Chpater 21 Advanced Compiler Use

21-1. Programming with Multiple Source Files

Until now, all your C programs have consisted of a single source-code file, exclusive of header files. A single source-code file is often all you need, particularly for small programs, but you can also divide the source code for a single program among two or more files, a practice called *modular programming*. Why would you want to do this? The following sections explain.

21-1-1. Advantages of Modular Programming

The primary reason to use modular programming is closely related to structured programming and its reliance on functions. As you become a more experienced programmer, you develop more general-purpose functions that you can use, not only in the program for which they were originally written, but in other programs as well. For example, you might write a collection of general-purpose functions for displaying information on-screen. By keeping these functions in a separate file, you can use them again in different programs that also display information on-screen. When you write a program that consists of multiple source-code files, each source file is called a *module*.

21-1-2. Modular Programming Techniques

A C program can have only one main() function. The module that contains the main() function is called the *main module,* and other modules are called *secondary modules*. A separate header file is usually associated with each secondary module (you'll learn why later in this chapter). For now, look at a few simple examples that illustrate the basics of multiple module programming. Listings 21.1, 21.2, and 21.3 show the main module, the secondary module, and the header file, respectively, for a program that inputs a number from the user and displays its square.

Listing 21.1. LIST21_1.C: the main module.

```
1: /* Inputs a number and displays its square. */
2:
3: #include <stdio.h>
4: #include "calc.h"
5:
6: main()
7: {
8:    int x;
9:
10:    printf("Enter an integer value: ");
11:    scanf("%d", &x);
12:    printf("\nThe square of %d is %ld.\n", x, sqr(x));
13:    return(0);
14: }
```

Listing 21.2. CALC.C: the secondary module.

```
1: /* Module containing calculation functions. */
2:
3: #include "calc.h"
4:
5: long sqr(int x)
6: {
7:    return ((long)x * x);
```

```
8: }
```

Listing 21.3. CALC.H: the header file for CALC.C.

```
1: /* CALC.H: header file for CALC.C. */
2:
3: long sqr(int x);
4:
5: /* end of CALC.H */
Enter an integer value: 100
The square of 100 is 10000.
```

ANALYSIS: Let's look at the components of these three files in greater detail. The header file, CALC.H, contains the prototype for the sqr() function in CALC.C. Because any module that uses sqr() needs to know sqr()'s prototype, the module must include CALC.H.

The secondary module file, CALC.C, contains the definition of the sqr() function. The #include directive is used to include the header file, CALC.H. Note that the header filename is enclosed in quotation marks rather than angle brackets. (You'll learn the reason for this later in this chapter.)

The main module, LIST21_1.C, contains the main() function. This module also includes the header file, CALC.H.

After you use your editor to create these three files, how do you compile and link the final executable program? Your compiler controls this for you. At the command line, enter

```
xxx list21_1.c calc.c
```

where *xxx* is your compiler's command. This directs the compiler's components to perform the following tasks:

1. Compile LIST21_1.C, creating LIST21_1.OBJ (or LIST21_1.O on a UNIX system). If it encounters any errors, the compiler displays descriptive error messages.
2. Compile CALC.C, creating CALC.OBJ (or CALC.O on a UNIX system). Again, error messages appear if necessary.
3. Link LIST21_1.OBJ, CALC.OBJ, and any needed functions from the standard library to create the final executable program LIST21_1.EXE.

21-1-3. Module Components

As you can see, the mechanics of compiling and linking a multiple-module program are quite simple. The only real question is what to put in each file. This section gives you some general guidelines.

The secondary module should contain general utility functions--that is, functions that you might want to use in other programs. A common practice is to create one secondary module for each type of function--for example, KEYBOARD.C for your keyboard functions, SCREEN.C for your screen display functions, and so on. To compile and link more than two modules, list all source files on the command line:

```
tcc mainmod.c screen.c keyboard.c
```

The main module should contain main(), of course, and any other functions that are program-specific (meaning that they have no general utility).

There is usually one header file for each secondary module. Each file has the same name as the associated module, with an .H extension. In the header file, put

- Prototypes for functions in the secondary module

- #define directives for any symbolic constants and macros used in the module
- Definitions of any structures or external variables used in the module

Because this header file might be included in more than one source file, you want to prevent portions of it from compiling more than once. You can do this by using the preprocessor directives for conditional compilation (discussed later in this chapter).

