

THE EXPERT'S VOICE® IN GAME DEVELOPMENT

Learn C++ for Game Development

Bruce Sutherland

Apress®

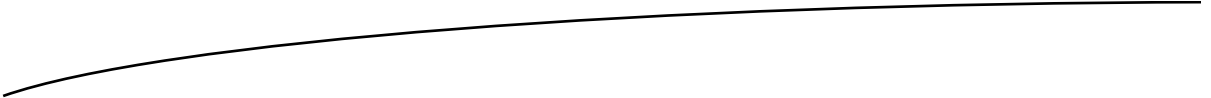
For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.



Contents at a Glance

About the Author	xv
About the Technical Reviewer	xvii
Acknowledgments	xix
Introduction	xxi
■ Chapter 1: Beginning C++	1
■ Part 1: Procedural Programming.....	5
■ Chapter 2: Writing a Guessing Game with C++ Types	7
■ Chapter 3: Creating Calculators with Operators.....	17
■ Chapter 4: Beginning C++ Game Development with Arrays	37
■ Chapter 5: Functions, the Building Blocks of C++	49
■ Chapter 6: Making Decisions with Flow Control.....	57
■ Chapter 7: Organizing Projects Using Files and Namespaces	71
■ Part 2: Object-Oriented Programming	79
■ Chapter 8 : Object-Oriented Programming with Classes	81
■ Chapter 9 : Controlling Data with Access Modifiers.....	97
■ Chapter 10: Building Games with Inheritance.....	111

■ Chapter 11: Designing Game Code with Polymorphism	127
■ Chapter 12: Copying and Assigning Data to Objects	139
■ Part 3: The STL.....	145
■ Chapter 13: The STL String Class	147
■ Chapter 14: STL Array and Vector	157
■ Chapter 15: STL List.....	163
■ Chapter 16: STL's Associative Containers.....	169
■ Chapter 17: STL's Stack and Queue	177
■ Chapter 18: STL's bitset	181
■ Chapter 19: Using the STL in Text Adventure.....	185
■ Part 4: Generic Programming	199
■ Chapter 20: Template Programming.....	201
■ Chapter 21: Practical Template Programming	205
■ Part 5: C++ Game Programming	223
■ Chapter 22: Managing Memory for Game Developers	225
■ Chapter 23: Useful Design Patterns for Game Development.....	239
■ Chapter 24: Using File IO to Save and Load Games	253
■ Chapter 25: Speeding Up Games with Concurrent Programming.....	267
■ Chapter 26: Supporting Multiple Platforms in C++	277
■ Chapter 27: Wrapping Up.....	285
■ Index.....	291



Introduction

The C++ programming language remains the de facto language for many game development studios around the world. This is true because C++ provides a convenient, low-level programming language that allows developers to straddle the line between high-level software engineering and low-level coding to the metal. This makes it the language of choice for many high-performance and real-time computer programming projects.

C++ is a language in active development. Many of the examples in this book use features that were ratified and released in the C++11 standard. Some of the features used are not yet supported by one of the major compiler vendors. This active and progressive language development is a major reason for C++ remaining a widely used and relevant language. The fact that it's the only way to write portable, in-house code across all of the types of computers and processors you find today makes learning C++ a handy skill to have. Devices that support C++ range from mobile phones, game consoles, ATMs, smart watches, to even glasses! C++ programmers will never be short of work. I hope you find this book an invaluable resource in your journey to learning the C++ programming language.

Chapter 1

Beginning C++

The C++ programming language was designed by Bjarne Stroustrup at Bell Labs, beginning in 1979. The goal of the language was to provide useful features from other languages of the time, such as Simula, with faster runtime performance. C was chosen as the basis for this new language due to its execution speed, portability, and wide adoption.

The first major extension Stroustrup added to C was class support. Classes allowed a new programming paradigm to be used with C: object-oriented programming (OOP). Stroustrup's new language quickly became known as *C with Classes*, eventually changed to C++ in 1983.

C++ has been in continual development since its inception. New features were added regularly throughout the 1980s and 1990s and many of these have become essential tools for game developers. Examples of these features that are covered in this book are *virtual functions*, *templates*, and the *Standard Template Library*.

An ISO standard for C++ was first published in 1998. Standardizing a language gives compiler writers a common set of features to implement if they wish to achieve C++ compatibility. Standardization also benefits programmers writing code in C++ as it allows them to use a common set of features that they can expect to behave in the same way when using multiple compilers or multiple operating systems and computer architectures. There have been four C++ standards released to date. The original standard is known as C++98. C++98 was added to with C++03, and new experimental features were added to the language with C++TR1 (C++ Technical Review 1). The current C++ standard is C++11.

