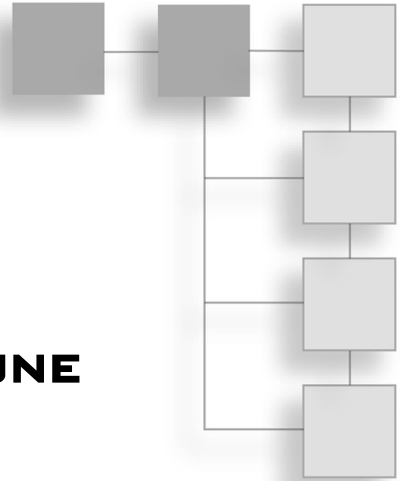


CHAPTER 1



TYPES, VARIABLES, AND STANDARD I/O: LOST FORTUNE

Game programming is demanding. It pushes both programmer and hardware to their limits. But it can also be extremely satisfying. In this chapter, you'll be introduced to the fundamentals of C++, the standard language for AAA game titles. Specifically, you'll learn to:

- Display output in a console window
- Perform arithmetic computations
- Use variables to store, manipulate, and retrieve data
- Get user input
- Work with constants and enumerations
- Work with strings

INTRODUCING C++

C++ is leveraged by millions of programmers around the world. It's one of the most popular languages for writing computer applications—and *the* most popular language for writing big-budget computer games.

Created by Bjarne Stroustrup, C++ is a direct descendant of the C language. In fact, C++ retains almost all of C as a subset. However, C++ offers better ways to do things as well as some brand-new capabilities.

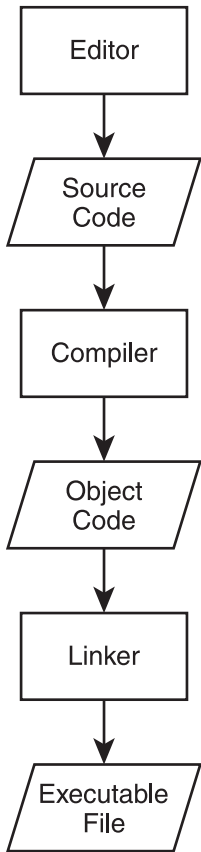
Using C++ for Games

There are a variety of reasons why game programmers choose C++. Here are a few:

- **It's fast.** Well-written C++ programs can be blazingly fast. One of C++'s design goals is performance. And if you need to squeeze out even more performance from your programs, C++ allows you to use *assembly language*—the lowest-level, human-readable programming language—to communicate directly with the computer's hardware.
- **It's flexible.** C++ is a multi-paradigm language that supports different styles of programming, including *object-oriented programming*. Unlike some other modern languages, though, C++ doesn't force one particular style on a programmer.
- **It's well-supported.** Because of its long history in the game industry, there's a large pool of assets available to the C++ game programmer, including graphics APIs and 2D, 3D, physics, and sound engines. All of this pre-existing code can be leveraged by a C++ programmer to greatly speed up the process of writing a new game.

Creating an Executable File

The file that you run to launch a program—whether you're talking about a game or a business application—is an *executable file*. There are several steps to creating an executable file from C++ *source code* (a collection of instructions in the C++ language). The process is illustrated in Figure 1.1.

**Figure 1.1**

The creation of an executable file from C++ source code.

1. First, the programmer uses an *editor* to write the C++ source code, a file that usually has the extension `.cpp`. The editor is like a word processor for programs; it allows a programmer to create, edit, and save source code.
2. After the programmer saves a source file, he or she invokes a C++ *compiler*—an application that reads source code and translates it into an *object file*. Object files usually have the extension `.obj`.
3. Next, a linker links the object file to any external files as necessary, and then creates the executable file, which generally ends with the extension `.exe`. At this point, a user (or gamer) can run the program by launching the executable file.

Hint

The process I've described is the simple case. Creating a complex application in C++ often involves multiple source code files written by a programmer (or even a team of programmers).

To help automate this process, it's common for a programmer to use an all-in-one tool for development, called an *IDE* (*Integrated Development Environment*). An IDE typically combines an editor, a compiler, and a linker, along with other tools. A popular (and free) IDE for Windows is Microsoft Visual Studio Express 2013 for Windows Desktop. You can find out more about this IDE (and download a copy) at www.visualstudio.com/downloads/download-visual-studio-vs.

Dealing with Errors

When I described the process for creating an executable from C++ source, I left out one minor detail: errors. If to err is human, then programmers are the most human of us. Even the best programmers write code that generates errors the first (or fifth) time through. Programmers must fix the errors and start the entire process over. Here are the basic types of errors you'll run into as you program in C++:

- **Compile errors.** These occur during code compilation. As a result, an object file is not produced. These can be *syntax errors*, meaning that the compiler doesn't understand something. They're often caused by something as simple as a typo. Compilers can issue warnings, too. Although you usually don't need to heed the warnings, you should treat them as errors, fix them, and recompile.
- **Link errors.** These occur during the linking process and may indicate that something the program references externally can't be found. These errors are usually solved by adjusting the offending reference and starting the compile/link process again.
- **Run-time errors.** These occur when the executable is run. If the program does something illegal, it can crash abruptly. But a more subtle form of run-time error, a *logical error*, can make the program simply behave in unintended ways. If you've ever played a game where a character walked on air (that is, a character who shouldn't be able to walk on air), then you've seen a logical error in action.

In the Real World

Like other software creators, game companies work hard to produce bug-free products. Their last line of defense is the quality assurance personnel (the game testers). Game testers play games for a living, but their jobs are not as fun as you might think. Testers must play the same parts of a game over and over—perhaps

hundreds of times—trying the unexpected and meticulously recording any anomalies. On top of monotonous work, the pay ain't great either. But being a tester is a terrific way to get into a game company on the proverbial bottom rung.

Understanding the ISO Standard

The *ISO (International Organization for Standardization) standard* for C++ is a definition of C++ that describes exactly how the language should work. It also defines a group of files, called the *standard library*, that contain building blocks for common programming tasks, such as *I/O*—getting input and displaying output. The standard library makes life easier for programmers and provides fundamental code to save them from reinventing the wheel. I'll use the standard library in all of the programs in this book.

Hint

The ISO standard is often called the *ANSI (American National Standards Institute) standard* or *ANSI/ISO standard*. These different names involve the acronyms of the various committees that have reviewed and established the standard. The most common way to refer to C++ code that conforms to the ISO standard is simply *Standard C++*.

I used Microsoft Visual Studio Express 2013 for Windows Desktop to develop the programs in this book. The compiler that's a part of this IDE is pretty faithful to the ISO standard, so you should be able to compile, link, and run all of the programs using some other modern compiler as well. However, if you're using Windows, I recommend using Visual Studio Express 2013 for Windows Desktop.

