Your First C++ Program

This chapter introduces you to some important C++ language commands and other elements. Before looking at the language more specifically, many people like to "walk through" a few simple programs to get an overall feel for what a C++ program involves. This is done here. The rest of the book covers these commands and elements more formally.

This chapter introduces the following topics:

- An overview of C++ programs and their structure
- Variables and literals
- Simple math operators
- Screen output format

This chapter introduces a few general tools you need to become familiar with the C++ programming language. The rest of the book concentrates on more specific areas of the actual language.

Looking at a C++ Program

Figure 3.1 shows the outline of a typical small C++ program. No C++ commands are shown in the figure. Although there is much more to a program than this outline implies, this is the general format of the beginning examples in this book.

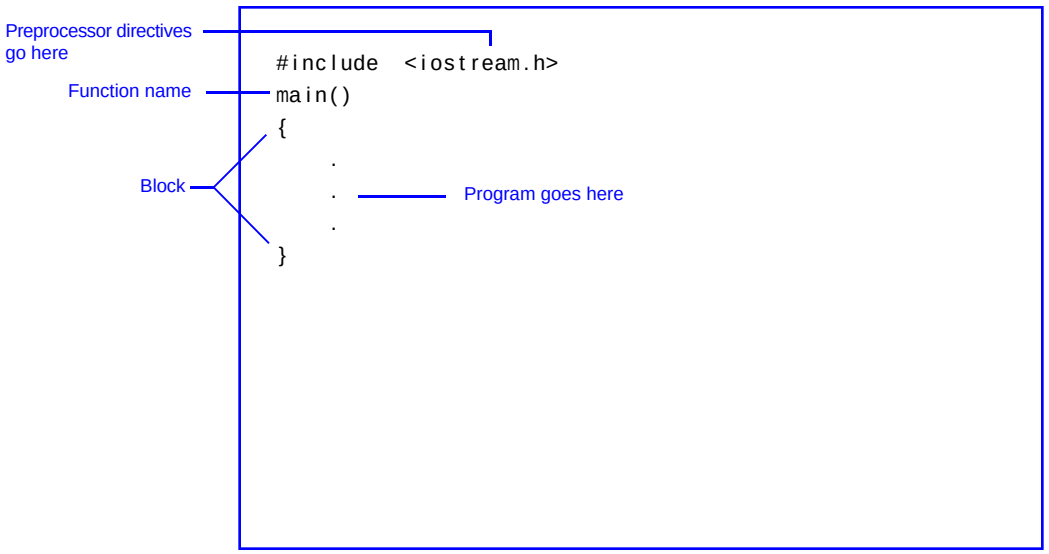


Figure 3.1. A skeleton outline of a simple C++ program.

To acquaint yourself with C++ programs as fast as possible, you should begin to look at a program in its entirety. The following is a listing of a simple example C++ program. It doesn't do much, but it enables you to see the general format of C++ programming. The next few sections cover elements from this and other programs. You might not understand everything in this program, even after finishing the chapter, but it is a good place to start.

```
// Filename: C3FIRST.CPP
// Initial C++ program that demonstrates the C++ comments
// and shows a few variables and their declarations.
```

```
#include <iostream.h>

main()
{
   int i, j;     // These three lines declare four variables.
   char c;
   float x;

   i = 4;        // i and j are both assigned integer literals.
   j = i + 7;
   c = 'A';      // All character literals are
                 // enclosed in single quotations.
   x = 9.087;    // x requires a floating-point value because it
                 // was declared as a floating-point variable.
   x = x * 4.5; // Change what was in x with a formula.

   // Sends the values of the variables to the screen.
   cout <<  i << ", " << j << ", " << c << ", " << x << "\n";

   return 0;    // ALWAYS end programs and functions with return.
                // The 0 returns to the operating system and
                // usually indicates no errors occurred.
}
```

For now, familiarize yourself with this overall program. See if you can understand any part or all of it. If you are new to programming, you should know that the computer reads each line of the program, starting with the first line and working its way down, until it has completed all the instructions in the program. (Of course, you first have to compile and link the program, as described in Chapter 2, "What Is a Program?".)