This book focuses on the most up-to-date version of the standard. However, this presents us with some challenges, as not all compilers have been updated to support all of the latest features of C++'. It can be taken for granted that when I refer to C++ in this book I mean C++11; any topics that require explicit reference to older C++ standards will be clear. The Microsoft' C++ compiler included with its Visual Studio integrated development environment (IDE), for example, does not include support for constant expressions or type aliasing. (Don't worry if you don't understand these terms; they are new C++11 features and are covered in the section relating to templates.) Notes are added to the text in sections where relevant features are not supported by one of the major compilers.

Compilers

C++ is a compiled language; this means that the code is read by a program that converts the text source files created by a programmer into instructions that can be executed by a processor. Different settings in compilers, or different compilers, can be used to turn source files into instructions suitable for completely different processor architectures, such as x86 and Armv7.

The major compilers currently in use are the following:

- Microsoft Visual C++ Compiler (MSVC)
- GNU Compiler Collection (GCC)
- Apple LLVM
- Clang

These are the compilers that this book is most concerned with, as they are the most likely to be encountered by game developers. MSVC is the compiler included with the Microsoft Visual Studio IDE and used when writing programs for Windows and Microsoft game consoles such as the Xbox One. GCC is common on Linux platforms and is the default compiler used when building native Android applications. Apple LLVM is the C++ compiler used when building OS X and iOS applications. Clang is a relatively new program that acts as the front end for the LLVM compiler (also used in the Apple ecosystem) and is included as an optional compiler by the current Android Native Development Kit (NDK).

Some of the samples included in this book might not compile on your compiler of choice if that compiler is incompatible with all of the latest features of C++11. Switching compilers if possible or updating your software to the latest versions are possible options to work with these features.

Programming Paradigms

Programming paradigms are closely related to high-level programming styles. C++ does not enforce a particular paradigm and provides flexible features to enable programmers to program in the following paradigms.

Procedural Programming

Procedural programming involves writing ordered statements in blocks known as functions (also known as procedures). Functions are used to encapsulate code that can be reused and to improve code readability. C is also a procedural language and C++ contains the entire C language as a subset, allowing C++ programs to be written in a fully procedural manner. Part 1 of this book covers the aspects of C++ necessary to write procedural programs.

Object-Oriented Programming

OOP is supported through the addition of classes to C++. OOP involves designing a program as a set of discrete objects. These objects hide their implementation and data from the calling code (encapsulation), which allows specific implementation details to be changed at a later time without

affecting other sections of the program. Inheritance and polymorphism are other important aspects of the C++ OOP paradigm. OOP programming is the focus of Part 2 in this book.

Generic Programming

Generic programming is most likely the least understood paradigm in C++ despite its widespread use via the Standard Template Library. Templates allow programmers to write classes that can operate on different types via specializations. Template metaprogramming is a powerful technique that confers the ability to write programs within the code that generate values and code at compile time. Templates and Metaprogramming are covered in Part 4 of this book.

C++ Game Programming

C++ has been a major language of choice for game developers since its inception. All major gaming platforms have supported and currently support C++ compilation. This includes Windows, Linux, OS X, iOS, Xbox One, PlayStation 4, Wii U, and Android.

Part 5 of this book covers advanced C++ topics that programmers will find useful as hardware becomes more powerful. These include design patterns, streams, memory management, concurrency (multithreaded programming), and a look at how we can write code that supports multiple game development platforms.

Our First C++ Program

Let's take a look at a simple C++ program before we get started with the details of C++. Listing 1-1 shows a simple C++ program written in Microsoft Visual Studio Express 2012 for Windows Desktop.

Listing 1-1. Our First C++ Program

```
// Chapter1.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <iostream>
#include <string>

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    cout << "What is your name?" << endl;

    string name {};
    cin >> name;

    cout << "You said your name is: " << name << endl;

    return 0;
}
```

The first two lines of this source code are comments. A double forward slash indicates that all text following the `//` on the same line should be ignored by the compiler. Block comments are also supported and are opened with `/*` and closed with `*/`. You will see several uses of both forms of comment throughout this book.

There are then several lines beginning with `#include`. This tells the compiler to include other files into our program. We cover the difference between source and header files in Chapter 6.

The next line tells the compiler that we want to use the standard namespace. Namespaces are also covered in Chapter 6.