Hint

For step-by-step instructions on how to create, save, compile, and run the Game Over program using Visual Studio Express 2013 for Windows Desktop, check out Appendix A, "Creating Your First C++ Program." If you're using another compiler or IDE, check its documentation.

WRITING YOUR FIRST C++ PROGRAM

Okay, enough theory. It's time to get down to the nitty-gritty and write your first C++ program. Although it is simple, the following program shows you the basic anatomy of a program. It also demonstrates how to display text in a console window.

Introducing the Game Over Program

The classic first task a programmer tackles in a new language is the Hello World program, which displays `Hello World` on the screen. The Game Over program puts a gaming twist on the classic and displays `Game Over!` instead. Figure 1.2 shows the program in action.

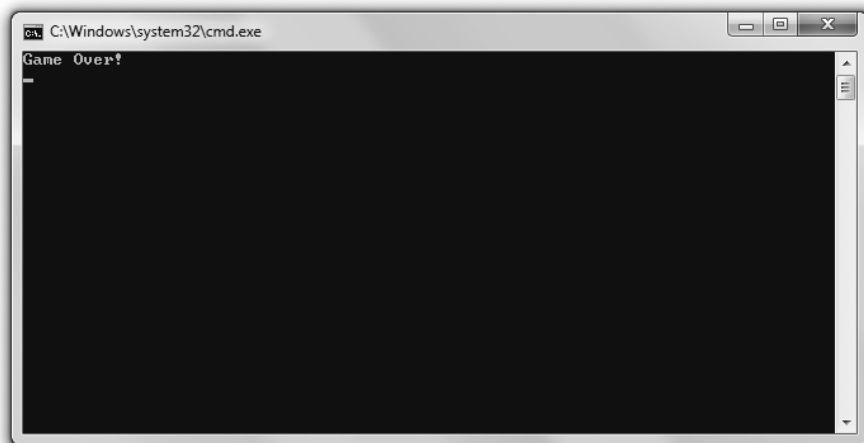


Figure 1.2

Your first C++ program displays the two most infamous words in computer gaming.

Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 1 folder; the filename is `game_over.cpp`.

Hint

You can download all of the source code for the programs in this book by visiting www.cengageptr.com/downloads and searching for this book. One way to search is by ISBN (the book's identification number), which is 9781305109919.

```
// Game Over
// A first C++ program
#include <iostream>

int main()
{
    std::cout << "Game Over!" << std::endl;
    return 0;
}
```

Commenting Code

The first two lines of the program are comments.

```
// Game Over  
// A first C++ program
```

Comments are completely ignored by the compiler; they're meant for humans. They can help other programmers understand your intentions. But comments can also help you. They can remind you how you accomplished something that might not be clear at first glance.

You can create a comment using two forward slashes in a row (`//`). Anything after this on the rest of the physical line is considered part of the comment. This means you can also include a comment after a piece of C++ code, on the same line.

Hint

You can also use what are called *C-style comments*, which can span multiple lines. All you have to do is start the comment with `/*` and end it with `*/`. Everything in between the two markers is part of the comment.

Using Whitespace

The next line in the program is a blank line. The compiler ignores blank lines. In fact, compilers ignore just about all *whitespace*—spaces, tabs, and new lines. Like comments, whitespace is just for us humans.

Judicious use of whitespace helps make programs clearer. For example, you can use blank lines to separate sections of code that belong together. I also use whitespace (a tab, to be precise) at the beginning of the two lines between the curly braces to set them off.

Including Other Files

The next line in the program is a preprocessor directive. You know this because the line begins with the `#` symbol.

```
#include <iostream>
```

The *preprocessor* runs before the compiler does its thing and substitutes text based on various directives. In this case, the line involves the `#include` directive, which tells the preprocessor to include the contents of another file.

I include the file `iostream`, which is part of the standard library, because it contains code to help me display output. I surround the filename with less than (`<`) and greater than (`>`)

characters to tell the compiler to find the file where it keeps all the files that came with the compiler. A file that you include in your programs like this is called a *header file*.

Defining the `main()` Function

The next non-blank line is the header of a function called `main()`.

```
int main()
```

A *function* is a group of programming code that can do some work and return a value. In this case, `int` indicates that the function will return an integer value. All function headers have a pair of parentheses after the function name.

All C++ programs must have a function called `main()`, which is the starting point of the program. The real action begins here.

The next line marks the beginning of the function.

```
{
```

And the very last line of the program marks the end of the function.

```
}
```

All functions are delimited by a pair of curly braces, and everything between them is part of the function. Code between two curly braces is called a *block* and is usually indented to show that it forms a unit. The block of code that makes up an entire function is called the *body* of the function.

Displaying Text through the Standard Output

The first line in the body of `main()` displays `Game Over!`, followed by a new line, in the console window.

```
std::cout << "Game Over!" << std::endl;
```

"Game Over!" is a *string*—a series of printable characters. Technically, it's a *string literal*, meaning it's literally the characters between the quotes.

`cout` is an object, defined in the file `iostream`, that's used to send data to the standard output stream. In most programs (including this one), the standard output stream simply means the console window on the computer screen.

I use the *output operator* (`<<`) to send the string to `cout`. You can think of the output operator like a funnel; it takes whatever's on the open side and funnels it to the pointy side. So the string is funneled to the standard output—the screen.

I use `std` to prefix `cout` to tell the compiler that I mean `cout` from the standard library. `std` is a *namespace*. You can think of a namespace like an area code of a phone number—it identifies the group to which something belongs. You prefix a namespace using the *scope resolution operator* (`::`).

Finally, I send `std::endl` to the standard output. `endl` is defined in `iostream` and is also an object in the `std` namespace. Sending `endl` to the standard output acts like pressing the Enter key in the console window. In fact, if I were to send another string to the console window, it would appear on the next line.

I understand this might be a lot to take in, so check out Figure 1.3 for a visual representation of the relationship between all of the elements I've just described.

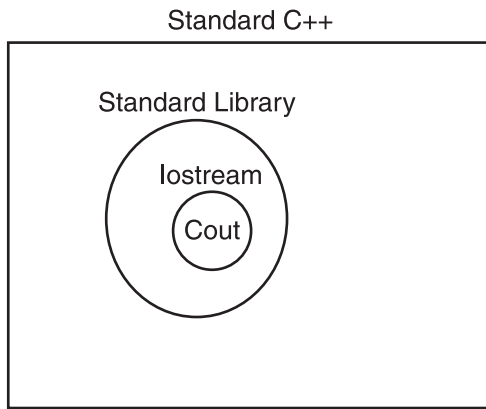


Figure 1.3

An implementation of Standard C++ includes a set of files called the standard library, which includes the file `iostream`, which defines various things including the object `cout`.

Terminating Statements

You'll notice that the first line of the function ends with a semicolon (`;`). That's because the line is a *statement*—the basic unit controlling the execution flow. All of your statements must end with a semicolon—otherwise, your compiler will complain with an error message and your program won't compile.