The output of this program is minimal: It simply displays four values on-screen after performing some assignments and calculations of arbitrary values. Just concentrate on the general format at this point.

The Format of a C++ Program

C++ is a free-form language.

Unlike some other programming languages, such as COBOL, C++ is a *free-form* language, meaning that programming statements

can start in any column of any line. You can insert blank lines in a program if you want. This sample program is called C3FIRST.CPP (you can find the name of each program in this book in the first line of each program listing). It contains several blank lines to help separate parts of the program. In a simple program such as this, the separation is not as critical as it might be in a longer, more complex program.

Generally, spaces in C++ programs are free-form as well. Your goal should not be to make your programs as compact as possible. Your goal should be to make your programs as readable as possible. For example, the C3FIRST.CPP program shown in the previous section could be rewritten as follows:

```
// Filename: C3FIRST.CPP Initial C++ program that demonstrates
// the C++ comments and shows a few variables and their
// declarations.
#include <iostream.h>
main(){int i,j;// These three lines declare four variables.
char c;float x;i=4;// i and j are both assigned integer literals.
j=i+7;c='A';// All character literals are enclosed in
//single quotations.
x=9.087;//x requires a floating-point value because it was
//declared as a floating-point variable.
x=x*4.5;//Change what was in x with a formula.
//Sends the values of the variables to the screen.
cout<<i<<", "<<j<<", "<<c<<", "<<x<<"\n";return 0;// ALWAYS
//end programs and functions with return. The 0 returns to
//the operating system and usually indicates no errors occurred.
}
```

To your C++ compiler, the two programs are exactly the same, and they produce exactly the same result. However, to people who have to read the program, the first style is much more readable.

Readability Is the Key

As long as programs do their job and produce correct output, who cares how well they are written? Even in today's world of fast computers and abundant memory and disk space, you should still

care. Even if nobody else ever looks at your C++ program, you might have to change it at a later date. The more readable you make your program, the faster you can find what needs changing, and change it accordingly.

If you work as a programmer for a corporation, you can almost certainly expect to modify someone else's source code, and others will probably modify yours. In programming departments, it is said that long-term employees write readable programs. Given this new global economy and all the changes that face business in the years ahead, companies are seeking programmers who write for the future. Programs that are straightforward, readable, abundant with *white space* (separating lines and spaces), and devoid of hard-to-read "tricks" that create messy programs are the most desirable.

Use ample white space so you can have separate lines and spaces throughout your programs. Notice the first few lines of C3FIRST.CPP start in the first column, but the body of the program is indented a few spaces. This helps programmers "zero in" on the important code. When you write programs that contain several sections (called *blocks*), your use of white space helps the reader's eye follow and recognize the next indented block.

Uppercase Versus Lowercase

Use lowercase abundantly in C++!

Your uppercase and lowercase letters are much more significant in C++ than in most other programming languages. You can see that most of C3FIRST.CPP is in lowercase. The entire C++ language is in lowercase. For example, you must type the keywords `int`, `char`, and `return` in programs using lowercase characters. If you use uppercase letters, your C++ compiler would produce many errors and refuse to compile the program until you correct the errors. Appendix E, "Keyword and Function Reference," shows a list of every command in the C++ programming language. You can see that none of the commands have uppercase letters.

Many C++ programmers reserve uppercase characters for some words and messages sent to the screen, printer, or disk file; they use lowercase letters for almost everything else. There is, however, one exception to this rule in Chapter 4, "Variables and Literals," dealing with the `const` keyword.

Braces and `main()`

All C++ programs require the following lines:

```
main()
{
```

The statements that follow `main()` are executed first. The section of a C++ program that begins with `main()`, followed by an opening brace, `{`, is called the *main function*. A C++ program is actually a collection of functions (small sections of code). The function called `main()` is always required and always the first *function* executed.

A C++ block is enclosed in two braces.

In the sample program shown here, almost the entire program is `main()` because the matching closing brace that follows `main()`'s opening brace is at the end of the program. Everything between two matching braces is called a *block*. You read more about blocks in Chapter 16, "Writing C++ Functions." For now, you only have to realize that this sample program contains just one function, `main()`, and the entire function is a single block because there is only one pair of braces.

All executable C++ statements must end with a semicolon (;).

All *executable* C++ statements must have a semicolon (;) after them so C++ is aware that the statement is ending. Because the computer ignores all comments, do *not* put semicolons after your comments. Notice that the lines containing `main()` and braces do not end with semicolons either, because these lines simply define the beginning and ending of the function and are not executed.

As you become better acquainted with C++, you learn when to include the semicolon and when to leave it off. Many beginning C++ programmers learn quickly when semicolons are required; your compiler certainly lets you know if you forget to include a semicolon where one is needed.

Figure 3.2 repeats the sample program shown in Figure 3.1. It contains additional markings to help acquaint you with these new terms as well as other items described in the remainder of this chapter.

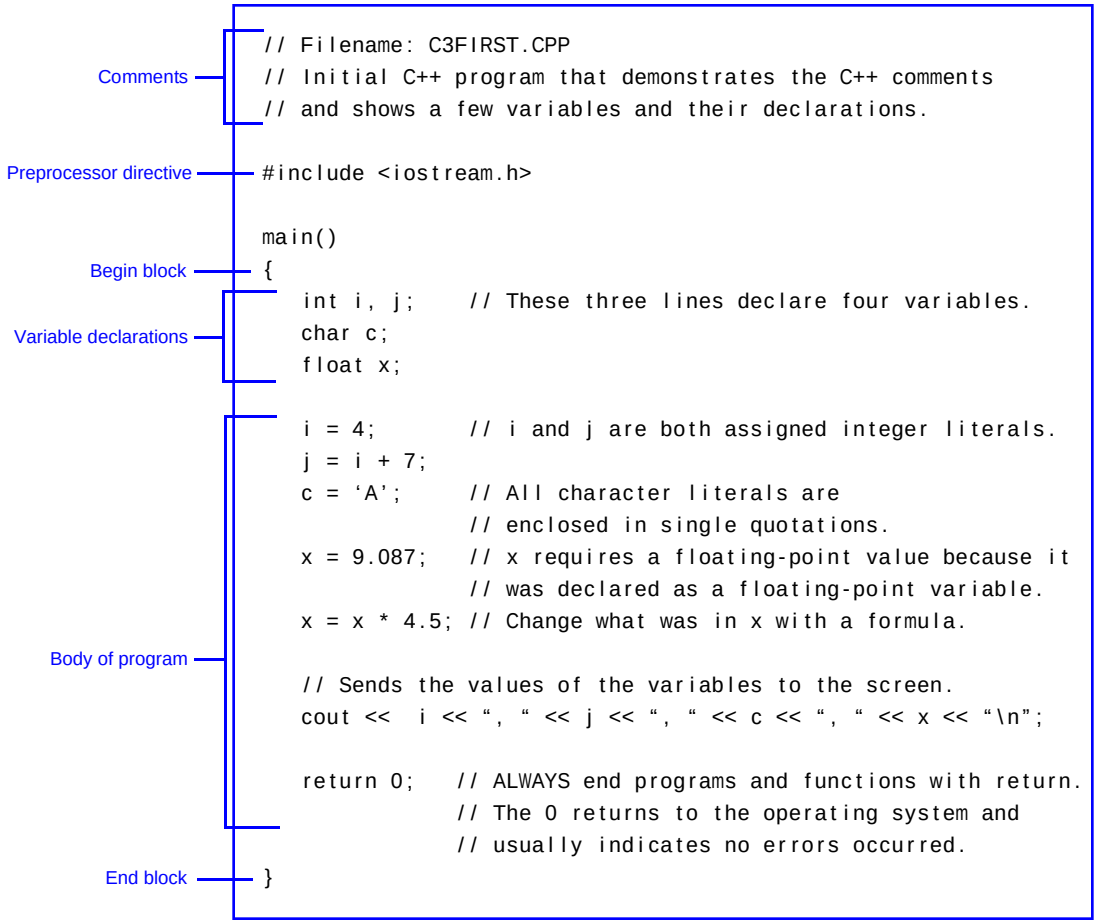


Figure 3.2. The parts of the sample program.

Comments in C++

In Chapter 2, "What Is a Program?," you learned the difference between a program and its output. Most users of a program do not see the actual program; they see the output from the execution of the program's instructions. Programmers, on the other hand, look at the program listings, add new routines, change old ones, and update for advancements in computer equipment.

Comments tell *people* what the program is doing.

As explained earlier, the readability of a program is important so you and other programmers can look through it easily. Nevertheless, no matter how clearly you write C++ programs, you can always enhance their readability by adding comments throughout.

Comments are messages that you insert in your C++ programs, explaining what is going on at that point in the program. For example, if you write a payroll program, you might put a comment before the check-printing routine that describes what is about to happen. You never put C++ language statements inside a comment, because a comment is a message for people—not computers. Your C++ compiler ignores all comments in every program.

NOTE: C++ comments always begin with a // symbol and end at the end of the line.

Some programmers choose to comment several lines. Notice in the sample program, C3FIRST.CPP, that the first three lines are comment lines. The comments explain the filename and a little about the program.

Comments also can share lines with other C++ commands. You can see several comments sharing lines with commands in the C3FIRST.CPP program. They explain what the individual lines do. Use abundant comments, but remember who they're for: people, not computers. Use comments to help explain your code, but do not *over*comment. For example, even though you might not be familiar with C++, the following statement is easy: It prints "C++ By Example" on-screen.


```
cout << "C++ By Example"; // Print C++ By Example on-screen.
```

This comment is redundant and adds nothing to your understanding of the line of code. It would be much better, in this case, to leave out the comment. If you find yourself almost repeating the C++ code, leave out that particular comment. Not every line of a C++ program should be commented. Comment only when code lines need explaining—in English—to the people looking at your program.

It does not matter if you use uppercase, lowercase, or a mixture of both in your comments because C++ ignores them. Most C++

programmers capitalize the first letter of sentences in comments, just as you would in everyday writing. Use whatever case seems appropriate for the letters in your message.

C++ can also use C-style comments. These are comments that begin with `/*` and end with `*/`. For instance, this line contains a comment in the C *and* C++ style:

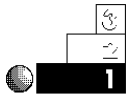
```
netpay = grosspay - taxes;  /* Compute take-home pay. */
```

Comment As You Go

Insert your comments as you write your programs. You are most familiar with your program logic at the time you are typing the program in the editor. Some people put off adding comments until after the program is written. More often than not, however, those comments are never added, or else they are written halfheartedly.

If you comment as you write your code, you can glance back at your comments while working on later sections of the program—instead of having to decipher the previous code. This helps you whenever you want to search for something earlier in the program.

Examples



1. Suppose you want to write a C++ program that produces a fancy boxed title containing your name with flashing dots around it (like a marquee). The C++ code to do this might be difficult to understand. Before such code, you might want to insert the following comment so others can understand the code later:

```
// The following few lines draw a fancy box around
// a name, then display flashing dots around the
// name like a Hollywood movie marquee.
```

This would not tell C++ to do anything because a comment is not a command, but it would make the next few lines of code more understandable to you and others. The comment explains in English, for people reading the program, exactly what the program is getting ready to do.

2. You should also put the disk filename of the program in one of the first comments. For example, in the C3FIRST.CPP program shown earlier, the first line is the beginning of a comment:

```
// Filename: C3FIRST.CPP
```

The comment is the first of three lines, but this line tells you in which disk file the program is stored. Throughout this book, programs have comments that include a possible filename under which the program can be stored. They begin with C*x*, where *x* is the chapter number in which they appear (for example, C*6*VARPR.CPP and C*10*LNIN.CPP). This method helps you find these programs when they are discussed in another section of the book.

TIP: It might be a good idea to put your name at the top of a program in a comment. If people have to modify your program at a later date, they first might want to consult with you, as the original programmer, before they change it.

Explaining the Sample Program

Now that you have an overview of a C++ program, its structure, and its comments, the rest of this chapter walks you through the entire sample program. Do not expect to become a C++ expert just by completing this section—that is what the rest of the book is for! For now, just sit back and follow this step-by-step description of the program code.

