1

A Crash Course in C++

The goal of this chapter is to cover briefly the most important parts of C++ so that you have a base of knowledge before embarking on the rest of the book. This chapter is not a comprehensive lesson in the C++ programming language. The very basic points (like what a program is and the difference between = and ==) are not covered. The very esoteric points (remember what a `union` is? how about the `volatile` keyword?) are also omitted. Certain parts of the C language that are less relevant in C++ are also left out, as are parts of C++ that get in-depth coverage in later chapters.

This chapter aims to cover the parts of C++ that programmers encounter on a daily basis. If you've been away from C++ for a while and you've forgotten the syntax for a `for` loop, you'll find that in this chapter. If you're fairly new to C++ and you don't understand what a reference variable is, you'll learn that here as well.

If you already have significant experience with C++, skim this chapter to make sure that there aren't any fundamental parts of the language on which you need to brush up. If you're new to C++, take the time to read this chapter carefully and make sure that you understand the examples. If you need additional introductory information, consult the titles listed in Appendix B.

The Basics of C++

The C++ language is often viewed as a "better C" or a "superset of C." Many of the annoyances or rough edges of the C language were addressed when C++ was designed. Because C++ is based on C, much of the syntax you'll see in this section will look familiar to you if are an experienced C programmer. The two languages certainly have their differences, though. As evidence, *The C++ Programming Language* by C++ creator Bjarne Stroustrup weighs in at 911 pages, while Kernighan and Ritchie's *The C Programming Language* is a scant 274 pages. So if you're a C programmer, be on the lookout for new or unfamiliar syntax!

The Obligatory Hello, World

In all its glory, the following code is the simplest C++ program you're likely to encounter.

```
// helloworld.cpp

#include <iostream>

int main(int argc, char** argv)
{
    std::cout << "Hello, World!" << std::endl;

    return 0;
}
```

This code, as you might expect, prints the message Hello, World! on the screen. It is a simple program and unlikely to win any awards, but it does exhibit several important concepts about the format of a C++ program.

Comments

The first line of the program is a *comment*, a message that exists for the programmer only and is ignored by the compiler. In C++, there are two ways to delineate a comment. In the preceding example, two slashes indicate that whatever follows on that line is a comment.

```
// helloworld.cpp
```

The same behavior (this is to say, none) would be achieved by using a *C-style comment*, which is also valid in C++. C-style comments start with `/*` and end with `*/`. In this fashion, C-style comments are capable of spanning multiple lines. The code below shows a C-style comment in action (or, more appropriately, inaction).

```
/* this is a multiline
 * C-style comment. The
 * compiler will ignore
 * it.
 */
```

Comments are covered in detail in Chapter 7.

Preprocessor Directives

Building a C++ program is a three-step process. First, the code is run through a *preprocessor*, which recognizes metainformation about the code. Next, the code is *compiled*, or translated into machine-readable object files. Finally, the individual object files are *linked* together into a single application. Directives that are aimed at the preprocessor start with the `#` character, as in the line `#include <iostream>` in the previous example. In this case, an include directive tells the preprocessor to take everything from the iostream header file and make it available to the current file. The most common use of header files is to declare functions that will be defined elsewhere. Remember, a *declaration* tells the compiler how a function is called. A *definition* contains the actual code for the function. The iostream header declares the input and output mechanisms provided by C++. If the program did not include it, it would be unable to perform its only task of outputting text.

In C, included files usually end in .h, such as `<stdio.h>`. In C++, the suffix is omitted for standard library headers, such as `<iostream>`. Your favorite standard headers from C still exist in C++, but with new names. For example, you can access the functionality from `<stdio.h>` by including `<cstdio>`.

The table below shows some of the most common preprocessor directives.

Preprocessor Directive	Functionality	Common Uses
`#include [file]`	The specified file is inserted into the code at the location of the directive.	Almost always used to include header files so that code can make use of functionality that is defined elsewhere.
`#define [key] [value]`	Every occurrence of the specified key is replaced with the specified value.	Often used in C to define a constant value or a macro. C++ provides a better mechanism for constants. Macros are often dangerous so `#define` is rarely used in C++. See Chapter 12 for details.