The program then begins with the `_tmain` function. This is the first function called by the program when writing Windows applications that target the console. Different operating systems have different entry points and you'll see these throughout this book as different examples are written to target different operating systems and compilers. You should be able to easily alter the samples to use the entry point for your system of choice.

The `cout` operator writes the text following the `<<` operator to the console window. The `cin` operator stops the program and waits for the user to enter text. We store the text entered into the `name` variable and then use it to echo back the input the user entered.

The last line returns the value 0 from the function to let the operating system know that we finished without error.

As you can see, this simple program asks for the user's name and then repeats that name back to the user. Congratulations: You've written your first C++ program.

Summary

This chapter has given you a very brief introduction to the C++ programming language. We've looked at the basic history of the language, the current main C++ compilers, its major paradigms, and a simple C++ program.

The next chapters look at how we can create some simple games using the C features of C++, beginning with a look at the fundamental built-in types and operators in C++.

The examples in Chapters 2 and 3 are simple programs that will help you to understand how C++ types and operators behave. The examples beginning in Chapter 4 see us begin to create a text adventure game that we will develop to completion throughout the remainder of the book. Text adventure games were very popular on early computers and they are an excellent way to learn C++ for game development without requiring you to learn the intricacies of a platform-specific windowing system or graphics application programming interface (API).

Procedural Programming

The roots of the C++ programming language come from the C programming language. C provides the basis for the C++ procedural programming paradigm.

Procedural programming involves using variables and functions to create reusable procedures that constitute a full program. This part of the book introduces you to the features provided in C++ that would allow you to write fully procedural programs.

This part of the book begins by looking at C++'s built-in types, operators, arrays, pointers and references, and functions before finally looking at flow control statements and methods of structuring your programs.

Writing a Guessing Game with C++ Types

You will begin the process of learning C++ by learning about the different built-in types that the language supports. C++ has built-in support for integers, floating point numbers, booleans, and enumerators. This chapter covers all of these types. We also look at the different types of operators that can be used to manipulate and compare these types of values. It is useful to understand how C++ handles variables before we begin to create programs in the language. To start, we take a look at the difference between dynamically and statically typed languages, which includes how we declare and define variables in C++.

Dynamic Versus Static Typed Languages

Variables in programming languages can be handled in two different ways. Dynamically typed languages do not require the programmer to state explicitly which type should be used to store the value. Dynamically typed languages can switch the type of a variable on the fly depending on how it is being used.

Statically typed languages require the programmer to tell the compiler explicitly which type to use to represent the data the variable will store. C++ is a statically typed language and creating variables in C++ is a two-step process.

Declaring Variables

When we are introducing a new variable into a program, we are said to be declaring the variable. This essentially means that we are telling the compiler that we would like to have a variable of a specific type and with a specific name. C++ compilers require that all variables be declared before they can be used in our programs. An example of a variable declaration would be:

```
int numberOfObjects;
```

This declaration tells the compiler that we would like to have an integer with the name `numberOfObjects`; it does not tell the compiler what the variable contains or what it should be used for.

Defining Variables

We can define our variable once we have declared it. Defining a variable involves telling the compiler what its initial value should be. This is an example of a variable definition:

```
numberOfObjects = 0;
```

It's fairly common that we will define and declare some variables at the same time. Combining the declaration and definition would resemble this line:

```
int numberOfObjects = 0;
```

We'll be looking at the places where it is appropriate to declare names to the compiler compared to defining them as we move through this book. The rules for declaring and defining different variables and types depend on the context in which they are being used.

Integers

The first set of types that we will be looking at in C++ is the integer types. The previous sections have already shown how to declare and define an integer variable. To recap:

```
int numberOfObjects = 0;
```

This line of code declares and defines an integer variable. What this means to the compiler is that we want to create a variable that is capable of storing whole numbers. Examples of integers are `-1000`, `0`, `24`, and `1345219`. Integers can contain both positive and negative numbers.

The type `int` is just one of several different types that can store integers. The others are `char`, `short`, and `long`. Table 2-1 shows the different ranges of values that can be stored in these types when using MSVC.

Table 2-1. *The MSVC Integral Types, with Minimum and Maximum Values*

Type Name	Number of Bytes	Minimum Value	Maximum Value
char	1	-128	127
short	2	-32,768	32,767
int	4	-2,147,483,648	2,147,483,647
long	4	-2,147,483,648	2,147,483,647
long long	8	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

As you can see there are five main types of integers when using C++. It's important to select a variable that can accommodate all of the possible values you might wish to store. Failing to use a type with enough possible values results in wrapping. That means that if you had a `char` variable that currently stored 127 and you tried to add a one, the value in the variable would *wrap* and your variable would then contain -128.

Unsigned Integers

C++ also supports unsigned versions of the integer types shown in Table 2-1. Table 2-2 shows the ranges that these unsigned variable types can store.

Table 2-2. MSVC's Unsigned Integral Type Ranges

Type Name	Number of Bytes	Minimum Value	Maximum Value
unsigned char	1	0	255
unsigned short	2	0	65,535
unsigned int	4	0	4,294,967,295
unsigned long	4	0	4,294,967,295
unsigned long long	8	0	18,446,744,073,709,551,615

Using unsigned types extends the range of available positive integers at the expense of losing all negative integers. These types should be used in all cases where negative numbers will not be needed.

Two's Complement

Before we move on to look at some other types of variables, it is important to understand how the integer values we have covered are represented by the computer. You've seen that unsigned numbers of varying numbers of bytes can store numbers ranging from 0 through to their maximum size. In the case of a single-byte `char` that maximum number is 255.

This happens because a single byte is made up of 8 individual bits. Each bit represents the next power of two from the last. Table 2-3 shows how you can think of the bits in a byte.

Table 2-3. The Bits Making Up a Single Byte: The Top Row Shows the Bit Number and the Bottom Row Shows the Value This Byte Represents When Using Unsigned Variables

8	7	6	5	4	3	2	1
128	64	32	16	8	4	2	1

Table 2-3 shows the bit values for an unsigned `char`. When we have an unsigned `char` variable that stores the value 255, all of the bits are *turned on*. This would be represented by 11111111 in binary. If we add a 1 to this number the values will *wrap* and we will end up back at 00000000 as the 8 bits cannot store a higher value. This is something that you will need to be aware of when using unsigned integer variables.

When you want to represent negative numbers using signed variables, the processor will use *Two's Complement*. Two's Complement works by using the highest bit to represent whether the number is positive or negative. The highest positive number that can be stored in a `char` variable is 127 and is represented in binary by 01111111. Adding 1 to this number gives us 10000000, which is -128 and once again shows you how wrapping occurs in practice.

Switching between a positive and negative number is a simple process. You have to switch all of the bits and add one. The number 1 is represented in binary by 00000001. To flip 1 to -1 we first flip all of the bits to 11111110 (which is -2), then add 1 to give us 11111111.

This is a brief introduction to how a processor represents the integer values your programs will store in memory. We will cover some more binary representations of numbers in Chapter 3 when we look at binary operators.

Floating Point Numbers

Floating point numbers in C++ are used to represent numbers with a decimal point. Examples of these are -10.5, 0.3337, and 89.8376.

Like the signed integer types, floats can be used to store positive and negative numbers. Unlike integers, though, there are no unsigned versions of floating point numbers. Defining a floating point is done in the following way:

```
float decimalNumber = 1.0f;
```

When defining floating point variables in code, the value must be followed by an `f`. The reason for this is that a number with a decimal point but no `f` is a double.

The current floating point standards require that floats be represented with 32 bits (or 4 bytes). Thirty-two-bit floating point numbers might not have a large enough range for what we would like to achieve under given circumstances, so C++ also supplies the double type. The float type can be thought of as a single precision floating point number and the double type is then a double precision floating point number. A double can be defined as follows:

```
double doubleNumber = 1.0;
```

Single precision floats are usually sufficient for all but a few scenarios in game development and are generally faster and more efficient for the types of tasks we carry out.

Boolean Values

Booleans are simple values that can only contain true or false. Booleans are usually, but not always, implemented as a single byte char. A value of 0 is usually used to represent false and any nonzero value is true. We rarely use numbers to represent booleans, as C++ contains the true and false keywords. A couple of boolean variable declarations would be:

```
bool isTrue = true;
bool isFalse = false;
```

We've already seen in the previous section that functions can return boolean values to help create statements that make sense to a human reader:

```
bool numberIsNaN = isnan(nan);
```

Booleans will also become useful once we start looking at flow control statements in Chapter 5.

Enums

C++ allows us to create an enumerated list of values. This is useful if we wish to convey sets of things that have readable names. An example of an enum in C++ is:

```
enum class Color
{
    Red,
    Green,
    Blue
};
```

This has effectively contained a new type in C++, Color. We can create new variables of type Color like this:

```
Color currentColor = Color::Red;
```

Using enums is a useful way to create code that is highly readable. Compare the preceding line to this:

```
unsigned int currentColor = 0;
```

This code essentially does exactly the same as the preceding line, but it is much more difficult to understand. We have no way of knowing that 0 was the color red and it was also possible to assign the number 10 to currentColor, which wouldn't be a valid color.