21-1-4. External Variables and Modular Programming

In many cases, the only data communication between the main module and the secondary module is through arguments passed to and returned from the functions. In this case, you don't need to take special steps regarding data visibility, but what about an external variable that needs to be visible in both modules?

Recall from Day 12, "Understanding Variable Scope," that an external variable is one declared outside of any function. An external variable is visible throughout the entire source code file in which it is declared. However, it is not automatically visible in other modules. To make it visible, you must declare the variable in each module, using the extern keyword. For example, if you have an external variable declared in the main module as

```
float interest_rate;
```

you make interest_rate visible in a secondary module by including the following declaration in that module (outside of any function):

```
extern float interest_rate;
```

The extern keyword tells the compiler that the original declaration of interest_rate (the one that set aside storage space for it) is located elsewhere, but that the variable should be made visible in this module. All extern variables have static duration and are visible to all functions in the module. Figure 21.1 illustrates the use of the extern keyword in a multiple-module program.

In Figure 21.1, the variable x is visible throughout all three modules. In contrast, y is visible only in the main module and secondary module 1.

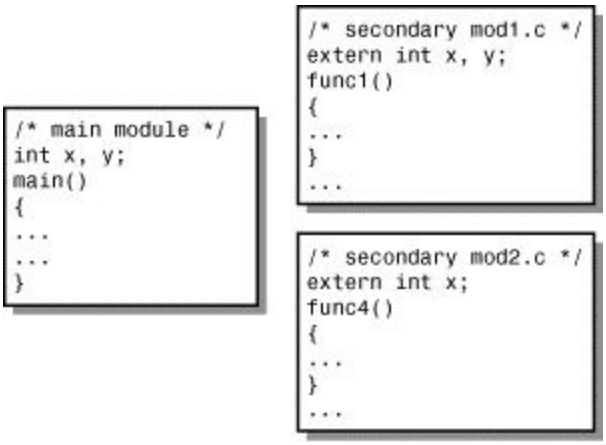

Figure 21-1. Using the extern keyword to make an external variable visible across modules.

21-1-5. Using .OBJ Files

After you've written and thoroughly debugged a secondary module, you don't need to recompile it every time you use it in a program. Once you have the object file for the module code, all you need to do is link it with each program that uses the functions in the module.

When you compile a program, the compiler creates an object file that has the same name as the C source code file and an .OBJ extension (or an .O extension on UNIX systems). For example, suppose you're developing a module called KEYBOARD.C and compiling it, along with the main module DATABASE.C, using the following command:

```
tcc database.c keyboard.c
```

The KEYBOARD.OBJ file is also on your disk. Once you know that the functions in KEYBOARD.C work properly, you can stop compiling it every time you recompile DATABASE.C (or any other program that uses it), linking the existing object file instead. To do this, use the command

```
tcc database.c keyboard.obj
```

The compiler then compiles DATABASE.C and links the resulting object file DATABASE.OBJ with KEYBOARD.OBJ to create the final executable file DATABASE.EXE. This saves time, because the compiler doesn't have to recompile the code in KEYBOARD.C. However, if you modify the code in KEYBOARD.C, you must recompile it. In addition, if you modify a header file, you must recompile all the modules that use it.

DON'T try to compile multiple source files together if more than one module contains a main() function. You can have only one main().

DO create generic functions in their own source files. This way, they can be linked into any other programs that need them.

DON'T always use the C source files when compiling multiple files together. If you compile a source file into an object file, recompile only when the file changes. This saves a great deal of time.

21-1-6. Using the make Utility

Almost all C compilers come with a make utility that can simplify and speed the task of working with multiple source-code files. This utility, which is usually called NMAKE.EXE, lets you write a so-called *make file* that defines the dependencies between your various program components. What does dependency mean?

Imagine a project that has a main module named PROGRAM.C and a secondary module named SECOND.C. There are also two header files, PROGRAM.H and SECOND.H. PROGRAM.C includes both of the header files, whereas SECOND.C includes only SECOND.H. Code in PROGRAM.C calls functions in SECOND.C.