As described earlier, this sample program contains several comments. The first three lines of the program are comments:

```
// Filename: C3FIRST.CPP
// Initial C++ program that demonstrates the C++ comments
// and shows a few variables and their declarations.
```

This comment lists the filename and explains the purpose of the program. This is not the only comment in the program; others appear throughout the code.

The next line beginning with `#include` is called a preprocessor directive and is shown here:

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

This strange looking statement is not actually a C++ command, but is a directive that instructs the C++ compiler to load a file from disk into the middle of the current program. The only purpose for this discussion is to ensure that the output generated with `cout` works properly. Chapter 6, "Preprocessor Directives," more fully explains this directive.

The next two lines (following the blank separating line) are shown here:

```
main()
{
```

This begins the `main()` function. Basically, the `main()` function's opening and closing braces enclose the body of this program and `main()`'s instructions that execute. C++ programs often contain more than one function, but they *always* contain a function called `main()`. The `main()` function does not have to be the first one, but it usually is. The opening brace begins the first and only block of this program.

When a programmer compiles and runs this program, the computer looks for `main()` and starts executing whatever instruction follows `main()`'s opening brace. Here are the three lines that follow:

```
int i, j;     // These three lines declare four variables.
char c;
float x;
```

These three lines declare variables. A *variable declaration* describes variables used in a block of code. Variable declarations describe the program's data storage.

A C++ program processes data into meaningful results. All C++ programs include the following:

- Commands
- Data

Data comprises *variables* and *literals* (sometimes called constants). As the name implies, a *variable* is data that can change (become variable) as the program runs. A literal remains the same. In life, a variable might be your salary. It increases over time (if you are lucky). A literal would be your first name or social security number, because each remains with you throughout life and does not (naturally) change.

Chapter 4, "Variables and Literals," fully explains these concepts. However, to give you an overview of the sample program's elements, the following discussion explains variables and literals in this program.

C++ enables you to use several kinds of literals. For now, you simply have to understand that a C++ literal is any number, character, word, or phrase. The following are all valid C++ literals:

```
5.6
-45

'Q'

"Mary"

18.67643

0.0
```

As you can see, some literals are numeric and some are character-based. The single and double quotation marks around two of the literals, however, are not part of the actual literals. A single-character literal requires single quotation marks around it; a string of characters, such as `"Mary"`, requires double quotation marks.

Look for the literals in the sample program. You find these:

```
4
7
'A'
9.087
4.5
```

A variable is like a box inside your computer that holds something. That "something" might be a number or a character. You can have as many variables as needed to hold changing data. After you define a variable, it keeps its value until you change it or define it again.

Variables have names so you can tell them apart. You use the assignment operator, the equal sign (=), to assign values to variables. The following statement,


```
sales=25000;
```

puts the literal value 25000 into the variable named sales. In the sample program, you find the following variables:

```
i
j
c
x
```

The three lines of code that follow the opening brace of the sample program declare these variables. This variable declaration informs the rest of the program that two integer variables named i and j as well as a character variable called c and a floating-point variable called x appear throughout the program. The terms *integer* and *floating-point* basically refer to two different types of numbers: Integers are whole numbers, and floating-point numbers contain decimal points.

The next few statements of the sample program assign values to these variables.

```
i = 4;        // i and j are both assigned integer literals.
j = i + 7;
c = 'A';      // All character literals are
              // enclosed in single quotations.
x = 9.087;    // x requires a floating-point value because it
              // was declared as a floating-point variable.
x = x * 4.5; // Change what was in x with a formula.
```

The first line puts 4 in the integer variable, i. The second line adds 7 to the variable i's value to get 11, which then is assigned to (or put into) the variable called j. The plus sign (+) in C++ works just like it does in mathematics. The other primary math operators are shown in Table 3.1.

Table 3.1. The primary math operators.