Enums also allow us to assign a value to each element. In the `Color` enum we currently have `Red` assigned the value 0, `Green` 1 and `Blue` 2. In the enum definition we could have used the following:

```
enum class Color
{
    Red,
    Green = 10,
    Blue
};
```

In the preceding enum, `Red` is still 0, and `Green` is 10; however, `Blue` is 11, as it directly follows `Green`. There are few occasions where we need to specify the values of enums as they are most useful for creating more readable code; but it is useful to understand that this is possible because you will come across code written by others that will use this feature.

It's also important to note that the `enum class` is a C++11 construct. Prior to C++11 the enum would have been created using the following:

```
enum Color
{
    Red,
    Green,
    Blue
};
```

C++11 style enums are known as strongly typed enums. The compiler will not allow you to accidentally use integers in place of the enum values. It also wouldn't have been possible to use the value `Red` in two different enums.

```
enum Color
{
    Red,
    Green,
    Blue,
};

enum TrafficLights
{
    Red,
    Amber,
    Green
};
```

This is not allowed with older enums, as the code to define a variable would look like:

```
Color currentColor = Red;
TrafficLights = currentLight = Red;
```

As you can see, we have no way of knowing which `Red` is to be used. This is even worse with the `Green` value, as it would have a different integer representation: 1 in `Color` and 2 in `TrafficLights`. This could cause bugs in our code as it would be valid to use the `TrafficLights` version of `Green`

when we really meant to use the `Color` version. The compiler would assume that we meant `Blue` as both `Color::Blue` and `TrafficLights::Green` are representations of 2. Using an enum class, however, means the code must look like this:

```
Color currentColor = Color::Red;
TrafficLights = currentLight = TrafficLights::Red;
```

As you can see, the strongly typed version of enums is much better and provides us with more explicit code. Always use an enum class where possible in new code.

Enums are the last type introduced in this chapter. We can now spend the rest of the chapter looking at the different types of operators that C++ supplies to work with our types.

Switching from One Type to Another

Sometimes while writing our programs we will have a value stored as one type, such as an `int`, but we would like to use the value from our variable along with a value stored in another variable. Fortunately C++ can automatically convert from some types to others. The following lines of code provide some examples:

```
char charNumber = 1;
int intNumber = charNumber;
float floatNumber = intNumber;
```

C++ can implicitly convert numbers from one type to another. Going from smaller types to larger types and from integer types to floats and floats to doubles as well as from integer types to bools are all legal and carried out automatically by the compiler.

static_cast

Whereas moving from a smaller type to a larger type is safe, moving from a larger type to a smaller type can cause a loss of data. For example:

```
int largeNumber = 1000;
char smallNumber = largeNumber;
```

In this case, the large number would likely be truncated from 1,000 to 127, as that is the largest number that a signed `char` variable can hold. This type of data loss is known as a narrowing conversion. C++ compilers give warnings about this type of conversion and many game development studios like to keep their code error-free or enable compiler settings that cause all warnings to become errors. Fortunately C++ provides us with a method to tell the compiler that we were converting this value on purpose. This method is known as a static cast. An example of the usage of `static_cast` would be:

```
char smallNumber = static_cast<char>(largeNumber);
```

This code tells the compiler that we are purposefully converting our type to a `char` and it will not output any warnings for the conversion.

It's also good practice to use `static_cast` when carrying out conversions to larger types (actually called widening conversions) to make it clear and obvious to other programmers that we have used the conversion on purpose and that it is not actually a mistake.

`static_cast` is evaluated by the compiler at compile time and will cause a compilation error if we are trying to convert between incompatible types.

A Simple Guessing Game

We'll use what you have learned about declaring and defining variables and integers to create a simple guessing game. In the beginning the game will be basic; however, we will add to the game throughout the remainder of this chapter to make the output more like a game.

For now we will ask the player to input a number and output that number and a random number selected by the program. Listing 2-1 shows the program written to compile using C++ 4.7.3.

