0

Getting started

Let us begin by looking at a small C++ program:

```
// a small C++ program
#include <iostream>

int main()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

Programmers often refer to such a program as a Hello, world! program. Despite its small size, you should take the time to compile and run this program on your computer before reading further. The program should write

```
Hello, world!
```

on the standard output, which will typically be a window on your display screen. If you have trouble, find someone who already knows C++ and ask for help, or consult our Website, http://www.acceleratedcpp.com, for advice.

This program is useful because it is so simple that if you have trouble, the most likely reasons are obvious typographical errors or misconceptions about how to use the implementation. Moreover, thoroughly understanding even such a small program can teach a surprising amount about the fundamentals of C++. In order to gain this understanding, we'll look in detail at each line of the program.

0.1 Comments

The first line of our program is

```
// a small C++ program
```

The // characters begin a ***comment***, which extends to the end of the line. The compiler ignores comments; their purpose is to explain the program to a human reader. In this book,

0.2 #include

In C++, many fundamental facilities, such as input-output, are part of the ***standard library***, rather than being part of the ***core language***. This distinction is important because the core language is always available to all C++ programs, but you must explicitly ask for the parts of the standard library that you wish to use.

Programs ask for standard-library facilities by using **#include *directives***. Such directives normally appear at the beginning of a program. The only part of the standard library that our program uses is input-output, which we request by writing

```
#include <iostream>
```

The name iostream suggests support for sequential, or stream, input-output, rather than random-access or graphical input-output. Because the name iostream appears in an #include directive and it is enclosed in ***angle brackets*** (< and >), it refers to a part of the C++ library called a ***standard header***.

The C++ standard does not tell us exactly what a standard header is, but it does define each header's name and behavior. Including a standard header makes the associated library facilities available to the program, but exactly how the implementation does so is its concern, not ours.

0.3 The main function

A ***function*** is a piece of program that has a name, and that another part of the program can ***call***, or cause to run. Every C++ program must contain a function named main. When we ask the C++ implementation to run a program, it does so by calling this function.

The main function is required to yield an integer as its result, the purpose of which is to tell the implementation whether the program ran successfully. A zero value indicates success; any other value means there was a problem. Accordingly, we begin by writing

```
int main()
```

to say that we are defining a function named main that returns a value of type int. Here, int is the name that the core language uses to describe integers. The parentheses after main enclose the parameters that our function receives from the implementation. In this particular example, there are no parameters, so there is nothing between the parentheses. We'll see how to use main's parameters in §10.4/179.

0.4 Curly braces

We continue our definition of the main function by following the parentheses with a sequence of ***statements*** enclosed in ***curly braces*** (often simply called braces):

```
int main()
{           // left brace
            // the statements go here
}           // right brace
```

In C++, braces tell the implementation to treat whatever appears between them as a unit. In this example, the left brace marks the beginning of the statements in our main function, and the right brace marks their end. In other words, the braces indicate that all the statements between them are part of the same function.

When there are two or more statements within braces, as there are in this function, the implementation executes them in the order in which they appear.

0.5 Using the standard library for output

The first statement inside the braces does our program's real work:

```
std::cout << "Hello, world!" << std::endl;
```

This statement uses the standard library's ***output operator***, <<, to write Hello, world! on the standard output, and then to write the value of std::endl.

Preceding a name by std:: indicates that the name is part of a ***namespace*** named std. A namespace is a collection of related names; the standard library uses std to contain all the names that it defines. So, for example, the iostream standard header defines the names cout and endl, and we refer to these names as std::cout and std::endl.

The name std::cout refers to the ***standard output stream***, which is whatever facility the C++ implementation uses for ordinary output from programs. In a typical C++ implementation under a windowing operating system, std::cout will denote the window that the implementation associates with the program while it is running. Under such a system, the output written to std::cout will appear in the associated window.

Writing the value of std::endl ends the current line of output, so that if this program were to produce any more output, that output would appear on a new line.

0.6 The return statement

A return statement, such as

```
return 0;
```

ends execution of the function in which it appears, and passes the value that appears between the return and the semicolon (0 in this example) back to the program that called the function that is returning. The value that is returned must have a type that is appropriate for the type that the function says it will return. In the case of main, the return type is int and the program to which main returns is the C++ implementation itself. Therefore, a return from main must include an integer-valued expression, which is passed back to the implementation.

Of course, there may be more than one point at which it might make sense to terminate a program; such a program may have more than one return statement. If the definition of a function promises that the function returns a value of a particular type, then every return statement in the function must return a value of an appropriate type.

0.7 A slightly deeper look

This program uses two additional concepts that permeate C++: expressions and scope. We will have much more to say about these concepts as this book progresses, but it is worthwhile to begin with some of the basics here.

An ***expression*** asks the implementation to compute something. The computation yields a ***result***, and may also have ***side effects***-that is, it may affect the state of the program or the implementation in ways that are not directly part of the result. For example, 3+4 is an expression that yields 7 as its result, and has no side effects, and

```
std::cout << "Hello, world!" << std::endl
```

is an expression that, as its side effect, writes Hello, world! on the standard output stream and ends the current line.

An expression contains operators and operands, both of which can take on many forms. In our Hello, world! expression, the two << symbols are operators, and std::cout, "Hello, world! " and std::endl are operands.

Every operand has a ***type***. We shall have much more to say about types, but essentially, a type denotes a data structure and the meanings of operations that make sense for that data structure. The effect of an operator depends on the types of its operands.

Types often have names. For example, the core language defines int as the name of a type that represents integers, and the library defines std::ostream as the type that provides stream-based output. In our program, std::cout has type std::ostream.

The << operator takes two operands, and yet we have written two << operators and three operands. How can this be? The answer is that << is ***left-associative***, which, loosely speaking, means that when << appears twice or more in the same expression, each << will use as much of the expression as it can for its left operand, and as little of it as it can for its right operand. In our example, the first << operator has "Hello, world! " as its right operand and std::cout as its left operand, and the second << operator has std::endl as its right operand and std::cout << "Hello, world! " as its left operand. If we use parentheses to clarify the relationship between operands and operators, we see that our output expression is equivalent to