PROGRAM.C is dependent on the two header files because it includes them both. If you make a change to either header file, you must recompile PROGRAM.C so that it will include those changes. In contrast, SECOND.C is dependent on SECOND.H but not on PROGRAM.H. If you change PROGRAM.H, there is no need to recompile SECOND.C--you can just link the existing object file SECOND.OBJ that was created when SECOND.C was last compiled.

A make file describes the dependencies such as those just discussed that exist in your project. Each time you edit one or more of your source code files, you use the NMAKE utility to "run" the make file. This utility examines the time and date stamps on the source code and object files and, based on the dependencies you have defined, instructs the compiler to recompile only those files that are dependent on the modified file(s). The result is that no unnecessary compilation is done, and you can work at the maximum efficiency.

For projects that involve one or two source code files, it usually isn't worth the trouble of defining a make file. For larger projects, however, it's a real benefit. Refer to your compiler documentation for information on how to use its NMAKE utility.

21-2. The C Preprocessor

The preprocessor is a part of all C compiler packages. When you compile a C program, the preprocessor is the first compiler component that processes your program. In most C compilers, the preprocessor is part of the compiler program. When you run the compiler, it automatically runs the preprocessor.

The preprocessor changes your source code based on instructions, or *preprocessor directives,* in the source code. The output of the preprocessor is a modified source code file that is then used as the input for the next compilation step. Normally you never see this file, because the compiler deletes it after it's used. However, later in this chapter you'll learn how to look at this intermediate file. First, you need to learn about the preprocessor directives, all of which begin with the # symbol.

21-2-1. The #define Preprocessor Directive

The #define preprocessor directive has two uses: creating symbolic constants and creating macros.

Simple Substitution Macros Using #define

You learned about substitution macros on Day 3, "Storing Data: Variables and Constants," although the term used to describe them in that chapter was *symbolic constants*. You create a substitution macro by using #define to replace text with other text. For example, to replace text1 with text2, you write

```
#define text1 text2
```

This directive causes the preprocessor to go through the entire source code file, replacing every occurrence of text1 with text2. The only exception occurs if text1 is found within double quotation marks, in which case no change is made.

The most frequent use for substitution macros is to create symbolic constants, as explained on Day 3. For example, if your program contains the following lines:

```
#define MAX 1000
x = y * MAX;
z = MAX - 12;
```

during preprocessing, the source code is changed to read as follows:

```
x = y * 1000;
z = 1000 - 12;
```

The effect is the same as using your editor's search-and-replace feature in order to change every occurrence of MAX to 1000. Your original source code file isn't changed, of course. Instead, a temporary copy is created with the changes. Note that #define isn't limited to creating symbolic numeric constants. For example, you could write

```
#define ZINGBOFFLE printf
ZINGBOFFLE("Hello, world.");
```

although there is little reason to do so. You should also be aware that some authors refer to symbolic constants defined with #define as being macros themselves. (Symbolic constants are also called *manifest constants*.) However, in this book, the word *macro* is reserved for the type of construction described next.

Creating Function Macros with #define

You can use the #define directive also to create function macros. A *function macro* is a type of shorthand, using something simple to represent something more complicated. The reason for the "function" name is that this type of macro can accept arguments, just like a real C function does. One advantage of function macros is that their arguments aren't type-sensitive. Therefore, you can pass any numeric variable type to a function macro that expects a numeric argument.

Let's look at an example. The preprocessor directive

```
#define HALFOF(value) ((value)/2)
```

defines a macro named HALFOF that takes a parameter named value. Whenever the preprocessor encounters the text HALFOF(value) in the source code, it replaces it with the definition text and inserts the argument as needed. Thus, the source code line

```
result = HALFOF(10);
```

is replaced by this line:

```
result = ((10)/2);
```

Likewise, the program line

```
printf("%f", HALFOF(x[1] + y[2]));
```

is replaced by this line:

```
printf("%f", ((x[1] + y[2])/2));
```

A macro can have more than one parameter, and each parameter can be used more than once in the replacement text. For example, the following macro, which calculates the average of five values, has five parameters:

```
#define AVG5(v, w, x, y, z) (((v)+(w)+(x)+(y)+(z))/5)
```

The following macro, in which the conditional operator determines the larger of two values, also uses each of its parameters twice. (You learned about the conditional operator on Day 4, "Statements, Expressions, and Operators.")

```
#define LARGER(x, y) ((x) > (y) ? (x) : (y))
```

A macro can have as many parameters as needed, but all of the parameters in the list must be used in the substitution string. For example, the macro definition

```
#define ADD(x, y, z) ((x) + (y))
```

is invalid, because the parameter z is not used in the substitution string. Also, when you invoke the macro, you must pass it the correct number of arguments.