`#ifdef [key]`
`#ifndef [key]`
`#endif`
	Code within the `ifdef` ("if defined") or `ifndef` ("if not defined") blocks are conditionally included or omitted based on whether the specified value has been defined with `#define`.	Used most frequently to protect against circular includes. Each included file defines a value initially and surrounds the rest of its code with a `#ifndef` and `#endif` so that it won't be included multiple times.

`#pragma`	Varies from compiler to compiler. Often allows the programmer to display a warning or error if the directive is reached during preprocessing.	Because usage of `#pragma` is not standard across compilers, we advocate not using it.

The main function

`main()` is, of course, where the program starts. An `int` is returned from `main()`, indicating the result status of the program. `main()` takes two parameters: `argc` gives the number of arguments passed to the program, and `argv` contains those arguments. Note that the first argument is always the name of the program itself.

I/O Streams

If you're new to C++ and coming from a C background, you're probably wondering what `std::cout` is and what has been done with trusty old `printf()`. While `printf()` can still be used in C++, a much better input/output facility is provided by the streams library.

I/O streams are covered in depth in Chapter 14, but the basics of output are very simple. Think of an output stream as a laundry chute for data. Anything you toss into it will be output appropriately. `std::cout` is the chute corresponding to the user console, or *standard out*. There are other chutes, including `std::cerr`, which outputs to the error console. The `<<` operator tosses data down the chute. In the preceding example, a quoted string of text is sent to standard out. Output streams allow multiple data of varying types to be sent down the stream sequentially on a single line of code. The following code outputs text, followed by a number, followed by more text.

```
std::cout << "There are " << 219 << " ways I love you." << std::endl;
```

`std::endl` represents an end of line character. When the output stream encounters `std::endl`, it will output everything that has been sent down the chute so far and move to the next line. An alternate way of representing the end of a line is by using the '`\n`' character. The `\n` character is an *escape character*, which refers to a new-line character. Escape characters can be used within any quoted string of text. The list below shows the most common escape characters.

- **`\n`** new line
- **`\r`** carriage return
- **`\t`** tab
- **`\\`** the backslash character
- **`\"`** quotation mark

Streams can also be used to accept input from the user. The simplest way to do this is to use the `>>` operator with an input stream. The `std::cin` input stream accepts keyboard input from the user. User input can be tricky because you can never know what kind of data the user will enter. See Chapter 14 for a full explanation of how to use input streams.

Namespaces

Namespaces address the problem of naming conflicts between different pieces of code. For example, you might be writing some code that has a function called `foo()`. One day, you decide to start using a third-party library, which also has a `foo()` function. The compiler has no way of knowing which version of `foo()` you are referring to within your code. You can't change the library's function name, and it would be a big pain to change your own.

Namespaces come to the rescue in such scenarios because you can define the context in which names are defined. To place code in a namespace, simply enclose it within a namespace block:

```
 // namespaces.h

namespace mycode {
    void foo();
}
```

The implementation of a method or function can also be handled in a namespace:

```
// namespaces.cpp

#include <iostream>
#include "namespaces.h"

namespace mycode {

    void foo() {
        std::cout << "foo() called in the mycode namespace" << std::endl;
    }
}
```

By placing your version of `foo()` in the namespace "mycode," it is isolated from the `foo()` function provided by the third-party library. To call the namespace-enabled version of `foo()`, prepend the namespace onto the function name as follows.

```
mycode::foo();    // Calls the "foo" function in the "mycode" namespace
```

Any code that falls within a "mycode" namespace block can call other code within the same namespace without explicitly prepending the namespace. This implicit namespace is useful in making the code more precise and readable. You can also avoid prepending of namespaces with the `using` directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace. The namespace is thus implied for the code that follows:

```
// usingnamespaces.cpp

#include "namespaces.h"

using namespace mycode;

int main(int argc, char** argv)
{
    foo();  // Implies mycode::foo();
}
```

A single source file can contain multiple `using` directives, but beware of overusing this shortcut. In the extreme case, if you declare that you're using every namespace known to humanity, you're effectively eliminating namespaces entirely! Name conflicts will again result if you are using two namespaces that contain the same names. It is also important to know in which namespace your code is operating so that you don't end up accidentally calling the wrong version of a function.

You've seen the namespace syntax before — we used it in the Hello, World program. `cout` and `endl` are actually names defined in the `std` namespace. We could have rewritten Hello, World with the `using` directive as shown here: