs.

5.1 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 the 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, this statement in C++

```
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 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** keywords in a class's declaration, the class inherits the ambient model of its base class(es). If the class has no base class(es), 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** keywords 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 `print_all()` 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 switch 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.

5.2 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 Microsoft 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

Microsoft 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 tiny, 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 like an array allocator.

The **new** operator for huge objects behaves like an array allocator, 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 2.

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 would use the following prototype:

```
void __based(void) *operator new( __segment segvar, size_t size, short
place );
```

This prototype permits expressions like

```
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

Microsoft C++ allows you to 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** functions, 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.

5.3 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 as 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, since 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.

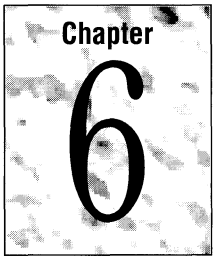
The easiest way to avoid this problem is to declare all your virtual functions as **__far**, so they can be called from any segment. However, if you want to declare your virtual functions as **__near**, you can avoid problems by declaring the virtual function to be based in the same segment as `test`:

```
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 will be 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.

Using the Inline Assembler



This chapter explains how to use the Microsoft C/C++ inline assembler. Assembly language serves many purposes, such as improving program speed, reducing memory needs, and controlling hardware. The inline assembler lets you 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 11, "Programming with Mixed Languages."

6.1 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 DOS and BIOS routines with the **INT** instruction
- Creating TSR (terminate-and-stay-resident) 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.

6.2 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.

Below is a simple **__asm** block enclosed in braces. (The code prints the "beep" character, ASCII 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
```

Since 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, since 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.

6.3 Using Assembly Language in __asm Blocks

The inline assembler has much in common with other assemblers. For example, it accepts any expression that is legal in MASM. This section describes the use of assembly-language features in **__asm** blocks.

Instruction Set

The 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. The inline assembler for CL3232.EXE recognizes 80386- and 80387-specific instructions, so those instructions are available when you write a 32-bit program; the inline assemblers for the other compilers do not recognize those instructions.

Expressions

Inline assembly code can use any MASM expression, which is any combination of operands and operators that evaluates to a single value or address.

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** or **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

While 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 eight-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. See "Using C and C++ in **__asm** Blocks," on page 115 for more information.

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 the value 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 123). An **__asm** block can also contain C-style comments, as noted below.

The _emit Pseudoinstruction

The **_emit** pseudoinstruction is similar to the **DB** directive of MASM. It allows you to 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 instructions, which the inline assembler does not support. The following fragment, 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 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. In source mode, the CodeView debugger lets you 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.

See the *Environment and Tools* manual for more information.

6.4 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 would be 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). This allows you to 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 2 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.

Since 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 allows you to access stack arguments and local stack variables by name.)

The POWER2.C program below 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. Since C automatically preserves BP, the **__asm** block doesn't need to do so. It can also dispense with the **RET** instruction, since 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 2 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.

6.5 Using and Preserving Registers

In general, you should not assume that a register will have 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, since 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. In addition, you must preserve the CX register in any function declared with **__fastcall**.

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 **__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 the **__fastcall** attribute.

As you may have noticed in the POWER2.C example in "Writing Functions" on page 118, 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 118 also shows that functions return values in registers. This is true for return values that are four 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

124). You should preserve the SP and BP registers unless you have some reason to change them—to switch stacks, for example.

6.6 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
jne 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:
```

6.7 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
   }
}
```

Since function arguments are passed on the stack, you simply push the needed arguments—string pointers, in the example above—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**.

6.8 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, since all the standard header files declare the library functions to have **extern "C"** linkage.

6.9 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. Since a block defined as a macro is one logical line, you must separate each instruction 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. (See "Using and Preserving Registers" on page 120.)

You can convert MASM macros to C macros.

Note that some MASM-style macros can be written as C macros. Below 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.

6.10 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.

Since 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

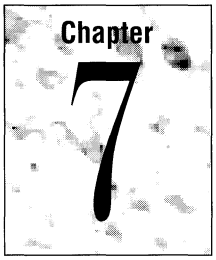
```
mov v1, v2
```

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.

Controlling Floating-Point Math Operations



This chapter describes how to control the way your Microsoft C/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 in the Programmer's WorkBench (PWB) or 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.

7.1 Declaring Floating-Point Types

Microsoft C/C++ supports three floating-point types that conform to the Institute of Electrical and Electronics Engineers (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 7.1 shows the relationship between significance and storage requirements.

Table 7.1 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 7.2 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 7.2 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 1,023; 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 1 and less than 2. 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 zero. By using 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 7.3 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 normal form; numbers can only be represented in denormal form when stored in 32- or 64-bit floating-point variables (type **float** and type **long**).

Table 7.3 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**.

Microsoft C/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 example below (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 NDP stack (or numeric data processor 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.

7.2 Run-Time Library Support of Type long double

Of the math packages offered by the Microsoft C/C++ compiler, 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.

7.3 Summary of Math Packages

The Microsoft C/C++ compiler offers 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 Microsoft C/C++, the SETUP program allows you to build combined libraries. These libraries 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 PWB or 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 *Getting Started* manual.

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**, **tan**) differ slightly from the corresponding functions performed on a coprocessor. This difference can cause a slight discrepancy (usually within two bits) between the results of these operations when performed with the software emulation instead of 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. See the discussion of the **_control87** function in Help for more information about 80*x*87 floating-point exceptions.

Math Coprocessor Package

The math coprocessor package utilizes 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 weaker error checking. The alternate math package does not support infinities, NANs ("not a number"), and denormal numbers.

7.4 Selecting Floating-Point Options (/FP)

You can select a floating-point library and the method of accessing floating-point routines by setting options in PWB or by specifying command-line options to CL. You can choose between the emulator, alternate, or math coprocessor library. 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 smallest and the fastest floating-point math option is the inline math coprocessor package because the compiler generates true 80*x*87 coprocessor instructions. If, however, you cannot depend on the target computer having a coprocessor, you must use either the emulator or alternate math options.

To specify floating-point options on the CL command line, you must specify an option from the list in Table 7.4. 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. See Table 7.4 and "Library Considerations for Floating-Point Options," on page 137 for more information about linking with different libraries.

Table 7.4 summarizes the floating-point options and their effects. These options are described in detail in the following sections.

Table 7.4 Summary of Floating-Point Options

Option for CL for PWB	Combined Use of Method	Effect	Coprocessor	Libraries Selected
/FPi Inline Emulation	Inline	Default; larger than /FPi87, but can work without a coprocessor; most efficient way to get maximum precision without a coprocessor	Uses coprocessor if present[1]	*m*LIBCE.LIB[2]
/FPi87 Inline Math Coprocessor	Inline	Smallest and fastest option available with a coprocessor	Requires coprocessor	*m*LIBC7.LIB
/FPc Calls to Emulator	Calls	Slower than /FPi, but allows use of alternate math library at link time	Uses coprocessor if present[1]	*m*LIBCE.LIB[2,3]
/FPc87 Calls to Math Coprocessor	Calls	Slower than /FPi87, but allows use of alternate math library at link time	Requires coprocessor unless library changed at link time[5]	*m*LIBC7.LIB[3,4]
/FPa Alternate Math	Calls	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 NO87.

[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 1, "Optimizing Your Programs."

You can specify floating-point options in the Programmer's WorkBench.

To specify floating-point options when using the Programmer's WorkBench, pull down the Language Options submenu of the Options menu. From that menu, open the C or C++ Compiler Options dialog box. From that dialog box, open the Additional Release Options dialog box and select one of the following floating-point math options:

Option	Effect
Emulation Calls	Generates calls; makes emulator math library the default (/FPc)
80*x*87 Calls	Generates calls; makes math coprocessor library the default (/FPc87)
Fast Alternate Math	Generates calls; makes alternate math library the default (/FPa)
Inline Emulation	Generates inline instructions; makes emulator math library the default (/FPi); this is the default option
Inline 80*x*87 Instructions	Generates inline instructions; selects math coprocessor library (/FPi87)

Inline Emulator Option (/FPi)

The inline emulator option (/FPi) 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 inline emulator option by default.

The inline emulator option is useful if you cannot be sure that an 80*x*87 coprocessor will be available on the target computer. Programs compiled using the inline emulator option work as described below:

- 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 inline emulator option offers the most efficient way to get maximum precision in floating-point results.

When you use the inline emulator option, the compiler does not generate inline 80*x*87 instructions. Instead, the compiler 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 inline math coprocessor option (/FPi87).

Inline Math Coprocessor Instructions Option (/FPi87)

The inline math coprocessor instructions option (/FPi87) 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 inline math coprocessor instructions 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 inline math coprocessor instructions option results in the smallest, fastest programs possible for handling floating-point results.

Calls to Emulator Option (/FPc)

The calls to emulator option (/FPc) 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 inline emulator option.

Using the calls to emulator 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 Microsoft C/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, selected the Generate Intrinsic Functions option from the Optimizations dialog box in the Programmer's WorkBench, or specified math functions in an **intrinsic** pragma).

Calls to Math Coprocessor Option (/FPc87)

The calls to math coprocessor option (/FPc87) generates function calls to routines in the math coprocessor library (*m*LIBC7.LIB) that issue the corresponding 80*x*87 instructions. As with the inline math coprocessor instructions option (/FPi87), at link time you can choose to link with an emulator library (*m*LIBCE.LIB). However, /FPc offers more flexibility in choosing libraries, since you can change your mind and link with the appropriate alternate math library as well (*m*LIBCA.LIB).

The disadvantages of using the calls to math coprocessor option as opposed to the inline coprocessor option are as follows:

- Your executable 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, selected the Generate Intrinsic Functions option from the Optimizations dialog box in the Programmer's WorkBench, or specified math functions in an **intrinsic** pragma).

You must have a math coprocessor installed to run programs compiled with the /FPc 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, since arithmetic of different precision can result, there may be slight differences in your results.

Use Alternate Math Option (/FPa)

The use alternate math option (/FPa) 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, selected the Generate Intrinsic Functions option from the Optimizations dialog box in the Programmer's WorkBench, or specified math functions in an **intrinsic** pragma).

7.5 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. You can control which library is used in one of two ways:

- In the Programmer's WorkBench, pull down the Project menu, open the Edit Project dialog box, and add the name of the C run-time library file to the project. Then pull down the Options menu, open the LINK Options dialog box, and select the No Default Library Search option.
- From the LINK command line, give the /NODEFAULTLIBRARYSEARCH (/NOD) option and then specify the name of the combined library file you want to use in the *link-libinfo* field of the CL command line. This 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 inline math coprocessor instructions (/FPi87) or inline emulator (/FPi) option. Otherwise, compile for floating-point function calls using the calls to math coprocessor (/FPc87), calls to emulator (/FPc), or alternate math (/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.

7.6 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 calls to emulator option (/FPc) 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 example below compiles the program `CALC.C` with the medium-model option (/AM). Because no floating-point option is specified, the default inline emulator option (/FPi) is used. The inline emulator 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 example below compiles `CALC.C` using the small (default) memory model and the alternate math option (/FPa). 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 to the alternate math library.

```
CL /FPa CALC.C /link SLIBCE /NOD
```

The example below compiles `CALC.C` with the calls to math coprocessor option (/FPc87), 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
```

7.7 Using the NO87 Environment Variable

Programs compiled using either the calls to emulator (/FPc) or the inline emulator (/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.

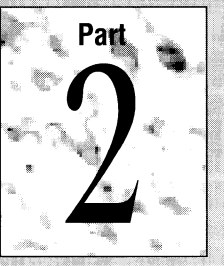
The NO87 variable takes effect with any program linked with an emulator library (*m*LIBCE.LIB). It has no effect on programs linked with math coprocessor libraries (*m*LIBC7.LIB) or on programs linked with alternate math libraries (*m*LIBCA.LIB).

7.8 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 PC, 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 will run 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 Microsoft C/C++. Any machine that sends the 80*x*87 exception to an 8259 Priority Interrupt Controller (master or master/slave) 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.

Special Environments

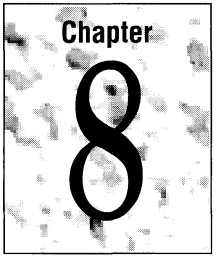


Special Environments

Microsoft C/C++ provides a platform from which you can build graphics applications and interface with programs written in other languages.

Chapter 8 describes the QuickWin library, which lets you easily convert simple DOS programs into Windows programs. Chapter 9 discusses the basic graphics functions to set video modes, draw basic shapes, and use graphic fonts. Chapter 10 describes the "presentation graphics" functions, which let you create sophisticated charts that show data relationships. Chapter 11 explains how to interface your C and C++ programs with routines written in other languages. Chapter 12 describes the portability of Microsoft C to other environments.

Compiling with the QuickWin Windows Library



QuickWin is a set of libraries that helps you turn non-Windows DOS programs into simple Windows applications.

Using QuickWin, many programs written for DOS can be compiled with Microsoft C/C++ to run in a Windows text-only window. A QuickWin window behaves like the DOS character-mode display. You can write to the window and receive input through it with standard C input and output routines, such as **printf** and **scanf**, or standard C++ iostream facilities, such as **cout** and **cin**.

Note To work with QuickWin, DOS programs must meet certain qualifications. Generally, your DOS program can be linked with the QuickWin libraries as long as it does not use graphics, Dynamic Data Exchange (DDE), serial port I/O, or cursor positioning and as long as it does not spawn processes.

QuickWin makes it easy to add Windows functions to DOS programs.

QuickWin makes it easy for you to add a simple subset of Windows functions to your DOS programs without having a detailed knowledge of Windows programming. Note that QuickWin offers only a portion of Windows capability. You cannot write a complete Windows application using QuickWin because you cannot call Windows application programming interface (API) functions from your QuickWin program. You can, however, add a Windows flavor to your applications, especially if you use the enhanced QuickWin features explained later in this chapter.

QuickWin is also useful for experienced Windows programmers. When you have a simple non-Windows program that you'd like to see in a window without completely overhauling the application, use QuickWin.

Additionally, QuickWin applications have access to all of the Windows address space and can share data with other Windows applications.

This chapter explains the user interface and the programming features provided by QuickWin and how to use them to build your own QuickWin applications.

8.1 What a QuickWin Program Provides

This section explains how to use QuickWin, the QuickWin user interface, standard and enhanced features, and the extent of Windows functionality that QuickWin provides.

Using QuickWin

You can use QuickWin in two ways.

Simple QuickWin Programs

The simplest way to use QuickWin is to link your DOS application with the QuickWin libraries without altering your source code. Your program then has the standard QuickWin user interface features described in "The QuickWin User Interface" section on page 147. Your simple QuickWin program:

- Runs in the Windows environment, in a window.
- Can be minimized or maximized, like any Windows 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 Windows Clipboard by providing Copy and Paste commands.
- Provides Help for the QuickWin features.
- Takes advantage of the virtual memory management capabilities of Windows, overriding the DOS 640K size limitation.

Enhanced QuickWin Programs

You can use QuickWin to take advantage of more Windows features (although not the functions in the Windows API). To use these enhanced features, you must alter your source code. You can:

- 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 window's text 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 currently active window (said to have the "input focus").

- Add an About dialog box customized with your text.
- Simulate mouse clicks in some of the QuickWin menus.
- Yield processing time to other Windows applications.
- Add custom application and document icons to your program.

The QuickWin User Interface

When a QuickWin program runs, it displays a Windows-style client window (also called an application window) titled with the program's name. The window has standard Windows controls, 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 standard menus; the status bar provides status information to the user. Within the client window is a child window titled "Stdin/Stdout/Stderr," which displays the standard C or C++ input/output streams. The child window also has controls and may have one or more scroll bars. QuickWin windows are text-only; text is black on white. Figure 8.1 shows the standard QuickWin user interface.

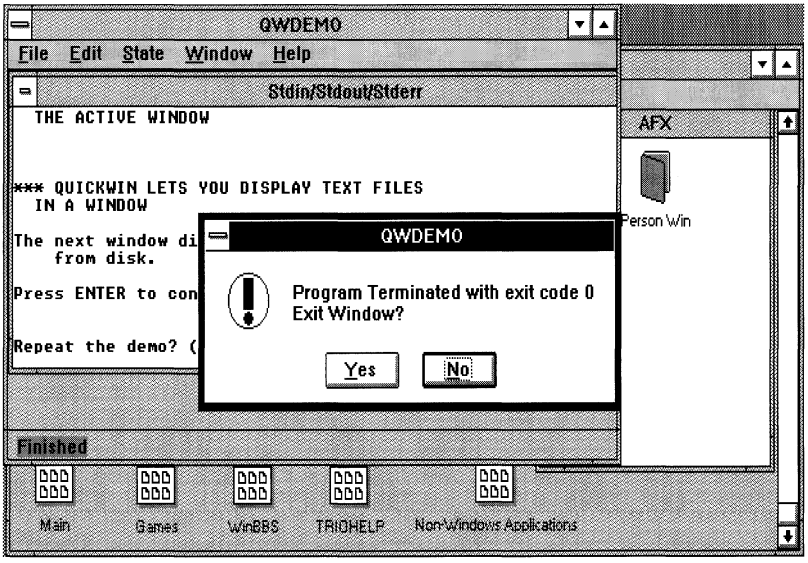


Figure 8.1 Standard QuickWin User Interface

QuickWin programs that take advantage of the enhanced features can display more than one child window. Among other things, the program can control the

size and position of windows and which window is currently the active window (the foremost window).

Standard Menus

When you run your QuickWin program, the client window always contains a standard QuickWin menu bar (you cannot add your own commands to the menus). The menu bar contains File, Edit, State, Window, and Help menus.

File Menu The QuickWin File menu has one command, Exit, which ends the program, closing all windows. If you exit the program in any other manner, such as by executing an **exit** function, three possible events can occur. By default, any of your program's windows that still exist remain on the screen.

You can call the **_wsetexit** function in your program to alter that default. You can specify that windows will remain on the screen (as in the default), that windows do not remain on the screen, or that the user can choose whether windows remain on the screen. If you specify user choice, a dialog box appears with the message "Program terminated with exit code *n*; Exit window?" Responding "No" to this dialog box allows the user to quit without closing windows. Figure 8.1 shows the dialog box's typical appearance. The user can examine window contents or select and copy text to the Clipboard, but further input or output is disabled. Exiting with the Exit menu command does not bring up a dialog box. For more information about the **_wsetexit** function, see "Writing Enhanced QuickWin Programs" on page 157.

Edit Menu The QuickWin Edit menu has commands for selecting, copying, and pasting text within or between windows or between applications.

- The Mark command puts the active window in Mark mode, ready for you to select text for copying to the Clipboard. The string "Mark – " is prefixed to the window title.

  You can select text with the keyboard or the mouse. Selecting text in a QuickWin window requires an extra step not required in selecting text in a standard Windows program. To select with the keyboard, you must first choose the Mark command. Then you can use the arrow keys to move the cursor from the upper left corner of the window to any corner of the text area that you want to select. 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 out a rectangle outlining the selection. For mouse selection, choosing Mark is unnecessary, but you may choose Mark and then select with the mouse. If you select with the mouse, the string "Select – " is prefixed to the window title instead of "Mark – ".

  Beginning a selection either with the keyboard or with the mouse pauses the program. The Pause command in the State menu is checked, the program does

not accept input, and processing time is yielded to other Windows applications. 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. The Resume command in the State menu is checked, 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 to the Clipboard.

- The Copy Tabs command copies the currently selected text to the Clipboard in CF_TEXT format: its characters are taken from the ANSI character set, each line ends with a carriage return and line-feed, and a null character terminates the block of text. Before the text is placed in 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.
- The Select All command selects and highlights all text in the active window. Using Select All is equivalent to selecting all of 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.

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 applications can run without competition for resources from the QuickWin program. The Resume command lets the program resume execution and removes any highlighting. The command you select, 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 Windows applications, such as a calendar application or a calculator. You do not have to pause, for example, to make one of your program's windows in the background the a ctive window or to perform other operations within your program.

Window Menu The QuickWin Window menu has commands for arranging windows, selecting the window with current 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 8.2 shows the Window menu as it appears in the example program QWDEMO.C, which will be 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 iconized (minimized) child windows along the bottom of the client window.
- The Input command activates the window with pending input. This command is enabled only when there is a window with input pending. (The Status Bar displays a message when a window has input pending.)
- The Clear Paste command clears the Paste Buffer.
- The Status Bar command toggles 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 active child window. You can make another window active by selecting its name from the menu.

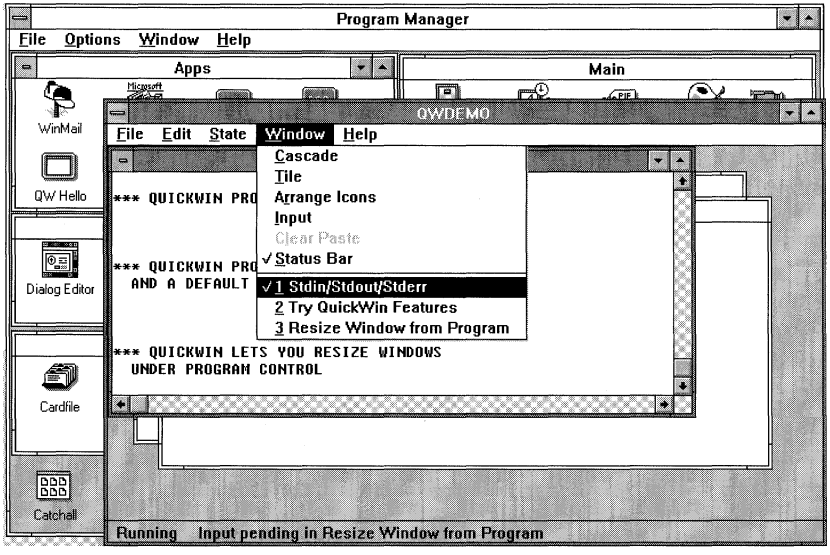


Figure 8.2 QWDEMO.C Example Program Menu

Help Menu The QuickWin Help menu has commands for calling up Windows help for the QuickWin interface. (Note that you cannot augment this help information with program-specific information.)

- The Index command calls up an index of Help for QuickWin.
- The Using Help command calls up 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 (see the next section).

Enhanced Capabilities of QuickWin

Many C and C++ programs require no changes in order to be compiled as QuickWin applications. You also have the option of giving your program more of a Windows look and greater flexibility using features described in this section. Details about using the features and calling QuickWin library functions are covered in "Writing Enhanced QuickWin Programs" on page 157.

About Dialog Box

You can customize the About dialog box.

In Windows, 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 in 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 8.3 shows the About dialog box as it appears in the example program QWDEMO.C, which is described later in the chapter.

Multiple Child Windows

By default, QuickWin displays a client window with a menu bar and one child window, titled "Stdin/Stdout/Stderr." The default input/output streams use this window. However, you have the option of opening additional child windows. Use the **_fwopen** or **_wopen** functions to open new windows. (These functions are described in "Writing Enhanced QuickWin Programs" on page 157.) If your program reads or writes multiple files, you can use document windows to display those files on the screen.

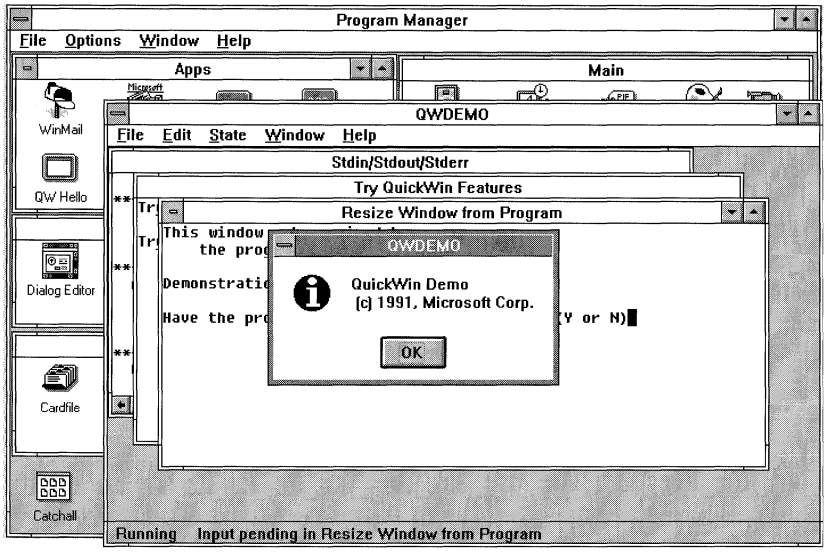


Figure 8.3 QWDEMO.C About Dialog Box

Active Window

When you open multiple child windows, the foremost window automatically becomes the active window. In Windows, the active window also is said to have the "input focus," which means that keyboard input is directed to this window. Use the **_wgetfocus** function to examine which window is active. Use the **_wsetfocus** function to make a particular window the active window. These routines are useful for bringing a hidden or partially obscured window to the foreground. The QWDEMO.C program supplied on the distribution disk demonstrates using these functions.

Program Control of Menus

Users of a Windows program 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. While your program cannot add menus of its own to the menu bar, it can have some control over QuickWin's default menus by simulating a mouse click in 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 in 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 activate a menu

command under program control. This feature is useful for setting up the initial configuration of windows and 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 executes.

Use the **_wgetsize** and **_wsetsize** functions to determine or to reset a window's current size and position. 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 an appropriately sized buffer so that users can scroll through the entire contents of the file.

Use the **_wsetexit** and **_wgetexit** functions to specify whether or not your program's windows remain on the screen when the program executes an **exit** function. Your program can behave in three possible ways at exit time:

- It leaves all windows on the screen by default.
- It leaves no windows on the screen.
- It allows user to choose windows or no windows 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 162.

QuickWin vs. Windows Applications

QuickWin provides a rich set of Windows features, but it does not provide total Windows capability. You cannot:

- Call Windows API functions.
- Detect and respond to mouse clicks in a window.
- Use graphics in your windows.
- Display and use your own menus, controls, and dialog boxes.
- Add your own customized help information to QuickWin's Help system.
- Run your QuickWin program in real mode.

Running QuickWin Programs

This section explains how to run your QuickWin programs from the DOS command line or from within Windows.

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 PATH, specify a path. For example,

