Compiling and Running Your First Program

IN THIS CHAPTER, YOU ARE INTRODUCED to the C language so that you can see what programming in C is all about. What better way to gain an appreciation for this language than by taking a look at an actual program written in C?

To begin with, you'll choose a rather simple example—a program that displays the phrase "Programming is fun." in your window. Program 3.1 shows a C program to accomplish this task.

Program 3.1 **Writing Your First C Program**

```
#include <stdio.h>

int main (void)
{
     printf ("Programming is fun.\n");

     return 0;
}
```

In the C programming language, lowercase and uppercase letters are distinct. In addition, in C, it does not matter where on the line you begin typing—you can begin typing your statement at any position on the line. This fact can be used to your advantage in developing programs that are easier to read. Tab characters are often used by programmers as a convenient way to indent lines.

Compiling Your Program

Returning to your first C program, you first need to type it into a file. Any text editor can be used for this purpose. Unix users often use an editor such as vi or emacs.

Most C compilers recognize filenames that end in the two characters "." and "c" as C programs. So, assume you type Program 3.1 into a file called `prog1.c`. Next, you need to compile the program.

Using the GNU C compiler, this can be as simple as issuing the `gcc` command at the terminal followed by the filename, like this:

```
$ gcc prog1.c
$
```

If you're using the standard Unix C compiler, the command is `cc` instead of `gcc`. Here, the text you typed is entered in bold. The dollar sign is your command prompt if you're compiling your C program from the command line. Your actual command prompt might be some characters other than the dollar sign.

If you make any mistakes keying in your program, the compiler lists them after you enter the `gcc` command, typically identifying the line numbers from your program that contain the errors. If, instead, another command prompt appears, as is shown in the preceding example, no errors were found in your program.

When the compiler compiles and links your program, it creates an executable version of your program. Using the GNU or standard C compiler, this program is called `a.out` by default. Under Windows, it is often called `a.exe` instead.

Running Your Program

You can now run the executable by simply typing its name on the command line[1]:

```
$ a.out
Programming is fun.
$
```

You can also specify a different name for the executable file at the time the program is compiled. This is done with the `-o` (that's the letter O) option, which is followed by the name of the executable. For example, the command line

```
$ gcc prog1.c -o prog1
```

compiles the program `prog1.c`, placing the executable in the file `prog1`, which can subsequently be executed just by specifying its name:

```
$ prog1
Programming is fun.
$
```

1. If you get an error like this: `a.out: No such file or directory`, then it probably means the current directory is not in your PATH. You can either add it to your PATH or type the following instead at the command prompt: `./a.out`.

Understanding Your First Program

Take a closer look at your first program. The first line of the program

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

should be included at the beginning of just about every program you write. It tells the compiler information about the `printf` output routine that is used later in the program. Chapter 13, "The Preprocessor," discusses in detail what this line does.

The line of the program that reads

```
int main (void)
```

informs the system that the name of the program is `main`, and that it *returns* an integer value, which is abbreviated "`int`." `main` is a special name that indicates precisely *where* the program is to begin execution. The open and close parentheses immediately following `main` specify that `main` is the name of a *function*. The keyword `void` that is enclosed in the parentheses specifies that the function `main` takes no arguments (that is, it is *void* of arguments). These concepts are explained in great detail in Chapter 8, "Working with Functions."

Now that you have identified `main` to the system, you are ready to specify precisely what this routine is to perform. This is done by enclosing all program statements of the routine within a pair of curly braces. All program statements included between the braces are taken as part of the `main` routine by the system. In Program 3.1, you have only two such statements. The first statement specifies that a routine named `printf` is to be invoked or *called*. The parameter or *argument* to be passed to the `printf` routine is the string of characters

```
"Programming is fun.\n"
```

The `printf` routine is a function in the C library that simply prints or displays its argument (or arguments, as you will see shortly) on your screen. The last two characters in the string, namely the backslash (`\`) and the letter `n`, are known collectively as the *newline* character. A newline character tells the system to do precisely what its name implies—that is, go to a new line. Any characters to be printed after the newline character then appear on the next line of the display. In fact, the newline character is similar in concept to the carriage return key on a typewriter. (Remember those?)

All program statements in C *must* be terminated by a semicolon (`;`). This is the reason for the semicolon that appears immediately following the closing parenthesis of the `printf` call.

The last statement in main that reads

```
return 0;
```

says to finish execution of `main`, and return to the system a status value of `0`. You can use any integer here. Zero is used by convention to indicate that the program completed successfully—that is, without running into any errors. Different numbers can be used to indicate different types of error conditions that occurred (such as a file not being found). This exit status can be tested by other programs (such as the Unix shell) to see whether the program ran successfully.