Returning a Value from `main()`

The last statement in the function returns 0 to the operating system.

```
return 0;
```

Returning 0 from `main()` is a way to indicate that the program ended without a problem. The operating system doesn't have to do anything with the return value. In general, you can simply return 0 like I did here.

Trick

When you run the Game Over program, you might only see a console window appear and disappear just as quickly. That's because C++ is so fast that it opens a console window, displays `Game Over!`, and closes the window all in a split second. However, in Windows, you can create a batch file that runs your console program and pauses, keeping the console window open so you can see the results of your program. Since the compiled program is named `game_over.exe`, you can simply create a batch file comprising the two lines

```
game_over.exe
pause
```

To create a batch file:

1. Open a text editor like Notepad (not Word or WordPad).
2. Type your text.
3. Save the file in the same folder with your `game_over.exe` file. Give the file a `.bat` extension—so, in this case, `game_over.bat` would be a good name.

Finally, run the batch file by double-clicking its icon. You should see the results of the program since the batch file keeps the console window open.

WORKING WITH THE STD NAMESPACE

Because it's so common to use elements from the `std` namespace, I'll show you two different methods for directly accessing these elements. This will save you the effort of using the `std::` prefix all the time.

Introducing the Game Over 2.0 Program

The Game Over 2.0 program produces the exact results of the original Game Over program, illustrated in Figure 1.2. But there's a difference in the way elements from the `std` namespace are accessed. You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 1 folder; the filename is `game_over2.cpp`.

```
// Game Over 2.0
// Demonstrates a using directive
```

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

int main()
{
    cout << "Game Over!" << endl;
    return 0;
}
```

Employing a using Directive

The program starts in the same way. I use two opening comments and then include `iostream` for output. But next, I have a new type of statement.

```
using namespace std;
```

This `using` directive gives me direct access to elements of the `std` namespace. Again, if a namespace is like an area code, then this line says that all of the elements in the `std` namespace should be like local phone numbers to me now. That is, I don't have to use their area code (the `std::` prefix) to access them.

I can use `cout` and `endl`, without any kind of prefix. This might not seem like a big deal to you now, but when you have dozens or even hundreds of references to these objects, you'll thank me.

Introducing the Game Over 3.0 Program

Okay, there's another way to accomplish what I did in Game Over 2.0: set up the file so that I don't have to explicitly use the `std::` prefix to access `cout` and `endl`. And that's exactly what I'll show you in the Game Over 3.0 program, which displays the same text as its predecessors. You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 1 folder; the filename is `game_over3.cpp`.

```
// Game Over 3.0
// Demonstrates using declarations

#include <iostream>
using std::cout;
using std::endl;
```

```
int main()
{
    cout << "Game Over!" << endl;
    return 0;
}
```

Employing using Declarations

In this version, I write two `using` declarations.

```
using std::cout;
using std::endl;
```

By declaring exactly which elements from the `std` namespace I want local to my program, I'm able to access them directly, just as in *Game Over 2.0*. Although it requires more typing than a `using` directive, the advantage of this technique is that it clearly spells out those elements I plan to use. Plus, it doesn't make local a bunch of other elements that I have no intention of using.

Understanding When to Employ using

Okay, you've seen two ways to make elements from a namespace local to your program. But which is the best technique?

A language purist would say you shouldn't employ either version of `using` and that you should always prefix each and every element from a namespace with its identifier. In my opinion, that's like calling your best friend by his first and last name all the time. It just seems a little too formal.

If you hate typing, you can employ the `using` directive. A decent compromise is to employ `using` declarations. In this book, I'll employ the `using` directive most of the time for brevity's sake.

In the Real World

I've laid out a few different options for working with namespaces. I've also tried to explain the advantages of each so you can decide which way to go in your own programs. Ultimately, though, the decision may be out of your hands. When you're working on a project, whether it's in the classroom or in the professional world, you'll probably receive coding standards created by the person in charge. Regardless of your personal tastes, it's always best to listen to those who hand out grades or paychecks.

USING ARITHMETIC OPERATORS

Whether you're tallying up the number of enemies killed or decreasing a player's health level, you need your programs to do some math. As with other languages, C++ has built-in arithmetic operators.

Introducing the Expensive Calculator Program

Most serious computer gamers invest heavily in a bleeding-edge, high-powered gaming rig. This next program, Expensive Calculator, can turn that monster of a machine into a simple calculator. The program demonstrates built-in arithmetic operators. Figure 1.4 shows off the results.

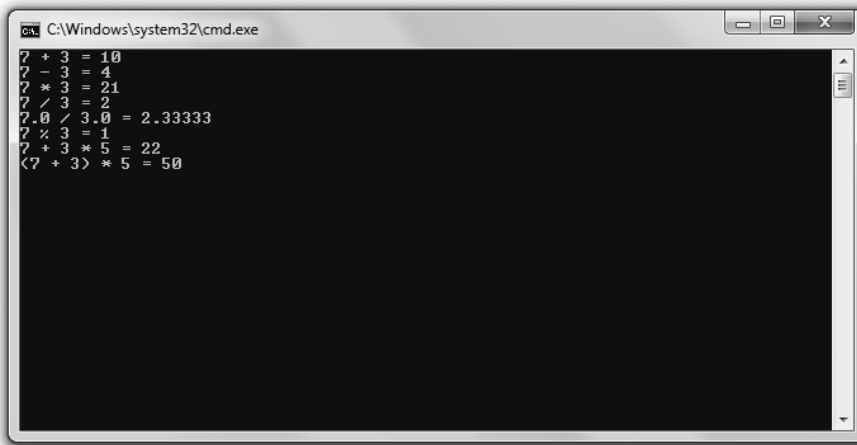


Figure 1.4

C++ can add, subtract, multiply, divide, and even calculate a remainder.

Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 1 folder; the filename is `expensive_calculator.cpp`.

```
// Expensive Calculator
// Demonstrates built-in arithmetic operators

#include <iostream>
using namespace std;

int main()
{
```

```

cout << "7 + 3 = " << 7 + 3 << endl;
cout << "7 - 3 = " << 7 - 3 << endl;
cout << "7 * 3 = " << 7 * 3 << endl;

cout << "7 / 3 = " << 7 / 3 << endl;
cout << "7.0 / 3.0 = " << 7.0 / 3.0 << endl;

cout << "7 % 3 = " << 7 % 3 << endl;

cout << "7 + 3 * 5 = " << 7 + 3 * 5 << endl;
cout << "(7 + 3) * 5 = " << (7 + 3) * 5 << endl;