When you write a macro definition, the opening parenthesis must immediately follow the macro name; there can be no white space. The opening parenthesis tells the preprocessor that a function macro is being defined and that this isn't a simple symbolic constant type substitution. Look at the following definition:

```
#define SUM (x, y, z) ((x)+(y)+(z))
```

Because of the space between SUM and (, the preprocessor treats this like a simple substitution macro. Every occurrence of SUM in the source code is replaced with (x, y, z) ((x)+(y)+(z)), clearly not what you wanted.

Also note that in the substitution string, each parameter is enclosed in parentheses. This is necessary to avoid unwanted side effects when passing expressions as arguments to the macro. Look at the following example of a macro defined without parentheses:

```
#define SQUARE(x) x*x
```

If you invoke this macro with a simple variable as an argument, there's no problem. But what if you pass an expression as an argument?

```
result = SQUARE(x + y);
```

The resulting macro expansion is as follows, which doesn't give the proper result:

```
result = x + y * x + y;
```

If you use parentheses, you can avoid the problem, as shown in this example:

```
#define SQUARE(x) (x)*(x)
```

This definition expands to the following line, which does give the proper result:

```
result = (x + y) * (x + y);
```

You can obtain additional flexibility in macro definitions by using the *stringizing operator* (#) (sometimes called the *string-literal operator*). When a macro parameter is preceded by # in the substitution string, the argument is converted into a quoted string when the macro is expanded. Thus, if you define a macro as

```
#define OUT(x) printf(#x)
```

and you invoke it with the statement

```
OUT(Hello Mom);
```

it expands to this statement:

```
printf("Hello Mom");
```

The conversion performed by the stringizing operator takes special characters into account. Thus, if a character in the argument normally requires an escape character, the # operator inserts a backslash before the character. Continuing with the example, the invocation

```
OUT("Hello Mom");
```

expands to

```
printf("\"Hello Mom\"");
```

The # operator is demonstrated in Listing 21.4. First, you need to look at one other operator used in macros, the *concatenation operator* (##). This operator concatenates, or joins, two strings in the macro expansion. It doesn't include quotation marks or special treatment of escape characters. Its main use is to create sequences of C source code. For example, if you define and invoke a macro as

```
#define CHOP(x) func ## x
salad = CHOP(3)(q, w);
```

the macro invoked in the second line is expanded to

```
salad = func3 (q, w);
```

You can see that by using the ## operator, you determine which function is called. You have actually modified the C source code.

Listing 21.4 shows an example of one way to use the # operator.

Listing 21.4. Using the # operator in macro expansion.

```
1: /* Demonstrates the # operator in macro expansion. */
2:
3: #include <stdio.h>
```

```
4:
5: #define OUT(x) printf(#x " is equal to %d.\n", x)
6:
7: main()
8: {
9:    int value = 123;
10:   OUT(value);
11:   return(0);
12: }
value is equal to 123.
```

ANALYSIS: By using the # operator on line 5, the call to the macro expands with the variable name value as a quoted string passed to the printf() function. After expansion on line 9, the macro OUT looks like this:

```
printf("value" " is equal to %d.",  value );
```

Macros Versus Functions

You have seen that function macros can be used in place of real functions, at least in situations where the resulting code is relatively short. Function macros can extend beyond one line but usually become impractical beyond a few lines. When you can use either a function or a macro, which should you use? It's a trade-off between program speed and program size.

A macro's definition is expanded into the code each time the macro is encountered in the source code. If your program invokes a macro 100 times, 100 copies of the expanded macro code are in the final program. In contrast, a function's code exists only as a single copy. Therefore, in terms of program size, the better choice is a true function.

When a program calls a function, a certain amount of processing overhead is required in order to pass execution to the function code and then return execution to the calling program. There is no processing overhead in "calling" a macro, because the code is right there in the program. In terms of speed, a function macro has the advantage.