```
(std::cout << "Hello, world!") << std::endl
```

Each << behaves in a way that depends on the types of its operands. The first << has

std::cout, which has type std::ostream, as its left operand. Its right operand is a string literal, which has a mysterious type that we shall not even discuss until §10.2/176. With those operand types, << writes its right operand's characters onto the stream that its left operand denotes, and its result is its left operand.

The left operand of the second << is therefore an expression that yields std::cout, which has type std::ostream; the right operand is std::endl, which is a ***manipulator***. The key property of manipulators is that writing a manipulator on a stream manipulates the stream, by doing something other than just writing characters to it. When the left operand of << has type std::ostream and the right operand is a manipulator, << does whatever the manipulator says to do to the given stream, and returns the stream as its result. In the case of std::endl, that action is to end the current line of output.

The entire expression therefore yields std::cout as its value, and, as a side effect, it writes Hello, world! on the standard output stream and ends the output line. When we follow the expression by a semicolon, we are asking the implementation to discard the value-which action is appropriate, because we are interested only in the side effects.

The ***scope*** of a name is the part of a program in which that name has its meaning. C++ has several different kinds of scopes, two of which we have seen in this program.

The first scope that we used is a namespace, which, as we've just seen, is a collection of related names. The standard library defines all of its names in a namespace named std, so that it can avoid conflicts with names that we might define for ourselves-as long as we are not so foolish as to try to define std. When we use a name from the standard library, we must specify that the name we want is the one from the library; for example, std::cout means cout as defined in the namespace named std.

The name std::cout is a ***qualified name***, which uses the :: operator. This operator is also known as the ***scope operator***. To the left of the :: is the (possibly qualified) name of a scope, which in the case of std::cout is the namespace named std. To the right of the :: is a name that is defined in the scope named on the left. Thus, std::cout means "the name cout that is in the (namespace) scope std."

Curly braces form another kind of scope. The body of main-and the body of every function-is itself a scope. This fact is not too interesting in such a small program, but it will be relevant to almost every other function we write.

0.8 Details

Although the program we've written is simple, we've covered a lot of ground in this chapter. We intend to build on what we've introduced here, so it is important for you to be sure that you understand this chapter fully before you continue.

To help you do so, this chapter-and every chapter except Chapter 16-ends with a section called *Details* and a set of exercises. The *Details* sections summarize and occasionally expand on the information in the text. It is worth looking at each *Details* section as a reminder of the ideas that the chapter introduced.

Program structure: C++ programs are usually in ***free form***, meaning that spaces are required only when they keep adjacent symbols from running together. In particular, newlines (i.e., the way in which the implementation represents the change from one line of the program to the next) are just another kind of space, and usually have no additional special meaning. Where you choose to put spaces in a program can make it much easier-or harder-to read. Programs are normally indented to improve readability.

There are three entities that are not free-form:

string literals

characters enclosed in double quotes; may not span lines

#include *name*

must appear on a line by themselves (except for comments)

// *comments*

// followed by anything; ends at the end of the current line

A comment that begins with /* is free-form; it ends with the first subsequent */ and can span multiple lines.

Types define data structures and operations on those data structures. C++ has two kinds of types: those built into the core language, such as int, and those that are defined outside the core language, such as std::ostream.

Namespaces are a mechanism for grouping related names. Names from the standard library are defined in the namespace called std.

String literals begin and end with double quotes ("); each string literal must appear entirely on one line of the program. Some characters in string literals have special meaning when preceded by a backslash (\):

\n newline character

\t tab character

\b backspace character

\" treats this symbol as part of the string rather than as the string terminator

\' same meaning as ' in string literals, for consistency with character literals (§1.2/14)

\\ includes a \ in the string, treating the next character as an ordinary character

We'll see more about string literals in §10.2/176 and §A.2.1.3/302.

Definitions and headers: Every name that a C++ program uses must have a corresponding definition. The standard library defines its names in headers, which programs access through #include. Names must be defined before they are used; hence, a #include must precede the use of any name from that header. The <iostream> header defines the library's input-output facilities.

The main function: Every C++ program must define exactly one function, named main, that returns an int. The implementation runs the program by calling main. A zero return from main indicates success; a nonzero return indicates failure. In general, functions must include at least one return statement and are not permitted to fall off the end of the function. The main function is special: It may omit the return; if it does so, the implementation will assume a zero return value. However, explicitly including a return from main is good practice.

Braces and semicolons: These inconspicuous symbols are important in C++ programs. They are easy to overlook because they are small, and they are important because forgetting one typically evokes compiler diagnostic messages that may be hard to understand.

A sequence of zero or more statements enclosed in braces is a statement, called a ***block***, which is a request to execute the constituent statements in the order in which they appear. The body of a function must be enclosed in braces, even if it is only a single statement. The statements between a pair of matching braces constitute a scope.

An expression followed by a semicolon is a statement, called an ***expression statement***, which is a request to execute the expression for its side effects and discard its result. The expression is optional; omitting it results in a ***null statement***, which has no effect.

Output: Evaluating std::cout << e writes the value of e on the standard-output stream, and yields std::cout, which has type ostream, as its value in order to allow chained output operations.

Exercises