Chapter 2 The Components of a C Program

2-1. A Short C Program

Listing 2.1 presents the source code for MULTIPLY.C. This is a very simple program. All it does is input two numbers from the keyboard and calculate their product. At this stage, don't worry about understanding the details of the program's workings. The point is to gain some familiarity with the parts of a C program so that you can better understand the listings presented later in this book.

Before looking at the sample program, you need to know what a function is, because functions are central to C programming. A *function* is an independent section of program code that performs a certain task and has been assigned a name. By referencing a function's name, your program can execute the code in the function. The program also can send information, called *arguments,* to the function, and the function can return information to the main part of the program. The two types of C functions are *library functions,* which are a part of the C compiler package, and *user-defined functions,* which you, the programmer, create. You will learn about both types of functions in this book.

Note that, as with all the listings in this book, the line numbers in Listing 2.1 are not part of the program. They are included only for identification purposes, so don't type them.

Listing 2.1. MULTIPLY.C.

```
1:  /* Program to calculate the product of two numbers. */
2:  #include <stdio.h>
3:
4:  int a,b,c;
5:
6:  int product(int x, int y);
7:
8:  main()
9:  {
10:     /* Input the first number */
11:     printf("Enter a number between 1 and 100: ");
12:     scanf("%d", &a);
13:
14:     /* Input the second number */
15:     printf("Enter another number between 1 and 100: ");
16:     scanf("%d", &b);
17:
18:     /* Calculate and display the product */
19:     c = product(a, b);
20:     printf ("%d times %d = %d\n", a, b, c);
21:
22:     return 0;
23: }
24:
25: /* Function returns the product of its two arguments */
26: int product(int x, int y)
27: {
28:     return (x * y);
29: }
Enter a number between 1 and 100: 35
```

```
Enter another number between 1 and 100: 23
35 times 23 = 805
```

2-2. The Program's Components

The following sections describe the various components of the preceding sample program. Line numbers are included so that you can easily identify the program parts being discussed.

2-2-1. The main() Function (Lines 8 Through 23)

The only component that is required in every C program is the main() function. In its simplest form, the main() function consists of the name main followed by a pair of empty parentheses (()) and a pair of braces ({}). Within the braces are statements that make up the main body of the program. Under normal circumstances, program execution starts at the first statement in main() and terminates at the last statement in main().

2-2-2. The #include Directive (Line 2)

The #include directive instructs the C compiler to add the contents of an include file into your program during compilation. An *include file* is a separate disk file that contains information needed by your program or the compiler. Several of these files (sometimes called *header files*) are supplied with your compiler. You never need to modify the information in these files; that's why they're kept separate from your source code. Include files should all have an .H extension (for example, STDIO.H).

You use the #include directive to instruct the compiler to add a specific include file to your program during compilation. The #include directive in this sample program means "Add the contents of the file STDIO.H." Most C programs require one or more include files. More information about include files is presented on Day 21, "Advanced Compiler Use."

2-2-3. The Variable Definition (Line 4)

A *variable* is a name assigned to a data storage location. Your program uses variables to store various kinds of data during program execution. In C, a variable must be defined before it can be used. A variable definition informs the compiler of the variable's name and the type of data it is to hold. In the sample program, the definition on line 4, int a,b,c;, defines three variables--named a, b, and c--that will each hold an integer value. More information about variables and variable definitions is presented on Day 3, "Storing Data: Variables and Constants."

2-2-4. The Function Prototype (Line 6)

A *function prototype* provides the C compiler with the name and arguments of the functions contained in the program. It must appear before the function is used. A function prototype is distinct from a *function definition,* which contains the actual statements that make up the function. (Function definitions are discussed in more detail later in this chapter.)

2-2-5. Program Statements (Lines 11, 12, 15, 16, 19, 20, 22, and 28)

The real work of a C program is done by its statements. C statements display information on-screen, read keyboard input, perform mathematical operations, call functions, read disk files, and carry out all the other operations that a program needs to perform. Most of this book is devoted to teaching you the various C statements. For now, remember that in your source code, C statements are generally written one per line and always end with a semicolon. The statements in MULTIPLY.C are explained briefly in the following sections.

printf()

The printf() statement (lines 11, 15, and 20) is a library function that displays information on-screen. The printf() statement can display a simple text message (as in lines 11 and 15) or a message and the value of one or more program variables (as in line 20).

scanf()

The scanf() statement (lines 12 and 16) is another library function. It reads data from the keyboard and assigns that data to one or more program variables.

The program statement on line 19 calls the function named product(). In other words, it executes the program statements contained in the function product(). It also sends the arguments a and b to the function. After the statements in product() are completed, product() returns a value to the program. This value is stored in the variable named c.

return

Lines 22 and 28 contain return statements. The return statement on line 28 is part of the function product(). It calculates the product of the variables x and y and returns the result to the program statement that called product(). The return statement on line 22 returns a value of 0 to the operating system just before the program ends.

2-2-6. The Function Definition (Lines 26 Through 29)

A function is an independent, self-contained section of code that is written to perform a certain task. Every function has a name, and the code in each function is executed by including that function's name in a program statement. This is known as *calling* the function.

The function named product(), in lines 26 through 29, is a user-defined function. As the name implies, user-defined functions are written by the programmer during program development. This function is simple. All it does is multiply two values and return the answer to the program that called it. On Day 5, "Functions: The Basics," you will learn that the proper use of functions is an important part of good C programming practice.

Note that in a real C program, you probably wouldn't use a function for a task as simple as multiplying two numbers. I've done this here for demonstration purposes only.

C also includes library functions that are a part of the C compiler package. Library functions perform most of the common tasks (such as screen, keyboard, and disk input/output) your program needs. In the sample program, printf() and scanf() are library functions.

2-2-7. Program Comments (Lines 1, 10, 14, 18, and 25)

Any part of your program that starts with /* and ends with */ is called a *comment*. The compiler ignores all comments, so they have absolutely no effect on how a program works. You can put anything you want into a comment, and it won't modify the way your program operates. A comment can span part of a line, an entire line, or multiple lines. Here are three examples:

```
/* A single-line comment */
int a,b,c; /* A partial-line comment */
/* a comment
spanning
multiple lines */
```

However, you shouldn't use *nested* comments (in other words, you shouldn't put one comment within another). Most compilers would not accept the following:

```
/*
/* Nested comment */
*/
```

Some compilers do allow nested comments. Although this feature might be tempting to use, you should avoid doing so. Because one of the benefits of C is portability, using a feature such as nested comments might limit the portability of your code. Nested comments also might lead to hard-to-find problems.

Many beginning programmers view program comments as unnecessary and a waste of time. This is a mistake! The operation of your program might be quite clear while you're writing it--particularly when you're writing simple programs. However, as your programs become larger and more complex, or when you need to modify a program you wrote six months ago, you'll find comments invaluable. Now is the time to develop the habit of using comments liberally to document all your programming structures and operations.

NOTE: Many people have started using a newer style of comments in their C programs. Within C++ and Java, you can use double forward slashes to signal a comment. Here are two examples:

```
// This entire line is a comment
int x;  // Comment starts with slashes.
```

The two forward slashes signal that the rest of the line is a comment. Although many C compilers support this form of comment, you should avoid it if you're interested in portability.

DO add abundant comments to your program's source code, especially near statements or functions that could be unclear to you or to someone who might have to modify it later.

DON'T add unnecessary comments to statements that are already clear. For example, entering

```
/* The following prints Hello World! on the screen */
printf("Hello World!");
```

might be going a little too far, at least once you're completely comfortable with the printf() function and how it works.

DO learn to develop a style that will be helpful. A style that's too lean or cryptic doesn't help, nor does one that's so verbose that you're spending more time commenting than programming!

2-2-8. Braces (Lines 9, 23, 27, and 29)

You use braces ({}) to enclose the program lines that make up every C function--including the main() function. A group of one or more statements enclosed within braces is called a *block*. As you will see in later chapters, C has many uses for blocks.

