CHAPTER

2

C++ Programming Basics

Assist. Prof. Dr.
Ahmed Hashim Mohammed
2018

IN THIS CHAPTER

In any language there are some fundamentals you need to know before you can write even the most elementary programs. This chapter introduces three such fundamentals: basic program construction, variables, and input/output (I/O). It also touches on a variety of other language features, including comments, arithmetic operators, the increment operator, data conversion, and library functions.

These topics are not conceptually difficult, but you may find that the style in C++ is a little austere compared with, say, BASIC or Pascal. Before you learn what it's all about, a C++ program may remind you more of a mathematics formula than a computer program. Don't worry about this. You'll find that as you gain familiarity with C++, it starts to look less forbidding, while other languages begin to seem unnecessarily fancy and verbose.

Getting Started

As we noted in the Introduction, you can use either a Microsoft or a Borland compiler with this book. Appendixes C and D provide details about their operation. (Other compilers may work as well.) Compilers take source code and transform it into executable files, which your computer can run as it does other programs. Source files are text files (extension .CPP) that correspond with the listings printed in this book. Executable files have the .EXE extension, and can be executed either from within your compiler, or, if you're familiar with MS-DOS, directly from a DOS window.

The programs run without modification on the Microsoft compiler or in an MS-DOS window. If you're using the Borland compiler, you'll need to modify the programs slightly before running them; otherwise the output won't remain on the screen long enough to see. Make sure to read Appendix D, "Borland C++Builder," to see how this is done.

Basic Program Construction

Let's look at a very simple C++ program. This program is called FIRST, so its source file is FIRST.CPP. It simply prints a sentence on the screen. Here it is:

```
#include <iostream>
using namespace std;

int main()
   {
   cout << "Every age has a language of its own\n";
   return 0;
   }
```

Despite its small size, this program demonstrates a great deal about the construction of C++ programs. Let's examine it in detail.

Functions

Functions are one of the fundamental building blocks of C++. The FIRST program consists almost entirely of a single function called `main()`. The only parts of this program that are not part of the function are the first two lines—the ones that start with `#include` and `using`. (We'll see what these lines do in a moment.)

We noted in Chapter 1, "The Big Picture," that a function can be part of a class, in which case it is called a member function. However, functions can also exist independently of classes. We are not yet ready to talk about classes, so we will show functions that are separate standalone entities, as `main()` is here.

Function Name

The parentheses following the word `main` are the distinguishing feature of a function. Without the parentheses the compiler would think that `main` refers to a variable or to some other program element. When we discuss functions in the text, we'll follow the same convention that C++ uses: We'll put parentheses following the function name. Later on we'll see that the parentheses aren't always empty. They're used to hold function arguments: values passed from the calling program to the function.

The word `int` preceding the function name indicates that this particular function has a return value of type `int`. Don't worry about this now; we'll learn about data types later in this chapter and return values in Chapter 5, "Functions."

Braces and the Function Body

The body of a function is surrounded by braces (sometimes called curly brackets). These braces play the same role as the `BEGIN` and `END` keywords in some other languages: They surround or delimit a block of program statements. Every function must use this pair of braces around the function body. In this example there are only two statements in the function body: the line starting with `cout`, and the line starting with `return`. However, a function body can consist of many statements.

Always Start with `main()`

When you run a C++ program, the first statement executed will be at the beginning of a function called `main()`. (At least that's true of the console mode programs in this book.) The program may consist of many functions, classes, and other program elements, but on startup, control always goes to `main()`. If there is no function called `main()` in your program, an error will be reported when you run the program.

In most C++ programs, as we'll see later, `main()` calls member functions in various objects to carry out the program's real work. The `main()` function may also contain calls to other standalone functions. This is shown in Figure 2.1.

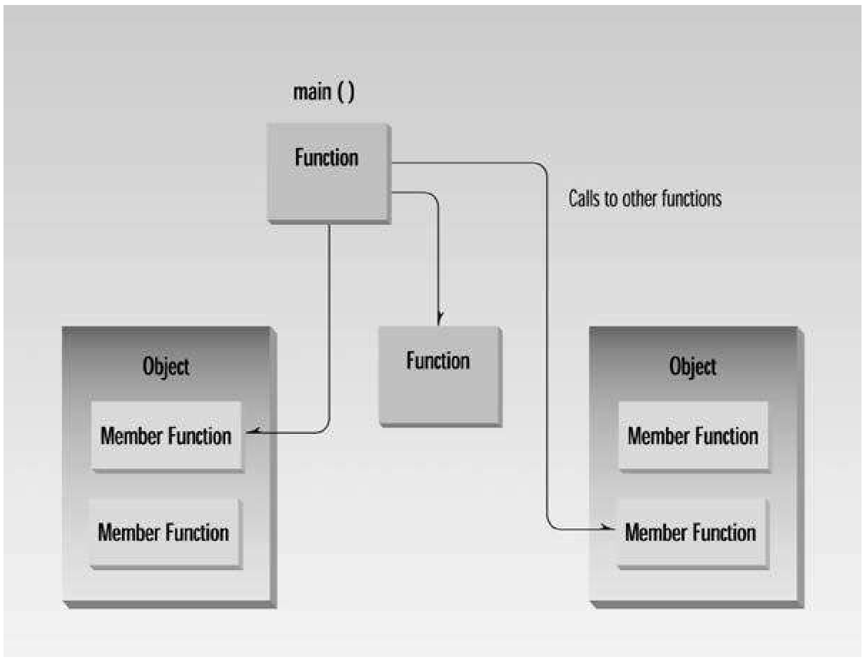

FIGURE 2.1
Objects, functions, and `main()`.