return 0;
}

```

Adding, Subtracting, and Multiplying

I use the built-in arithmetic operators for addition (the plus sign, +), subtraction (the minus sign, -), and multiplication (an asterisk, *). The results depicted in Figure 1.4 are just what you'd expect.

Each arithmetic operator is part of an *expression*—something that evaluates to a single value. So, for example, the expression $7 + 3$ evaluates to 10, and that's what is sent to `cout`.

Understanding Integer and Floating Point Division

The symbol for division is the forward slash (/), so that's what I use in the next line of code. However, the output might surprise you. According to C++ (and that expensive gaming rig), 7 divided by 3 is 2. What's going on? Well, the result of any arithmetic calculation involving only *integers* (numbers without fractional parts) is always another integer. And since 7 and 3 are both integers, the result must be an integer. The fractional part of the result is thrown away.

To get a result that includes a fractional part, at least one of the values needs to be a *floating point* (a number with a fractional part). I demonstrate this in the next line with the expression $7.0 / 3.0$. This time the result is a more accurate 2.33333.

Trap

You might notice that while the result of $7.0 / 3.0$ (2.33333) includes a fractional part, it is still truncated. (The true result would stretch out 3s after the decimal point forever.) It's important to know that computers generally store only a limited number of significant digits for floating point numbers. However, C++ offers categories of floating point numbers to meet the most demanding needs—even those of computationally intensive 3D games.

Using the Modulus Operator

In the next statement, I use an operator that might be unfamiliar to you—the modulus operator (%). The modulus operator returns the remainder of integer division. In this case, $7 \% 3$ produces the remainder of $7 / 3$, which is 1.

Understanding Order of Operations

Just as in algebra, arithmetic expressions in C++ are evaluated from left to right. But some operators have a higher precedence than others and are evaluated first, regardless of position. Multiplication, division, and modulus have equal precedence, which is higher than the precedence level that addition and subtraction share.

The next line of code provides an example to help drive this home. Because multiplication has higher precedence than addition, you calculate the results of the multiplication first. So the expression $7 + 3 * 5$ is equivalent to $7 + 15$, which evaluates to 22.

If you want an operation with lower precedence to occur first, you can use parentheses, which have higher precedence than any arithmetic operator. So in the next statement, the expression $(7 + 3) * 5$ is equivalent to $10 * 5$, which evaluates to 50.

Hint

For a list of C++ operators and their precedence levels, see Appendix B, “Operator Precedence.”

DECLARING AND INITIALIZING VARIABLES

A *variable* represents a particular piece of your computer’s memory that has been set aside for you to use to store, retrieve, and manipulate data. So if you wanted to keep track of a player’s score, you could create a variable for it, then you could retrieve the score to display it. You could also update the score when the player blasts an alien enemy from the sky.

Introducing the Game Stats Program

The Game Stats program displays information that you might want to keep track of in a space shooter game, such as a player’s score, the number of enemies the player has destroyed, and whether the player has his shields up. The program uses a group of variables to accomplish all of this. Figure 1.5 illustrates the program.

**Figure 1.5**

Each game stat is stored in a variable.

Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 1 folder; the filename is `game_stats.cpp`.

```
// Game Stats
// Demonstrates declaring and initializing variables

#include <iostream>
using namespace std;

int main()
{
    int score;
    double distance;
    char playAgain;
    bool shieldsUp;
    short lives, aliensKilled;

    score = 0;
    distance = 1200.76;
    playAgain = 'y';
    shieldsUp = true;
    lives = 3;
    aliensKilled = 10;
```



```
double engineTemp = 6572.89;

cout << "\nscore: "          << score << endl;
cout << "distance: "         << distance << endl;
cout << "playAgain: "        << playAgain << endl;
//skipping shieldsUp since you don't generally print Boolean values
cout << "lives: "            << lives << endl;
cout << "aliensKilled: " << aliensKilled << endl;
cout << "engineTemp: "       << engineTemp << endl;

int fuel;
cout << "\nHow much fuel? ";
cin >> fuel;
cout << "fuel: " << fuel << endl;

typedef unsigned short int ushort;
ushort bonus = 10;
cout << "\nbonus: " << bonus << endl;

return 0;
}
```

Understanding Fundamental Types

Every variable you create has a *type*, which represents the kind of information you can store in the variable. It tells your compiler how much memory to set aside for the variable and it defines exactly what you can legally do with the variable.

Fundamental types—those built into the language—include `bool` for Boolean values (true or false), `char` for single character values, `int` for integers, `float` for single-precision floating point numbers, and `double` for double-precision floating point numbers.

Understanding Type Modifiers

You can use modifiers to alter a type. `short` is a modifier that can reduce the total number of values a variable can hold. `long` is a modifier that can increase the total number of values a variable can hold. `short` may decrease the storage space required for a variable while `long` may increase it. `short` and `long` can modify `int`. `long` can also modify `double`.

`signed` and `unsigned` are modifiers that work only with integer types. `signed` means that a variable can store both positive and negative values, while `unsigned` means that a variable can store only positive values. Neither `signed` nor `unsigned` change the total number of values a variable can hold; they only change the range of values. `signed` is the default for integer types.

Okay, confused with all of your type options? Well, don't be. Table 1.1 summarizes commonly used types with some modifiers thrown in. The table also provides a range of values for each.

Table 1.1 Commonly Used Types	
Type	Values
short int	−32,768 to 32,767
unsigned short int	0 to 65,535
int	−2,147,483,648 to 2,147,483,647
unsigned int	0 to 4,294,967,295
long int	−2,147,483,648 to 2,147,483,647
unsigned long int	0 to 4,294,967,295
float	3.4E +/- 38 (seven significant digits)
double	1.7E +/- 308 (15 significant digits)
long double	1.7E +/- 308 (15 significant digits)
char	256 character values
bool	true or false

Trap

The range of values listed is based on my compiler. Yours might be different. Check your compiler's documentation.

Hint

For brevity's sake, short int can be written as just short, and long int can be written as just long.

Declaring Variables

All right, now that you've got a basic understanding of types, it's time to get back to the program. One of the first things I do is *declare* a variable (request that it be created) with the line:

```
int score;
```

In this code, I declare a variable of type `int`, which I name `score`. You use a variable name to access the variable. You can see that to declare a variable you specify its type followed by a name of your choosing. Because the declaration is a statement, it must end with a semicolon.

I declare three more variables of yet three more types in the next three lines. `distance` is a variable of type `double`. `playAgain` is a variable of type `char`. And `shieldsUp` is a variable of type `bool`.

Games (and all major applications) usually require lots of variables. Fortunately, C++ allows you to declare multiple variables of the same type in a single statement. That's just what I do next in the line.