2-2-9. Running the Program

Take the time to enter, compile, and run MULTIPLY.C. It provides additional practice in using your editor and compiler. Recall these steps from Day 1, "Getting Started with C":

1. Make your programming directory current.
2. Start your editor.
3. Enter the source code for MULTIPLY.C exactly as shown in Listing 2.1, but be sure to omit the line numbers and colons.

4. Save the program file.
5. Compile and link the program by entering the appropriate command(s) for your compiler. If no error messages are displayed, you can run the program by entering multiply at the command prompt.
6. If one or more error messages are displayed, return to step 2 and correct the errors.

2-2-10. A Note on Accuracy

A computer is fast and accurate, but it also is completely literal. It doesn't know enough to correct your simplest mistake; it takes everything you enter exactly as you entered it, not as you meant it!

This goes for your C source code as well. A simple typographical error in your program can cause the C compiler to choke, gag, and collapse. Fortunately, although the compiler isn't smart enough to correct your errors (and you'll make errors--everyone does!), it *is* smart enough to recognize them as errors and report them to you. (You saw in the preceding chapter how the compiler reports error messages and how you interpret them.)

2-3. A Review of the Parts of a Program

Now that all the parts of a program have been described, you should be able to look at any program and find some similarities. Look at Listing 2.2 and see whether you can identify the different parts.

Listing 2.2. LIST_IT.C.

```
1:  /* LIST_IT.C--This program displays a listing with line numbers!
*/
2:  #include <stdio.h>
3:  #include <stdlib.h>
4:
5:  void display_usage(void);
6:  int line;
7:
8:  main( int argc, char *argv[] )
9:  {
10:    char buffer[256];
11:    FILE *fp;
12:
13:    if( argc < 2 )
14:    {
15:       display_usage();
16:       exit(1);
17:    }
18:
19:    if (( fp = fopen( argv[1], "r" )) == NULL )
20:    {
21:         fprintf( stderr, "Error opening file, %s!", argv[1] );
22:         exit(1);
23:    }
24:
25:    line = 1;
26:
27:    while( fgets( buffer, 256, fp ) != NULL )
28:       fprintf( stdout, "%4d:\t%s", line++, buffer );
29:
30:    fclose(fp);
```

```
31:    return 0;
32: }
33:
34: void display_usage(void)
35: {
36:       fprintf(stderr, "\nProper Usage is: " );
37:       fprintf(stderr, "\n\nLIST_IT filename.ext\n" );
38: }
C:\>list_it list_it.c
1: /* LIST_IT.C - This program displays a listing with line numbers!
*/
2:   #include <stdio.h>
3:   #include <stdlib.h>
4:
5:   void display_usage(void);
6:   int line;
7:
8:   main( int argc, char *argv[] )
9:   {
10:       char buffer[256];
11:       FILE *fp;
12:
13:       if( argc < 2 )
14:       {
15:            display_usage();
16:            exit(1);
17:       }
18:
19:       if (( fp = fopen( argv[1], "r" )) == NULL )
20:       {
21:            fprintf( stderr, "Error opening file, %s!", argv[1] );
22:            exit(1);
23:       }
24:
25:       line = 1;
26:
27:       while( fgets( buffer, 256, fp ) != NULL )
28:          fprintf( stdout, "%4d:\t%s", line++, buffer );
29:
30:       fclose(fp);
31:       return 0;
32:  }
33:
34:  void display_usage(void)
35:  {
36:       fprintf(stderr, "\nProper Usage is: " );
37:       fprintf(stderr, "\n\nLIST_IT filename.ext\n" );
38:  }
```

ANALYSIS: LIST_IT.C is similar to PRINT_IT.C, which you entered in exercise 7 of Day 1. Listing 2.2 displays saved C program listings on-screen instead of printing them on the printer.