Program Statements

The program statement is the fundamental unit of C++ programming. There are two statements in the FIRST program: the line

```
cout << “Every age has a language of its own\n”;
```

and the return statement

```
return 0;
```

The first statement tells the computer to display the quoted phrase. Most statements tell the computer to do something. In this respect, statements in C++ are similar to statements in other languages. In fact, as we’ve noted, the majority of statements in C++ are identical to statements in C.

A semicolon signals the end of the statement. This is a crucial part of the syntax but easy to forget. In some languages (like BASIC), the end of a statement is signaled by the end of the line, but that’s not true in C++. If you leave out the semicolon, the compiler will often (although not always) signal an error.

The last statement in the function body is `return 0;`. This tells `main()` to return the value 0 to whoever called it, in this case the operating system or compiler. In older versions of C++ you could give `main()` the return type of `void` and dispense with the return statement, but this is not considered correct in Standard C++. We'll learn more about `return` in Chapter 5.

Whitespace

We mentioned that the end of a line isn't important to a C++ compiler. Actually, the compiler ignores whitespace almost completely. Whitespace is defined as spaces, carriage returns, line-feeds, tabs, vertical tabs, and formfeeds. These characters are invisible to the compiler. You can put several statements on one line, separated by any number of spaces or tabs, or you can run a statement over two or more lines. It's all the same to the compiler. Thus the FIRST program could be written this way:

```
#include <iostream>
using
namespace std;

int main () { cout
<<
“Every age has a language of its own\n”
; return
0;}
```

We don't recommend this syntax—it's nonstandard and hard to read—but it does compile correctly.

There are several exceptions to the rule that whitespace is invisible to the compiler. The first line of the program, starting with `#include`, is a preprocessor directive, which must be written on one line. Also, string constants, such as `“Every age has a language of its own”`, cannot be broken into separate lines. (If you need a long string constant, you can insert a backslash (\) at the line break or divide the string into two separate strings, each surrounded by quotes.)

Output Using cout

As you have seen, the statement

```
cout << “Every age has a language of its own\n”;
```

causes the phrase in quotation marks to be displayed on the screen. How does this work? A complete description of this statement requires an understanding of objects, operator overloading, and other topics we won't discuss until later in the book, but here's a brief preview.

The identifier `cout` (pronounced “C out”) is actually an object. It is predefined in C++ to correspond to the standard output stream. A stream is an abstraction that refers to a flow of data. The standard output stream normally flows to the screen display—although it can be redirected to other output devices. We’ll discuss streams (and redirection) in Chapter 12, “Streams and Files.”

The operator `<<` is called the insertion or put to operator. It directs the contents of the variable on its right to the object on its left. In FIRST it directs the string constant `“Every age has a language of its own\n”` to `cout`, which sends it to the display.

(If you know C, you’ll recognize `<<` as the left-shift bit-wise operator and wonder how it can also be used to direct output. In C++, operators can be overloaded. That is, they can perform different activities, depending on the context. We’ll learn about overloading in Chapter 8, “Operator Overloading.”)

Although the concepts behind the use of `cout` and `<<` may be obscure at this point, using them is easy. They’ll appear in almost every example program. Figure 2.2 shows the result of using `cout` and the insertion operator `<<`.

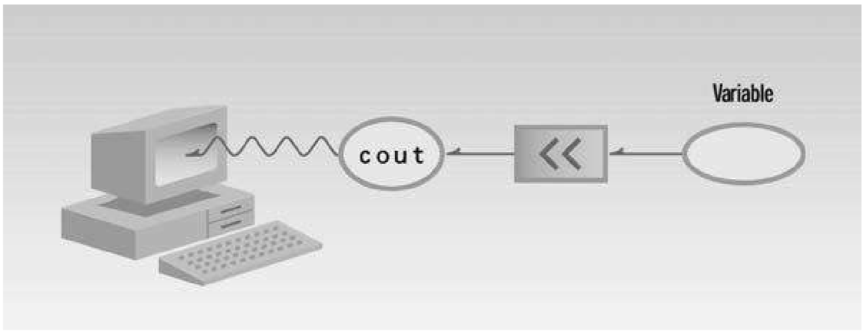

FIGURE 2.2
Output with `cout`.

String Constants

The phrase in quotation marks, `“Every age has a language of its own\n”`, is an example of a string constant. As you probably know, a constant, unlike a variable, cannot be given a new value as the program runs. Its value is set when the program is written, and it retains this value throughout the program’s existence.