Listing 2-1. A Simple C++ Guessing Game

```
#include <ctime>
#include <cstdlib>
#include <iostream>
#include <string>

using namespace std;

int main()
{
    // Generate unique random numbers using the current time
    srand(time(NULL));
    // Get a random number between 0 and 99
    unsigned int numberToGuess = rand() % 100;

    cout << "Guess a number between 0 and 99" << endl;

    unsigned int playersNumber {};
    cin >> playersNumber;

    cout << "You guessed: "
         << playersNumber
         << " The actual number was: "
         << numberToGuess
         << endl;

    return 0;
}
```

Our guessing game source code begins by including the C++ header files necessary for the features we use and declares that we will be using the `std` namespace.

```
#include <ctime>
#include <cstdlib>
#include <iostream>
#include <string>
```

```
using namespace std;
```

The entry point for our program when using GCC is:

```
int main()
```

Listing 1-1 showed that the entry point when using MSVC was:

```
int _tmain(int argc, _TCHAR* argv[])
```

The first four lines of the main function are used to obtain a random number. The `srand` function carries out an operation known as seeding. This uses the current time returned from the time function to generate a sequence of random numbers.

```
// Generate unique random numbers using the current time
srand(time(NULL));
```

We declare and define a variable, `numberToGuess`, to store the value that we would like our player to guess. The `rand` function returns the number and we use the modulus operator (%) to ensure that the number remains less than 100. We cover the modulus operator in more detail in Chapter 3.

```
// Get a random number between 0 and 99
unsigned int numberToGuess = rand() % 100;
```

The remaining code is similar to Listing 1-1. We use `cout` to write text for the player to read and use `cin` to allow the player to enter a guessed number. We finish by returning 0.

```
cout << "Guess a number between 0 and 99" << endl;
```

```
unsigned int playersNumber {};
cin >> playersNumber;
```

```
cout << "You guessed: "
      << playersNumber
      << " The actual number was: "
      << numberToGuess
      << endl;
```

```
return 0;
```

Summary

This chapter has given you a brief introduction to the built-in data types provided by C++ as well as also taking a look at enums and the usage of `static_cast` to convert between these built-in types.

We then used this knowledge to create a very simple program to guess a number chosen by the computer. At the moment, we do not have the ability to test whether the player guessed correctly; that will come later in Part 1 when we look at flow control statements in Chapter 6. Chapter 3 covers the operators that are available in C++. Operators are the means with which C++ allows us to create logical programs that calculate and compute outcomes from our statements.

Creating Calculators with Operators

C++ provides operators that allow you to express all sorts of mathematical formulae in computer programs. You have access to operators that allow us to carry out algebra, compare values, and even manipulate the individual bit pattern of a variable's value directly. This chapter is going to cover these C++ operators and end by having you create a simple command line calculator.

The Assignment Operator

The first operator we are going to look at is the assignment operator. We have already been using the assignment operator in the previous chapters as it is the operator that allows us to assign a value to a variable. This operator uses the = symbol:

```
int number1 = 1;
```

As you can see we have assigned the value of 1 to an integer variable named number1. We can also assign variable values to other variables:

```
int number2 = number1;
```

Here the program will read the value stored in number1 and assign that value to the variable number2. As we saw in the previous chapter, the assignment operator can also automatically convert from one variable type to another.

```
char charNumber = 1;  
int intNumber = charNumber;
```

The assignment is one of the most widely used operators in C++; thankfully, it's also easy to use and fairly obvious in nature.

The assignment operator can also be used in some pieces of code that aren't necessarily very readable. Take the following line, for instance:

```
number1 = number2 = number3 = 1;
```

C++ compilers will evaluate these statements from right to left. 1 is assigned to `number3`, which would be assigned to `number2`, which would be assigned to `number1`, and all three variables will end up storing 1. I wouldn't recommend writing code like this.

Arithmetic Operators

C++ also allows us to carry out basic arithmetic operations and we have five of these available:

- Addition operator
- Subtraction operator
- Multiplication operator
- Division operator
- Modulo operator

The Addition Operator

The addition operator behaves exactly as you would expect: It allows us to add two numbers together. The addition operator can add a value from the right to a value on the left and the result can be used in a statement, like this:

```
int sum = number1 + number2;
```

Here we have added `number2` to `number1` and stored the result in a third variable named `sum`.

It is also possible to use the same values in all of the places in our addition statement, for example:

```
int number = 1;  
number = number + number;
```

Here we end up with 2 being stored in the variable `number` after this code has executed.

The Subtraction Operator

The subtraction operator works exactly like the addition operator, although unsurprisingly it subtracts one number from another and creates a result:

```
int number1 = 2;  
int number2 = 1;  
int result = number1 - number2;
```

This code would cause the value 1 to be stored in the variable `result`.