Operator	*Meaning*	*Example*
+	Addition	4 + 5
–	Subtraction	7 – 2
*	Multiplication	12 * 6
/	Division	48 / 12

The character literal A is assigned to the c variable. The number 9.087 is assigned to the variable called x, then x is immediately overwritten with a new value: itself (9.087) multiplied by 4.5. This helps illustrate why computer designers use an asterisk (*) for multiplication and not a lowercase x as people generally do to show multiplication; the computer would confuse the variable x with the multiplication symbol, x, if both were allowed.

TIP: If mathematical operators are on the right side of the equal sign, the program completes the math before assigning the result to a variable.

The next line (after the comment) includes the following special—and, at first, confusing—statement:

```
cout << i << ", " << j << ", " << c << ", " << x << "\n";
```

When the program reaches this line, it prints the contents of the four variables on-screen. The important part of this line is that the four values for `i`, `j`, `c`, and `x` print on-screen.

The output from this line is

```
4, 11, A, 40.891499
```

Because this is the only `cout` in the program, this is the only output the sample program produces. You might think the program is rather long for such a small output. After you learn more about C++, you should be able to write more useful programs.

The `cout` is not a C++ command. You might recall from Chapter 2, "What Is a Program?," that C++ has no built-in input/output commands. The `cout` is an operator, described to the compiler in the `#include` file called iostream.h, and it sends output to the screen.

C++ also supports the `printf()` function for formatted output. You have seen one function already, `main()`, which is one for which you write the code. The C++ programming designers have already written the code for the `printf` function. At this point, you can think of `printf` as a command that outputs values to the screen, but it is actually a built-in function. Chapter 7, "Simple Input/Output" describes the `printf` function in more detail.

NOTE: To differentiate `printf` from regular C++ commands, parentheses are used after the name, as in `printf()`. In C++, all function names have parentheses following them. Sometimes these parentheses have something between them, and sometimes they are blank.

Put a `return` statement at the end of each function.

The last two lines in the program are shown here:

```
return 0; // ALWAYS end programs and functions with return.
}
```

The `return` command simply tells C++ that this function is finished. C++ returns control to whatever was controlling the program before it started running. In this case, because there was only one function, control is returned either to DOS or to the C++ editing environment. C++ requires a return value. Most C++ programmers return a `0` (as this program does) to the operating system. Unless you use operating-system return variables, you have little use for a return value. Until you have to be more specific, always return a `0` from `main()`.

Actually, many `return` statements are optional. C++ would know when it reached the end of the program without this statement. It is a good programming practice, however, to put a `return` statement at the end of every function, including `main()`. Because some functions require a `return` statement (if you are returning values), it is better to get in the habit of using them, rather than run the risk of leaving one out when you really need it.

You will sometimes see parentheses around the `return` value, as in:

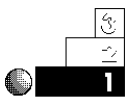
```
return (0); // ALWAYS end programs and functions with return.
```

The parentheses are unnecessary and sometimes lead beginning C++ students into thinking that `return` is a built-in function. However, the parentheses are recommended when you want to return an expression. You read more about returning values in Chapter 19, “Function Return Values and Prototypes.”

The closing brace after the `return` does two things in this program. It signals the end of a block (begun earlier with the opening brace), which is the end of the `main()` function, and it signals the end of the program.

Review Questions

The answers to the review questions are in Appendix B, aptly named “Answers to Review Questions.”



1. What must go before each comment in a C++ program?
2. What is a variable?
3. What is a literal?

4. What are four C++ math operators?
5. What operator assigns a variable its value? (*Hint:* It is called the assignment operator.)

6. True or false: A variable can consist of only two types: integers and characters.
7. What is the operator that writes output to the screen?
8. Is the following a variable name or a string literal?

```
city
```

9. What, if anything, is wrong with the following C++ statement?

```
RETURN;
```

Summary

This chapter focused on teaching you to write helpful and appropriate comments for your programs. You also learned a little about variables and literals, which hold the program's data. Without them, the term *data processing* would no longer be meaningful (there would be no data to process).

Now that you have a feel for what a C++ program looks like, it is time to begin looking at specifics of the commands. Starting with the next chapter, you begin to write your own programs. The next chapter picks up where this one left off; it takes a detailed look at literals and variables, and better describes their uses and how to choose their names.