These size/speed considerations aren't usually of much concern to the beginning programmer. Only with large, time-critical applications do they become important.

Viewing Macro Expansion

At times, you might want to see what your expanded macros look like, particularly when they aren't working properly. To see the expanded macros, you instruct the compiler to create a file that includes macro expansion after the compiler's first pass through the code. You might not be able to do this if your C compiler uses an Integrated Development Environment (IDE); you might have to work from the command prompt. Most compilers have a flag that should be set during compilation. This flag is passed to the compiler as a command-line parameter.

For example, to precompile a program named PROGRAM.C with the Microsoft compiler, you would enter

```
cl /E program.c
```

On a UNIX compiler, you would enter

```
cc -E program.c
```

The preprocessor makes the first pass through your source code. All header files are included, #define macros are expanded, and other preprocessor directives are carried out. Depending on your compiler, the output goes either to stdout (that is, the screen) or to a disk file with the program name and a special extension. The Microsoft compiler

sends the preprocessed output to stdout. Unfortunately, it's not at all useful to have the processed code whip by on your screen! You can use the redirection command to send this output to a file, as in this example:

```
cl /E program.c > program.pre
```

You can then load the file into your editor for printing or viewing.

DO use #defines, especially for symbolic constants. Symbolic constants make your code much easier to read. Examples of things to put into defined constants are colors, true/false, yes/no, the keyboard keys, and maximum values. Symbolic constants are used throughout this book.

DON'T overuse macro functions. Use them where needed, but be sure they are a better choice than a normal function.

21-2-2. The #include Directive

You have already learned how to use the #include preprocessor directive to include header files in your program. When it encounters an #include directive, the preprocessor reads the specified file and inserts it at the location of the directive. You can't use the * or ? wildcards to read in a group of files with one #include directive. You can, however, nest #include directives. In other words, an included file can contain #include directives, which can contain #include directives, and so on. Most compilers limit the number of levels deep that you can nest, but you usually can nest up to 10 levels.

There are two ways to specify the filename for an #include directive. If the filename is enclosed in angle brackets, such as #include <stdio.h> (as you have seen throughout this book), the preprocessor first looks for the file in the standard directory. If the file isn't found, or no standard directory is specified, the preprocessor looks for the file in the current directory.

"What is the standard directory?" you might be asking. In DOS, it's the directory or directories specified by the DOS INCLUDE environment variable. Your DOS documentation contains complete information on the DOS environment. To summarize, however, you set an environment variable with a SET command (usually, but not necessarily, in your AUTOEXEC.BAT file). Most compilers automatically set the INCLUDE variable in the AUTOEXEC.BAT file when the compiler is installed.

The second method of specifying the file to be included is enclosing the filename in double quotation marks: #include "myfile.h". In this case, the preprocessor doesn't search the standard directories; instead, it looks in the directory containing the source code file being compiled. Generally speaking, header files that you write should be kept in the same directory as the C source code files, and they are included by using double quotation marks. The standard directory is reserved for header files supplied with your compiler.

21-2-3. Using #if, #elif, #else, and #endif

These four preprocessor directives control conditional compilation. The term *conditional compilation* means that blocks of C source code are compiled only if certain conditions are met. In many ways, the #if family of preprocessor directives operates like the C language's if statement. The difference is that if controls whether certain statements are executed, whereas #if controls whether they are compiled.

The structure of an #if block is as follows:

```
#if condition_1
statement_block_1
#elif condition_2
```

```
statement_block_2
...
#elif condition_n
statement_block_n
#else
default_statement_block
#endif
```

The test expression that #if uses can be almost any expression that evaluates to a constant. You can't use the sizeof() operator, typecasts, or the float type. Most often you use #if to test symbolic constants created with the #define directive.

Each *statement_block* consists of one or more C statements of any type, including preprocessor directives. They don't need to be enclosed in braces, although they can be.

The #if and #endif directives are required, but #elif and #else are optional. You can have as many #elif directives as you want, but only one #else. When the compiler reaches an #if directive, it tests the associated condition. If it evaluates to TRUE (nonzero), the statements following the #if are compiled. If it evaluates to FALSE (zero), the compiler tests, in order, the conditions associated with each #elif directive. The statements associated with the first TRUE #elif are compiled. If none of the conditions evaluates as TRUE, the statements following the #else directive are compiled.

Note that, at most, a single block of statements within the #if...#endif construction is compiled. If the compiler finds no #else directive, it might not compile any statements.

The possible uses of these conditional compilation directives are limited only by your imagination. Here's one example. Suppose you're writing a program that uses a great deal of country-specific information. This information is contained in a header file for each country. When you compile the program for use in different countries, you can use an #if...#endif construction as follows:

```
#if ENGLAND == 1
#include "england.h"
#elif FRANCE == 1
#include "france.h"
#elif ITALY == 1
#include "italy.h"
#else
#include "usa.h"
#endif
```

Then, by using #define to define the appropriate symbolic constant, you can control which header file is included during compilation.

21-2-4. Using #if...#endif to Help Debug

Another common use of #if...#endif is to include conditional debugging code in the program. You could define a DEBUG symbolic constant set to either 1 or 0. Throughout the program, you can insert debugging code as follows:

```
#if DEBUG == 1
debugging code here
#endif
```

During program development, if you define DEBUG as 1, the debugging code is included to help track down any bugs. After the program is working properly, you can redefine DEBUG as 0 and recompile the program without the debugging code.

The defined() operator is useful when you write conditional compilation directives. This operator tests to see whether a particular name is defined. Thus, the expression

```
defined( NAME )
```

evaluates to TRUE or FALSE, depending on whether NAME is defined. By using defined(), you can control compilation, based on previous definitions, without regard to the specific value of a name. Referring to the previous debugging code example, you could rewrite the #if...#endif section as follows:

```
#if defined( DEBUG )
debugging code here
#endif
```

You can also use defined() to assign a definition to a name only if it hasn't been previously defined. Use the NOT operator (!) as follows:

```
#if !defined( TRUE )     /* if TRUE is not defined. */
#define TRUE 1
#endif
```

Notice that the defined() operator doesn't require that a name be defined as anything in particular. For example, after the following program line, the name RED is defined, but not as anything in particular:

```
#define RED
```

Even so, the expression defined(RED) still evaluates as TRUE. Of course, occurrences of RED in the source code are removed and not replaced with anything, so you must use caution.

21-2-5. Avoiding Multiple Inclusions of Header Files

As programs grow, or as you use header files more often, you run the risk of accidentally including a header file more than once. This can cause the compiler to balk in confusion. Using the directives you've learned, you can easily avoid this problem. Look at the example shown in Listing 21.5.

Listing 21.5. Using preprocessor directives with header files.

```
1: /* PROG.H - A header file with a check to prevent multiple
includes! */
2:
3. #if defined( PROG_H )
4: /* the file has been included already */
5: #else
6: #define PROG_H
7:
8: /* Header file information goes here... */
9:
10:
11:
12: #endif
```

ANALYSIS: Examine what this header file does. On line 3, it checks whether PROG_H is defined. Notice that PROG_H is similar to the name of the header file. If PROG_H is defined, a comment is included on line 4, and the program looks for the #endif at the end of the header file. This means that nothing more is done.

How does PROG_H get defined? It is defined on line 6. The first time this header is included, the preprocessor checks whether PROG_H is defined. It won't be, so control goes to the #else statement. The first thing done after the #else is to define PROG_H so that any other inclusions of this file skip the body of the file. Lines 7 through 11 can contain any number of commands or declarations.

21-2-6. The #undef Directive

The #undef directive is the opposite of #define--it removes the definition from a name. Here's an example:

```
#define DEBUG 1
/* In this section of the program, occurrences of DEBUG     */
/* are replaced with 1, and the expression defined( DEBUG ) */
/* evaluates to TRUE. *.
#undef DEBUG
/* In this section of the program, occurrences of DEBUG  */
/* are not replaced, and the expression defined( DEBUG ) */
/* evaluates to FALSE. */
```

You can use #undef and #define to create a name that is defined only in parts of your source code. You can use this in combination with the #if directive, as explained earlier, for more control over conditional compilations.