Now that you've finished analyzing your first program, you can modify it to also display the phrase "And programming in C is even more fun." This can be done by the simple addition of another call to the `printf` routine, as shown in Program 3.2. Remember that every C program statement must be terminated by a semicolon.

Program 3.2

```
#include <stdio.h>

int main (void)
{
     printf ("Programming is fun.\n");
     printf ("And programming in C is even more fun.\n");

     return 0;
}
```

If you type in Program 3.2 and then compile and execute it, you can expect the following output in your program's output window, sometimes called the "console."

Program 3.2 **Output**

```
Programming is fun.
And programming in C is even more fun.
```

As you will see from the next program example, it is not necessary to make a separate call to the `printf` routine for each line of output. Study the program listed in Program 3.3 and try to predict the results before examining the output. (No cheating now!)

Program 3.3 **Displaying Multiple Lines of Output**

```
#include <stdio.h>

int main (void)
{
     printf ("Testing...\n..1\n...2\n....3\n");

     return 0;
}
```

Program 3.3 **Output**

```
Testing...
..1
...2
....3
```

Displaying the Values of Variables

The `printf` routine is the most commonly used routine in this book. This is because it provides an easy and convenient means to display program results. Not only can simple phrases be displayed, but the values of *variables* and the results of computations can also be displayed. In fact, Program 3.4 uses the `printf` routine to display the results of adding two numbers, namely 50 and 25.

Program 3.4 **Displaying Variables**

```
#include <stdio.h>

int main (void)
{
     int sum;

     sum = 50 + 25;
     printf ("The sum of 50 and 25 is %i\n", sum);

     return 0;
}
```

Program 3.4 **Output**

```
The sum of 50 and 25 is 75
```

In Program 3.4, the first C program statement *declares* the *variable* `sum` to be of type *integer*. C requires that all program variables be declared before they are used in a program. The declaration of a variable specifies to the C compiler how a particular variable will be used by the program. This information is needed by the compiler to generate the correct instructions to store and retrieve values into and out of the variable. A variable declared as type `int` can only be used to hold integral values; that is, values without decimal places. Examples of integral values are 3, 5, –20, and 0. Numbers with decimal places, such as 3.14, 2.455, and 27.0, for example, are known as *floating-point* or *real* numbers.

The integer variable `sum` is used to store the result of the addition of the two integers 50 and 25. A blank line was intentionally left following the declaration of this variable to visually separate the variable declarations of the routine from the program statements; this is strictly a matter of style. Sometimes, the addition of a single blank line in a program can help to make the program more readable.

The program statement

```
sum = 50 + 25;
```

reads as it would in most other programming languages: The number `50` is added (as indicated by the plus sign) to the number `25`, and the result is stored (as indicated by the *assignment operator*, the equal sign) in the variable `sum`.

The printf routine call in Program 3.4 now has two items or *arguments* enclosed within the parentheses. These arguments are separated by a comma. The first argument to the printf routine is always the character string to be displayed. However, along with the display of the character string, you might also frequently want to have the value of certain program variables displayed. In this case, you want to have the value of the variable sum displayed at the terminal after the characters

```
The sum of 50 and 25 is
```

are displayed. The percent character inside the first argument is a special character recognized by the printf function. The character that immediately follows the percent sign specifies what *type* of value is to be displayed at that point. In the preceding program, the letter i is recognized by the printf routine as signifying that an integer value is to be displayed.[2]

Whenever the printf routine finds the %i characters inside a character string, it automatically displays the value of the next argument to the printf routine. Because sum is the next argument to printf, its value is automatically displayed after the characters "The sum of 50 and 25 is" are displayed.

Now try to predict the output from Program 3.5.

Program 3.5 **Displaying Multiple Values**

```
#include <stdio.h>

int main (void)
{
     int  value1, value2, sum;

     value1 = 50;
     value2 = 25;
     sum = value1 + value2;
     printf ("The sum of %i and %i is %i\n", value1, value2, sum);

     return 0;
}
```

Program 3.5 **Output**

```
The sum of 50 and 25 is 75
```

2. Note that printf also allows you to specify %d format characters to display an integer. This book consistently uses %i throughout the remaining chapters.

The first program statement declares three variables called `value1`, `value2`, and `sum` all to be of type `int`. This statement could have equivalently been expressed using three separate declaratory statements as follows:

```
int value1;
int value2;
int sum;
```

After the three variables have been declared, the program assigns the value `50` to the variable `value1` and then assigns `25` to `value2`. The sum of these two variables is then computed, and the result is assigned to the variable `sum`.