The Multiplication and Division Operators

These operators are also fairly self-explanatory; the multiplication operator multiplies numbers together and the division operator divides the number on the left by the number on the right:

```
int number1 = 4;
int number2 = 2;
int multiplied = number1 * number2;
int divided = number1 / number2;
```

The result of the multiplication would be 8 and the result of the division would be 2.

The division operator, like the subtraction operator, has a result that depends on the order of the variables.

```
int divided1 = number1 / number2;
int divided2 = number2 / number1;
```

After executing these lines of code, `divided1` would store 2 as earlier but `divided2` would store 0 as 4 cannot divide into 2. You can think of the results of the division as being rounded down to the nearest whole integer. The following example shows a similar result.

```
int number1 = 5;
int number2 = 2;
int divided1 = number1 / number2;
int divided2 = number2 / number1;
```

In the preceding code, `divided1` will contain 2 after execution. The actual math would be $5/2 = 2.5$ or 2 remainder 1, but as integers can only store whole numbers we lose the extra 0.5 or remainder. $2/5 = 0$ remainder 5, and again we lose the remainder; therefore `divided2` will store 0.

```
float divided2 = static_cast<float>(number2) / static_cast<float>(number1);
```

Casting both numerator and denominator to float values (or dividing two float variables) would have yielded a result of 0.5f. This is why it is important to use the correct type of variable for the job at hand.

The Modulo Operator

Our guessing game in Listing 2-1 has already shown a usage of the modulo operator. This can be thought of as a remainder operator, as it will return the remainder of dividing one number by another.

```
int numerator = 12;
int denominator = 10;
int remainder = numerator % denominator;
```

The variable `remainder` shown here will store the value 2 as 12 divided by 10 equals 1 remainder 2.

A Simple Arithmetic Calculator

We will now write a simple program that can show the results of the arithmetic operators. Listing 3-1 shows the code that will take two numbers and add them together. This sample code was created using Xcode and the main function is suitable for creating command-line programs for OS X.

Listing 3-1. Adding Two Numbers

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

int main(int argc, const char * argv[])
{
    cout << "Enter your first number: " << endl;
    float number1 = 0.0f;
    cin >> number1;

    cout << "Enter your second number: " << endl;
    float number2 = 0.0f;
    cin >> number2;

    float result = number1 + number2;
    cout << "The result of adding your two numbers is: " << result << endl;
    return 0;
}
```

We once again use `cin` and `cout` to communicate with users and ask them to enter two numbers. We then store the result of an addition in the variable `result` and print the output to the console. An example of the output from this program is shown here:

```
Enter your first number:
40
Enter your second number:
10
The result of adding your two numbers is: 50
```

Listing 3-2 shows a subtraction version of this program.

Listing 3-2. Subtracting Two Numbers

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

int main(int argc, const char * argv[])
{
    cout << "Enter your first number: " << endl;
    float number1 = 0.0f;
    cin >> number1;

    cout << "Enter your second number: " << endl;
    float number2 = 0.0f;
    cin >> number2;
```

```
float result = number1 - number2;
cout << "The result of subtracting your two numbers is: " << result << endl;
return 0;
}
```

Listing 3-3 shows a multiplication version of this program.

Listing 3-3. Multiplying Two Numbers

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

int main(int argc, const char * argv[])
{
    cout << "Enter your first number: " << endl;
    float number1 = 0.0f;
    cin >> number1;

    cout << "Enter your second number: " << endl;
    float number2 = 0.0f;
    cin >> number2;

    float result = number1 * number2;
    cout << "The result of multiplying your two numbers is: " << result << endl;
    return 0;
}
```

Listing 3-4 shows a division version of this program.

Listing 3-4. Dividing Two Numbers

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

int main(int argc, const char * argv[])
{
    cout << "Enter your first number: " << endl;
    float number1 = 0.0f;
    cin >> number1;

    cout << "Enter your second number: " << endl;
    float number2 = 0.0f;
    cin >> number2;

    float result = number1 / number2;
    cout << "The result of dividing your second number into your first is: "
        << result << endl;
    return 0;
}
```

Last but not least, Listing 3-5 shows a modulo version of this program.