21-3. Predefined Macros

Most compilers have a number of predefined macros. The most useful of these are __DATE__, __TIME__, __LINE__, and __FILE__. Notice that each of these are preceded and followed by double underscores. This is done to prevent you from redefining them, on the theory that programmers are unlikely to create their own definitions with leading and trailing underscores.

These macros work just like the macros described earlier in this chapter. When the precompiler encounters one of these macros, it replaces the macro with the macro's code. __DATE__ and __TIME__ are replaced with the current date and time. This is the date and time the source file is precompiled. This can be useful information as you're working with different versions of a program. By having a program display its compilation date and time, you can tell whether you're running the latest version of the program or an earlier one.

The other two macros are even more valuable. __LINE__ is replaced by the current source-file line number. __FILE__ is replaced with the current source-code filename. These two macros are best used when you're trying to debug a program or deal with errors. Consider the following printf() statement:

View Code

If these lines were part of a program called MYPROG.C, they would print

```
Program MYPROG.C: (32) Error opening file
```

This might not seem important at this point, but as your programs grow and spread across multiple source files, finding errors becomes more difficult. Using __LINE__ and __FILE__ makes debugging much easier.

DO use the __LINE__ and __FILE__ macros to make your error messages more helpful.

DON'T forget the #endif when using the #if statement.

DO put parentheses around the value to be passed to a macro. This prevents errors. For example, use this:

```
#define CUBE(x)   (x)*(x)*(x)
```

instead of this:

```
#define CUBE(x)   x*x*x
```

21-4. Using Command-Line Arguments

Your C program can access arguments passed to the program on the command line. This refers to information entered after the program name when you start the program. If you start a program named PROGNAME from the C:\> prompt, for example, you could enter

```
C:\>progname smith jones
```

The two command-line arguments smith and jones can be retrieved by the program during execution. You can think of this information as arguments passed to the program's main() function. Such command-line arguments permit information to be passed to the program at startup rather than during execution, which can be convenient at times. You can pass as many command-line arguments as you like. Note that command-line arguments can be retrieved only within main(). To do so, declare main() as follows:

```
main(int argc, char *argv[])
{
/* Statements go here */
}
```

The first parameter, argc, is an integer giving the number of command-line arguments available. This value is always at least 1, because the program name is counted as the first argument. The parameter argv[] is an array of pointers to strings. The valid subscripts for this array are 0 through argc - 1. The pointer argv[0] points to the program name (including path information), argv[1] points to the first argument that follows the program name, and so on. Note that the names argc and argv[] aren't required--you can use any valid C variable names you like to receive the command-line arguments. However, these two names are traditionally used for this purpose, so you should probably stick with them.

The command line is divided into discrete arguments by any white space. If you need to pass an argument that includes a space, enclose the entire argument in double quotation marks. For example, if you enter

```
C:>progname smith "and jones"
```

smith is the first argument (pointed to by argv[1]), and and jones is the second (pointed to by argv[2]). Listing 21.6 demonstrates how to access command-line arguments.

Listing 21.6. Passing command-line arguments to main().

```
1: /* Accessing command-line arguments. */
2:
3: #include <stdio.h>
4:
5: main(int argc, char *argv[])
6: {
7:     int count;
8:
9:     printf("Program name: %s\n", argv[0]);
```

```
10:
11:     if (argc > 1)
12:     {
13:         for (count = 1; count < argc; count++)
14:             printf("Argument %d: %s\n", count, argv[count]);
15:     }
16:     else
17:         puts("No command line arguments entered.");
18:     return(0);
19: }
list21_6
Program name: C:\LIST21_6.EXE
No command line arguments entered.
list21_6 first second "3 4"
Program name: C:\LIST21_6.EXE
Argument 1: first
Argument 2: second
Argument 3: 3 4
```

ANALYSIS: This program does no more than print the command-line parameters entered by the user. Notice that line 5 uses the argc and argv parameters shown previously. Line 9 prints the one command-line parameter that you always have, the program name. Notice this is argv[0]. Line 11 checks to see whether there is more than one command-line parameter. Why more than one and not more than zero? Because there is always at least one--the program name. If there are additional arguments, a for loop prints each to the screen (lines 13 and 14). Otherwise, an appropriate message is printed (line 17).