```
C:WIN C:\PROGRAMS\HELLO
```

Note QuickWin programs that contain the enhanced features cannot be run as DOS programs. They can only be run under Windows in Standard or Enhanced mode.

With the Windows Run Command In Windows, choose Run from the File menu. Type the program name, prefixed with a path if needed. Click the OK button.

From the Windows File Manager In the Windows File Manager, double-click the name of the program's .EXE file.

From a Windows Icon In Windows, use the New command 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 displays an icon for your program, which you can double click with the mouse to run the program.

8.2 Compiling QuickWin Programs

Many DOS programs can be compiled as QuickWin programs simply by linking them with QuickWin libraries. A DOS program can generally become a QuickWin program as long as it doesn't make graphics calls (including cursor addressing calls) and doesn't spawn processes. When testing your QuickWin programs, remember that QuickWin programs do not run in real mode.

This section explains how to compile DOS programs as QuickWin programs from the command line and from within the Programmer's WorkBench.

Compiling from the DOS Command Line

Use the /Mq compiler option with CL.EXE to compile a QuickWin program from the DOS command line. The /Mq option has the following effects:

- The **_WINDOWS** constant is defined using the /D compiler option, causing the QuickWin constants and functions defined in IO.H and other include files to be compiled.
- The linker is invoked with the libraries *x*LIBC*y*WQ and LIBW, where *x* = S, M, C, or L, specifying the memory model used (determined by other compiler options) and *y* = E or A, specifying the floating point model used. (S, M, C, and L stand for Small, Medium, Compact, and Large, respectively. E and A specify how the compiler does floating-point math in your program: with an alternate math library (A) or with either a math coprocessor chip or an emulator library (E).) If you specify only /Mq, the default memory and math models for your programming environment are used. QuickWin supports the same models as other Windows .EXEs. You can also use the /A and /FP options to specify different models. (Windows 3.*x* does not support the tiny model.)
- If no other .DEF file is given, QuickWin uses the default .DEF file, CL.DEF. Every Windows program needs a module definition file (.DEF file) to define its name, segments, memory requirements, and exported functions. (See the *Microsoft Windows Software Development Kit Guide to Programming* for more information about module definition files.) CL.DEF provides reasonable defaults for a Windows application. In particular, CL.DEF specifies **PROTMODE**, which tells the linker to mark the application for execution in Windows standard or enhanced mode, and sets **HEAPSIZE** to 1024 bytes and **STACKSIZE** to 8096 bytes. You may want to provide your own .DEF file to change **HEAPSIZE** or **STACKSIZE**, but you are unlikely to need to redefine any of CL.DEF's other default values.

You can specify other options, filenames, and libraries in the command line as well, depending on your program's needs. To build a QuickWin C or C++ application, the compiler must have access to the module-definition file CL.DEF. Make sure this file is in the same directory as CL.EXE. In addition, the program WINSTUB.EXE must be in the current directory or in a directory listed in PATH. (WINSTUB displays the message "This program requires Microsoft Windows" if an attempt is made to run a QuickWin program from the DOS command line.)

The program QWHELLO.C, supplied on the distribution disks, can be compiled to run under either DOS or Windows, depending on the compiler options you choose. This is because it contains no enhanced QuickWin features. The following command compiles QWHELLO.C as a QuickWin program:

```
CL /Mq QWHELLO.C
```

You can immediately run the resulting program, QWHELLO.EXE, in Windows. QWHELLO writes the text "Hello, Windows!" in the standard output window. For instructions on starting the program, see "Running QuickWin Programs" on page 154.

Compiling from the Programmer's WorkBench

If you use the Programmer's WorkBench (PWB) to compile a QuickWin program, specify the "Windows QuickWin EXE" project template and set any other compile, link, browse, and debugging options you require. Then build your program. This section describes the steps briefly. For more information about using PWB, see the PWB tutorial in the *Environment and Tools* manual.

Preparing your PWB QuickWin project requires the following steps:

1. Set up your PATH, INCLUDE, and LIB environment variables so the compiler can find your source, include, and library files. The environment should be correct if you ran the SETUP program to install Microsoft C/C++. For more information, see the *Getting Started* manual.
2. In PWB, create a new PWB project and choose the C or C++ run-time support option and the "Windows QuickWin EXE" project template.
3. Set up the project's file list using the Edit Project dialog box that appears when you click the OK button in the New Project dialog box. Use the Edit Project dialog box to add files to your project. Add your files with the .C or .CPP extension. You can also add .OBJ, .LIB, and .RC files if you supply custom icons for your client and child windows (see "Using Custom Icons" on page 164). QuickWin programs also require a module definition file (with the .DEF extension). If you do not supply your own .DEF file, you must add the default module definition file, CL.DEF, to your project. For more information about .DEF files, see "Compiling from the DOS Command Line" on page 154, and consult the *Microsoft Windows Software Development Kit Guide to Programming.*
4. Set build options for the project (either Debug or Release) in the dialog box brought up by the Options/Build Options menu command.
5. Set any additional compiler, browse, link, NMake, or CodeView options you need. These options can be set in the Options menu.
6. Choose Build or Rebuild All from the Project menu.

Caution If you are running PWB from within Windows, do not execute the program from within PWB after you build it. This will spawn a new instance of Windows that runs in real mode. QuickWin programs cannot be run in real mode. Instead, execute your program from your original Windows session.

The project template you selected tells the compiler which libraries to use and sets the appropriate default compiler options. It also defines the **_WINDOWS** constant for you. This ensures that the QuickWin functions are defined, in case your program uses them. For more details about the libraries and other information required to compile QuickWin programs, see "Compiling from the DOS Command Line" on page 154.

8.3 Writing Enhanced QuickWin Programs

This section explains how to program with the enhanced features of QuickWin to improve Windows appearance and the behavior of your programs. See "Enhanced Capabilities of QuickWin" on page 151 for an overview.

The QWDEMO.C Program

The program QWDEMO.C, supplied on the distribution disks, can be compiled as an enhanced QuickWin program. It demonstrates the enhanced QuickWin features. The following command compiles QWDEMO.C as a QuickWin program:

```
CL /Mq QWINDEMO.C
```

Note QWDEMO.C cannot be run as a DOS program. Because it contains QuickWin enhancements, it can only be run under Windows in Standard or Enhanced mode.

Figures 8.2 and 8.3 show the output of QWDEMO.C.

Customizing the About Dialog Box

Use the **_wabout** function to specify the text in your program's About dialog box. This text will appear in a dialog box if the user chooses the About command from the QuickWin Help menu. For example, QWDEMO.C uses the following line of code:

```
result = _wabout("QuickWin Demo");
```

Pass the function a pointer to a null-terminated string. The function returns an **int**. If you don't call the **_wabout** function, the About dialog box displays an OK button and default information about QuickWin.

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 windows (child windows) in which to display your program's data.

Depending on your needs, you can use one of two QuickWin functions to open new child windows. The **_wopen** function is a low-level routine that returns a file handle, which you can use for 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 library functions such as **_write** and **_read**. Using these functions is explained later in this section.

In order to write to a 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 standard input/output routines, such as **fprintf** and **fscanf**, which require a stream argument.

Note If you open windows with **_fwopen**, you can use the standard **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.

Both **_wopen** and **_fwopen** require arguments of type **_wopeninfo** and **_wsizeinfo**. These are defined as C structures in the Windows version of IO.H. 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 **_WINVER**, 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 is 2,048.

The **_wsizeinfo** struct is declared as

```
struct _wsizeinfo {
     unsigned int _version;     /* Use _WINVER */
     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 **_WINVER**. For use in opening windows, the **_type** field specifies the size of the window as one of the following constants:

_WINSIZEMIN
Minimizes the window

_WINSIZEMAX
Maximizes the window

_WINSIZECHAR
Uses the listed coordinates in the **x**, **y**, **h**, **w** fields for the 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 document window, first declare variables of the **_wopeninfo** and **_wsizeinfo** types and fill in their fields. Then call either **_wopen** or **_fwopen**.

The **_wopen** function also takes a third argument of type **int**, specifying the access flags. Flags accepted are **_O_BINARY, _O_RDONLY, _O_RDWR, _O_TEXT**, and **_O_WRONLY**. Note that **_wopen** does not allow the **_O_CREAT, _O_TRUNC**, or **_O_EXCL** flags.

The **_fwopen** function takes an argument of type pointer to **char** to specify stream mode. The **_fwopen** function accepts the following mode values: "r", "w","r+", and "w+". You can also append a "t" (for text) or a "b" (for binary) to the mode string.

If you pass **NULL** for either the **_wsizeinfo** or **_wopeninfo** arguments, 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 window if successful, or –1 if not. The **_fwopen** function returns a stream pointer to the new window if successful, or **NULL** if not.

Reading from and Writing to Child Windows

Reading from or writing to a window resembles reading from or writing to a file. QuickWin windows behave as input/output streams. You can pass the file pointer obtained from the **_fwopen** function as the stream argument to standard input/output 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 */
.
.
.
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.

See the examples of using document windows for input and output in QWDEMO.C.

Resizing and Positioning Child Windows

To resize or reposition a window, use the **_wsetsize** function. Pass it an argument of type **_wsizeinfo** (see "Opening Child Windows" on page 157 for information about the **_wsizeinfo** structure).

You can also examine the current size and position of a window by calling the **_wgetsize** function.

A child window cannot be larger than its client window.

Both resizing functions require a file handle argument and a **_wsizeinfo** argument. 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 (it 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**, which allows you to specify (in the remaining fields of the **_wsizeinfo** structure) the coordinates of the 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 = _WINVER;      /* 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.

See QWDEMO.C for additional examples.

Note A child window cannot be larger than its client window.

Setting the Amount of Scrollable Text

By default, the screen buffer associated with each QuickWin document window can store 2,048 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 **_setscreenbuf** 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, up to the buffer's capacity, are stored. 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.

See QWDEMO.C for further examples.

Making a Child Window Active

When the user selects a document 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 called the active window.

To make a document window the active window (to bring it to the front), call the **_wsetfocus** function.

For example, before writing to one of several cascaded windows, you can bring the target window to the top 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 **_wsetfocus** function returns 0 if successful or –1 if not. The **_wgetfocus** function returns an integer handle to the window with the focus if successful, or –1 if not.

See QWDEMO.C for further examples.

Closing a Child Window

Once you finish using a document window, you usually close it. For windows opened with the **_wopen** function, you can call QuickWin's **_wclose** function. For windows opened with **_fwopen**, you can call the standard C **fclose** or **_fcloseall** functions. The **_wclose** function takes a second argument to specify whether the 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 input/output calls, but you can select and copy text from it, scroll through its text, and continue to use the 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** 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 );
```

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** 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. But you can alter this default behavior by calling the **_wsetexit** function. You can get the current exit setting at any time by calling the **_wgetexit** function.

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

_WINEXITNOPERSIST
Windows do not remain on the screen

_WINEXITPERSIST
Windows remain on the screen (default value)

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 current setting (one of the values above) 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 == _WINPERSIST )
    _wsetexit( _WINNOPERSIST );
```

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 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
Toggle the status bar's visibility

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.

See QWDEMO.C for further examples.

Yielding Time to Other Windows Applications

If your QuickWin program runs concurrently with other Windows applications, 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 Windows message processing 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 Windows message loop.

The **_wyield** function returns void.

Using Custom Icons

The QuickWin run-time 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 will display them instead of the default icons.

To add icons to your QuickWin program, follow these steps:

1. Create the icon files, using SDKPaint (provided with the Microsoft Windows Software Development Kit) or a similar tool provided by another Windows programming system.
2. Create a resource script with the contents

```
FRAMEICON ICON frame.ico
CHILDICON ICON child.ico
```

where *frame.ico* and *child.ico* are the names of the files containing the frame and child icons. The icon resources must have the resource names **FRAMEICON** and **CHILDICON**.

3. Using the Microsoft Windows Resource Compiler (provided with the Microsoft Windows Software Development Kit) or a similar tool, compile the icon resources and add them to your executable file.

For information on using SDKPaint and the Resource Compiler, see the *Tools* manual distributed with the Microsoft Windows Software Development Kit (version 3.0) and available as a trade book from Microsoft Press.

Providing Help

A Help file, QWIN.HLP, is provided with Microsoft C/C++. The file contains information on the QuickWin user interface. It should be stored in the same directory as your QuickWin application or in a directory named in the PATH environment variable.

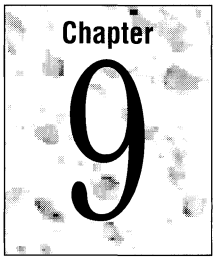
You can view Help by

- Choosing Index from the Help menu
- Highlighting any command on a QuickWin menu and pressing F1

For information on moving among screens in the Help file, choose Using Help from the Help menu.

Note QuickWin's Help is limited to information about the QuickWin user interface. You cannot add your own context-sensitive help to a QuickWin program..

Communicating with Graphics



A map, a chart, an illustration, a graph, or some other visual aid can often communicate information more quickly and more vividly than would several screens of text.

The extensive Microsoft C graphics library allows you to communicate your ideas graphically. The functions in this library range from the simple to the complex; from functions that turn on a pixel to functions that draw graphs and charts complete with labels and legends.

This chapter describes low-level graphics functions that draw basic shapes such as lines, circles, and rectangles. It introduces video modes, color palettes, coordinate systems, and synopses of the graphics and font functions. For complete function prototypes and example programs, see Help or the *Run-Time Library Reference.*

Note The ANSI C standard does not define any standard graphics functions. The functions described in this section are unique to Microsoft C/C++ and are not portable to other implementations of C. These functions can be called from C++.

9.1 Video Modes

Graphics adapters are boards or cards inside the computer that are responsible for displaying text and graphics on the screen. Commonly used adapters include:

- CGA (Color Graphics Adapter)
- EGA (Enhanced Graphics Adapter)
- HGC (Hercules Graphics Card)
- MCGA (Multicolor Graphics Array)
- MDPA (Monochrome Display Printer Adapter)
- VGA (Video Graphics Array)
- Super VGA (VGA plus extra high-resolution modes)

In addition, there are Olivetti versions of the CGA, EGA, and VGA (called OCGA, OEGA, and OVGA in this chapter).

Video modes available depend on your graphics adapter/ monitor.

Adapters can enter one or more "video modes." The video mode controls the resolution and number of colors on the video display. Microsoft C/C++ supports 25 video modes, which fall into two broad categories:

- Text modes, where characters are displayed
- Graphics modes, where individual pixels can be turned on and off

The graphics adapter and the type of monitor in use determine which of the 25 video modes are available at run time. See "Setting a Video Mode" on page 169 for a list of video modes.

Sample Low-Level Graphics Program

The following program, ERESBOX.C, shows the steps you follow to enter and exit a graphics mode. It sets the video mode **_ERESCOLOR**, draws a box, waits for a keypress, and returns to default mode, which is the video mode in effect when the program began running.

```
/* ERESBOX.C - Enters _ERESCOLOR mode and draws a box */

#include <graph.h> /* graphics functions */
#include <stdio.h> /* puts */
#include <conio.h> /* _getch */

main()
{
   if( _setvideomode( _ERESCOLOR ) ) /* EGA 640x350 mode */
   {
      _rectangle( _GBORDER, 10, 10, 110, 110 ); /* draw */
      _getch(); /* wait for a keypress */
      _setvideomode( _DEFAULTMODE ); /* return to default */
   } else puts( "Can't enter _ERESCOLOR graphics mode." )
}
```

ERESBOX.C illustrates the steps you follow to display graphics:

1. Include the header file GRAPH.H. It contains function prototypes, macros, useful structures, and symbolic constants such as **_ERESCOLOR**, **_GBORDER**, and **_DEFAULTMODE**.

   ```
   #include <graph.h>
   ```

2. Call the **_setvideomode** function, which sets the desired video mode. The function returns 0 if the hardware does not support the requested mode. (See "Setting a Video Mode" on this page.)

```
if( _setvideomode( _ERESCOLOR ) )
```

3. Draw the graphics on the screen. The example program calls the **_rectangle** function. (See "Drawing Points, Lines, and Shapes" on page 189.)

```
_rectangle( _GBORDER, 10, 10, 110, 110 )
```

4. Exit the graphics mode and return to whatever video mode was in effect before the program began. Call **_setvideomode**, passing the constant **_DEFAULTMODE**. In some cases, you might want to skip this step, exiting the program with the graphics screen still in place.

```
_setvideomode( _DEFAULTMODE );
```

In addition, you must link with the GRAPHICS.LIB library, which contains the function code. (If you instructed SETUP to include GRAPHICS.LIB in the combined libraries, you don't need to link with GRAPHICS.LIB explicitly.) If you use window-coordinate functions (which require floating-point calculations) and if you have not created a standard combined library containing a floating-point component, you must explicitly link with a floating-point math library.

Setting a Video Mode

The **_setvideomode** function turns on one of the 25 available video modes. Pass it a single integer that tells it which mode to display. The constants in Table 9.1 are defined in the GRAPH.H file. The dimensions are listed in pixels for video graphics mode and in columns for video text mode.

Table 9.1 Constants That Represent Video Modes

Constant (Name)	Description	Mode/Hardware
_MAXCOLORMODE	Graphics mode with the most colors	Graphics/All†
_MAXRESMODE	Graphics mode with the highest resolution	Graphics/All†
_DEFAULTMODE	Restores the original mode	All/All
_TEXTBW40	40 column text, 16 gray	Text/CGA
_TEXTC40	40 column text, 16/8 color	Text/CGA
_TEXTBW80	80 column text, 16 gray	Text/CGA
_TEXTC80	80 column text, 16/8 color	Text/CGA
_MRES4COLOR	320 × 200, 4 color	Graphics/CGA
_MRESNOCOLOR	320 × 200, 4 gray	Graphics/CGA

Table 9.1 *(continued)*

Constant (Name)	Description	Mode/Hardware
_HRESBW	640 × 200, BW	Graphics/CGA
_TEXTMONO	80 column text, monochrome	Text/MDPA
_HERCMONO*	720 × 348, monochrome	Graphics/HGC
_MRES16COLOR	320 × 200, 16 color	Graphics/EGA
_HRES16COLOR	640 × 200, 16 color	Graphics/EGA
_ERESNOCOLOR	640 × 350, 4 monochrome	Graphics/EGA
_ERESCOLOR	640 × 350, 4 or 16 color	Graphics/EGA
_VRES2COLOR	640 × 480, BW	Graphics/VGA
_VRES16COLOR	640 × 480, 16 color	Graphics/VGA
_MRES256COLOR	320 × 200, 256 color	Graphics/VGA
_ORESCOLOR	640 × 400, 1 of 16 colors	Graphics/Olivetti
_ORES256COLOR	640 × 400, 256 colors	Graphics/SVGA
_VRES256COLOR	640 × 480, 256 colors	Graphics/SVGA
_SRES16COLOR[1]	800 × 600, 16 colors	Graphics/SVGA
_SRES256COLOR[1]	800 × 600, 256 colors	Graphics/SVGA
_XRES16COLOR[2]	1024 × 768, 16 colors	Graphics/SVGA
_XRES256COLOR[2]	1024 × 768, 256 colors	Graphics/SVGA
_ZRES16COLOR[3]	1280 × 1024, 16 colors	Graphics/SVGA
_ZRES256COLOR[3]	1280 × 1024, 256 colors	Graphics/SVGA

* Before attempting to enter **_HERCMONO** mode, you must install the terminate-and-stay-resident program MSHERC.COM, which comes in the Microsoft C/C++ package. If you have both a Hercules adapter and an additional graphics adapter in the same computer, use the /H option to put the Hercules into **HALF** mode to avoid unpredictable and undesirable results.