```
short lives, aliensKilled;
```

This line establishes two short variables—`lives` and `aliensKilled`.

Even though I've defined a bunch of variables at the top of my `main()` function, you don't have to declare all of your variables in one place. As you'll see later in the program, I often define a new variable just before I use it.

Naming Variables

To declare a variable, you must provide a name, known as an *identifier*. There are only a few rules you have to follow to create a legal identifier.

- An identifier can contain only numbers, letters, and underscores.
- An identifier can't start with a number.
- An identifier can't be a C++ keyword.

A *keyword* is a special word that C++ reserves for its own use. There aren't many, but to see a full list, check out Appendix C, "Keywords."

In addition to the rules for creating *legal* variable names, following are some guidelines for creating *good* variable names.

- **Choose descriptive names.** Variable names should be clear to another programmer. For example, use `score` instead of `s`. (One exception to this rule involves variables used for a brief period. In that case, single-letter variable names, such as `x`, are fine.)
- **Be consistent.** There are different schools of thought about how to write multiword variable names. Is it `high_score` or `highScore`? In this book, I use the second style, where the initial letter of the second word (and any other words) is capitalized, which

is known as *camel case*. But as long as you're consistent, it's not important which method you use.

- **Follow the traditions of the language.** Some naming conventions are just traditions. For example, in most languages (C++ included) variable names start with a lowercase letter. Another tradition is to avoid using an underscore as the first character of your variable names. Names that begin with an underscore can have special meaning.
- **Keep the length in check.** Even though `playerTwoBonusForRoundOne` is descriptive, it can make code hard to read. Plus, long names increase the risk of a typo. As a guideline, try to limit your variable names to fewer than 15 characters. Ultimately, though, your compiler sets an actual upper limit.

Trick

Self-documenting code is written in such a way that it's easy to understand what is happening in the program independent of any comments. Choosing good variable names is an excellent step toward this kind of code.

Assigning Values to Variables

In the next group of statements, I assign values to the six variables I declared. I'll go through a few assignments and talk a little about each variable type.

Assigning Values to Integer Variables

In the following assignment statement, I assign the value of 0 to `score`.

```
score = 0;
```

Now `score` stores 0.

You assign a value to a variable by writing the variable name followed by the assignment operator (=) followed by an expression. (Yes, technically 0 is an expression, which evaluates to, well, 0.)

Assigning Values to Floating Point Variables

In the following statement, I assign `distance` the value 1200.76.

```
distance = 1200.76;
```

Because `distance` is of type `double`, I can use it to store a number with a fractional part, which is just what I do.

Assigning Values to Character Variables

In the following statement, I assign `playAgain` the single-character value `'y'`.

```
playAgain = 'y';
```

As I did here, you can assign a character to a variable of type `char` by surrounding the character with single quotes.

Variables of type `char` can store the 128 ASCII character values (assuming that your system uses the ASCII character set). *ASCII*, short for *American Standard Code for Information Interchange*, is a code for representing characters. To see a complete ASCII listing, check out Appendix D, “ASCII Chart.”

Assigning Values to Boolean Variables

In the following statement, I assign `shieldsUp` the value `true`.

```
shieldsUp = true;
```

In my program, this means that the player’s shields are up.

`shieldsUp` is a `bool` variable, which means it’s a Boolean variable. As such, it can represent either `true` or `false`. Although intriguing, you’ll have to wait until Chapter 2, “Truth, Branching, and the Game Loop: Guess My Number,” to learn more about this kind of variable.

Initializing Variables

You can both declare and assign a value to variables in a single initialization statement. That’s exactly what I do next.

```
double engineTemp = 6572.89;
```

This line creates a variable of type `double` named `engineTemp`, which stores the value `6572.89`.

Just as you can declare multiple variables in one statement, you can initialize more than one variable in a statement. You can even declare and initialize different variables in a single statement. Mix and match as you choose!

Hint

Although you can declare a variable without assigning it a value, it’s best to initialize a new variable with a starting value whenever you can. This makes your code clearer, plus it eliminates the chance of accessing an uninitialized variable, which may contain any value.

Displaying Variable Values

To display the value of a variable of one of the fundamental types, just send it to `cout`. That's what I do next in the program. Note that I don't try to display `shieldsUp` because you don't normally display `bool` values.

Trick

In the first statement of this section I use what's called an *escape sequence*—a pair of characters that begins with a backslash (`\`), which represents special printable characters.

```
cout << "\nscore: " << score << endl;
```

The escape sequence I used is `\n`, which represents a new line. When sent to `cout` as part of a string, it's like pressing the Enter key in the console window. Another useful escape sequence is `\t`, which acts as a tab.

There are other escape sequences at your disposal. For a list of escape sequences, see Appendix E, "Escape Sequences."

Getting User Input

Another way to assign a value to a variable is through user input. So next, I assign the value of a new variable, `fuel`, based on what the user enters. To do so I use the following line:

```
cin >> fuel;
```

Just like `cout`, `cin` is an object defined in `iostream` which lives in the `std` namespace. To store a value in the variable, I use `cin` followed by `>>` (the extraction operator), followed by the variable name. You can use `cin` and the extraction operator to get user input into variables of other fundamental types too. To prove that everything worked, I display `fuel` to the user.

Defining New Names for Types

You can define a new name for an existing type. In fact, that's what I do next in the line:

```
typedef unsigned short int ushort;
```

This code defines the identifier `ushort` as another name for the type `unsigned short int`. To define new names for existing types, use `typedef` followed by the current type, followed by the new name. `typedef` is often used to create shorter names for types with long names.

You can use your new type name just like the original type. I initialize a `ushort` variable (which is really just an `unsigned short int`) named `bonus` and display its value.

Understanding Which Types to Use

You have many choices when it comes to the fundamental types. So how do you know which type to use? Well, if you need an integer type, you're probably best off using `int`. That's because `int` is generally implemented so that it occupies an amount of memory that is most efficiently handled by the computer. If you need to represent integer values greater than the maximum `int` or values that will never be negative, feel free to use an `unsigned int`.

If you're tight on memory, you can use a type that requires less storage. However, on most computers, memory shouldn't be much of an issue. (Programming on game consoles or mobile devices is another story.)

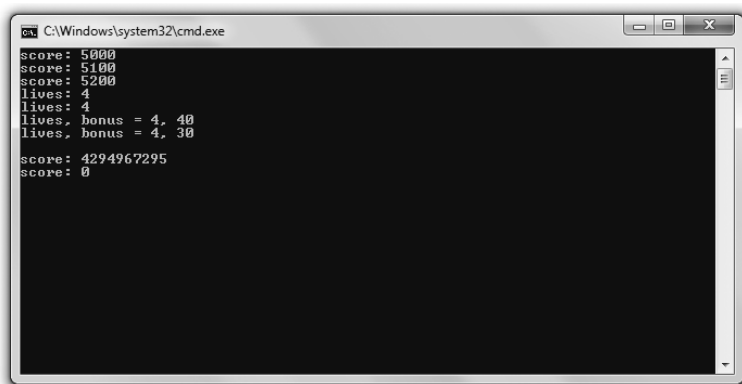
Finally, if you need a floating-point number, you're probably best off using `float`, which again is likely to be implemented so that it occupies an amount of memory that is most efficiently handled by the computer.

PERFORMING ARITHMETIC OPERATIONS WITH VARIABLES

Once you have variables with values, you'll want to change their values during the course of your game. You might want to add a bonus to a player's score for defeating a boss, increasing the score. Or you might want to decrease the oxygen level in an airlock. By using operators you've already met (along with some new ones), you can accomplish all of this.

Introducing the Game Stats 2.0 Program

The Game Stats 2.0 program manipulates variables that represent game stats and displays the results. Figure 1.6 shows the program in action.