Listing 3-5. Finding the Remainder Between Two Numbers

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

int main(int argc, const char * argv[])
{
    cout << "Enter your first number: " << endl;
    int number1 = 0;
    cin >> number1;

    cout << "Enter your second number: " << endl;
    int number2 = 0;
    cin >> number2;

    int result = number1 % number2;
    cout << "The remainder from dividing your second number into your first is: "
        << result << endl;
    return 0;
}
```

We have to change the variables in Listing 3-5 to `int` from `float` as the modulo operator can only operate on integer values.

Now that we have covered the arithmetic operators, we will take a look at relational operators.

Relational Operators

C++ allows us to compare variable values using relational operators. These operators allow us to work out if two variables are equal, not equal, one is greater than the other, or one is less than the other. There are also operators to work out if variables are less than or equal, and greater than or equal.

Each of these operators provides a true or false result that can be stored in a `bool`. C++ provides the following relational operators:

- Equality operator
- Greater-than operators
- Less-than operators

Equality Operators

The equals operator is `==`. We can use this in the following way:

```
bool isEqual = number1 == number2;
```

`isEqual` will contain `true` if `number1` and `number2` store the same value; otherwise it will contain `false`.

C++ also contains an inequality operator that can be thought of as a not equals operation. It can be used as follows.

```
bool isNotEqual = number1 != number2;
```

As we can see, the inequality operator uses an exclamation mark before the equals symbol. This operator provides true when the two values are not the same and false if they are the same.

Greater-Than Operators

C++ also allows us to work out if a value stored in a variable is greater than another. We can do this as follows:

```
bool isGreater = number1 > number2;
```

We can also tell if a number is greater than or equal to another:

```
bool isGreaterThanOrEqual = number1 >= number2;
```

Less-Than Operators

We can work out if a value is less than another using the less-than operator:

```
bool isLessThan = number1 < number2;
```

As with greater-than, there is a less-than or equal to operator:

```
bool isLessThanOrEqual = number1 <= number2;
```

We will create some small programs to show off the relational operators just as we did with the arithmetic operators.

Simple Comparison Calculators

Listing 3-6 shows the source code for a simple program that tests the equality of two values.

Listing 3-6. Using the Equality Operator

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

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    cout << "Enter your first number: " << endl;
    int number1 = 0;
    cin >> number1;
```

```
    cout << "Enter your second number: " << endl;
    int number2 = 0;
    cin >> number2;

    bool result = number1 == number2;
    cout << "It is "
         << result
         << " that your first number is equal your second."
         << endl;

    return 0;
}
```

This sample application was created using Microsoft Visual Studio. The program will end very quickly after the second value has been entered, so a good trick is to set a breakpoint on the return line using the F9 key and to use debug mode to run the program using the F5 key.

The boolean value in this sample will be printed out as a 1 when the operation results in a true value and 0 when the result is false.

Listing 3-7 shows a simple program using the inequality operator.

Listing 3-7. The Inequality Operator

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

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    cout << "Enter your first number: " << endl;
    int number1 = 0;
    cin >> number1;

    cout << "Enter your second number: " << endl;
    int number2 = 0;
    cin >> number2;

    bool result = number1 != number2;
    cout << "It is "
         << result
         << " that your first number is not equal your second."
         << endl;

    return 0;
}
```

Listing 3-8 contains the source code for an example of the greater-than operator.

Listing 3-8. The Greater-Than Operator

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

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    cout << "Enter your first number: " << endl;
    int number1 = 0;
    cin >> number1;

    cout << "Enter your second number: " << endl;
    int number2 = 0;
    cin >> number2;

    bool result = number1 > number2;
    cout << "It is "
         << result
         << " that your first number is greater than your second."
         << endl;

    return 0;
}
```

Listing 3-9 shows a use of the greater-than or equal to operator.

Listing 3-9. The Greater-Than or Equal To Operator

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

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    cout << "Enter your first number: " << endl;
    int number1 = 0;
    cin >> number1;

    cout << "Enter your second number: " << endl;
    int number2 = 0;
    cin >> number2;

    bool result = number1 >= number2;
    cout << "It is "
         << result
         << " that your first number is greater than"
         << " or equal to your second."
         << endl;

    return 0;
}
```

Listing 3-10 shows an example of the less-than operator.

Listing 3-10. The Less Than Operator

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

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    cout << "Enter your first number: " << endl;
    int number1 = 0;
    cin >> number1;

    cout << "Enter your second number: " << endl;
    int number2 = 0;
    cin >> number2;

    bool result = number1 < number2;
    cout << "It is "
         << result
         << " that your first number is less than your second."
         << endl;

    return 0;
}
```

Finally, Listing 3-11 shows a use of the less-than or equal to operator.

Listing 3-11. The Less-Than or Equal To Operator

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

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    cout << "Enter your first number: " << endl;
    int number1 = 0;
    cin >> number1;

    cout << "Enter your second number: " << endl;
    int number2 = 0;
    cin >> number2;

    bool result = number1 <= number2;
    cout << "It is "
         << result
         << " that your first number is less than "
```