† **_MAXRESMODE** and **_MAXCOLORMODE** support all adapters except the MDPA. These constants never select **_SRES**, **_XRES**, or **_ZRES** modes. See "Maximizing Resolution or Color," on page 172 for definitions of these two modes.

[1] Requires NEC MultiSync 3D, equivalent, or better.

[2] Requires NEC MultiSync 4D, equivalent, or better.

[3] Requires NEC MultiSync 5D, equivalent, or better.

If the hardware does not support the selected mode, **_setvideomode** returns 0.

Warning Do not attempt to use any of the **_SRES**, **_XRES**, or **_ZRES** modes unless your display monitor supports them. Otherwise, you risk damaging your display monitor. **Micrososft assumes no responsibility for damage to video monitors resulting from use of this software.** Consult your owner's manual for details.

Some graphics adapters are able to enter additional video modes:

- EGA adapters can display all CGA modes and **_TEXTMONO**.
- HGC adapters can enter **_TEXTMONO** mode.
- MCGA adapters can display all CGA modes, plus **_VRES2COLOR** and **_MRES256COLOR**.
- VGA adapters can display all EGA modes.
- SVGA adapters can display all VGA modes. SVGA adapters may also support nonstandard modes that conform to the limitations listed in "Limitations of VESA Support" on page 175.

Reading the _videoconfig Structure

At any time, you can inquire about the current video configuration by passing the **_getvideoconfig** function a structure of type **_videoconfig**. The structure contains 11 members, all of which are short integers. They are listed in Table 9.2.

Table 9.2 Members of a _videoconfig Structure

Member	Description
numxpixels	Number of pixels on the *x* axis
numypixels	Number of pixels on the *y* axis
numtextcols	Number of text columns available
numtextrows	Number of text rows available
numcolors	Number of color indexes
bitsperpixel	Number of bits per pixel
numvideopages	Number of video pages available
mode*	Current video mode
adapter*	Active display adapter
monitor*	Active display monitor
memory	Adapter video memory in kilobytes

* Possible values for the mode, adapter, and monitor items are listed in the GRAPH.H file.

The **_getvideoconfig** function initializes these values. Most of the values are self-explanatory. For example, if **numxpixels** holds 640, the current video mode contains 640 horizontal pixels, numbered 0 – 639.

The READVC.C example program below illustrates how to initialize and examine a **_videoconfig** structure:

```
/* READVC.C - Reads the _videoconfig structure */
```

```
#include <graph.h>
#include <stdio.h>

main()
{
   struct _videoconfig vc;

   _getvideoconfig( &vc );
   printf( "Text Rows = %i.\n", vc.numtextrows );
}
```

First, the program declares a structure `vc` of type **_videoconfig**. Next, it calls **_getvideoconfig** to initialize the structure. Finally, it prints a member of the structure.

Maximizing Resolution or Color

The constant **_MAXRESMODE** selects the highest possible resolution for the graphics adapter and monitor currently in use. The constant **_MAXCOLORMODE** selects the graphics mode with the greatest number of colors. These constants work with all graphics adapters except the MDPA. These constants never select the **_SRES**, **_XRES**, or **_ZRES** mode to avoid possible monitor damage and to guarantee that the selected mode works. (See Table 9.3.)

Table 9.3 Constants for Maximum Resolution and Color

Adapter/Monitor	_MAXRESMODE	_MAXCOLORMODE
CGA	**_HRESBW**	**_MRES4COLOR**
EGA color	**_HRES16COLOR**	**_HRES16COLOR**
EGA ecd 64K	**_ERESCOLOR**	_HRES16COLOR
EGA ecd 256K	**_ERESCOLOR**	**_ERESCOLOR**
EGA mono	**_ERESNOCOLOR**	**_ERESNOCOLOR**
HGC	**_HERCMONO**	**_HERCMONO**
MCGA	**_VRES2COLOR**	**_MRES256COLOR**
MDPA	Fails	Fails
OCGA	**_ORESCOLOR**	**_MRES4COLOR**
OEGA color	**_ORESCOLOR**	**_ERESCOLOR**
VGA/OVGA	**_VRES16COLOR**	**_MRES256COLOR**
SVGA*	**_VRES256COLOR**	_VRES256COLOR

* If your SVGA adapter does not support **_VRES256COLOR**, **_MAXCOLORMODE** selects **_ORES256COLOR**. If **_ORES256COLOR** is not supported either, the VGA modes for maximum color and maximum resolution are used.

Selecting Your Own Video Modes

A program that will run only on a single machine with a known graphics adapter can enter the appropriate video mode immediately. However, if you attempt to run the program on another machine with a different adapter, it may not run correctly, if at all.

If your program might run on a variety of computers and you prefer to select your own video modes, initialize a **_videoconfig** structure by calling the **_getvideoconfig** function. Then check the **adapter** member and use a **switch** statement to enter the selected video mode.

For example, suppose you know that a program will run on monochrome systems equipped with either an EGA adapter or a Hercules adapter. To enter the appropriate mode, use code such as this:

```
struct _videoconfig vc;

_getvideoconfig( &vc );

switch( vc.adapter )
{
    case _EGA:
    _setvideomode( _ERESNOCOLOR );
    break;
    case _HGC:
    _setvideomode( _HERCMONO );
    break;
}
```

Super VGA Support

"Super VGA" (SVGA) does not describe a standard display adapter. Instead, it refers to any VGA-compatible video adapter that also provides higher resolution modes. SVGA adapters made by different manufacturers may support different extended resolution modes.

Microsoft C graphics libraries support the VESA interface.

To allow your programs to take advantage of different SVGA adapters, the Microsoft C graphics libraries support the interface defined by the Video Electronics Standards Association (VESA). VESA has defined a standard interface for accessing the extended features of different SVGA adapters, and this interface is widely supported by video hardware manufacturers. This allows your graphics program to run with virtually any adapter that is VESA-compliant.

The **_setvideomode** function supports eight extended resolution modes, some or all of which are available on VESA-compliant SVGA adapters: **_ORES256COLOR**, **_VRES256COLOR**, **_SRES16COLOR**, **_SRES256COLOR**, **_XRES16COLOR**, **_XRES256COLOR**,

_ZRES16COLOR and **_ZRES256COLOR**. (These modes represent BIOS numbers 0x0100 through 0x0107, respectively, in the VESA standard.) Consult the owner's manual to see which modes your adapter supports.

Note that having an SVGA adapter is not sufficient for using one of these extended modes. You must also have a display monitor that supports the higher resolution. Only the first two of the extended modes can be displayed on a standard VGA analog monitor. The other extended modes require special monitors.

Warning Do not attempt to use any of the **_SRES**, **_XRES**, or **_ZRES** modes unless your display monitor supports them. Otherwise, you risk damaging your display monitor. If you use one of the extended modes in a program intended for use by others, inform the program's users of its monitor requirements. This is especially important with software intended for resale or for wide distribution. **Microsoft assumes no responsibility for damage to video monitors resulting from use of this software.** Consult your owner's manual for details.

You can also use **_setvideomode** to select a nonstandard graphics mode that is specific to a particular manufacturer's adapter, if the adapter is VESA-compliant. Consult your adapter's owner's manual for the BIOS number of a given mode, and pass that number as the argument to **_setvideomode**. The BIOS number must be between 0x15 and 0x7F. Typically, these additional modes differ from the eight modes listed above only in resolution, not in number of colors. The **_setvideomode** function may not support all of an adapter's extended modes.

VESA TSRs

Some SVGA adapters provide the VESA interface in ROM. Other adapters require that you install a TSR (Terminate-and-Stay Resident) program. Microsoft C/C++ includes TSRs for several adapters (see the file PACKING.LST for a list of TSRs supplied). If the TSR for your adapter is not included, contact your dealer or video adapter manufacturer.

These TSRs are executable programs with names of the form *xxx*VESA.COM or *xxx*VESA.EXE. You install the TSR by running the program. You must install the TSR before you run a graphics program that uses one of the VESA extended modes. If you don't install the TSR, your SVGA adapter behaves like a standard VGA adapter.

NO WARRANTY These drivers are provided on an as-is, unsupported basis, without any claim as to their correctness or suitability. Neither Microsoft nor the TSR vendor makes any representations or warranties regarding the capabilities or performance of the TSR software. Should you want to distribute any of the supplied TSRs with a software program developed using Microsoft C/C++, it is your responsibility to obtain permission from VESA and/or the TSR vendor.

Limitations of VESA Support

The graphics libraries may not work with all hardware and TSR combinations. Furthermore, VESA support has the following limitations:

- Version 1.0 of the VESA Super VGA Standard (#VS891001) is supported.
- Only color graphics modes are supported.
- Output functions must be supported by the BIOS.
- Only single window systems, as defined in section 5.2.1 of the VESA Super VGA Standard, are supported. The single window (window A) must be both readable and writeable.
- The only window size (`WinSize`) supported is 64K. Some adapters have an option of using either a 64K single-window, or a 32K double-window mode. Your adapter must be configured for 64K mode.
- The window granularity (`WinGranularity`) must be a power of 2.
- The memory model must be either 4-plane planar (16-color) or packed pixel (256-color).

Consult the VESA Super VGA Standard and your adapter's owner's manual for details. For a copy of the Super VGA Standard, write to the Video Electronics Standards Association (VESA) in San Jose, California.

9.2 Mixing Colors and Changing Palettes

Depending on the graphics card installed and the video mode in effect, you can display 2, 4, 8, 16, or 256 colors on the screen at the same time. You specify a color by selecting a color index (sometimes called a "pixel value" or "color attribute"). The color indexes are numbered from 0 to n–1, where n is the number of colors in the palette.

CGA adapters offer four palettes containing predefined fixed color sets.

All video modes that support color offer a color palette.

EGA, MCGA, and VGA adapters have palettes that can be redefined to suit your needs. You can change the visible color associated with any color index by remapping to a color index a color value that describes the true color (the amount of red, green, and blue) you want to display.

Olivetti adapters (OCGA, OEGA, and OVGA) support the standard CGA, EGA, and VGA modes (and palettes), plus an additional Olivetti mode described in "Olivetti Palettes" on page 177.

Note The distinction between a color index and a color value is important. A color index is always a short integer. A color value is always a long integer. The only exception to this rule involves **_setbkcolor**, which uses a color index cast to a long integer in CGA and text modes.

CGA Palettes

The CGA (Color Graphics Adapter) supports two color video modes, **_MRES4COLOR** and **_MRESNOCOLOR**, which display four colors selected from one of several predefined palettes of colors. They display these foreground colors against a background color that can be any one of the 16 available colors. With the CGA hardware, the palette of foreground colors is predefined and cannot be changed. Each palette number is an integer. (See Table 9.4.)

Table 9.4 CGA Palettes in _MRES4COLOR Mode

	Color Index		
Palette Number	**1**	**2**	**3**
0	Green	Red	Brown
1	Cyan	Magenta	Light Gray
2	Light Green	Light Red	Yellow
3	Light Cyan	Light Magenta	White

_MRESNOCOLOR produces palettes with shades of gray on monochrome monitors.

The **_MRESNOCOLOR** video mode produces palettes containing various shades of gray on monochrome monitors. However, the **_MRESNOCOLOR** mode displays colors when used with a color display. Only two palettes are available in this mode. Table 9.5 shows the colors available in the two palettes.

Table 9.5 CGA Palettes in _MRESNOCOLOR Mode

	Color Index		
Palette Number	**1**	**2**	**3**
0	Blue	Red	Light Gray
1	Light Blue	Light Red	White

You can use the **_selectpalette** function only in the **_MRES4COLOR**, **_MRESNOCOLOR**, and **_ORESCOLOR** graphics modes. To change palettes in other video modes, use the **_remappalette** or **_remapallpalette** functions.

Olivetti Palettes

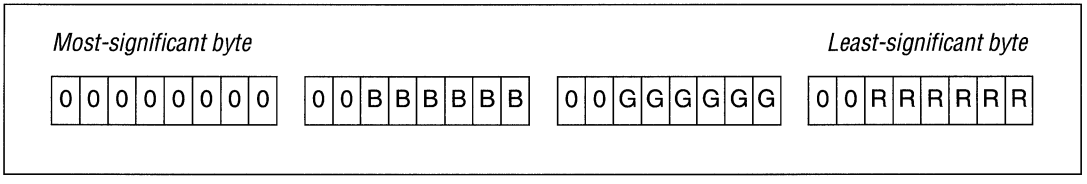
Olivetti graphics adapters are found in most Olivetti computers (including the M24, M28, M240, M280, and M380) and in the AT&T 6300 series computers. These adapters function the same as their non-Olivetti equivalents; that is, the OCGA, OEGA, and OVGA adapters support CGA, EGA, and VGA modes, respectively. In addition, Olivetti adapters can enter the high resolution **_ORESCOLOR** mode.

In **_ORESCOLOR** mode, you can choose one of 16 foreground colors by passing a value in the range 0 –15 to the **_selectpalette** function. The background color is always black.

VGA Palettes

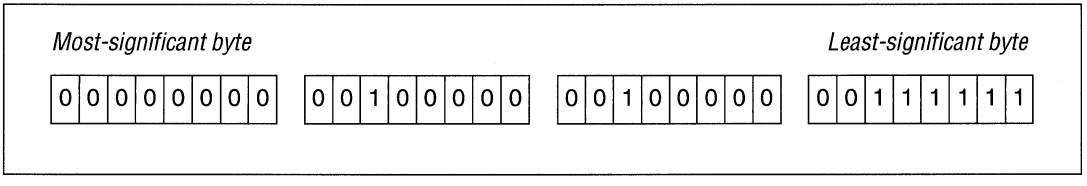
Depending on the video mode currently in effect, a VGA (Video Graphics Array) screen has 2, 16, or 256 color indexes chosen from a pool of 262,144 (256K) color values.

To name a color value, specify a level of intensity ranging from 0 – 63 for each of the red, green, and blue components. The long integer that defines a color value contains four bytes (32 bits):



The most-significant byte should contain zeros. The two high bits in the remaining three bytes should also be zero (these bits are ignored).

To mix a light red (pink), turn red all the way up, and mix in some green and blue:



The number 0x0020203FL represents this value in hexadecimal notation. You can also use the following macro:

```
#define RGB ( r, g, b ) (0x3F3F3FL & ((long)(b) << 16 | (g) << 8 | (r)))
```

To create pure yellow (100% red plus 100% green) and assign it to a variable `yel`, use this line:

```
yel = RGB( 63, 63, 0 );
```

For white, turn all the colors on: `RGB( 63, 63, 63)`. For black, set all colors to 0: `RGB( 0, 0, 0 )`.

Once you have the color value,

1. Call **_remappalette**, passing a color index and a color value.
2. Call **_setcolor** to make that color index the current color.
3. Draw something.

The program YELLOW.C below shows how to remap a color. It draws a rectangle in color index 3 and then changes index 3 to the color value 0x00003F3FL (yellow).

```
/* YELLOW.C - Draws a yellow box on the screen */
/* Requires VGA or EGA */

#include <graph.h> /* graphics functions */
#include <conio.h> /* _getch */

main()
{
   short int index3 = 3;
   long int yellow = 0x00003F3FL;
   long int old3;
   if( _setvideomode( _HRES16COLOR ) )
   {
            /* set current color to index 3*/
      _setcolor( index3 );
            /* draw a rectangle in that color */
      _rectangle( _GBORDER, 10, 10, 110, 110 );
            /* wait for a keypress */
      _getch();
            /* change index 3 to yellow */
      old3 = _remappalette( index3, yellow );
            /* wait for a keypress */
      _getch();
            /* restore the old color */
      _remappalette( index3, old3 );
```

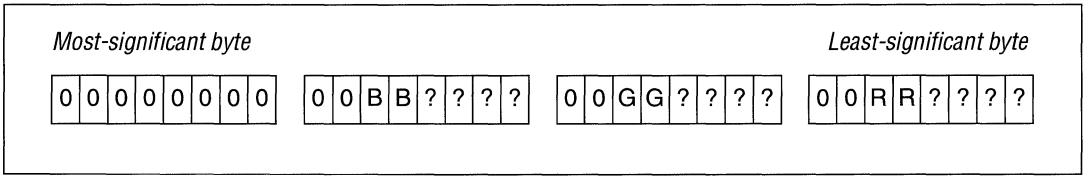
```
        _getch();
                /* back to default mode */
        _setvideomode( _DEFAULTMODE );
    } else _outtext( "This program requires EGA or VGA." );
}
```

MCGA Palettes

In terms of color mixing, the MCGA (Multicolor Graphics Array) adapter is the same as the VGA. It can display any of 256K colors. It cannot enter all of the VGA video modes, however. It is limited to CGA modes and **_VRES2COLOR** and **_MRES256COLOR**.

EGA Palettes

Mixing colors in EGA (Enhanced Graphics Adapter) is similar to the VGA mixing described in "VGA Palettes" on page 177, but there are fewer levels of intensity for the red, green, and blue (RGB) components. In the modes that offer 64 colors, the RGB values include two bits and can range in value from 0 – 3. The long integer that defines a color value looks like this:



The bits marked `0` should be zeros; the bits marked `?` are ignored. EGA color values are defined this way to maintain compatibility with VGA color values.

To form a pure red color value, use the constant 0x00000030L. For cyan (blue plus green), use 0x00303000L. The RGB macro defined above for VGA color mixing can be used as is, or you can modify it for EGA monitors:

```
#define EGARGB( r, g, b ) (0x303030L & ((long)(b) << 20 | (g) << 12 | (r << 4)))
```

In this macro, you would pass values in the range 0 – 3 instead of 0 – 63.

For an example program that remaps a color index to a color value, see YELLOW.C in "VGA Palettes" on page 177.

Symbolic Constants

The GRAPH.H file defines the following constants, which can be used as ready-made color values for EGA and VGA adapters:

_BLACK	**_GREEN**	**_LIGHTYELLOW**
_BLUE	**_LIGHTBLUE**	**_MAGENTA**
_BRIGHTWHITE	**_LIGHTCYAN**	**_RED**
_BROWN	**_LIGHTGREEN**	**_WHITE**
_CYAN	**_LIGHTMAGENTA**	
_GRAY	**_LIGHTRED**	

For example, to change color index 1 to red, use the following line:

```
_remappalette( 1, _RED );
```

This causes any object currently drawn with color index 1 to change to red. The default color value associated with index 1 is blue.

9.3 Specifying Points Within Coordinate Systems

A coordinate system describes points on the screen in terms of their horizontal (x) and vertical (y) positions. You specify a certain location by providing two values that map to a unique position.

Graphics functions usually use viewport and window coordinates.

Coordinates on the physical screen never change. Only five functions, listed in "Physical Coordinates" on this page use physical coordinates. All other graphics functions use one of these two coordinate systems:

- Viewport coordinates (short integers)
- Window coordinates (double-precision floating-point numbers)

Viewports and windows can occupy all of the physical screen or just part of it. The three coordinate systems and conventions for naming points and regions of the screen are described below.

Physical Coordinates

Within the physical screen, the upper-left corner is called the "origin." The x and y coordinates for the origin are always (0, 0). The x axis extends in the positive direction left to right, while the y axis extends in the positive direction top to bottom.

For example, the video mode **_VRES16COLOR** has a resolution of 640-by-480, which means the x axis contains the values 0 – 639 (left to right), and the y axis contains 0 – 479 (top to bottom). (See Figure 9.1.)

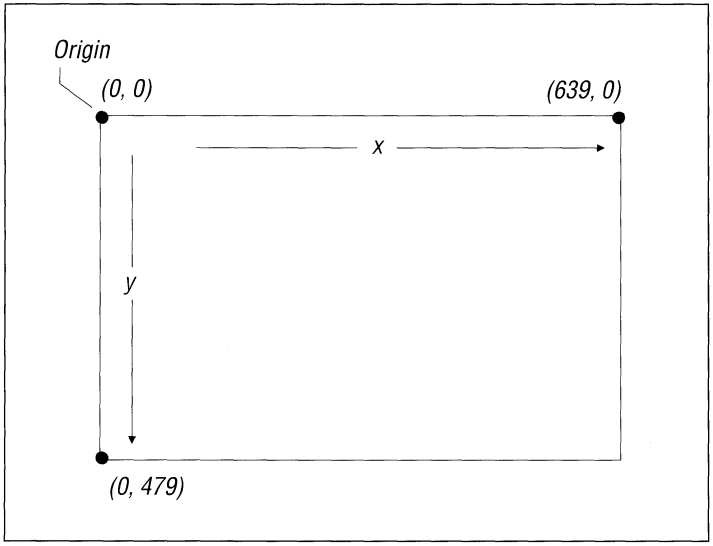


Figure 9.1 Physical Screen Coordinates

Only five functions use physical coordinates: **_setcliprgn**, **_setvieworg**, **_setviewport**, **_getviewcoord**, and **_getphyscoord**.

The **_setcliprgn** function establishes a "clipping region." Attempts to draw inside the region succeed, while attempts to draw outside the region are clipped (ignored). When you first enter a graphics mode, the clipping region defaults to the entire screen.

The **_setvieworg** function changes the current location of the origin. When a program first enters a graphics mode, the physical origin and the viewport origin are in the upper-left corner. The following code moves the viewport origin to the physical screen location (50, 100):

```
_setvieworg( 50, 100 );
```

The effect on the screen is illustrated in Figure 9.2. Note that the number of pixels remains constant, but the range of legal *x* values changes from a range of 0 to 639 (physical screen) to –50 to 589. The legal *y* values change as well.

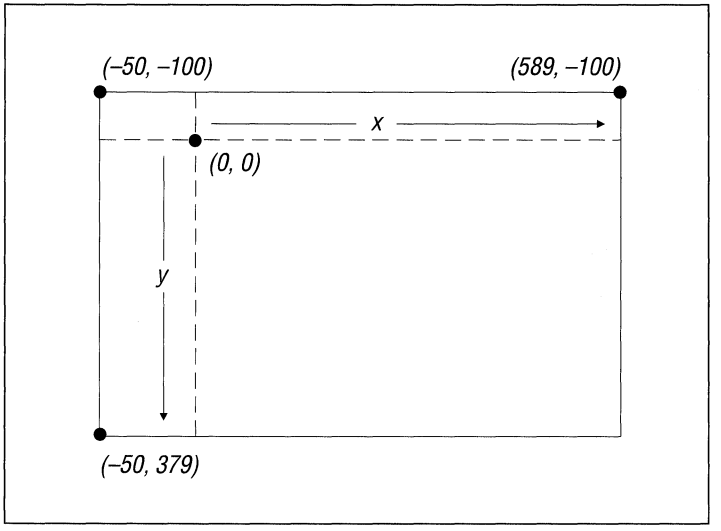


Figure 9.2 Coordinates Changed by _setvieworg

All graphics functions are affected by the new origin, including **_arc**, **_ellipse**, **_lineto**, **_moveto**, **_outgtext**, **_pie**, and **_rectangle**.

The third function that uses physical coordinates is **_setviewport**, described below, which establishes the boundaries of the current viewport.

Viewport Coordinates

The default viewport coordinate system is identical to the physical screen coordinate system. The **_setviewport** function creates a new viewport within the boundaries of the physical screen. A standard viewport has two distinguishing features:

- The origin of a viewport initially lies in the upper-left corner of the viewport, not the upper-left corner of the physical screen.
- The clipping region matches the outer boundaries of the viewport.

In other words, the **_setviewport** function does the same thing as would two separate calls to **_setvieworg** and **_setcliprgn**. All graphics output functions require values that are either viewport coordinates or window coordinates.

For example,

```
_setviewport( 50, 50, 200, 100 );
```

creates the viewport illustrated in Figure 9.3. The values passed to the **_setviewport** function are physical screen locations of opposite corners. After the viewport is created, the viewport origin lies in the upper-left corner.

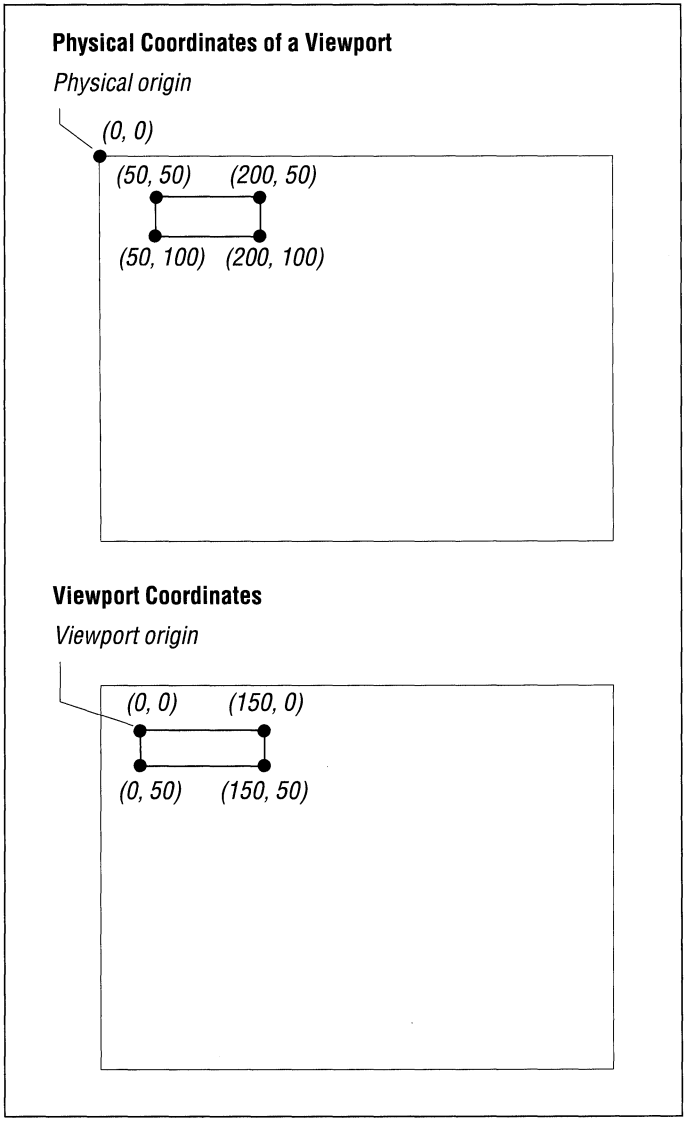


Figure 9.3 A Viewport

Window Coordinates

The **_setwindow** function allows you to use floating-point coordinates instead of integers. More importantly, it scales the screen coordinates to almost any size within the current viewport. Window functions take double-precision arguments and have names that end with the suffixes **_w** or **_wxy**. The function **_lineto_w** is the window-coordinate equivalent of the viewport function **_lineto**.

To create a window for charting 12 months of average temperatures ranging from −40 to 100, use this line:

```
_setwindow( TRUE, 1.0, -40.0, 12.0, 100.0 );
```

The first argument is the invert flag, which puts the lowest *y* value at the bottom of the screen instead of the top. The minimum and maximum coordinates follow. The new organization of the screen is shown in Figure 9.4.

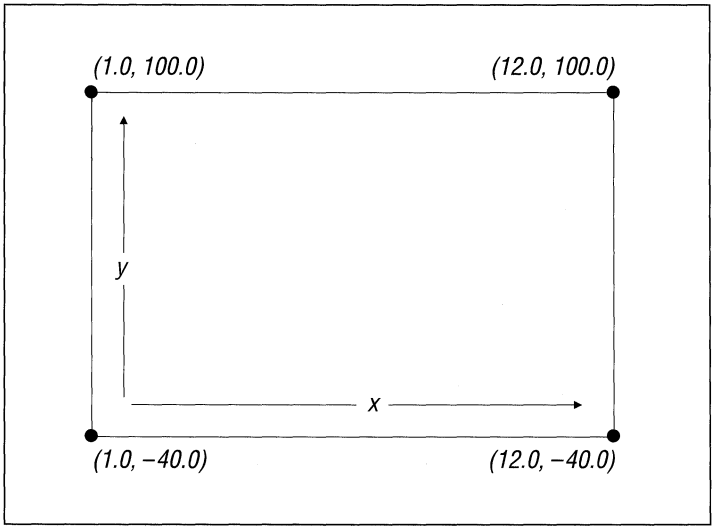


Figure 9.4 Window Coordinates

If you plot a point with **_setpixel_w** or draw a line with **_lineto_w**, the values are automatically scaled to the established window.

Window-coordinate graphics provide a lot of flexibility. You can fit an axis into a small range (such as 151.25 to 151.45) or into a large range (–50,000 to 80,000), depending on the type of data to be graphed. In addition, by changing the window coordinates and redrawing a figure, you can create the effects of zooming in or panning across a figure.

Screen Locations

A coordinate system needs two values (a horizontal and a vertical position) to describe the location of a point on the screen. There are times, however, when it is more convenient to use one variable instead of two.

Some graphics functions require you to pass the location of a point on the screen. Others return a value that represents a location. The GRAPH.H file defines two structures that allow you to refer to a point with a single variable.

- An **_xycoord** structure contains two short integers called **xcoord** and **ycoord** for use in viewport graphics.
- A **_wxycoord** structure contains two **doubles** called **wx** and **wy** for use in window-coordinate graphics.

For example, you pass four **doubles** to the **_rectangle_w** function: an *x* and *y* position for the upper-left corner of the window and an *x* and *y* position for the lower-right corner. The **_rectangle_wxy** function takes two **_wxycoord** structures.

Bounding Rectangles

Certain figures such as arcs and ellipses are centered within a "bounding rectangle," specified by two points that define the opposite corners of the rectangle. The center of the rectangle becomes the center of the figure, and the rectangle's borders determine the size of the figure. Figure 9.5 shows start and end vectors and a bounding rectangle in which a pie shape has been drawn with the **_pie** function. The first two sets of coordinates are *x1*, *y1*, *x2*, and *y2*. They define the boundaries of the rectangle. The pie shape needs two other sets of points, *x3*, *y3*, *x4*, and *y4*, which indicate the starting and ending lines.

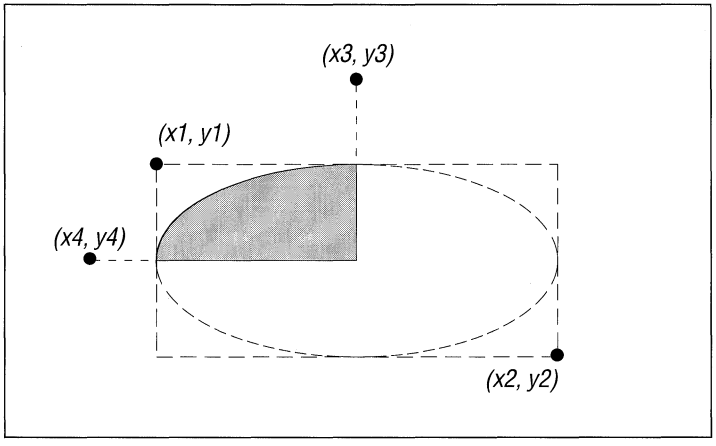


Figure 9.5 Bounding Rectangle

The Pixel Cursor

A "pixel cursor" is a location on the screen. The **_moveto** function positions this cursor at a given spot. Nothing visible appears. If you call **_lineto**, a line is drawn from the current pixel cursor to another point. The **_lineto** function also changes the location of the pixel cursor. When you call **_outgtext** to display fonted text, the characters are drawn at the current pixel cursor location.

To draw a series of connected lines, call **_lineto** several times.

The **_getcurrentposition** function returns the cursor location in an **_xycoord** structure.

9.4 Graphics Functions

This section lists the functions that work in one or more bit-mapped graphics modes. Most of these functions are present in several forms. The function names that end with **_w** use **double** values as arguments and the window coordinate system. Functions that end with **_wxy** use the window coordinate system and a **_wxycoord** structure to define the coordinates. Functions with no suffix use the viewport coordinate system.

Controlling Video Modes

The functions described below affect the current video mode, coordinate systems, clipping regions, viewports, and windows. For more information, use Help.

_clearscreen Erases the text or graphics screen and fills it with the current background color (note that setting the video mode automatically clears the screen). Pass one of the constants **_GCLEARSCREEN**, **_GVIEWPORT**, or **_GWINDOW**. No return value.

_getphyscoord Converts viewport coordinates to physical coordinates. Pass an *x* and *y* coordinate from the viewport. The function returns an **_xycoord** structure, which includes an *x* and a *y* position from the physical screen.

_getvideoconfig Obtains the status of the current graphics environment. Pass it the address of a structure of type **_videoconfig**. See "Reading the _videoconfig Structure" on page 171.

_getviewcoord Converts physical coordinates to viewport coordinates. Pass two integers: an *x* and *y* coordinate. The function returns an **_xycoord** structure containing the equivalent position within the viewport.

_getviewcoord_w Converts window coordinates to viewport coordinates. Pass two **doubles** that name points within the window. Returns the equivalent viewport coordinates as an **_xycoord** structure.

_getviewcoord_wxy Converts window coordinates to viewport coordinates in an **_xycoord** structure. Pass a **_wxycoord** structure.

_getwindowcoord Converts viewport coordinates to window coordinates. Pass two integers representing viewport coordinates. Returns a **_wxycoord** structure.

_setcliprgn Limits graphic output to part of the screen, called the "clipping region." Pass four values: the *x* and *y* coordinate of the upper-left corner (on the physical screen) and the coordinates of the lower-right corner. The default clipping region is the entire screen. See "Physical Coordinates" on page 180.

_setvideomode Selects an operating mode for the display screen. Pass a constant, such as **_HRES16COLOR**. Returns 0 if the video mode selected is not supported by the hardware. See "Setting a Video Mode" on page 169.

_setvideomoderows Sets the video mode and the number of rows for text operations. Pass two values: a video mode and the desired number of text rows (25, 30, 43, 50, or 60). Pass the symbolic constant **_MAXTEXTROWS** to get the largest available number of rows. Returns the number of rows or 0 if unsuccessful.

_setvieworg Repositions the viewport origin. Pass an *x* and *y* position: the physical screen location that will become the new origin. Returns the previous origin in an **_xycoord** structure.

_setviewport Creates a viewport, including a clipping region and a new origin in the upper-left corner of the viewport. Subsequent calls to graphics routines will be limited to the viewport area. Pass four short integers that indicate the physical screen locations of the *x* and *y* coordinates in the upper-left and lower-right corners of the viewport. No return value.

_setwindow Defines a window coordinate system. Pass five values: a short invert flag (TRUE or FALSE) and four **doubles** that represent the extreme values in the upper-left and lower-right portions of the current viewport. See "Window Coordinates" on page 184.

Changing Colors

The functions below control colors and color palettes. For an introduction to this topic, see "Mixing Colors and Changing Palettes" on page 175. For function prototypes and more information, consult Help.

_getbkcolor Reports the current background color as a long integer. In EGA, MCGA, and VGA video modes, this is a color value. In CGA and text modes, it is a color index.

_getcolor Returns the current color index.

_remapallpalette Assigns new color values to all color indexes. Pass a pointer to an array of color values. Returns 0 if unsuccessful.

_remappalette Assigns a color value to a specific color index. Pass a short color index and a long color value (which specifies the amount of red, green, and blue). Returns the previous color value for that index or –1 if unsuccessful. See "CGA Palettes" on page 176.

_selectpalette Selects a predefined palette. This function applies only to the CGA video modes **_MRES4COLOR** and **_MRESNOCOLOR** and the Olivetti graphics mode **_ORESCOLOR**. To change palettes in other color video modes, use **_remappalette** instead. Pass a short integer in the range 0 – 4 for CGA, or 0 –15 for Olivetti mode. Returns the value of the previous palette.

_setbkcolor Sets the current background color. Always pass a long integer. In EGA, MCGA, and VGA modes, this value is a color value. In CGA and text modes, this is a color index cast to a long integer. Returns the old background color or –1 if unsuccessful.

_setcolor Sets the color index to be used for graphic output. It affects later calls to functions such as **_arc**, **_ellipse**, **_floodfill**, **_lineto**, **_outgtext**, **_outtext**, **_pie**, **_rectangle**, and **_setpixel**. Returns the previous color or –1 if unsuccessful.

Drawing Points, Lines, and Shapes

The functions described below draw points, lines, and shapes. For a definition of bounding rectangle and pixel cursor, see "Bounding Rectagles" on page 185 and "The Pixel Cursor" on page 186.

_arc Draws an elliptical arc. Pass eight short integers: four pairs of *x* and *y* coordinates. The first two pairs are the corners of the bounding rectangle. The third and fourth are the starting and ending points of the arc. Returns 0 if unsuccessful.

_arc_wxy Draws an arc within the window. Pass four **wxycoord** structures. The first two are the corners of the bounding rectangle. The third and fourth are the starting and ending points of the arc. Returns 0 if unsuccessful.

_ellipse Draws an ellipse or a circle. Pass a short fill flag (**_GBORDER** or **_GFILLINTERIOR**) and four short integers representing the corners of the bounding rectangle. Returns 0 if unsuccessful.

_ellipse_w Draws an ellipse or a circle within a window. Pass a short fill flag (**_GBORDER** or **_GFILLINTERIOR**) and four **doubles** representing the corners of the bounding rectangle. Returns 0 if unsuccessful.

_ellipse_wxy Draws an ellipse or a circle. Pass a short fill flag (**_GBORDER** or **_GFILLINTERIOR**) and two **_wxycoord** structures representing the two corners of the bounding rectangle. Returns 0 if unsuccessful.

_getcurrentposition Returns the current pixel cursor position in viewport coordinates as an **_xycoord** structure. The current position can be changed by **_arc**, **_lineto**, and **_moveto**. The default position is the center of the viewport.

_getcurrentposition_w Returns the current position of the pixel cursor as a **_wxycoord** structure containing the *x* and *y* coordinates. Pass nothing.

_getpixel Returns a pixel's color index. Pass a short *x* and *y* coordinate (in viewport coordinates). If the point is outside the clipping region, the function returns –1.

_getpixel_w Returns a pixel's color index. Pass two doubles: an *x* and *y* coordinate.

_lineto Draws a line from the current pixel cursor position to a specified point. Pass a short *x* and a short *y* position. Returns 0 if unsuccessful.

_lineto_w Draws a line from the current pixel position to a specified window coordinate point. Pass a **double** *x* and *y* position. Returns 0 if unsuccessful.

_moveto Moves the pixel cursor to a specified point (with no graphic output). Pass an *x* and *y* position. Returns the coordinates of the previous position in an **_xycoord** structure.

_moveto_w Moves the pixel cursor to a specified point in a window. Pass two doubles: an *x* and a *y* coordinate. Returns the previous position as a **_wxycoord** structure.

_pie Draws a figure shaped like a pie slice. Pass a short fill flag and eight short integers. The first four describe the bounding rectangle. The final four represent the starting vector and ending vector. Returns 0 if unsuccessful.

_pie_wxy Draws a pie-slice figure within a window. Pass a short fill flag and four **_wxycoord** structures. The first two describe the bounding rectangle. The second two represent the starting vector and ending vector. Returns 0 if unsuccessful.

_rectangle Draws a rectangle in the current line style. Pass a short fill flag (**_GFILLINTERIOR** or **_GBORDER**) and four short integers: the *x* and *y* coordinates of opposite corners. Returns 0 if unsuccessful.

_rectangle_w Draws a rectangle in the current line style. Pass a short fill flag (**_GFILLINTERIOR** or **_GBORDER**) and four doubles: the *x* and *y* window coordinates of opposite corners. Returns 0 if unsuccessful.

_rectangle_wxy Draws a rectangle in the current line style. Pass a short fill flag (**_GFILLINTERIOR** or **_GBORDER**) and two **_wxycoord** structures describing the *x* and *y* coordinates of opposite corners. Returns 0 if unsuccessful.

_setpixel Sets a pixel to the current color (which is selected by **_setcolor**). Pass it integer *x* and *y* coordinates. Returns the previous value of the pixel or –1 if unsuccessful.

_setpixel_w Sets a pixel to the current color (which is selected by **_setcolor**). Pass it **double** *x* and *y* coordinates describing a position within the window. Returns the previous value of the pixel or –1 if unsuccessful.

Defining Patterns

The following functions control the style in which straight lines are drawn and the fill pattern used for solid shapes. For more information, use Help.

_floodfill Fills a bounded shape with the fill pattern set by **_setfillmask** in the current color established by **_setcolor**. Pass an *x* and *y* coordinate and a boundary color (the color index that marks the edge of the shape to be filled). Returns 0 if unsuccessful.

_floodfill_w Fills a bounded shape with the fill pattern set by **_setfillmask**. Pass **doubles** that describe an *x* and *y* position within the window and a boundary color (the color index that marks the edge of the shape to be filled). Returns 0 if unsuccessful.

_getfillmask Returns the address of the current fill mask, an eight-character array, or 0 if the fill mask is not currently defined.

_getlinestyle Returns the line style, a short integer whose bits correspond to the screen pixels turned on or off within a line.

_setfillmask Sets the current fill mask used by **_floodfill** and functions that draw solid shapes (**_ellipse**, **_pie**, and **_rectangle**). Pass the address of an array of eight unsigned characters, where each bit represents a pixel. The pixels are drawn in the current color. No return value.

_setlinestyle Sets the current style, which is used to draw the straight lines within **_lineto**, **_rectangle**, and **_pie**. Pass an unsigned short integer within which the bits correspond to the pixels on screen. For example, 0xFFFF represents a solid line, 0xAAAA is a dotted line, and 0xF0F0 is dashed.

Manipulating Images

The functions described below can be used to create animated graphics. The **_getimage** and **_putimage** functions act like a rubber stamp; after capturing a shape, you can make copies anywhere on the screen.

_getimage Stores a screen image in memory. Pass four integers (the coordinates of the bounding rectangle) and a pointer to a storage buffer. Call **_imagesize** to find out how much memory is required. No return value.

_getimage_w Stores a screen image in memory. Pass four **doubles** (the coordinates of the bounding rectangle) and a pointer to a storage buffer. Call **_imagesize_w** to find out how much memory is required. No return value.

_getimage_wxy Same as **_getimage_w**, but you pass two **_wxycoord** structures and a pointer to memory.

_imagesize Returns a long integer representing the size of an image in bytes. Call this function in preparation for a call to **_getimage**. Pass four integers: the *x* and *y* coordinates of opposite corners of the portion of the screen to be saved.

_imagesize_w Returns the size of an image in bytes in preparation for a call to **_getimage_w** and **_putimage_w**. Pass four doubles: the *x* and *y* window coordinates of opposite corners of the portion of the screen to be saved.

_imagesize_wxy Same as **_imagesize_w**, but you pass two **_wxycoord** structures.

_putimage Retrieves an image from memory and displays it on the active screen page. The image should previously have been saved to memory with **_getimage**. Pass two short integers (coordinates where the image is to be placed), a pointer to the image, and a short integer indicating what kind of action to take: **_GAND**, **_GOR**, **_GPRESET**, **_GPSET**, or **_GXOR**. No return value.

_putimage_w Displays an image from memory within a window. The image should previously have been saved to memory with **_getimage_w**. Pass two **doubles** (coordinates where the image is to be placed), a pointer to the image, and a short integer indicating what kind of action to take: **_GAND**, **_GOR**, **_GPRESET**, **_GPSET**, or **_GXOR**. No return value.

9.5 Using Graphic Fonts

A "font" is a collection of stylized text characters. Each font consists of a typeface with several type sizes.

A "typeface" is the name of the displayed text—Courier, for example, or Roman. The list on the next page shows six of the typefaces available with the Microsoft C font library.