```
C:\Windows\system32\cmd.exe
score: 5000
score: 5100
score: 5200
lives: 4
lives, bonus = 4. 40
lives, bonus = 4. 30
score: 4294967295
score: 0
```

Figure 1.6

Each variable is altered in a different way.

Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 1 folder; the filename is `game_stats2.cpp`.

```
// Game Stats 2.0
// Demonstrates arithmetic operations with variables

#include <iostream>
using namespace std;

int main()
{
    unsigned int score = 5000;
    cout << "score: " << score << endl;

    //altering the value of a variable
    score = score + 100;
    cout << "score: " << score << endl;

    //combined assignment operator
    score += 100;
    cout << "score: " << score << endl;

    //increment operators
    int lives = 3;
    ++lives;
    cout << "lives: " << lives << endl;

    lives = 3;
    lives++;
    cout << "lives: " << lives << endl;

    lives = 3;
    int bonus = ++lives * 10;
    cout << "lives, bonus = " << lives << ", " << bonus << endl;

    lives = 3;
    bonus = lives++ * 10;
    cout << "lives, bonus = " << lives << ", " << bonus << endl;

    //integer wrap around
    score = 4294967295;
    cout << "\nscore: " << score << endl;
    ++score;
    cout << "score: " << score << endl;

    return 0;
}
```


Trap

When you compile this program, you may get a warning similar to, “[Warning] this decimal constant is unsigned.” Fortunately, the warning does not stop the program from compiling and being run. The warning is the result of something called integer wrap around that you’ll probably want to avoid in your own programs; however, the wrap around is intentional in this program to show the results of the event. You’ll learn about integer wrap around in the discussion of this program, in the section “Dealing with Integer Wrap Around.”

Altering the Value of a Variable

After I create a variable to hold the player’s score and display it, I alter the score by increasing it by 100.

```
score = score + 100;
```

This assignment statement says to take the current value of `score`, add 100, and assign the result back to `score`. In effect, the line increases the value of `score` by 100.

Using Combined Assignment Operators

There’s an even shorter version of the preceding line, which I use next.

```
score += 100;
```

This statement produces the same results as `score = score + 100;`. The `+=` operator is called a *combined assignment operator* because it combines an arithmetic operation (addition, in this case) with assignment. This operator is shorthand for saying “add whatever’s on the right to what’s on the left and assign the result back to what’s on the left.”

There are versions of the combined assignment operator for all of the arithmetic operators you’ve met. To see a list, check out Table 1.2.

Table 1.2 Combined Assignment Operators

Operator	Example	Equivalent To
<code>+=</code>	<code>x += 5;</code>	<code>x = x + 5;</code>
<code>-=</code>	<code>x -= 5;</code>	<code>x = x - 5;</code>
<code>*=</code>	<code>x *= 5;</code>	<code>x = x * 5;</code>
<code>/=</code>	<code>x /= 5;</code>	<code>x = x / 5;</code>
<code>%=</code>	<code>x %= 5;</code>	<code>x = x % 5;</code>

Using Increment and Decrement Operators

Next, I use the *increment operator* (`++`), which increases the value of a variable by one. I use the operator to increase the value of `lives` twice. First, I use it in the following line:

```
++lives;
```

Then I use it again in the following line:

```
lives++;
```

Each line has the same net effect; it increments `lives` from 3 to 4.

As you can see, you can place the operator before or after the variable you're incrementing. When you place the operator before the variable, the operator is called the *prefix increment operator*; when you place it after the variable, it's called the *postfix increment operator*.

At this point, you might be thinking that there's no difference between the postfix and prefix versions, but you'd be wrong. In a situation where you only increment a single variable (as you just saw), both operators produce the same final result. But in a more complex expression, the results can be different.

To demonstrate this important difference, I perform a calculation that would be appropriate for the end of a game level. I calculate a bonus based on the number of lives a player has, and I increment the number of lives. However, I perform this calculation in two different ways. The first time, I use the prefix increment operator.

```
int bonus = ++lives * 10;
```

The prefix increment operator increments a variable *before* the evaluation of a larger expression involving the variable. `++lives * 10` is evaluated by first incrementing `lives`, and then multiplying that result by 10. Therefore, the code is equivalent to `4 * 10`, which is 40, of course. This means that now `lives` is 4 and `bonus` is 40.

After setting `lives` back to 3, I calculate `bonus` again, this time using the postfix increment operator.

```
bonus = lives++ * 10;
```

The postfix increment operator increments a variable *after* the evaluation of a larger expression involving the variable. `lives++ * 10` is evaluated by multiplying the current value of `lives` by 10. Therefore, the code is equivalent to `3 * 10`, which is 30, of course. Then, after this calculation, `lives` is incremented. After the line is executed, `lives` is 4 and `bonus` is 30.

C++ also defines the *decrement operator*, `--`. It works just like the increment operator, except it decrements a variable. It comes in the two flavors (prefix and postfix) as well.

Dealing with Integer Wrap Around

What happens when you increase an integer variable beyond its maximum value? It turns out you don't generate an error. Instead, the value "wraps around" to the type's minimum value. Next up, I demonstrate this phenomenon. First, I assign `score` the largest value it can hold.

```
score = 4294967295;
```

Then I increment the variable.