The call to the `printf` routine now contains four arguments. Once again, the first argument, commonly called the *format string*, describes to the system how the remaining arguments are to be displayed. The value of `value1` is to be displayed immediately following the display of the characters "The sum of." Similarly, the values of `value2` and `sum` are to be printed at the appropriate points, as indicated by the next two occurrences of the `%i` characters in the format string.

Comments

The final program in this chapter (Program 3.6) introduces the concept of the *comment*. A comment statement is used in a program to document a program and to enhance its readability. As you will see from the following example, comments serve to tell the reader of the program—the programmer or someone else whose responsibility it is to maintain the program—just what the programmer had in mind when he or she wrote a particular program or a particular sequence of statements.

Program 3.6 **Using Comments in a Program**

```
/* This program adds two integer values
   and displays the results             */

#include <stdio.h>

int main (void)
{
    // Declare variables
    int  value1, value2, sum;

    // Assign values and calculate their sum
    value1 = 50;
    value2 = 25;
    sum = value1 + value2;
```

Program 3.6 **Continued**

```
    // Display the result
    printf ("The sum of %i and %i is %i\n", value1, value2, sum);

    return 0;
}
```

Program 3.6 **Output**

```
The sum of 50 and 25 is 75
```

There are two ways to insert comments into a C program. A comment can be initiated by the two characters `/` and `*`. This marks the *beginning* of the comment. These types of comments have to be *terminated*. To end the comment, the characters `*` and `/` are used without any embedded spaces. All characters included between the opening `/*` and the closing `*/` are treated as part of the comment statement and are ignored by the C compiler. This form of comment is often used when comments span several lines in the program. The second way to add a comment to your program is by using two consecutive slash characters `//`. Any characters that follow these slashes up to the end of the line are ignored by the compiler.

In Program 3.6, four separate comment statements were used. This program is otherwise identical to Program 3.5. Admittedly, this is a contrived example because only the first comment at the head of the program is useful. (Yes, it is possible to insert so many comments into a program that the readability of the program is actually degraded instead of improved!)

The intelligent use of comment statements inside a program cannot be overemphasized. Many times, a programmer returns to a program that he coded perhaps only six months ago, only to discover to his dismay that he could not for the life of him remember the purpose of a particular routine or of a particular group of statements. A simple comment statement judiciously inserted at that particular point in the program might have saved a significant amount of time otherwise wasted on rethinking the logic of the routine or set of statements.

It is a good idea to get into the habit of inserting comment statements into the program as the program is being written or typed in. There are good reasons for this. First, it is far easier to document the program while the particular program logic is still fresh in your mind than it is to go back and rethink the logic after the program has been completed. Second, by inserting comments into the program at such an early stage of the game, you get to reap the benefits of the comments during the debug phase, when program logic errors are being isolated and debugged. A comment can not only help you read through the program, but it can also help point the way to the source of the logic mistake. Finally, I have yet to discover a programmer who actually enjoyed documenting a program. In fact, after you have finished debugging your program, you will probably not relish the idea of going back to the program to insert comments. Inserting comments while developing the program makes this sometimes tedious task a bit easier to swallow.

This concludes this introductory chapter on developing programs in C. By now, you should have a good feel as to what is involved in writing a program in C, and you should be able to develop a small program on your own. In the next chapter, you begin to learn some of the finer intricacies of this wonderfully powerful and flexible programming language. But first, try your hand at the following exercises to make certain you understand the concepts presented in this chapter.

Exercises

1. Type in and run the six programs presented in this chapter. Compare the output produced by each program with the output presented after each program in the text.
2. Write a program that prints the following text at the terminal.
    1. In C, lowercase letters are significant.
    2. main is where program execution begins.
    3. Opening and closing braces enclose program statements in a routine.
    4. All program statements must be terminated by a semicolon.
3. What output would you expect from the following program?

```
#include <stdio.h>

int main (void)
{
    printf ("Testing...");
    printf ("....1");
    printf ("...2");
    printf ("..3");
    printf ("\n");

    return 0;
}
```

4. Write a program that subtracts the value `15` from `87` and displays the result, together with an appropriate message, at the terminal.
5. Identify the syntactic errors in the following program. Then type in and run the corrected program to ensure you have correctly identified all the mistakes.

```
#include <stdio.h>

int main (Void)
(
        INT  sum;
        /* COMPUTE RESULT
        sum = 25 + 37 - 19
        /* DISPLAY RESULTS //
        printf ("The answer is %i\n" sum);
        return 0;
}
```

6. What output might you expect from the following program?

```
#include <stdio.h>

int main (void)
{
      int answer, result;

      answer = 100;
      result = answer - 10;
      printf ("The result is %i\n", result + 5);

      return 0;
}
```