"Type size" measures the screen area occupied by individual characters in units of screen pixels. For example, "Courier 12 × 9" denotes text of Courier typeface, with each character occupying a screen area of 12 vertical pixels by 9 horizontal pixels.

A font's spacing can be fixed or proportional. "Fixed" means that all characters have the same width in pixels. "Proportional" means the width varies. An *i*, for example, is thinner than an *M*.

The Microsoft C font functions use two methods to create fonts. The first technique generates Courier, Helv, and Tms Rmn fonts through a "bit-mapping" (or "raster-mapping") technique. Bit-mapping defines character images with binary data. Each bit in the map corresponds to a screen pixel. If a bit is 1, its associated pixel is set to the current screen color.

The second method creates the remaining three type styles—Modern, Script, and Roman—as "vector-mapped" fonts. Vector-mapping represents each character in terms of lines and arcs.

Each method has advantages and disadvantages. Bit-mapped characters are more fully formed since the pixel mapping is predetermined. However, they cannot be scaled. Vector-mapped text can be scaled to any size, but the characters tend to lack the solid appearance of the bit-mapped characters.

The following list shows six sample typefaces:

Typeface	Sample Text
Courier	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
Helv	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
Tms Rmn	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
Modern	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
Script	*ABCDEFGHIJKLMNOPQRSTUVWXYZ* *abcdefghijklmnopqrstuvwxyz*
Roman	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz

Table 9.6 lists available sizes for each font. Note that the bit-mapped fonts come in preset sizes as measured in pixels. The vector-mapped fonts can be scaled to any size.

Table 9.6 Typefaces and Type Sizes in the C Library

Typeface	Mapping	Size (in Pixels)	Spacing
Courier	Bit	10 × 8, 12 × 9, 15 × 12	Fixed
Helv	Bit	10 × 5, 12 × 7, 15 × 8, 18 × 9, 22 × 12, 28 × 16	Proportional
Tms Rmn	Bit	10 × 5, 12 × 6, 15 × 8, 16 × 9, 20 × 12, 26 × 16	Proportional
Modern	Vector	Scaled	Proportional
Script	Vector	Scaled	Proportional
Roman	Vector	Scaled	Proportional

Using the C Font Library

Data for both bit-mapped and vector-mapped fonts reside in .FON files. For example, the files MODERN.FON, ROMAN.FON, and SCRIPT.FON hold data for the three vector-mapped fonts.

You can use Microsoft Windows .FON files.

The Microsoft C .FON files are identical to the .FON files used in the Microsoft Windows operating environment. If you have access to Windows, you can use any of its .FON files with Microsoft C font functions. In addition, several vendors offer software that creates or modifies .FON files, allowing you to design your own fonts.

Your programs should follow these three steps to display fonted text:

1. Register the fonts.
2. Set the current font from the register.
3. Display text using the current font.

The following sections describe each of the three steps in detail. An example program on page 198 demonstrates these steps.

Registering the Fonts

The fonts must first be organized into a list in memory, a process called "registering." Register fonts by calling the function **_registerfonts**. This function reads header information from specified .FON files, building a list of file information but not reading any mapping data from the files.

The GRAPH.H file prototypes the **_registerfonts** function as

```
short far _registerfonts( unsigned char far * );
```

The argument points to a string containing a filename. The filename is the name of the .FON file for the desired font. The filename can include wild cards, allowing you to register several fonts with one call to **_registerfonts**.

If it successfully reads one or more .FON files, **_registerfonts** returns the number of fonts. If the function fails, it returns a negative error code.

Setting the Current Font

Call the function **_setfont** to select a current font. This function checks to see if the requested font is registered, then reads the mapping data from the appropriate .FON file. A font must be registered and marked current before your program can display text in that font.

The GRAPH.H file prototypes the**_setfont** function as

```
short far _setfont( unsigned char far * );
```

The function's argument is a pointer to a character string. The string consists of letter codes that describe the desired font, as outlined here:

Option Code	Meaning
b	The best fit from the registered fonts. This option instructs _setfont to accept the closest-fitting font if a font of the specified size is not registered. If at least one font is registered, the **b** option always sets a current font. If you do not specify the **b** option and an exact matching font is not registered, the **_setfont** function will fail. In this case, any existing current font remains current. Refer to Help for a description of error codes returned by**_setfont**. The **_setfont** function uses four criteria for selecting the best fit. In descending order of precedence, the four criteria are pixel height, typeface, pixel width, and spacing (fixed or proportional). If you request a vector-mapped font, **_setfont** sizes the font to correspond with the specified pixel height and width. If you request a raster-mapped (bitmapped) font, **_setfont** chooses the closest available size. If the requested type size for a raster-mapped font fits exactly between two registered fonts, the smaller size takes precedence.
f	Fixed-spaced font.
h*y*	Character height, where *y* is the height in pixels.

Option Code	Meaning
n*x*	Font number *x*, where *x* is less than or equal to the value returned by **_registerfonts**. For example, the option **n**3 makes the third registered font current, if three or more fonts are registered.
p	Proportional-spaced font.
r	Raster-mapped (bit-mapped) font.
t'*fontname*'	Typeface of the font in single quotes. The *fontname* string is one of the following: courier modern helv script tms rmn roman Note the space in tms rmn. Additional font files use other names for *fontname*. Refer to the vendor's documentation for these names.
v	Vector-mapped font.
w*x*	Character width, where x is the width in pixels.

Option codes are not case sensitive and can be listed in any order. You can separate codes with spaces or any other character that is not a valid option code. The **_setfont** function ignores all invalid codes.

The **_setfont** function updates a data area with parameters of the current font. The data area is in the form of a structure, defined in GRAPH.H as follows:

```
struct _fontinfo
{
   int     type;          /* set = vector,clear = bit map */
   int     ascent;        /* pix dist from top to base */
   int     pixwidth;      /* character width in pixels */
   int     pixheight;     /* character height in pixels */
   int     avgwidth;      /* average character width */
   char    filename[81];  /* file name including path */
   char    faceName[32];  /* font name */
};
```

If you want to retrieve the parameters of the current font, call the function **_getfontinfo**.

Displaying Text

The last step, displaying text, consists of two parts. First you must select a screen position for the text with the graphics function **_moveto**. Then display fonted text at that position with the function **_outgtext**. The **_moveto** function takes pixel coordinates as arguments. The coordinates locate the top left of the first character in the text string.

Sample Program

The program SAMPLER.C displays sample text in all the available fonts, then exits when a key is pressed. Make sure the .FON files are in the current directory before running the program.

```
/* SAMPLER.C: Displays sample text in various fonts. */

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <graph.h>
#include <string.h>
#define NFONTS 6

main()

{
   static unsigned char *text[2*NFONTS] =
   {
       "COURIER",          "courier",
       "HELV",             "helv",
       "TMS RMN",          "tms rmn",
       "MODERN",           "modern",
       "SCRIPT",           "script",
       "ROMAN",            "roman"
   };
   static unsigned char *face[NFONTS] =
   {
       "t'courier'",
       "t'helv'",
       "t'tms rmn'",
       "t'modern'",
       "t'script'",
       "t'roman'"
   };

static unsigned char list[20];
   struct _videoconfig vc;
   int mode = _VRES16COLOR;
   register i;

   /*   Read header info from all .FON files in
    *   current directory
    */

   if( _registerfonts( "*.FON" ) < 0 )
   {
      _outtext( "Error: can't register fonts" );
      exit( 0 );
   }
```

```
    /*   Set highest available video mode */

    if( _setvideomode( _MAXRESMODE ) == 0 )
       exit ( 0 );

    /*   Copy video configuration into structure vc */

    _getvideoconfig( &vc );

    /*   Display six lines of sample text */

    for( i = 0; i < NFONTS; i++ )
    {
       strcpy( list, face[i] );
       strcat( list, "h30w24b" );

       if( _setfont( list ) >= 0 )
       {
           _setcolor( i + 1 );
           _moveto( 0, (i * vc.numypixels) / NFONTS );
           _outgtext( text[i * 2] );
           _moveto( vc.numxpixels / 2,
                       (i * vc.numypixels) / NFONTS );
           _outgtext( text[(i * 2) + 1] );
       }
       else
       {
           _setvideomode( _DEFAULTMODE );
           _outtext( "Error: can't set font" );
           exit( 0 );
       }
    }

_getch();
    _setvideomode( _DEFAULTMODE );

    /* Return memory when finished with fonts */

    _unregisterfonts();
    exit( 0 );
}
```

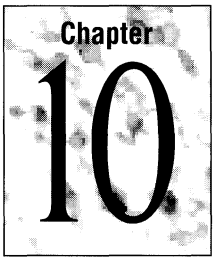
Using Fonts Effectively

Displaying fonts is simply another form of graphics; using fonts effectively requires little programming effort. Still, there are a few things to watch:

- Remember that the video mode should be set only once. If you generate an image with presentation graphics and want to add text to it, do not reset the video mode prior to calling the font routines. Doing so will blank the screen, destroying the original image.

- The **_setfont** function reads specified .FON files to obtain mapping data for the current font. Each call to **_setfont** causes a disk access and overwrites the old font data in memory. If you want to show text of different styles on the same screen, display all text of one font before moving on to the others. Minimizing the number of calls to **_setfont** saves time spent in disk I/O and memory reloads.
- When your program finishes using the fonts library, you may want to free the memory occupied by the register list by calling **_unregisterfonts**. This function frees the memory allocated by **_registerfonts**. The register information for each type size of each font takes up approximately 140 bytes of memory.
- Aesthetic suggestions for the printed page also apply to screen text. Typefaces are more effective when they do not compete with each other for attention. Restricting the number of styles per screen to one or two generally results in a more pleasing, less cluttered image.

Creating Charts and Graphs



The low-level graphics functions described in Chapter 9, "Communicating with Graphics," draw points, lines, and shapes. Although it is possible to use these functions to generate charts and graphs, an additional set of high-level graphics functions is better suited to this task.

"Presentation graphics" is a set of high-level functions that displays presentation-quality graphics. These functions transform numeric data into pie charts, bar and column charts, line graphs, and scatter diagrams.

This chapter describes how to use presentation graphics.

10.1 Overview of Presentation Graphics

The presentation graphics library PGCHART.LIB contains 22 functions. They are listed in Table 10.1 for convenient reference.

Table 10.1 Presentation Graphics Functions

Primary Functions	Secondary Functions	
_pg_chart	_pg_analyzechart	_pg_hlabelchart
_pg_chartms	_pg_analyzechartms	_pg_resetpalette
_pg_chartpie	_pg_analyzepie	_pg_resetstyleset
_pg_chartscatter	_pg_analyzescatter	_pg_setchardef
_pg_chartscatterms	_pg_analyzescatterms	_pg_setpalette
_pg_defaultchart	_pg_getchardef	_pg_setstyleset
_pg_initchart	_pg_getpalette	_pg_vlabelchart
	_pg_getstyleset	

The seven primary functions initialize variables and display selected chart types.

In most cases, you will be using only seven "primary functions." These functions initialize variables and display selected chart types. The 15 "secondary functions" of presentation graphics do not directly display charts. Most of them retrieve or set data in the presentation graphics chart environment.

Among the secondary functions are the "analysis functions," identified by the prefix **_pg_analyze**. These five functions calculate default values that pertain to a given chart type and data set. Calling an analysis function has the same effect as calling a corresponding primary function, except that the chart is not displayed. This allows you to pass on to the library the burden of calculating values. You can then make modifications to the resulting values and call a primary routine to display the chart.

Use the **_pg_hlabelchart** and **_pg_vlabelchart** functions to display text that is not part of a title or axis label on your chart. These functions enable you to attach notes or other messages to your chart.

10.2 Parts of a Graph

This section describes the terms used to refer to the different kinds of information that can be plotted. The various types of charts and graphs are also defined.

Data Series

Data that are related by a common idea or purpose constitute a "series." For example, the prices of a futures commodity over the course of a year form a single series of data. The volume forms a second data series.

When you include several series in one chart, characteristics such as color and pattern can help distinguish one from another. You can more readily differentiate series on a color monitor than you can on a monochrome monitor. The number of series that can appear on the same chart depends on the chart type and the number of available colors.

Categories

"Categories" are nonnumeric data. A set of categories forms a frame of reference for the comparison of numeric data. For example, the months of the year are categories against which numeric data such as inches of rainfall can be plotted.

Regional sales provide another example. A chart can compare a company's sales in different parts of the country. Each region forms a category.

Values

"Values" are numeric data. Sales, stock prices, air temperatures, and populations are all series of values that can be plotted against categories or against other values.

Presentation graphics allows you to overlay different series of value data on a single graph. For example, average monthly temperatures or monthly sales of heating oil during different years—or a combination of temperatures and sales—can be plotted together on the same graph.

Pie Charts

"Pie charts" are used to represent data by showing the relationship of each part to the whole. A good example is a company's annual budget. A pie chart allows you to view each area of revenue or spending by its relative size within the context of the entire company budget.

Presentation graphics can display either a standard or an "exploded" pie chart. The exploded view shows the pie with one or more pieces separated for emphasis. You can label each slice of a pie chart with a percentage figure if you wish.

Bar and Column Charts

As the name implies, a "bar chart" shows data as horizontal bars. Bar charts show comparisons among items rather than absolute value.

"Column charts" are vertical bar charts. Column charts are frequently used to show variations over a period of time, since they suggest time flow better than a bar chart.

Line Graphs

"Line graphs" illustrate trends or changes in data. They show how a series of values varies against a particular category—for example, average temperatures throughout one year.

Traditionally, line graphs show a collection of data points connected by lines. Presentation graphics can also plot points that are not connected by lines.

Scatter Diagrams

A "scatter diagram" is the only type of graph available in presentation graphics that directly compares values with values. A scatter diagram simply plots points.

Scatter diagrams illustrate the relationship between numeric values in different groups of data. They graphically show trends and correlations not easily detected from rows and columns of raw numbers.

Scatter diagrams are most useful with large amounts of data. Consider, for example, the relationship between personal income and family size. If you poll one thousand wage earners for their income and family size, you have a scatter diagram with one thousand points. If you combine your results so that you are left with one average income for each family size, you have a line graph.

Axes

All presentation graphics charts except pie charts are displayed with two perpendicular reference axes. The vertical, or *y*, axis runs from top to bottom of the chart and is placed against the left side of the screen. The horizontal, or *x*, axis runs from left to right across the bottom of the screen.

The *x* axis is the category axis for column and line charts and the value axis for bar charts. The *y* axis is the value axis for column and line charts and the category axis for bar charts.

Chart Windows

The "chart window" defines that part of the screen on which the chart is drawn. By default, the window fills the entire screen, but presentation graphics allows you to resize the window for smaller graphs. By redefining the chart window to different screen locations, you can view separate graphs together on the same screen.

Data Windows

While the chart window defines the entire graph including axes and labels, the "data window" defines only the actual plotting area. This is the portion of the graph to the right of the *y* axis and above the *x* axis. You cannot specify or adjust the size of the data window. Presentation graphics automatically determines its size based on the dimensions of the chart window.

Chart Styles

Each of the five types of presentation graphics charts can appear in two different "chart styles," as described in Table 10.2.

Table 10.2 Presentation Graphics Chart Styles

Chart Type	Chart Style #1	Chart Style #2
Pie	With percentages	Without percentages
Bar	Side-by-side	Stacked
Column	Side-by-side	Stacked
Line	Points with lines	Points only
Scatter	Points with lines	Points only

Bar and column charts have only one style when displaying a single series of data. The styles "side-by-side" and "stacked" are applicable when more than one series appears on the same chart. The first style arranges the bars or columns for the different series side by side, showing relative heights or lengths. The stacked style, illustrated for a column chart in Figure 10.3, emphasizes relative sizes between bars or columns.

Legends

Legends help identify individual data series.

When displaying more than one data series on a chart, presentation graphics uses different colors, line styles, or patterns to differentiate them. Presentation graphics also can display a "legend" that labels the different series of a chart. For a pie chart, the legend labels individual slices of the pie.

A sample of the color and pattern used to graph the series appears next to the series label. This identifies the set of data to which the labels belong.

You may change the font displayed by calling the **_registerfonts** and **_setfont** functions (see Chapter 9, "Communicating with Graphics," for more information about using fonts). If you don't select a font, presentation graphics defaults to an internal font.

10.3 Writing a Presentation Graphics Program

To write a C or C++ program that uses presentation graphics, follow these steps:

1. Include the required header files, GRAPH.H and PGCHART.H, as well as any other header files your program may need.
2. Set the video mode to a graphics mode. See Chapter 9, "Communicating with Graphics," for a description of video modes.
3. Initialize the presentation graphics chart environment. Presentation graphics places charting parameters in data structures. The amount of initialization that must be done by your program depends on how extensively it relies on the defaults.

4. Assemble the plot data. Data can be collected in a variety of ways: by calculating it elsewhere in the program, reading it from files, or entering it from the keyboard. All plot data must be assembled in arrays because the presentation graphics functions locate them through pointers.
5. Call presentation graphics functions to display the chart. Pause while the chart is on the screen.
6. Reset the video mode. When your program detects the signal to continue, it should reset the video to its original (default) mode.

After compiling the program, link it to the library modules PGCHART.LIB and GRAPHICS.LIB.

Note If your program uses the alternate math package (i.e., if it is compiled with /FPa), it cannot use the PGCHART.LIB module.

The sample programs on pages 206 – 212 use 5 of the 22 presentation graphics functions: **_pg_initchart**, **_pg_defaultchart**, **_pg_chartpie**, **_pg_chart**, and **_pg_chartscatter**. Each program is commented so that you can recognize the steps given in this section.

Pie Charts

The following program uses presentation graphics to display a pie chart for monthly sales of orange juice over a year. The chart, which is shown in Figure 10.1, remains on the screen until a key is pressed.

```
/* PIE.C:  Create sample pie chart.  */

#include <conio.h>
#include <string.h>
#include <graph.h>
#include <pgchart.h>

#define MONTHS 12

typedef enum {FALSE, TRUE} boolean;

float far value[MONTHS] =
{
    33.0, 27.0, 42.0, 64.0,106.0,157.0,
   182.0,217.0,128.0, 62.0, 43.0, 36.0
};
char far *category[MONTHS] =
{
   "Jan", "Feb", "Mar", "Apr",
```

```
    "May", "Jun", "Jly", "Aug",
    "Sep", "Oct", "Nov", "Dec"
};
short far explode[MONTHS] = {0};

main()
{
    _chartenv env;
    int mode = _VRES16COLOR;

    /* Set highest video mode available */

    if( _setvideomode( _MAXRESMODE ) == 0 )
        exit( 0 );

    /* Initialize chart library and a default pie chart */

    _pg_initchart();
    _pg_defaultchart( &env, _PG_PIECHART, _PG_PERCENT );

    /* Add titles and some chart options */

    strcpy( env.maintitle.title, "Good Neighbor Grocery" );
    env.maintitle.titlecolor = 6;
    env.maintitle.justify = _PG_RIGHT;
    strcpy( env.subtitle.title, "Orange Juice Sales" );
    env.subtitle.titlecolor = 6;
    env.subtitle.justify = _PG_RIGHT;
    env.chartwindow.border = FALSE;

    /* Parameters for call to _pg_chartpie are:
     *
     *    env        - Environment variable
     *    category   - Category labels
     *    value      - Data to chart
     *    explode    - Separated pieces
     *    MONTHS     - Number of data values
     */
    if( _pg_chartpie( &env, category, value,
                      explode, MONTHS ) )
    {
        _setvideomode( _DEFAULTMODE );
        _outtext( "Error:  can't draw chart" );
    }
    else
    {
        _getch();
        _setvideomode( _DEFAULTMODE );
    }
    return( 0 );
}
```

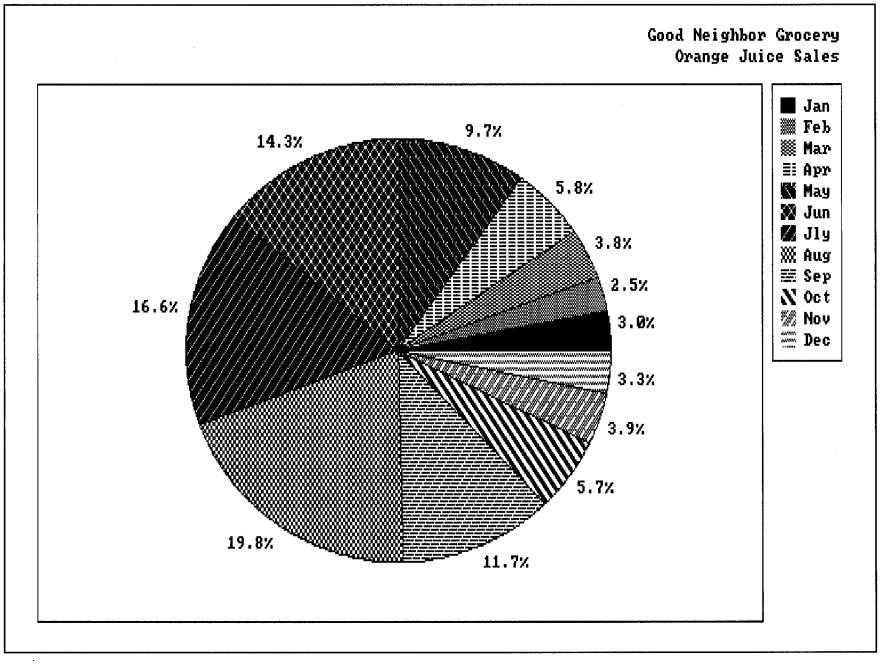


Figure 10.1 Example Pie Chart

Bar, Column, and Line Charts

The code for the PIE.C program needs only minor alterations to produce bar, column, and line charts for the same data:

- Replace the call to **_pg_chartpie** with **_pg_chart**. This function produces bar, column, and line charts depending on the value of the second argument for **_pg_defaultchart**.
- Give new arguments to **_pg_defaultchart** that specify chart type and style.
- Assign titles for the *x* axis and *y* axis in the structure `env`.
- Remove references to array `explode`, which is applicable only to pie charts.

The following example produces a bar chart for the store owner's data. The result is shown in Figure 10.2.

```
/* BAR.C:  Create sample bar chart. */

#include <conio.h>
#include <string.h>
#include <graph.h>
#include <pgchart.h>
```

```
#define MONTHS 12

typedef enum {FALSE, TRUE} boolean;

float far value[MONTHS] =
{
    33.0, 27.0, 42.0, 64.0,106.0,157.0,
   182.0,217.0,128.0, 62.0, 43.0, 36.0
};
char far *category[MONTHS] =
{
   "Jan", "Feb", "Mar", "Apr",
   "May", "Jun", "Jly", "Aug",
   "Sep", "Oct", "Nov", "Dec"
};

main()
{
   _chartenv env;
   int mode = _VRES16COLOR;

   /* Set highest video mode available */

   if( _setvideomode( _MAXRESMODE ) == 0 )
      exit( 0 );

   /* Initialize chart library and a default bar chart */
   _pg_initchart();
   _pg_defaultchart( &env, _PG_BARCHART, _PG_PLAINBARS );

   /* Add titles and some chart options */

   strcpy( env.maintitle.title, "Good Neighbor Grocery" );
   env.maintitle.titlecolor = 6;
   env.maintitle.justify = _PG_RIGHT;
   strcpy( env.subtitle.title, "Orange Juice Sales" );
   env.subtitle.titlecolor = 6;
   env.subtitle.justify = _PG_RIGHT;
   strcpy( env.yaxis.axistitle.title, "Months" );
   strcpy( env.xaxis.axistitle.title, "Quantity (cases)" );
   env.chartwindow.border = FALSE;

   /* Parameters for call to _pg_chart are:
    *    env        - Environment variable
    *    category   - Category labels
    *    value      - Data to chart
    *    MONTHS     - Number of data values
    */
   if( _pg_chart( &env, category, value, MONTHS ) )
   {
      _setvideomode( _DEFAULTMODE );
      _outtext( "Error:  can't draw chart" );
   }
```

```
    else
    {
        _getch();
        _setvideomode( _DEFAULTMODE );
    }
    return( 0 );
}
```

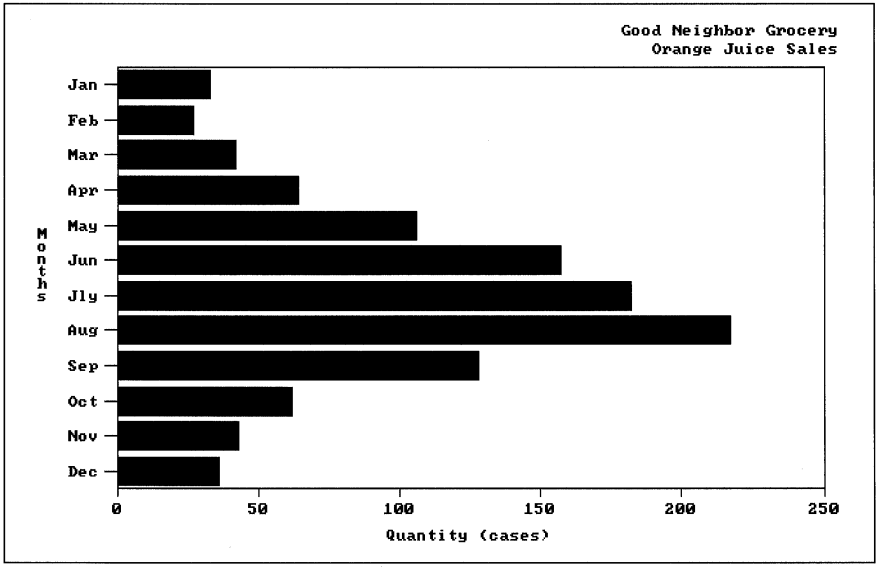


Figure 10.2 Example Bar Chart

The grocer's bar chart becomes a column chart in two easy steps. Simply specify the new chart type when calling **_pg_defaultchart** and change the axis titles. To produce a column chart for the grocer's data, replace the call to **_pg_defaultchart** with

```
_pg_defaultchart( &env, _PG_COLUMNCHART, _PG_PLAINBARS );
```

Replace the last two calls to **strcpy** with

```
strcpy( env.xaxis.axistitle.title, "Months" );
strcpy( env.yaxis.axistitle.title, "Quantity (cases)" );
```

Note that now the *x* axis is labeled "Months" and the *y* axis is labeled "Quantity (cases)." Figure 10.3 shows the resulting column chart.

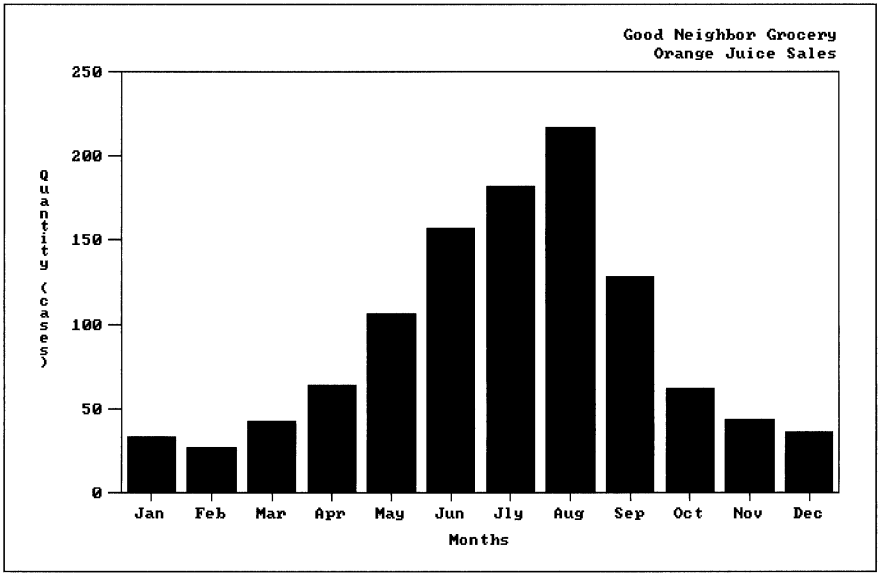


Figure 10.3 Example Column Chart

Creating an equivalent line chart requires only one change. Use the same code as for the column chart and replace the call to **_pg_defaultchart** with

```
_pg_defaultchart( &env, _PG_LINECHART, _PG_POINTANDLINE );
```

Figure 10.4 shows the line chart for the grocer's data.

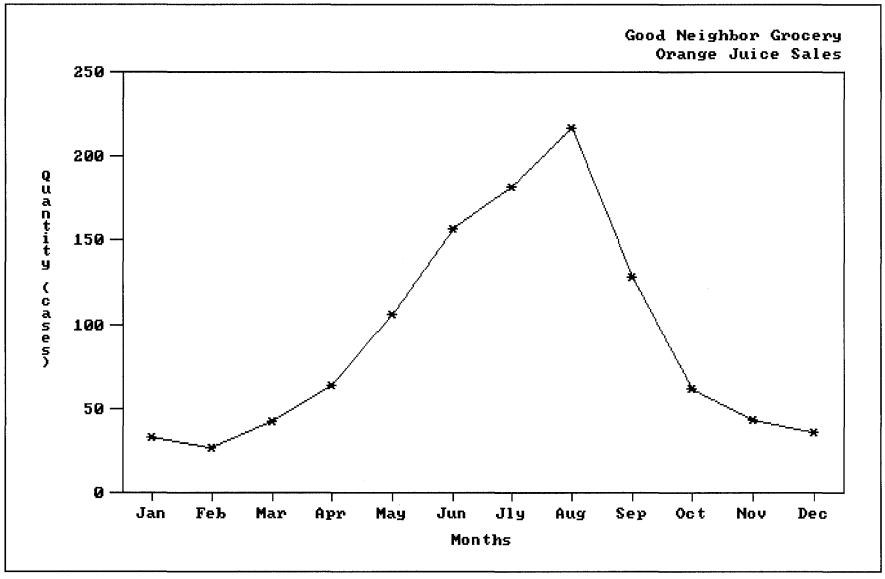


Figure 10.4 Example Line Chart

Scatter Diagram

The program SCATTER.C displays a scatter diagram that illustrates the relationship between the sales of orange juice and hot chocolate throughout a 12-month period. Figure 10.5 shows the results of SCATTER.C. Notice that the scatter points form a slightly curved line, indicating that a correlation exists between the sales of the two products. The demand for orange juice is roughly inverse to the demand for hot chocolate.

```
/* SCATTER.C:  Create sample scatter diagram. */

#include <conio.h>
#include <string.h>
#include <graph.h>
#include <pgchart.h>

#define MONTHS 12

typedef enum {FALSE, TRUE} boolean;

/* Orange juice sales */
```

```
float far xvalue[MONTHS] =
{
    33.0, 27.0, 42.0, 64.0,106.0,157.0,
   182.0,217.0,128.0, 62.0, 43.0, 36.0
};

/* Hot chocolate sales */

float far yvalue[MONTHS] =
{
   37.0, 37.0, 30.0, 19.0, 10.0,  5.0,
    2.0,  1.0,  7.0, 15.0, 28.0, 39.0
};

main()
{
   _chartenv env;
   int mode = _VRES16COLOR;

   /* Set highest video mode available */

   if( _setvideomode( _MAXRESMODE ) == 0 )
      exit( 0 );
   /* Initialize chart library and default
    * scatter diagram
    */
   _pg_initchart();
   _pg_defaultchart( &env, _PG_SCATTERCHART,
                     _PG_POINTONLY );

   /* Add titles and some chart options */

   strcpy( env.maintitle.title, "Good Neighbor Grocery" );
   env.maintitle.titlecolor = 6;
   env.maintitle.justify = _PG_RIGHT;
   strcpy( env.subtitle.title,
           "Orange Juice vs Hot Chocolate" );
   env.subtitle.titlecolor = 6;
   env.subtitle.justify = _PG_RIGHT;
   env.yaxis.grid = TRUE;
   strcpy( env.xaxis.axistitle.title,
           "Orange Juice Sales" );
   strcpy( env.yaxis.axistitle.title,
           "Hot Chocolate Sales" );
   env.chartwindow.border = FALSE;

   /* Parameters for call to _pg_chartscatter are:
    *    env        - Environment variable
    *    xvalue     - X-axis data
    *    yvalue     - Y-axis data
    *    MONTHS     - Number of data values
    */
```

```
    if( _pg_chartscatter( &env, xvalue,
                          yvalue, MONTHS ) )
    {
       _setvideomode( _DEFAULTMODE );
       _outtext( "Error:  can't draw chart" );
    }
    else
    {
       _getch();
       _setvideomode( _DEFAULTMODE );
    }
    return( 0 );
}
```

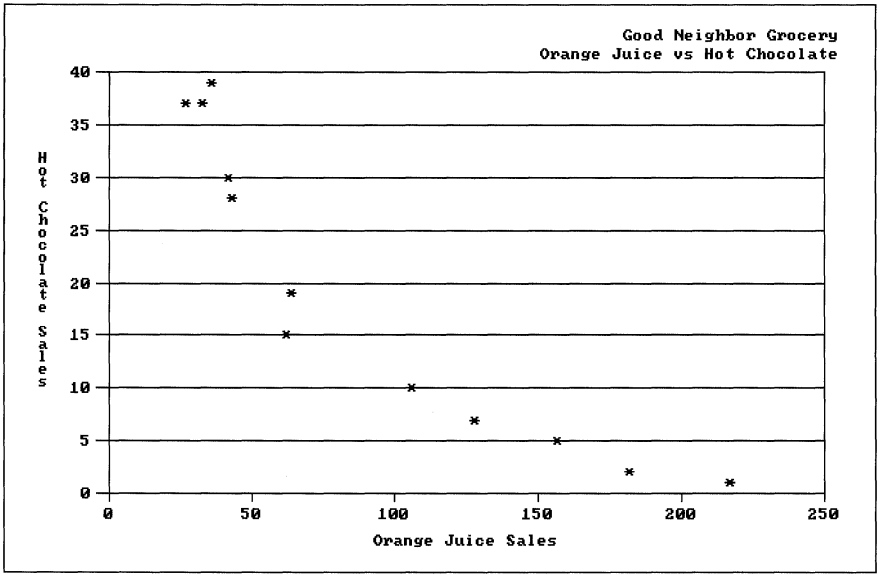


Figure 10.5 Example Scatter Diagram

10.4 Manipulating Colors and Patterns

Presentation graphics displays each data series in a way that makes it discernible from other series. It does this by defining a separate "palette" for every data series in a chart. Palettes consist of entries that determine color, line style, fill pattern, and point character used to graph the series.

Presentation graphics maintains its palettes as an array of structures. The header file PGCHART.H defines the palette structures as follows:

```
/* Typedef for pattern bitmap */
typedef unsigned char _fillmap[8];

/* Typedef for palette entry definition */
typedef struct
{
   unsigned short color;
   unsigned short style;
   _fillmap        fill;
   char           plotchar;
} _paletteentry;

/* Typedef for palette definition */
typedef _paletteentry _palettetype[_PG_PALETTELEN];
```

Do not confuse the presentation graphics palettes with the adapter display palettes, which are register values kept by the video controller. The function **_selectpalette** described in Chapter 9, "Communicating with Graphics," sets the display palette. It does not define the data series palettes used by presentation graphics.

Color Pool

The color pool determines the colors of graphic elements.

Presentation graphics organizes all chart colors into a "color pool." The color pool holds the color index values valid for the current graphics mode. (Refer to Chapter 9, "Communicating with Graphics," for more information about the color index.) Palette structures contain color codes that refer to the color pool. A palette's color index determines the colors used to graph the data series associated with the palette. The colors of labels, titles, legends, and axes are determined by the contents of the color pool.

The first element of the color pool is always 0, which is the color index for the screen background color. The second element is always the highest color index available for the graphics mode. The remaining elements repeat the sequences of available pixel values, beginning with 1.

As shown in the example in "Manipulating Colors and Patterns" on page 214, the first member of a palette data structure is

```
unsigned short color;
```

This member defines the color index for the data series associated with the palette.

An example should make this clearer. A graphics mode of **_MRES4COLOR** (320-by-200 pixels) provides four colors for display. Color index values from 0 to 3 determine the possible colors—say, black, green, red, and brown, respectively.

The first eight elements of this color pool are as follows.

Color Pool Index	Color Index	Color
0	0	Black
1	3	Brown
2	1	Green
3	2	Red
4	3	Brown
5	1	Green
6	2	Red
7	3	Brown

Notice that the sequence of available foreground colors repeats from the third element. The first data series in this case would be plotted in brown, the second series in green, the third series in red, the fourth series again in brown, and so forth.

Video adapters such as the EGA or the Hercules InColor Card allow 16 on-screen colors. This allows presentation graphics to graph more series without duplicating colors.

Style Pool

Presentation graphics matches the color pool with a collection of different line styles called the "style pool." Entries in the style pool define the appearance of lines such as axes and grids. Lines can be solid, dotted, dashed, or some combination of styles.

The second member of a palette structure defines a style code as

```
unsigned short style;
```

Each palette contains a style code that refers to an entry in the style pool in the same way that it contains a color code that refers to an entry in the color pool. The style code value in a palette is applicable only to line graphs and lined scatter diagrams. The style code determines the appearance of the lines drawn between points.

Use the different line styles in the style pool to differentiate series.

The palette's style code adds further variety to the lines of a multiseries graph. It is most useful when the number of lines in a chart exceeds the number of available colors. For example, a graph of nine different data series must repeat colors if only three foreground colors are available for the display. However, the style code for each color repetition will be different, ensuring that none of the lines looks the same.

Pattern Pool

Presentation graphics also maintains a pool of "fill patterns" that determine the fill design for column, bar, and pie charts. The third member of the palette structure holds the fill pattern. The pattern member is an array:

```
_fillmap fill;
```

where `_fillmap` is type-defined as

```
typedef unsigned char _fillmap[8];
```

Each fill pattern array holds an 8-by-8 bit map that defines the fill pattern for the data series associated with the palette. Table 10.3 shows how a fill pattern of diagonal stripes is created with the `fill` pattern array.

The bit map in Table 10.3 corresponds to screen pixels. Each of the eight layers of the map is a binary number, where a solid circle signifies 1 and an open circle signifies 0. Thus the first layer of the map—that is, the first byte—represents the binary number 10011001, which is the decimal number 153.

Table 10.3 Fill Patterns

Bit Map	Value in Fill
● ○ ○ ● ● ○ ○ ●	`fill[0] = 153`
● ● ○ ○ ● ● ○ ○	`fill[1] = 204`
○ ● ● ○ ○ ● ● ○	`fill[2] = 102`
○ ○ ● ● ○ ○ ● ●	`fill[3] = 51`
● ○ ○ ● ● ○ ○ ●	`fill[4] = 153`
● ● ○ ○ ● ● ○ ○	`fill[5] = 204`
○ ● ● ○ ○ ● ● ○	`fill[6] = 102`
○ ○ ● ● ○ ○ ● ●	`fill[7] = 51`

For example, if you want to create the pattern in Table 10.3 for your chart's first data series, you must reset the `fill` array for the first palette structure. You can do this in five steps:

1. Declare a structure of type **_palettetype** to hold the palette parameters.
2. Call **_pg_initchart** to initialize the palettes with default values.
3. Call the presentation graphics function **_pg_getpalette** to retrieve a copy of the current palette data.
4. Assign the values given in Table 10.3 to the array `fill` for the first palette.
5. Call the presentation graphics function **_pg_setpalette** to load the modified palette values.

The following lines of code demonstrate these five steps:

```
/* Declare a structure array for palette data. */

_palettetype palette_struct;
.
.
.
/* Initialize chart library */

_pg_initchart();
.
.
.
/* Copy current palette data into palette_struct */

_pg_getpalette( palette_struct );

/* Reinitialize fill pattern for first palette using
   values in Table 10.3 */

palette_struct[1].fill[0] = 153;
palette_struct[1].fill[1] = 204;
palette_struct[1].fill[2] = 102;
palette_struct[1].fill[3] =  51;
palette_struct[1].fill[4] = 153;
palette_struct[1].fill[5] = 204;
palette_struct[1].fill[6] = 102;
palette_struct[1].fill[7] =  51;

/* Load new palette data */

_pg_setpalette( palette_struct );
```

Now when you display your bar or column chart, the first series appears filled with the striped pattern shown in Table 10.3.

Palette structures are used differently with pie charts. Instead of clarifying multiple series, fill patterns, line styles, and colors, palette structures are used to distinguish individual slices in a pie chart. Palettes are recycled if the number of slices exceeds **_PG_PALETTELEN**. Thus, the first palette dictates not only the appearance of the first slice, but of slice number **_PG_PALETTELEN** as well. The second palette determines the appearance of both the second slice and of slice number **_PG_PALETTELEN** + 1, and so forth.

Character Pool

The last member of a palette structure is an index number in a pool of ASCII characters:

```
char plotchar;
```

The member **plotchar** represents plot points on line graphs and scatter diagrams. Each palette uses a different character to distinguish plot points between data series.

10.5 Customizing the Chart Environment

The presentation graphics functions are designed to be flexible. You can use the system of default values to produce professional-looking charts with a minimum of programming effort. Or you can fine-tune the appearance of your charts by overriding default values and initializing variables explicitly in your program.

The header file PGCHART.H defines a structure type **_chartenv**, which organizes the chart environment variables. The chart environment describes everything about a chart except the plots themselves. It is the blank page, in other words, ready for plotting data. The environment determines the appearance of text, axes, grid lines, and legends.

Colors and line styles in the chart environment are taken from palettes. In this way, the appearance of titles and axis lines matches the colors and line styles of plotted data series.

You can reset any variable in the environment.

Calling the **_pg_defaultchart** function fills the chart environment with default values. Presentation graphics allows you to reset any variable in the environment before displaying a chart. Except for adjusting the palette values, all initialization of data is done through a **_chartenv** type structure.

The sample chart programs provided in "Writing a Presentation Graphics Program," on page 205 illustrate how to adjust variables in the chart environment. These programs create a structure `env` of type **_chartenv**. The structure `env` contains the chart environment variables, initialized by the call to the **_pg_defaultchart** function. Environment variables such as the chart title are then given specific values, as in

```
strcpy( env.maintitle.title, "Good Neighbor Grocery" );
```

Environment variables that determine colors and line styles deserve special mention. The chart environment holds several such variables, which can be recognized by their names. For example, the variable **titlecolor** specifies the color of title text. Similarly, the variable **gridstyle** specifies the line style used to draw the chart grid.

These variables are index numbers, but do not refer directly to the color pool or line pool. They correspond instead to palette numbers. If you set **titlecolor** to 2, presentation graphics uses the color code in the second palette to determine the title's color. Thus, the title in this case would be the same color as the chart's second data series. If you change the color code in the palette, you'll also change the title's color.

A structure of type **_chartenv** consists of four types of secondary structures. The file PGCHART.H type-defines these secondary structures: **_titletype**, **_axistype**, **_windowtype**, and **_legendtype**.

The remainder of this section describes the chart environment of presentation graphics. It first examines structures of the four secondary structures that make up the chart environment structure. The section concludes with a description of the **_chartenv** structure type. Each section begins with a brief explanation of the structure's purpose, followed by a listing of the structure type definition as it appears in the PGCHART.H file. All symbolic constants are defined in the file PGCHART.H.

_titletype Structures

Structures of type **_titletype** determine text, color, and placement of titles appearing in the graph. The PGCHART.H file defines the structure type as

```
typedef struct
{
   char     title[_PG_TITLELEN];  /* Title text */
   short    titlecolor;           /* Palette color
                                     for title text */
   short    justify;              /* _PG_LEFT, _PG_CENTER,
                                     _PG_RIGHT */
} _titletype;
```

The following list describes **_titletype** members:

justify
An integer specifying how the title is justified within the chart window. The symbolic constants defined in PGCHART.H for this variable are **_PG_LEFT**, **_PG_CENTER**, and **_PG_RIGHT**.

titlecolor
An integer between 1 and **_PG_PALETTELEN** that specifies a title's color. The default value for *titlecolor* is 1.

title[_PG_TITLELEN]
A character array containing title text. For example, if `env` is a structure of type **_chartenv**, then `env.maintitle.title` holds the character string used for the main title of the chart. Similarly, `env.xaxis.axistitle.title` contains the *x* axis title. The number of characters in a title must be one less than **_PG_TITLELEN** to allow room for a null terminator.

_axistype Structures

Structures of type **_axistype** contain variables for the axes such as color, scale, grid style, and tick marks. The PGCHART.H file defines the structure type as the following:

```
typedef struct
{
   short       grid;          /* TRUE=grid lines drawn;
                                 FALSE=no lines */
   short       gridstyle;     /* Style bytes for grid */
   _titletype   axistitle;    /* Title definition
                                 for axis */
   short       axiscolor;     /* Color for axis */
   short       labeled;       /* TRUE=ticks marks and titles
                                 drawn */
   short       rangetype;     /* _PG_LINEARAXIS,
                                 _PG_LOGAXIS */
   float       logbase;       /* Base used if log axis */
   short       autoscale;     /* TRUE=next 7 values
                                 calculated by system */
   float       scalemin;      /* Minimum value of scale */
   float       scalemax;      /* Maximum value of scale */
   float       scalefactor;   /* Scale factor for data on
                                 this axis */
   _titletype   scaletitle;   /* Title definition for
                                 scaling factor */
   float       ticinterval;   /* Distance between tick marks
                                 (world coord.) */
   short       ticformat;     /* _PG_EXPFORMAT or
                                 _PG_DECFORMAT */
   short       ticdecimals;   /* Number of decimals for tick
                                 labels (max=9) */
} _axistype;
```

The following list describes **_axistype** member variables:

autoscale
A Boolean variable. If *autoscale* is set to **TRUE**, presentation graphics automatically determines values for *scalefactor*, *scalemax*, *scalemin*, *scaletitle*, *ticdecimals*, *ticformat*, and *ticinterval* (see below). If *autoscale* equals **FALSE**, these seven variables must be specified in your program.

axiscolor
An integer between 1 and **_PG_PALETTELEN** that specifies the color used for the axis and parallel grid lines. (See description for *gridstyle* below.) Note that this member does not determine the color of the axis title. That selection is made through the **axistitle** structure.

axistitle
A **_titletype** structure that defines the title of the associated axis. The title of the *y* axis displays vertically to the left of the *y* axis, and the title of the *x* axis displays horizontally below the *x* axis.

grid
A Boolean true/false value that determines whether grid lines are drawn for the associated axis. Grid lines span the data window perpendicular to the axis.

gridstyle
An integer between 1 and **_PG_PALETTELEN** that specifies the grid's line style. Lines can be solid, dashed, dotted, or some combination. The default value for *gridstyle* is 1.

Note that the color of the parallel axis determines the color of the grid lines. Thus, the *x* axis grid is the same color as the *y* axis, and the *y* axis grid is the same color as the *x* axis.

labeled
A Boolean value that determines whether tick marks and labels are drawn on the axis. Axis labels should not be confused with axis titles. Axis labels are numbers or descriptions such as "23.2" or "January" attached to each tick mark.

logbase
If *rangetype* is logarithmic,, the *logbase* variable determines the log base used to scale the axis. The default value is 10.

rangetype
An integer that determines whether the scale of the axis is linear or logarithmic. The variable *rangetype* applies only to value data.

Specify a linear scale with **_PG_LINEARAXIS**. A linear scale is best when the difference between axis minimum and maximum is relatively small. For example, a linear axis range 0 – 10 results in 10 tick marks evenly spaced along the axis.

Use **_PG_LOGAXIS** to specify a logarithmic *rangetype*. Logarithmic scales are useful when the range is very large or when the data varies exponentially. Line graphs of exponentially varying data can be made straight with a logarithmic *rangetype*.

scalefactor
All numeric data are scaled by dividing each value by *scalefactor*. For relatively small values, *scalefactor* should be 1, which is the default. But data with large values should be scaled by an appropriate factor. For example, data in the range 2 million – 20 million should be plotted with *scalemin* set to 2, *scalemax* set to 20, and *scalefactor* set to 1 million.

If *autoscale* is set to **TRUE**, presentation graphics automatically determines a suitable value for *scalefactor* based on the range of data to be plotted. Presentation graphics selects only values that are a factor of 1 thousand—that is, values such as 1 thousand, 1 million, or 1 billion. It then labels the *scaletitle* appropriately (see the following). If you desire some other value for scaling, you must set *autoscale* to **FALSE** and set *scalefactor* to the desired scaling value.

scalemax
Highest value represented by the axis.

scalemin
Lowest value represented by the axis.

scaletitle
A **_titletype** structure defining a string of text that describes the value of *scalefactor*. If *autoscale* is **TRUE**, presentation graphics automatically writes a scale description to *scaletitle*. If *autoscale* equals **FALSE** and *scalefactor* is 1, *scaletitle.title* should be blank. Otherwise your program should copy an appropriate scale description to *scaletitle.title*, such as "(× 1000)," "(in millions of units)," or "times 10 thousand dollars."

For the *y* axis, the *scaletitle* text displays vertically between the axis title and the *y* axis. For the *x* axis, the scale title appears below the *x* axis title.

ticdecimals
Number of digits to display after the decimal point in tick labels. Maximum value is 9. (This variable applies only to axes with value data and is ignored for the category axis.)

ticformat
An integer that determines format of the labels assigned to each tick mark. Set *ticformat* to **_PG_EXPFORMAT** for exponential format or to **_PG_DECFORMAT** for decimal. The default is **_PG_DECFORMAT**. (This variable applies only to axes with value data and is ignored for the category axis.)

ticinterval
Sets interval between tick marks on the axis. The tick interval is measured in the same units as the numeric data associated with the axis. For example, if 2 sequential tick marks correspond to the values 20 and 25, the tick interval between them is 5. (This variable applies only to axes with value data and is ignored for the category axis.)

_windowtype Structures

Structures of type **_windowtype** contain sizes, locations, and color codes for the three windows produced by presentation graphics: the chart window, the data window, and the legend. Windows are located on the screen relative to the screen's logical origin. By changing the logical origin, you can display charts that are partly or completely off the screen.

The PGCHART.H file defines **_windowtype** as the following:

```
typedef struct
{
   short  x1;            /* Left edge of window in
                            pixels */
   short  y1;            /* Top edge of window in
                            pixels */
   short  x2;            /* Right edge of window in
                            pixels */
   short  y2;            /* Bottom edge of window in
                            pixels */
   short  border;        /* TRUE for border, FALSE
                            otherwise */
   short  background;    /* Internal palette color for
                            window background */
   short  borderstyle;   /* Style bytes for window
                            border */
   short  bordercolor;   /* Internal palette color for
                            window border */
} _windowtype;
```

The following list describes **_windowtype** member variables:

background
An integer between 1 and **_PG_PALETTELEN** that specifies the window's background color. The default value for *background* is 1.

border
A Boolean variable that determines whether a border frame is drawn around a window.

bordercolor
An integer between 1 and **_PG_PALETTELEN** that specifies the color of the window's border frame. The default value is 1.

borderstyle
An integer between 1 and **_PG_PALETTELEN** that specifies the line style of the window's border frame. The default value is 1.

x1, y1, x2, y2
Window coordinates in pixels. The ordered pair (*x1*, *y1*) specifies the coordinate of the upper-left corner of the window. The ordered pair (*x2*, *y2*) specifies the coordinate of the lower-right corner.

The reference point for the coordinates depends on the type of window. The chart window is located relative to the logical origin, usually the upper-left corner of the screen. The data and legend windows are located relative to the upper-left corner of the chart window. This allows you to change the position of the chart window without having to redefine coordinates for the other two windows.

_legendtype Structures

Structures of type **_legendtype** contain size, location, and colors of the chart legend. The PGCHART.H file defines the structure type as the following:

```
typedef struct
{
   short       legend;          /* TRUE=draw legend;
                                   FALSE=no legend */
   short       place;           /* _PG_RIGHT, _PG_BOTTOM,
                                   _PG_OVERLAY */
   short       textcolor;       /* Palette color for text*/
   short       autosize;        /* TRUE=system calculates
                                   legend size */
   _windowtype legendwindow;    /* Window definition for
                                   legend */
} _legendtype;
```

The following list describes **_legendtype** member variables:

autosize
A Boolean true/false variable that determines whether presentation graphics is to automatically calculate the size of the legend. If *autosize* equals **FALSE**, the legend window must be specified in the *legendwindow* structure (see the following).

legend
A Boolean true/false variable that determines whether a legend is to appear on the chart. The *legend* variable is ignored by functions that graph single-series charts.

legendwindow
A **_windowtype** structure that defines coordinates, background color, and border frame for the legend. Coordinates given in *legendwindow* are ignored if *autosize* is set to **TRUE**.

place
An integer that specifies the location of the legend relative to the data window. Setting *place* equal to the constant **_PG_RIGHT** positions the legend to the right of the data window. Setting *place* to **_PG_BOTTOM** positions the legend below the data window. Setting *place* to **_PG_OVERLAY** positions the legend within the data window.

These settings influence the size of the data window. If *place* equals **_PG_RIGHT** or **_PG_BOTTOM**, presentation graphics automatically sizes the data window to accommodate the legend. If *place* equals **_PG_OVERLAY**, the data window is sized without regard to the legend.

textcolor
An integer between 1 and **_PG_PALETTELEN** that specifies the color of text within the legend window.

_chartenv Structures

A structure of type **_chartenv** defines the chart environment. The following listing shows that a **_chartenv** type structure consists almost entirely of structures of the four types described above.

The PGCHART.H file defines the **_chartenv** structure type as the following:

```
typedef struct
{
   short       charttype;      /* Chart type */
   short       chartstyle;     /* Chart style */
   _windowtype  chartwindow;   /* Window definition for
                                  overall chart */
   _windowtype  datawindow;    /* Window definition for data
                                  part of chart */
   _titletype   maintitle;     /* Main chart title */
   _titletype   subtitle;      /* Chart subtitle */
   _axistype    xaxis;         /* Definition for x axis */
   _axistype    yaxis;         /* Definition for y axis */
   _legendtype  legend;        /* Definition for legend */
} _chartenv;
```

Initialize the chart environment with the _pg_defaultchart function.

The data in a **_chartenv** type structure is initialized by calling the function **_pg_defaultchart**. If your program does not call **_pg_defaultchart**, it must explicitly define every variable in the chart environment—a tedious procedure. The recommended method for adjusting the appearance of your chart is to initialize variables for the proper chart type by calling the **_pg_defaultchart** function, and then to reassign selected environment variables such as titles.

The following list describes **_chartenv** member variables:

chartstyle
An integer that determines the style of the chart (see Table 10.2). Legal values for *chartstyle* are **_PG_PERCENT** and **_PG_NOPERCENT** for pie charts; **_PG_PLAINBARS** and **_PG_STACKEDBARS** for bar and column charts; and **_PG_POINTONLY** and **_PG_POINTANDLINE** for line graphs and scatter diagrams. This variable corresponds to the third argument for the **_pg_defaultchart** function.

charttype
An integer that determines the type of chart displayed. The value of *charttype* is **_PG_BARCHART**, **_PG_COLUMNCHART**, **_PG_LINECHART**, **_PG_SCATTERCHART**, or **_PG_PIECHART**. This variable corresponds to the second argument for the **_pg_defaultchart** function.

chartwindow
A **_windowtype** structure that defines the appearance of the chart window.

datawindow
A **_windowtype** structure that defines the appearance of the data window.

legend
A _**legendtype** structure that defines the appearance of the legend window.

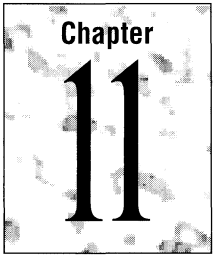
maintitle
A _**titletype** structure that defines the appearance of the main title of the chart.

subtitle
A _**titletype** structure that defines the appearance of the chart's subtitle.

xaxis
An _**axistype** structure that defines the appearance of the *x* axis. (This variable is not applicable for pie charts.)

yaxis
An _**axistype** structure that defines the appearance of the *y* axis. (This variable is not applicable for pie charts.)

Programming with Mixed Languages



There are times when your Microsoft 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 need 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.

11.1 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 would like 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 above, you would compile the main-module source file with the BASIC compiler, another source file (written in C) with the C compiler, and then link the two object files.

Figure 11.1 illustrates how the syntax of a mixed-language call works, using the instance mentioned above.

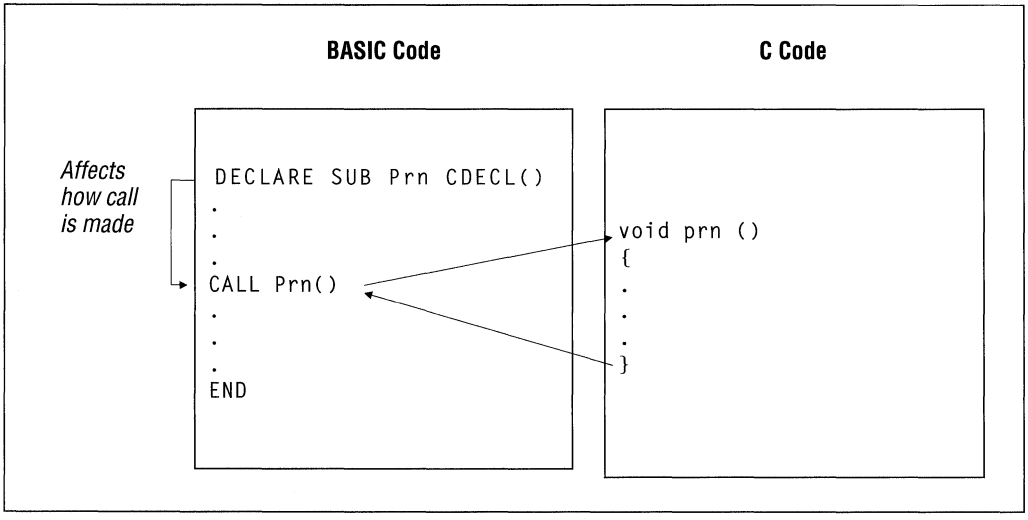


Figure 11.1 Mixed-Language Call

In Figure 11.1, 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 (which is described in detail in the *Microsoft BASIC Language Reference* and the *Microsoft BASIC Programmer's Guide*) is an example of a mixed-language "interface" statement. 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. (In this chapter, "routine" refers to any function, procedure, or subroutine that can be called from another module.)

Table 11.1 shows the correspondence between calls to routines in different languages.

Table 11.1 Language Equivalents for Routine Calls

Language	Return Value	No Return Value
Assembly Language	Procedure	Procedure
BASIC	FUNCTION procedure	Subprogram
C/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.

11.2 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 convention)
- How the subprogram call is implemented (calling convention)
- How parameters are passed (parameter-passing convention)

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 the routine before placing it in an object file. Languages may alter the identifier names differently. You can choose between 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 will not be able to find a match. It will instead report 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. The linker can then 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 (American Standard Code for Information Interchange) 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 will recognize, additional characters are simply not placed in the object file.

Note Versions of Microsoft FORTRAN previous to version 5.0 truncated identifiers to six 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.

The C compiler does not translate any letters to uppercase. It inserts a leading underscore (_) in front of the name of each routine. The C compiler recognizes the first 31 characters of a name (or 32 including the underscore). You can change the number of characters it recognizes with the /H option; see Chapter 13, "CL Command Reference," in the *Environment and Tools* manual for more information.

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 six 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 (generate Pascal-style function calls) when you compile, or if you declare a function or variable with the **__pascal** keyword, the compiler will translate your identifiers to uppercase.

Figure 11.2 illustrates a complete mixed-language development example, showing how naming conventions enter into the process.

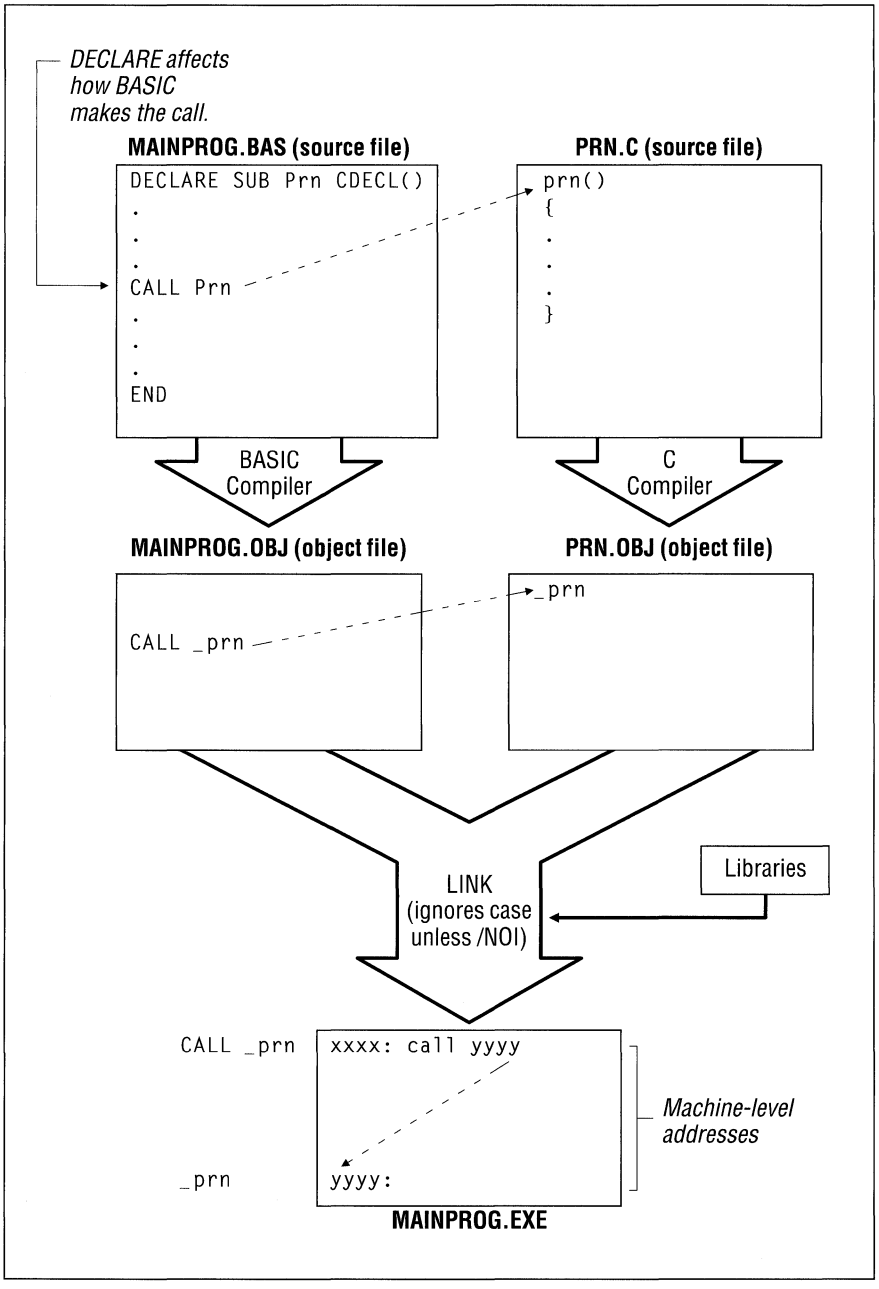


Figure 11.2 Naming Convention

In Figure 11.2, 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 will also convert 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 would use its own convention to call another C function, and would use 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 were 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 **__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 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:

Method	Description
Near reference	Passes a variable's near (offset) address. This address is expressed as an offset from the default data segment. This 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	Passes a variable's far (segmented) address. This 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.)
Value	Passes only the variable's value, not its address. With this 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 affect 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 that allow you to change parameter-passing methods.
- You may want to choose a specific parameter-passing method rather than using the defaults of any language.

Table 11.2 summarizes the parameter-passing defaults for each language.

Table 11.2 Parameter-Passing Defaults

Language	Near Reference	Far Reference	By Value
BASIC	All	—	—
C/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.

11.3 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 /A*X* command-line options), 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 the /A*X* command-line option to specify 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 4, "Managing Memory in C," and Chapter 5, "Managing Memory in C++."

Linking with Language Libraries

In most cases, you can easily link modules compiled with different languages. 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 cases above, 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 (no default libraries) 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 of the two language products 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
```

11.4 C Calls to High-Level Languages

Just as you can call Microsoft C routines from other Microsoft languages, you can call routines written in Microsoft FORTRAN and Pascal 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 257.

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 Pascal 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 example below 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 example below 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, since it indicates that the function returns no value.

```
void ( __fortran * func )( long larg );
```

The example below 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 example below 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 );
```

11.5 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 will perform 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 11.3 illustrates how to do this.