Command-line arguments generally fall into two categories: those that are required because the program can't operate without them, and those that are optional, such as flags that instruct the program to act in a certain way. For example, imagine a program that sorts the data in a file. If you write the program to receive the input filename from the command line, the name is required information. If the user forgets to enter the input filename on the command line, the program must somehow deal with the situation. The program could also look for the argument /r, which signals a reverse-order sort. This argument isn't required; the program looks for it and behaves one way if it's found and another way if it isn't.

DO use argc and argv as the variable names for the command-line arguments for main(). Most C programmers are familiar with these names.

DON'T assume that users will enter the correct number of command-line parameters. Check to be sure they did, and if not, display a message explaining the arguments they should enter.

21-5. Summary

This chapter covered some of the more advanced programming tools available with C compilers. You learned how to write a program that has source code divided among multiple files or modules. This practice, called modular programming, makes it easy to reuse general-purpose functions in more than one program. You saw how you can use preprocessor directives to create function macros, for conditional compilation, and other tasks. Finally, you saw that the compiler provides some function macros for you.

Q&A

Q When compiling multiple files, how does the compiler know which filename to use for the executable file?
A You might think the compiler uses the name of the file containing the main() function; however, this isn't usually the case. When compiling from the command line, the first file listed is used to determine the name. For example, if you compiled the following with Borland's Turbo C, the executable would be called FILE1.EXE:

```
tcc file1.c main.c prog.c
```

Q Do header files need to have an .H extension?
A No. You can give a header file any name you want. It is standard practice to use the .H extension.
Q When including header files, can I use an explicit path?
A Yes. If you want to state the path where a file to be included is, you can. In such a case, you put the name of the include file between quotation marks.
Q Are all the predefined macros and preprocessor directives presented in this chapter?
A No. The predefined macros and directives presented in this chapter are ones common to most compilers. However, most compilers also have additional macros and constants.
Q Is the following header also acceptable when using main() with command-line parameters?

```
main( int argc, char **argv);
```

A You can probably answer this one on your own. This declaration uses a pointer to a character pointer instead of a pointer to a character array. Because an array is a pointer, this definition is virtually the same as the one presented in this chapter. This declaration is also commonly used. (See Day 8, "Using Numeric Arrays," and Day 10, "Characters and Strings," for more details.)

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. What does the term *modular programming* mean?
2. In modular programming, what is the main module?
3. When you define a macro, why should each argument be enclosed in parentheses?
4. What are the pros and cons of using a macro in place of a regular function?
5. What does the defined() operator do?
6. What must always be used if #if is used?
7. What extension do compiled C files have? (Assume that they have not been linked.)
8. What does #include do?
9. What is the difference between this line of code:

```
#include <myfile.h>
```

and this line of code:

```
#include "myfile.h"
```

10. What is __DATE__ used for?
11. What does argv[0] point to?

Exercises

Because many solutions are possible for the following exercises, answers are not provided.

1. Use your compiler to compile multiple source files into a single executable file. (You can use Listings 21.1, 21.2, and 21.3 or your own listings.)
2. Write an error routine that receives an error number, line number, and module name. The routine should print a formatted error message and then exit the program. Use the predefined macros for the line number and module name (pass the line number and module name from the location where the error occurs). Here's a possible example of a formatted error:

```
module.c (Line ##): Error number ##
```

3. Modify exercise 2 to make the error message more descriptive. Create a text file with your editor that contains an error number and message. Call this file ERRORS.TXT. It could contain information such as the following:

```
1    Error number 1
2    Error number 2
90   Error opening file
100  Error reading file
```

Have your error routine search this file and display the appropriate error message based on a number passed to it.

4. Some header files might be included more than once when you're writing a mod-ular program. Use preprocessor directives to write the skeleton of a header file that compiles only the first time it is encountered during compilation.

5. Write a program that takes two filenames as command-line parameters. The program should copy the first file into the second file. (See Day 16, "Using Disk Files," if you need help working with files.)

6. This is the last exercise of the book, and its content is up to you. Select a programming task of interest to you that also meets a real need you have. For example, you could write programs to catalog your compact disc collection, keep track of your checkbook, or calculate financial figures related to a planned house purchase. There's no substitute for tackling a real-world programming problem in order to sharpen your programming skills and help you remember all the things you learned in this book.