Looking at this listing, you can summarize where the different parts are. The required main() function is in lines 8 through 32. Lines 2 and 3 have #include directives. Lines 6, 10, and 11 have variable definitions. A function prototype, void display_usage(void), is in line 5. This program has many statements (lines 13, 15, 16, 19, 21, 22, 25, 27, 28, 30, 31, 36, and 37). A function definition for display_usage() fills lines 34 through 38. Braces enclose blocks throughout the program. Finally, only line 1 has a comment. In most programs, you should probably include more than one comment line.

LIST_IT.C calls many functions. It calls only one user-defined function, display_usage(). The library functions that it uses are exit() in lines 16 and 22; fopen() in line 19; fprintf() in lines 21, 28, 36, and 37; fgets() in line 27; and fclose() in line 30. These library functions are covered in more detail throughout this book.

2-4. Summary

This chapter was short, but it's important, because it introduced you to the major components of a C program. You learned that the single required part of every C program is the main() function. You also learned that the program's real work is done by program statements that instruct the computer to perform your desired actions. This chapter also introduced you to variables and variable definitions, and it showed you how to use comments in your source code.

In addition to the main() function, a C program can use two types of subsidiary functions: library functions, supplied as part of the compiler package, and user-defined functions, created by the programmer.

Q&A

Q What effect do comments have on a program?
A Comments are for the programmer. When the compiler converts the source code to object code, it throws the comments and the white space away. This means that they have no effect on the executable program. Comments do make your source file bigger, but this is usually of little concern. To summarize, you should use comments and white space to make your source code as easy to understand and main-tain as possible.
Q What is the difference between a statement and a block?
A A block is a group of statements enclosed in braces ({}). A block can be used in most places that a statement can be used.
Q How can I find out what library functions are available?
A Many compilers come with a manual dedicated specifically to documenting the library functions. They are usually in alphabetical order. Another way to find out what library functions are available is to buy a book that lists them. Appendix E, "Common C Functions," lists many of the available functions. After you begin to understand more of C, it would be a good idea to read these appendixes so that you don't rewrite a library function. (There's no use reinventing the wheel!)

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 is the term for a group of one or more C statements enclosed in braces?
2. What is the one component that must be present in every C program?
3. How do you add program comments, and why are they used?
4. What is a function?
5. C offers two types of functions. What are they, and how are they different?
6. What is the #include directive used for?
7. Can comments be nested?
8. Can comments be longer than one line?
9. What is another name for an include file?
10. What is an include file?

Exercises

1. Write the smallest program possible.
2. Consider the following program:

```
1:  /* EX2-2.C */
2:  #include <stdio.h>
3:
4:  void display_line(void);
5:
6:  main()
7:  {
8:      display_line();
9:      printf("\n Teach Yourself C In 21 Days!\n");
10:     display_line();
11:
12:     return 0;
13: }
14:
15: /* print asterisk line */
16: void display_line(void)
17: {
18:     int counter;
19:
20:     for( counter = 0; counter < 21; counter++ )
21:         printf("*" );
22: }
23: /* end of program */
```

a. What line(s) contain statements?
b. What line(s) contain variable definitions?
c. What line(s) contain function prototypes?
d. What line(s) contain function definitions?
e. What line(s) contain comments?
3. Write an example of a comment.
4. What does the following program do? (Enter, compile, and run it.)

```
1: /* EX2-4.C */
2: #include <stdio.h>
3:
4: main()
5: {
6:     int ctr;
7:
8:     for( ctr = 65; ctr < 91; ctr++ )
9:         printf("%c", ctr );
10:
11:    return 0;
12: }
13: /* end of program */
```

5. What does the following program do? (Enter, compile, and run it.)

```
1: /* EX2-5.C */
2: #include <stdio.h>
3: #include <string.h>
4: main()
5: {
```

```
6:     char buffer[256];
7:
8:      printf( "Enter your name and press <Enter>:\n");
9:      gets( buffer );
10:
11:     printf( "\nYour name has %d characters and spaces!",
12                          strlen( buffer ));
13:
14:     return 0;
15: }
```