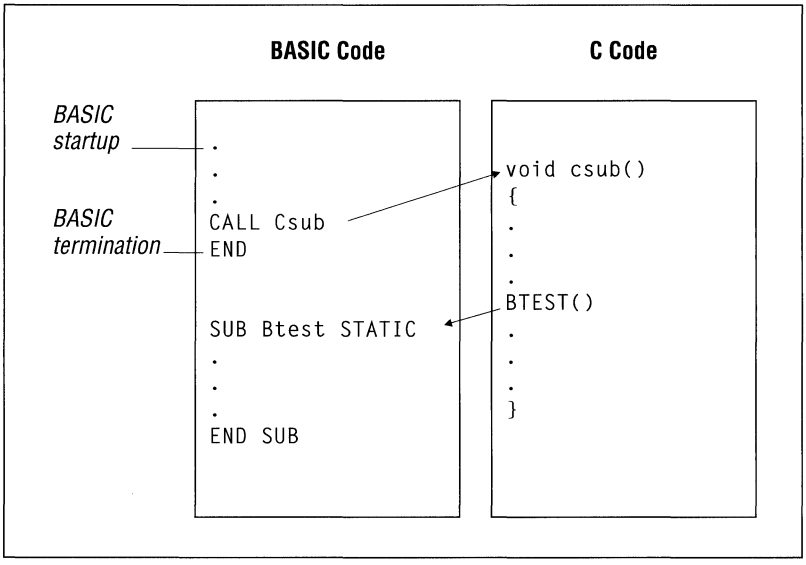


Figure 11.3 C Call to BASIC

Follow these rules when you call BASIC from C:

1. Start up in a BASIC main module. You will need to 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 are 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 are in the default data segment. If you want to pass data that are 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 example below 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, since 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 QuickBASIC later than 4.0 provide a "user entry point," **B_OnExit**, which can be called directly from C. The **B_OnExit** function enables you to make sure you have performed an orderly termination. The following code 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.     */
```

11.6 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, while subroutines do not. The examples in the next sections illustrate how to handle the difference between function and subroutine calls.

Calling a FORTRAN Subroutine from C

The example below 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, 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.
- Since 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 could have been specified by using far pointers in C. In that case, you would not 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 example below 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 example above, 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.

11.7 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, while procedures do not. The examples in the next sections 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 params, 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 example above, 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.
- Since 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 could have been 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 example below demonstrates a C main module calling 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 example above, 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 will corrupt the parameter's value in the calling module. Passing by value is the default method for both C and Pascal.

11.8 C Calls to Assembly Language

In Microsoft C/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 6, "Using the 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 is a list of 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 will be able to write recursive procedures (procedures that call themselves), and you will be able to use the CodeView debugger to locate errors in the code.

Note This section discusses only the simplified segment directives provided with the Microsoft Macro Assembler, 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 the assembly-language procedure with the 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 models, **FAR** for medium, large, or huge models).

  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 models.
- 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 will want to 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 the instructions:

```
push    bp
mov     bp,sp
```

To set up the stack frame in a 32-bit program, issue the 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 are 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 instruction sequence above preserves the value of BP, since it will be needed in the calling procedure as soon as your assembly-language procedure returns. It then transfers the value in SP 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 example above, `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 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 will place incorrect data in registers. Follow the rules below 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. So, in the preceding example, 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 11.4 shows how functions that observe the C calling convention use the stack frame.

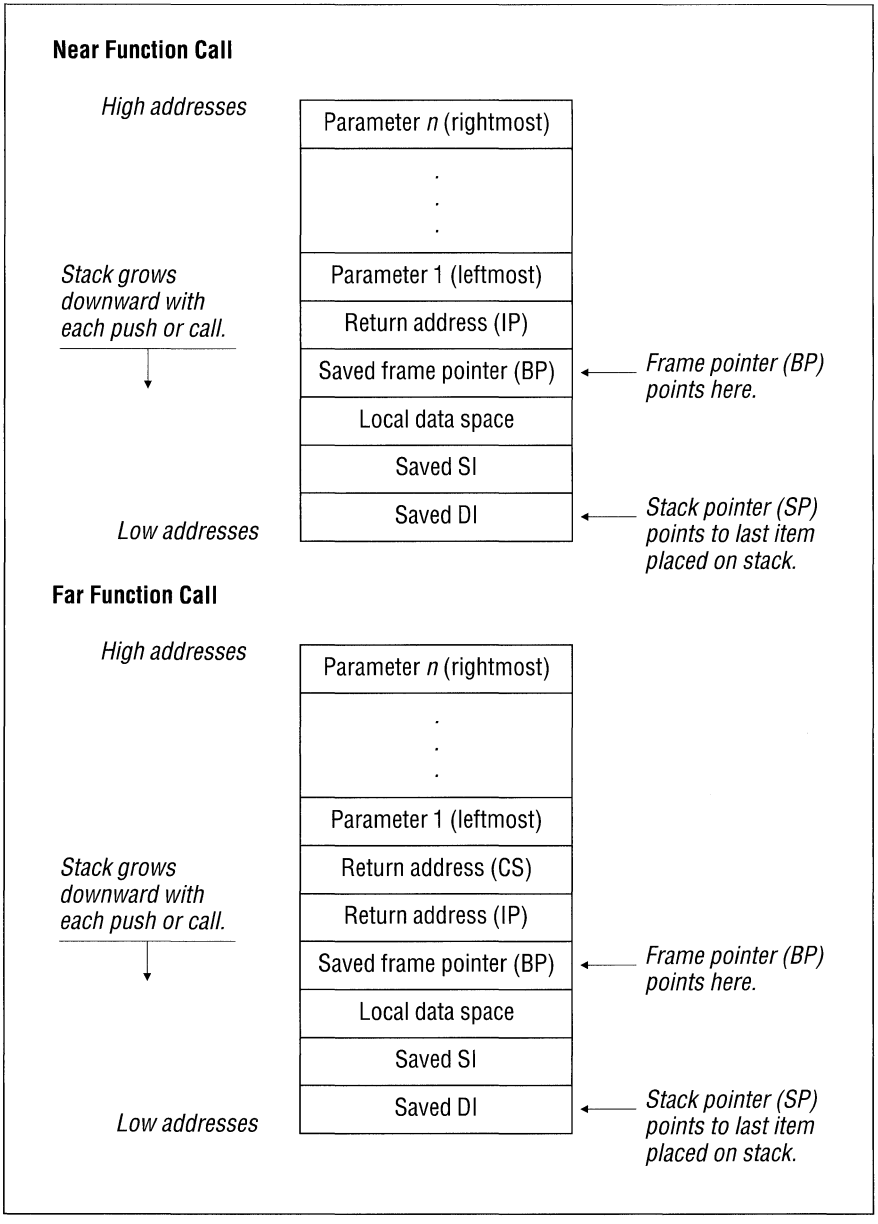


Figure 11.4 C Stack Frame

The stack frame for the assembly-language procedure shown in Figure 11.4 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 will ultimately return) to be placed on the stack. This address can be either two bytes long (for near calls) or four 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 return address plus 2 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** (two 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]
```

Once 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, Microsoft C pushes the segment address before pushing the offset address. When pushing arguments larger than two bytes, high-order words are always pushed before low-order words, and parameters larger than two bytes are stored on the stack in most-significant, least-significant order.

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 four bytes or less are passed in registers. Far pointers to return values larger than four 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 11.3 shows the register conventions for returning simple data types to a C program.

Table 11.3 Register Conventions for Simple Return Values

Data Type	Registers
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 (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 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 would a structure.

To return a value of type **long double**, you must place the value on the **NDP** (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. Some of these steps are dependent on 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 the reverse order from 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
```

11.9 C++ Calls to High-Level Languages

C++ lets you specify a linkage specification to permit communication between a C++ module and modules written in other languages. Microsoft C/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*.

11.10 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 (Table 11.4).

Table 11.4 Default Naming and Calling Conventions

Language	Calling Convention	Naming Convention	Parameter Passing
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 the *Microsoft BASIC Language Reference* or the *Microsoft BASIC Programmer's Guide.*

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 the *Microsoft FORTRAN Reference.*

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 the *Microsoft Pascal Compiler User's Guide.*

Numeric Data Representation

Table 11.5 shows how to declare numeric variables of similar type in different languages.

Table 11.5 Equivalent Numeric Data Types

BASIC	C/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 will be 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 not implemented in C. **LOGICAL*2** is stored as a one-byte Boolean indicator followed by an unused byte; **LOGICAL*4** is stored as a one-byte Boolean indicator followed by three unused bytes.

Strings

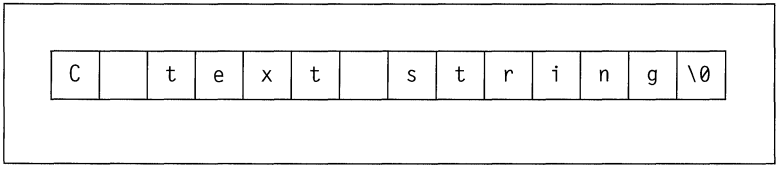
Each language implements strings differently. This section describes the ways that strings are implemented in Microsoft languages.

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 follows:

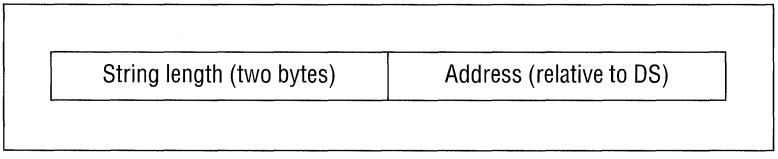


Because `c_string` is an array like any other, C and C++ pass it by reference in function calls.

Note that this does not apply to string classes written in C++.

BASIC String Format

BASIC stores strings as four-byte descriptors pointing to the actual string data. The format of the descriptor is as follows:



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,

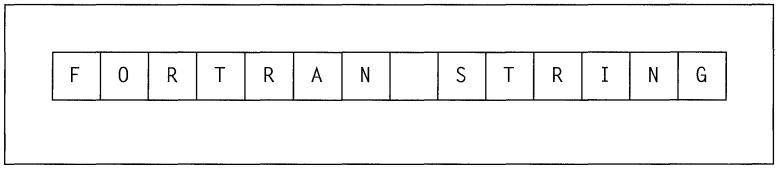
```
char c_string[] = "C String Data";

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 follows:



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 normally 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 259. 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 could change, and the address that was passed to the routine would become 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. Pascal lets you begin indexing at any integer value. Recent

versions of BASIC and FORTRAN also give you 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.

  This issue only affects 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 as well as the difference in the assumed lower bound between C and FORTRAN. Table 11.6 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 11.6 Equivalent Array Declarations

Language	Array Declaration	Notes
BASIC	**DIM** *x*(*r*–1, *c*–1)	With default lower bounds 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 bounds 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 will not be 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. See "Parameter-Passing Requirement" on page 235 for more information about the language you are using.

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. (See "Common Blocks," on page 267 for more information about sharing external data with FORTRAN programs.)

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 will not be 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 will 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 by value. Otherwise you will be 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 example below, 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 265. 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.

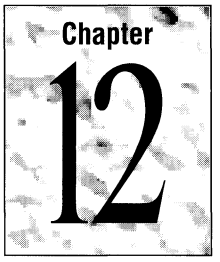
Note that the technique of accessing common blocks directly works with FORTRAN common blocks, but not with BASIC common blocks. If your C or C++ module must work with both FORTRAN and BASIC common blocks, pass the address of the common block as a parameter to the function..

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 normally 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.

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. Appendix B in the *C Language Reference* manual summarizes implementation-defined behavior for Microsoft C.

12.1 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. "Assumptions About the Compiler," on page 285 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. Normally, variables of type **char** are one 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 (two-byte) or a 32-bit quantity.
- An **int** is not always large enough to hold array indexes. 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 indexes 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 below, 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 below, 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
LDBL_MAX	Maximum value a variable of type **long double** can hold

Constant	Value
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

Microsoft C Type Sizes

Table 12.1 summarizes the size of the basic types in Microsoft C.

Table 12.1 Size of Basic Types in Microsoft 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
double	8
long double	10

* 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 one byte be aligned to two-byte or four-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` will be 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** will be 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 271.

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 );
}
```

Union Order and Alignment

Programmers use unions most often for two purposes: to store data whose exact type is not known until run time or 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" on page 277.)

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 machines. 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 eight 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 below:

```
#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 will be 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 will fail 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 );
```

Microsoft C Specific

Microsoft C normally aligns data types longer than one byte to an even-byte address for improved performance. See the /Zp compiler option in Chapter 13, "CL Command Reference," of the *Environment & Tools* manual; see the **pack** pragma in the *C Language Reference* or in Help for information about controlling structure packing in Microsoft C.

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 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 );
```

Microsoft C Specific

When performing disk input and output in Microsoft C, structures may be different sizes depending on the structure-packing option you have selected (see the /Zp

compiler option in Chapter 13, "CL Command Reference," of the *Environment and Tools* manual; see the **pack** pragma in the *C Language Reference* or in Help).

Bit Fields in Structures

The Microsoft C compiler implements bit fields. However, many C compilers do not.

Bit fields allow you to access the individual bits within a data item. While 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.) See "Storage Order and Alignment" on page 274 for more information.

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 < 1,024 */
};
```

The bit field `short_bitfield` is likely to be supported by more implementations than `long_bitfield`.

Microsoft 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, Microsoft C does not allow an element in a structure to cross a word boundary. The first two elements, `day` and `month`, take up nine 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. There is thus a seven-bit gap between the `month` and `year` elements in Microsoft C's 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 Microsoft C's 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 will not be portable. For example, two's-complement machines represent the eight-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 will fill a byte or a word with ones, and use it to construct a mask template that they later shift. This will not work correctly on one's-complement machines, but the error will 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.

Microsoft C Specific

Microsoft 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 error often corrupts memory unpredictably.

Casting Pointers

Be sure you do not make nonportable assumptions when casting 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 277.

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 machine 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 Microsoft 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 will cause 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**.

Microsoft 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 Microsoft 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 Microsoft 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 12.2 and 12.3, respectively.

Table 12.2 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 12.3 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 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 OS/2 or 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 under Windows can use an extraordinary amount of the global heap and not return it to the free pool. This type of behavior will cause 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 eight-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 code fragment 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 example above 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 machine.

Case Translation

Translation of characters from upper- to lowercase or from lower- 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 6. It is extremely efficient but nonportable. To write portable code, use the case-translation macros **toupper** and **tolower** (defined in CTYPE.H).

12.2 Assumptions About the Compiler

Different compilers translate C source code into object code in different ways. The ANSI draft 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, which can make your programs nonportable. For a complete description of how Microsoft C handles implementation-defined operations, see Appendix B of 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 code fragment 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` will be set to –3. If the implementation defines type **char** as unsigned, however, sign extension will not occur, and `i1` will be 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 will never evaluate 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 will cause sign extension on some machines. Consider the following code:

```
char c = 128;

printf( "%d\n", c );
```

Microsoft C Specific

Microsoft C allows you to 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 right bitwise 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 4,095.

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 right bitwise 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 2 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.

Microsoft 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**.

Since 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.

Microsoft 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.
PWB	Select the Global Register Allocation option in the Optimizations dialog box.
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* 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.

Microsoft C run-time libraries and macros allow you to use 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 fragment 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 will be 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 one of the following:
  - Logical AND (**&&**)
  - Logical OR (||)
  - Conditional (**?**)
  - Comma separator (**,**) when 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
  - 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 will not be 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` will be called only once, and its return value will be translated to uppercase. However, if **toupper** is implemented as a macro, `getc` will be called once or twice, depending on whether `c` is upper- 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` will always be 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 I/O. 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 dim text.

12.3 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.

12.4 Portability Concerns Specific to Microsoft C

Microsoft C offers extensions that let you take advantage of the full capabilities of the computer. These extensions are not portable to other compilers or environments. See Chapter 1, "Elements of C," of the *C Language Reference* for a list of Microsoft-specific keywords.

The *Run-Time Library Reference* contains compatibility information for every function in the run-time library. Any function or macro that does not have the ANSI box marked may not be portable to other compilers or computer systems.

12.5 Microsoft C Byte Ordering

Tables 12.4 and 12.5 summarize Microsoft 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.

Since byte ordering is machine specific, any program that uses this byte ordering will not be portable.

Table 12.4 Byte Ordering for Short Types

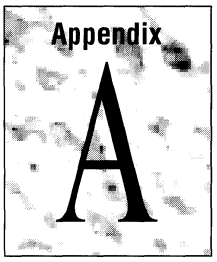
CPU	Byte Order
8086	b0 b1
80286	b0 b1
80386	b0 b1
PDP-11	b0 b1
VAX-11	b0 b1
M68000	b1 b0
Z8000	b1 b0

Table 12.5 Byte Ordering for Long Types

CPU	Byte Order
8086	b0 b1 b2 b3
80286	b0 b1 b2 b3
80386	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

Appendix

P-Code Instruction Tables



The following tables provide a complete list of one-byte and extended p-code instructions. Each instruction is represented by a specific hexadecimal number. For example, LDCW2 is in the row marked "60" and the column marked "1" or "9." Thus, it has a value of 61 hexadecimal. LDCW5 is in the same column but one row down, so it has a value of 69 hexadecimal.

The syntax for p-code instruction names is described in Chapter 3, "Reducing Program Size with P-Code." The p-code opcodes are fully described in the Help file PCODE.HLP.

Table A.1 One-Byte Instructions

	0	1	2	3	4	5	6	7
	8	9	A	B	C	D	E	F
0	NOP	LdfWb	LdfWw	IncfWwb	IncfWbb	IncgWbb	LdfW16	LdfW15
8	LdfW14	LdfW13	LdfW12	LdfW11	LdfW10	LdfW9	LdfW8	LdfW7
10	LdfW6	LdfW5	LdfW4	LdfW3	LdfW2	LdfW1	LdfW0	LdfAb
18	LdfAw	LdfA9	LdfA8	LdfA7	LdfA6	LdfA5	LdfA4	LdfA3
20	LdfA2	LdfA1	LdfA0	StfWb	StfWw	StfpWb	StfW13	StfW12
28	StfW11	StfW10	StfW9	StfW8	StfW7	StfW6	StfW5	StfW4
30	StfW3	StfW2	StfW1	StfW0	IncfWb	IncfW8	IncfW7	IncfW6
38	IncfW5	IncfW4	IncfW3	IncfW2	IncfW1	IncfW0	LdgWb	LdgW0
40	LdgW1	LdgW2	LdgW3	LdgW4	LdgW5	LdgW6	LdgW7	LdgW8
48	LdgW9	LdgW10	LdgW11	LdgW12	LdgW13	StfpLb	ClrfBb	LdgAb
50	LdgA0	LdgA1	LdgA2	LdgA3	LdgA4	LdinWb	LdinW0	LdinW1
58	LdinW2	LdinW3	LdinW4	LdinW5	LdinW6	LdinW7	LdcWub	LdcWw
60	LdcW0	LdcW2	LdcW4	LdcW6	LdcW8	LdcW10	LdcWm1	LdcW1
68	LdcW3	LdcW5	LdcW7	LdcW9	IncWb	IncW2	IncW4	IncW6
70	IncW8	IncW10	IncW12	IncW14	IncW16	IncW1	IncW3	CvtWuQb
78	RepW	Gotonb	AddW	SubW	LdinL0	StinL0	Quoteb	Equote
80	StinWb	LdinoW	StinoW	LdinoB	StinoB	StinW0	StinW1	StinW2
88	StinB0	StinpB0	StinQWb	CallfpWb	CallfpW2	CallfpW1	CallfpW0	CallfcWb
90	CallfcW2	CallfcW1	CallfcW0	CallfcV2	CallfcV1	CallfcV0	IncWw	CallfcL1
98	CallfcL0	CallfpV0	StfpBb	StgWb	StgWw	StgW0	StgW1	StgW2 StgW3
a0	StgW4	StgW5	StgW6	StgW7	LdinBb	LdinB1	LdinB3	LdinB0
a8	LdinB2	LdinB4	LdtW	SttW	LdintA	StintW	LdintB	StintB
b0	Gotob	Gotow	Goto2	Goto4	Goto6	Goto1	Goto3	Goto5
b8	JeqWb	JzrWb	JzrW2	JzrW4	JzrW6	JzrW8	JzrW1	JzrW3
c0	JzrW5	JzrW7	JneWb	JneW2	JneW4	JneW1	JneW3	JneW5
c8	JnzWb	JnzW2	JnzW4	IncfBb	INT3	SetfBb	JleWb	JltWb
d0	JgtWb	JgeWb	DecW1	AndW	OrW	ShlW	ShlW1	ShrW
d8	MulW	DivW	DropW	NegW	--	Ret	CaseW	CmpuW
e0	SwapW	AddWP	CmpsL	AddL	SubL	CvtWL	Exo	--
e8	LdgBb	StgBb	PushIPb	DecWb	LdifBb	LdifWb	LdifLb	PopIP
f0	CvtBW	LdfBb	StfBb	LdgLb	JnzW1	JnzW3	JnzW5	LdfLb
f8	StfLb	LdpLb	LdpWb	StgLb	LdpW0	LdpW1	LdpW2	LdpW3

Table A.2 Extended Instructions

	0	1	2	3	4	5	6	7
	8	9	A	B	C	D	E	F
0	NOP	XorW	PushWb	PopWb	MulL	DivL	SarW	LdintW
8	RepL	--	MulWWL	DivLWW			BltNNB	BltFFB
10	StifBb	StifWb	StifLb	StinBb	LdihoB	StinLb	LdihoW	StihoW
18	LdinoL	StinoL	LdinLb	CmpuL	CmpD	StifQWb	CvtWsQb	RetMB
20	BltNNBN	BltFFBF	--	-- --	--	--	--	--
28	AddHLH	LdtL	LdiftW	StiftW	LdiftB	StiftB		LdihrW
30	LdifoB	StifoB	LdcL1	StifoW	LdifoL	CallfcVb	--	--
38	RemuW	DivuW	NegL	RemuL	DivuL	CallfcLb	NotW	RemW
40	RemL	AndL	OrL	ShlL	ShlL1	ShrL	NotL	XorL
48	SarL	CaseL	OverW	CalinpV	CallifpV	CallincV	CallifcV	CallinpW
50	CallifpW	CallincW	CallifcW	CallinpL	CallifpL	CallincL	CallifcL	PushDS
58	PushCS	PushSS	--	CallnpVb	CallnpWb	CallnpLb	CallncVb	CallncWb
60	CallncLb	--	DivB	RemB	IncfWw		CallfpV2	CallfpV1
68	CallfpV0	CallfpLb	CallfpL2	CallfpL1	CallfpL0	Quotew	JeqW3	JltW3
70	JgtW3	JleW3	JgeW3	LdfgA	--	--	--	IncfBw
78	LdfBw	StfBw	LdfLw	StfLw	--	--	--	--
80	IncfWbw	IncfWww	IncfBbb	IncfFbb	IncfBwb	IncfFbw	IncfLbb	IncfLbw
88	IncfLbl	IncfLwb	IncfLww	IncfLwl	IncgBbb	IncgFbb	IncgWbw	IncgLbb
90	IncgLbw	IncgLbl	IncfFwb	IncfFww	IncfHbb	IncfHbw	IncfHwb	IncfHww
98	IncgFbw	IncgHbb	IncgHbw	PushIPw	--	--	--	--
a0	--	--	--	--	--	--	--	--
a8	--	--	--	--	--	--	--	--
b0	--	--	--	--	--	--	--	--
b8	--	--	--	--	--	--	--	--
c0	--	--	--	--	--	--	--	--
c8	LdcRr	LdcDd	--	--	--	--	--	--
d0	LdcTt	LdfRw	LdfDw	LdfTw	LdgRb	LdgDb	LdgTb	LdinR
d8	LdinD	LdinT	StfRw	StfDw	StfTw	StgRb	StgDb	StgTb
e0	StinR	StinD	StinT	PushR	PushD	PushT	SpillTb	UnspillTb
e8	CvtWT	CvtLT	CvtTW	CvtTL	AddT	SubT	MulT	DivT
f0	NegT	CmpT	DupT	SwapT	DropT	LdifR	LdifD	LdifT
f8	StifR	StifD	StifT	PopR	CvtuWT	CvtuLT	CvtTuW	CvtTuL

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399

Microsoft®

1191 Part No. 24775