```
++score;
```

As a result, `score` becomes 0 because the value wrapped around, much like a car odometer does when it goes beyond its maximum value (see Figure 1.7).



Figure 1.7

A way to visualize an unsigned `int` variable "wrapping around" from its maximum value to its minimum.

Decrementing an integer variable beyond its minimum value "wraps it around" to its maximum.

Hint

Make sure to pick an integer type that has a large enough range for its intended use.

WORKING WITH CONSTANTS

A *constant* is an unchangeable value that you name. Constants are useful if you have an unchanging value that comes up frequently in your program. For example, if you were writing a space shooter in which each alien blasted out of the sky is worth 150 points, you could define a constant named `ALIEN_POINTS` that is equal to 150. Then, any time you need the value of an alien, you could use `ALIEN_POINTS` instead of the literal 150.

Constants provide two important benefits. First, they make programs clearer. As soon as you see `ALIEN_POINTS`, you know what it means. If you were to look at some code and see 150, you might not know what the value represents. Second, constants make changes easy. For example, suppose you do some playtesting with your game and you decide that each alien should really be worth 250 points. With constants, all you'd have to do is change the initialization of `ALIEN_POINTS` in your program. Without constants, you'd have to hunt down every occurrence of 150 and change it to 250.

Introducing the Game Stats 3.0 Program

The Game Stats 3.0 program uses constants to represent values. First, the program calculates a player's score, and then it calculates the upgrade cost of a unit in a strategy game. Figure 1.8 shows the results.

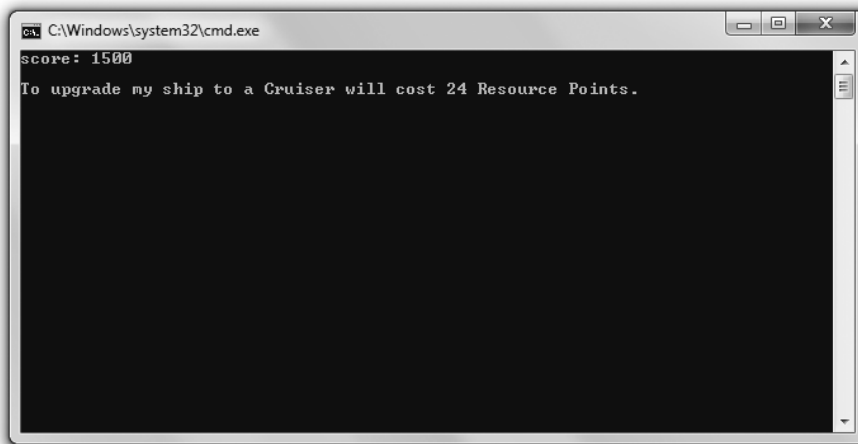


Figure 1.8

Each calculation involves a constant, making the code behind the scenes clearer.

Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 1 folder; the filename is `game_stats3.cpp`.

```
// Game Stats 3.0
// Demonstrates constants

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

```
int main()
{
    const int ALIEN_POINTS = 150;
    int aliensKilled = 10;
    int score = aliensKilled * ALIEN_POINTS;
    cout << "score: " << score << endl;

    enum difficulty {NOVICE, EASY, NORMAL, HARD, UNBEATABLE};
    difficulty myDifficulty = EASY;

    enum shipCost {FIGHTER_COST = 25, BOMBER_COST, CRUISER_COST = 50};
    shipCost myShipCost = BOMBER_COST;
    cout << "\nTo upgrade my ship to a Cruiser will cost "
         << (CRUISER_COST - myShipCost) << " Resource Points.\n";

    return 0;
}
```

Using Constants

I define a constant, `ALIEN_POINTS`, to represent the point value of an alien.

```
const int ALIEN_POINTS = 150;
```

I simply use the keyword `const` to modify the definition. Now I can use `ALIEN_POINTS` just like any integer literal. Also, notice that the name I chose for the constant is in all capital letters. This is just a convention, but it's a common one. An identifier in all caps tells a programmer that it represents a constant value.

Next, I put the constant to use in the following line:

```
int score = aliensKilled * ALIEN_POINTS;
```

I calculate a player's score by multiplying the number of aliens killed by the point value of an alien. Using a constant here makes the line of code quite clear.

Trap

You can't assign a new value to a constant. If you try, you'll generate a compile error.

Using Enumerations

An *enumeration* is a set of unsigned `int` constants, called *enumerators*. Usually the enumerators are related and have a particular order. Here's an example of an enumeration:

```
enum difficulty {NOVICE, EASY, NORMAL, HARD, UNBEATABLE};
```

This defines an enumeration named `difficulty`. By default, the value of enumerators begins at zero and increases by one. So `NOVICE` is 0, `EASY` is 1, `NORMAL` is 2, `HARD` is 3, and `UNBEATABLE` is 4. To define an enumeration of your own, use the keyword `enum` followed by an identifier, followed by a list of enumerators between curly braces.

Next, I create a variable of this new enumeration type.

```
difficulty myDifficulty = EASY;
```

The variable `myDifficulty` is set to `EASY` (which is equal to 1). `myDifficulty` is of type `difficulty`, so it can only hold one of the values defined in the enumeration. That means `myDifficulty` can only be assigned `NOVICE`, `EASY`, `NORMAL`, `HARD`, `UNBEATABLE`, 0, 1, 2, 3, or 4.

Next, I define another enumeration.

```
enum shipCost {FIGHTER_COST = 25, BOMBER_COST, CRUISER_COST = 50};
```

This line of code defines the enumeration `shipCost`, which represents the cost in Resource Points for three kinds of ships in a strategy game. In it, I assign specific integer values to some of the enumerators. The numbers represent the Resource Point value of each ship. You can assign values to the enumerators if you want. Any enumerators that are not assigned values get the value of the previous enumerator plus one. Because I didn't assign a value to `BOMBER_COST`, it's initialized to 26.

Next, I define a variable of this new enumeration type.

```
shipCost myShipCost = BOMBER_COST;
```

Then I demonstrate how you can use enumerators in arithmetic calculations.

```
(CRUISER_COST - myShipCost)
```

This piece of code calculates the cost of upgrading a Bomber to a Cruiser. The calculation is the same as $50 - 26$, which evaluates to 24.

INTRODUCING LOST FORTUNE

The final project for this chapter, *Lost Fortune*, is a personalized adventure game in which the player enters a few pieces of information, which the computer uses to enhance a basic adventure story. Figure 1.9 shows a sample run.

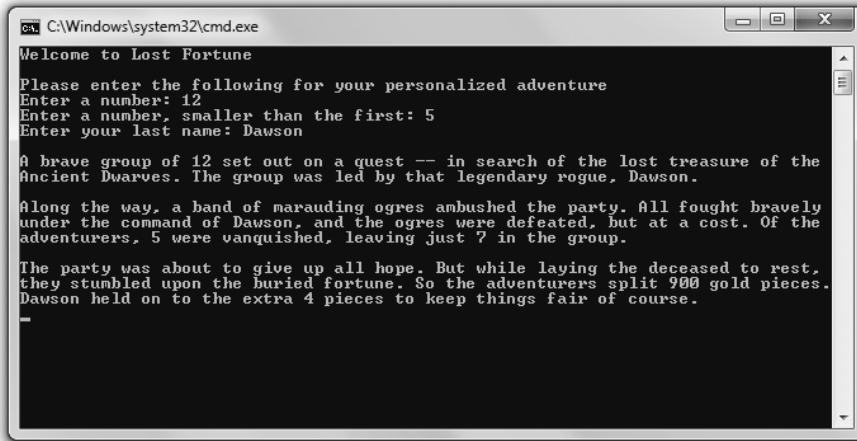


Figure 1.9

The story incorporates details provided by the player.

Used with permission from Microsoft.

Instead of presenting all the code at once, I'll go through it one section at a time. You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 1 folder; the filename is `lost_fortune.cpp`.

Setting Up the Program

First I create some initial comments, include two necessary files, and write a few using directives.

```
// Lost Fortune
// A personalized adventure

#include <iostream>
#include <string>

using std::cout;
using std::cin;
using std::endl;
using std::string;
```

I include the file `string`, part of the standard library, so I can use a `string` object to access a string through a variable. There's a lot more to `string` objects, but I'm going to keep you in suspense. You'll learn more about them in Chapter 3, "for Loops, Strings, and Arrays: Word Jumble."