As we’ll see later, the situation regarding strings is rather complicated in C++. Two ways of handling strings are commonly used. A string can be represented by an array of characters, or it can be represented as an object of a class. We’ll learn more about both kinds of strings in Chapter 7, “Arrays and Strings.”

The '\n' character at the end of the string constant is an example of an escape sequence. It causes the next text output to be displayed on a new line. We use it here so that the phrases such as "Press any key to continue," inserted by some compilers for display after the program terminates, will appear on a new line. We'll discuss escape sequences later in this chapter.

Directives

The two lines that begin the FIRST program are directives. The first is a preprocessor directive, and the second is a `using` directive. They occupy a sort of gray area: They're not part of the basic C++ language, but they're necessary anyway

Preprocessor Directives

The first line of the FIRST program

```
#include <iostream>
```

might look like a program statement, but it's not. It isn't part of a function body and doesn't end with a semicolon, as program statements must. Instead, it starts with a number sign (#). It's called a preprocessor directive. Recall that program statements are instructions to the computer to do something, such as adding two numbers or printing a sentence. A preprocessor directive, on the other hand, is an instruction to the compiler. A part of the compiler called the preprocessor deals with these directives before it begins the real compilation process.

The preprocessor directive `#include` tells the compiler to insert another file into your source file. In effect, the `#include` directive is replaced by the contents of the file indicated. Using an `#include` directive to insert another file into your source file is similar to pasting a block of text into a document with your word processor.

`#include` is only one of many preprocessor directives, all of which can be identified by the initial # sign. The use of preprocessor directives is not as common in C++ as it is in C, but we'll look at a few additional examples as we go along. The type file usually included by `#include` is called a header file.

Header Files

In the FIRST example, the preprocessor directive `#include` tells the compiler to add the source file IOSTREAM to the FIRST.CPP source file before compiling. Why do this? IOSTREAM is an example of a header file (sometimes called an include file). It's concerned with basic input/output operations, and contains declarations that are needed by the `cout` identifier and the << operator. Without these declarations, the compiler won't recognize `cout` and will think `<<` is being used incorrectly. There are many such include files. The newer Standard C++ header files don't have a file extension, but some older header files, left over from the days of the C language, have the extension .H.

If you want to see what's in IOSTREAM, you can find the `include` directory for your compiler and display it as a source file in the Edit window. (See the appropriate appendix for hints on how to do this.) Or you can look at it with the WordPad or Notepad utilities. The contents won't make much sense at this point, but you will at least prove to yourself that IOSTREAM is a source file, written in normal ASCII characters.

We'll return to the topic of header files at the end of this chapter, when we introduce library functions.

The `using` Directive

A C++ program can be divided into different namespaces. A namespace is a part of the program in which certain names are recognized; outside of the namespace they're unknown. The directive

```
using namespace std;
```

says that all the program statements that follow are within the `std` namespace. Various program components such as `cout` are declared within this namespace. If we didn't use the `using` directive, we would need to add the `std` name to many program elements. For example, in the FIRST program we'd need to say

```
std::cout << "Every age has a language of its own.";
```

To avoid adding `std::` dozens of times in programs we use the `using` directive instead. We'll discuss namespaces further in Chapter 13, "Multifile Programs."

Comments

Comments are an important part of any program. They help the person writing a program, and anyone else who must read the source file, understand what's going on. The compiler ignores comments, so they do not add to the file size or execution time of the executable program.

Comment Syntax

Let's rewrite our FIRST program, incorporating comments into our source file. We'll call the new program COMMENTS:

```
// comments.cpp
// demonstrates comments
#include <iostream>              //preprocessor directive
using namespace std;             //"using" directive
```

```
int main()                       //function name “main”
   {                             //start function body
   cout << “Every age has a language of its own\n”; //statement
   return 0;                     //statement
   }                             //end function body
```

Comments start with a double slash symbol (//) and terminate at the end of the line. (This is one of the exceptions to the rule that the compiler ignores whitespace.) A comment can start at the beginning of the line or on the same line following a program statement. Both possibilities are shown in the COMMENTS example.

When to Use Comments

Comments are almost always a good thing. Most programmers don’t use enough of them. If you’re tempted to leave out comments, remember that not everyone is as smart as you; they may need more explanation than you do about what your program is doing. Also, you may not be as smart next month, when you’ve forgotten key details of your program’s operation, as you are today.

Use comments to explain to the person looking at the listing what you’re trying to do. The details are in the program statements themselves, so the comments should concentrate on the big picture, clarifying your reasons for using a certain statement or group of statements.

Alternative Comment Syntax

There’s a second comment style available in C++:

```
/* this is an old-style comment */
```

This type of comment (the only comment originally available in C) begins with the `/*` character pair and ends with `*/` (not with the end of the line). These symbols are harder to type (since `/` is lowercase while `*` is uppercase) and take up more space on the line, so this style is not generally used in C++. However, it has advantages in special situations. You can write a multiline comment with only two comment symbols:

```
/* this
is a
potentially
very long
multiline
comment
*/
```

This is a good approach to making a comment out of a large text passage, since it saves inserting the `//` symbol on every line.