Also, I employ using directives to spell out the objects in the `std` namespace that I plan to access. As a result, you can clearly see that `string` is in namespace `std`.

Getting Information from the Player

Next, I get some information from the player.

```
int main()
{
    const int GOLD_PIECES = 900;
    int adventurers, killed, survivors;
    string leader;

    //get the information
    cout << "Welcome to Lost Fortune\n\n";
    cout << "Please enter the following for your personalized adventure\n";

    cout << "Enter a number: ";
    cin >> adventurers;

    cout << "Enter a number, smaller than the first: ";
    cin >> killed;

    survivors = adventurers - killed;

    cout << "Enter your last name: ";
    cin >> leader;
```

`GOLD_PIECES` is a constant that stores the number of gold pieces in the fortune the adventurers seek. `adventurers` stores the number of adventurers on the quest. `killed` stores the number that are killed in the journey. I calculate `survivors` for the number of adventurers that remain. Finally, I get the player's last name, which I'll be able to access through `leader`.

Trap

This simple use of `cin` to get a string from the user only works with strings that have no whitespace (such as tabs or spaces) in them. There are ways to compensate for this, but that really requires a discussion of something called *streams*, which is beyond the scope of this chapter. So, use `cin` in this way, but be aware of its limitations.

Telling the Story

Next, I use the variables to tell the story.

```
//tell the story
cout << "\nA brave group of " << adventurers << " set out on a quest ";
cout << "-- in search of the lost treasure of the Ancient Dwarves. ";
cout << "The group was led by that legendary rogue, " << leader << ".\n";
```



```

cout << "\nAlong the way, a band of marauding ogres ambushed the party. ";
cout << "All fought bravely under the command of " << leader;
cout << ", and the ogres were defeated, but at a cost. ";
cout << "Of the adventurers, " << killed << " were vanquished, ";
cout << "leaving just " << survivors << " in the group.\n";

cout << "\nThe party was about to give up all hope. ";
cout << "But while laying the deceased to rest, ";
cout << "they stumbled upon the buried fortune. ";
cout << "So the adventurers split " << GOLD_PIECES << " gold pieces.";
cout << leader << " held on to the extra " << (GOLD_PIECES % survivors);
cout << " pieces to keep things fair of course.\n";

return 0;
}

```

The code and thrilling narrative are pretty clear. I will point out one thing, though. To calculate the number of gold pieces that the leader keeps, I use the modulus operator in the expression `GOLD_PIECES % survivors`. The expression evaluates to the remainder of `GOLD_PIECES / survivors`, which is the number of gold pieces that would be left after evenly dividing the stash among all of the surviving adventurers.

SUMMARY

In this chapter, you should have learned the following concepts:

- C++ is the primary language used in AAA game programming.
- A program is a series of C++ statements.
- The basic lifecycle of a C++ program is idea, plan, source code, object file, executable.
- Programming errors tend to fall into three categories—compile errors, link errors, and run-time errors.
- A function is a group of programming statements that can do some work and return a value.
- Every program must contain a `main()` function, which is the starting point of the program.
- The `#include` directive tells the preprocessor to include another file in the current one.
- The standard library is a set of files that you can include in your program files to handle basic functions like input and output.

- `iostream`, which is part of the standard library, is a file that contains code to help with standard input and output.
- The `std` namespace includes elements from the standard library. To access an element from the namespace, you need to prefix the element with `std::` or employ `using`.
- `cout` is an object, defined in the file `iostream`, that's used to send data to the standard output stream (generally the computer screen).
- `cin` is an object, defined in the file `iostream`, that's used to get data from the standard input stream (generally the keyboard).
- C++ has built-in arithmetic operators, such as the familiar addition, subtraction, multiplication, and division—and even the unfamiliar modulus.
- C++ defines fundamental types for Boolean, single-character, integer, and floating-point values.
- The C++ standard library provides a type of object (`string`) for strings.
- You can use `typedef` to create a new name for an existing type.
- A constant is a name for an unchangeable value.
- An enumeration is a sequence of `unsigned int` constants.

QUESTIONS AND ANSWERS

Q: Why do game companies use C++?

A: C++ combines speed, low-level hardware access, and high-level constructs better than just about any other language. In addition, most game companies have a lot invested in C++ resources (both in reusable code and in programmer experience).

Q: How is C++ different than C?

A: C++ is the next iteration of the C programming language. To gain acceptance, C++ essentially retained all of C. However, C++ defines new ways to do things that can replace some of the traditional C mechanisms. In addition, C++ adds the ability to write object-oriented programs.

Q: How is C++ different from C#?

A: C# is a programming language created by Microsoft intended to be both simple and general purpose. C# was influenced by and bears much similarity to C++, but the two are separate and distinct languages.

Q: How should I use comments?

A: To explain code that is unusual or unclear. You should not comment the obvious.

Q: What's a programming block?

A: One or more statements surrounded by curly braces that form a single unit.

Q: What's a compiler warning?

A: A message from your compiler stating a potential problem. A warning will not stop the compilation process.

Q: Can I ignore compiler warnings?

A: You can, but you generally shouldn't. You should address the warning and fix the offending code.

Q: What is whitespace?

A: A set of non-printing characters that create space in your source files, including tabs, spaces, and new lines.

Q: What are literals?

A: Elements that represent explicit values. "Game Over!" is a string literal, while 32 and 98.6 are numeric literals.

Q: Why should I always try to initialize a new variable with a value?

A: Because the contents of an uninitialized variable could be any value—even one that doesn't make sense for your program.

Q: What are variables of type `bool` for?

A: They can represent a condition that is true or false, such as whether a chest is locked or a playing card is face up.

Q: How did the `bool` type get its name?

A: The type is named in honor of the English mathematician George Boole.

Q: Must the names of constants be in uppercase letters?

A: No. Using uppercase is just an accepted practice—but one you should use because it's what other programmers expect.

Q: How can I store more than one character with a single variable?

A: With a string object.

DISCUSSION QUESTIONS

1. How does having a widely adopted C++ standard help game programmers?
2. What are the advantages and disadvantages of employing the `using` directive?
3. Why might you define a new name for an existing type?
4. Why are there two versions of the increment operator? What's the difference between them?
5. How can you use constants to improve your code?

EXERCISES

1. Create a list of six legal variable names—three good choices and three bad choices. Explain why each name falls into the good or bad category.
2. What's displayed by each line in the following code snippet? Explain each result.

```
cout << "Seven divided by three is " << 7 / 3 << endl;  
cout << "Seven divided by three is " << 7.0 / 3 << endl;  
cout << "Seven divided by three is " << 7.0 / 3.0 << endl;
```
3. Write a program that gets three game scores from the user and displays the average.