**CHAPTER 1**

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

**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.

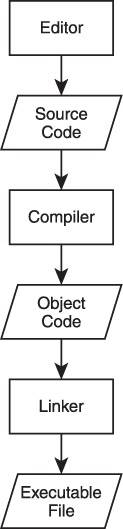
**Creating an Executable File**

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.

**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.

**WRITING YOUR FIRST C++ PROGRAM**

**// Game Over**

**// A first C++ program**

**#include <iostream>**

**int main()**

**{**

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

**return 0;**

**}**

**Game Over!**

**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.

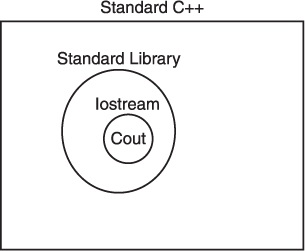
**cout** is an object, defined in the file **iostream** , that’s used to send data to the standard output stream. In most programs (including this one), the standard output stream simply means the console window on the computer screen.

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.

**WORKING WITH THE STD NAMESPACE**

**// Game Over 2.0**

**// Demonstrates a using directive**

**#include <iostream>**

**using namespace std;**

**int main()**

**{**

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

**return 0;**

**}**

**Game Over!**

**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** name-space 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**

**// Game Over 3.0**

**// Demonstrates using declarations**

**#include <iostream>**

**using std::cout;**

**using std::endl;**

**int main()**

**{**

**cout << "Game Over!" << endl;**

**return 0;**

**}**

**Game Over!**

**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.

**USING ARITHMETIC OPERATORS**

**// 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;**

**}**

**7 + 3 = 10**

**7 - 3 = 4**

**7 \* 3 = 21**

**7 / 3 = 2**

**7.0 / 3.0 = 2.33333**

**7 % 3 = 1**

**7 + 3 \* 5 = 22**

**(7 + 3) \* 5 = 50**

**Adding, Subtracting, and Multiplying**

I use the built-in arithmetic operators for addition (the plus sign, **+**), subtraction (the minus sign, **-**), and multiplication (an asterisk, **\***).

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* **3** *s 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**

**// 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 << "\ndistance: " << distance << endl;**

**cout << "\nplayAgain: " << playAgain << endl;**

**// Skipping shieldsUp since you don't generally print Boolean values**

**cout << "\nlives: " << lives << endl;**

**cout << "\naliensKilled: " << aliensKilled << endl;**

**cout << "\nengineTemp: " << 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;**

**}**

**score: 0**

**distance: 1200.76**

**playAgain: y**

**lives: 3**

**aliensKilled: 10**

**engineTemp: 6572.89**

**How much fuel? 78**

**fuel: 78**

**bonus: 10**

**Understanding Fundamental Types**

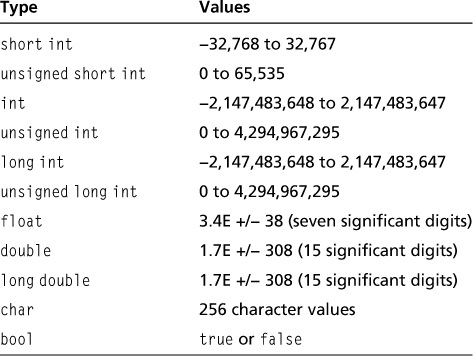
Every variable you create has a type, which represents the kind of information you can store in the variable. It tells your compiler how much memory to set aside for the variable and it defines exactly what you can legally do with the variable.

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*

**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**

**// 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;**

**}**

**score: 5000**

**score: 5100**

**score: 5200**

**lives: 4**

**lives: 4**

**lives, bonus = 4, 40**

**lives, bonus = 4, 30**

**score: 4294967295**

**score: 0**

**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**.

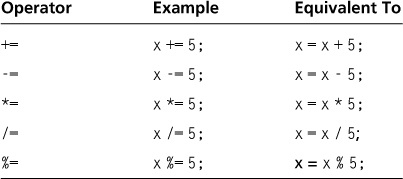
**Using Combined Assignment Operators**

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*

**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**.

**CONSTANST**

**// 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 Cruiser will cost "**

**<< (CRUISER\_COST - myShipCost) << " Resource Points.\n";**

**return 0;**

**}**

**score: 1500**

**To upgrade my ship to Cruiser will cost 24 Resource Points.**

**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 numerators 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.

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**

**// Lost Fortune**

**// A Personalized adventure**

**#include <iostream>**

**#include <string>**

**using namespace std;**

**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 first: ";**

**cin >> killed;**

**survivors = adventurers - killed;**

**cout << "Enter your last name: ";**

**cin >> leader;**

**// 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 adventures, " << killed << " were vanguished, ";**

**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 the extra " << (GOLD\_PIECES % survivors);**

**cout << " pieces to keep things fair of course.\n";**

**return 0;**

**}**

**Welcome to Lost Fortune**

**Please enter the following for your personalized adventure**

**Enter a number: 17**

**Enter a number, smaller than first: 9**

**Enter your last name: McKean**

**A brave group of 17 set out on a quest -- in search of the lost treasure of the Ancient Dwarves. The group was led by that legendary rogue, McKean.**

**Along the way, a band of marauding ogres ambushed the party. All fought bravely under the command of McKean, and the ogres were defeated, but at a cost. Of the adventures, 9 were vanguished, leaving just 8 in the group.**

**The party was about to give up all hope. But while laying the deceased to rest, they stumbled upon the buried fortune. So the adventurers split 900 gold pieces. McKean held on the extra 4 pieces to keep things fair of course.**

**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.

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.

**CHAPTER 2**

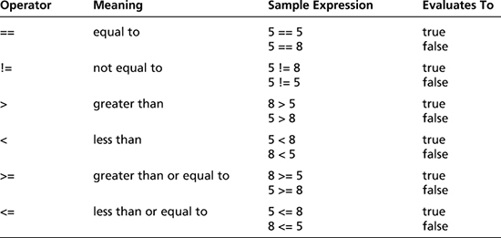
**TRUTH, BRANCHING, AND THE GAME LOOP: GUESS MY NUMBER**

**UNDERSTANDING TRUTH**

Truth is black and white, at least as far as C++ is concerned. You can represent **true** and **false** with their corresponding keywords, **true** and **false**. You can store such a Boolean value with a **bool** variable, as you saw in Chapter 1, “Types, Variables, and Standard I/O: Lost Fortune.” Here’s a quick refresher:

**bool fact = true, fiction = false;**

This code creates two **bool** variables, **fact** and **fiction**. **fact** is **true** and **fiction** is **false**. Although the keywords **true** and **false** are handy, any expression or value can be interpreted as **true** or **false** too. Any ***non-zero value*** can be interpreted as **true**, while **0** can be interpreted as **false**. A common kind of expression interpreted as **true** or **false** involves comparing things. Comparisons are often made by using ***built-in relational operators***. Table 2.1 lists the operators and a few sample expressions.



***Table 2.1*** *Relational Operators*

**USING THE IF STATEMENT**

Okay, it’s time to put the concepts of true and false to work. You can use an **if** statement to test an expression for truth and execute some code based on it.

Here’s a simple form of the if statement:

**if (expression)**

**statement;**

If expression is **true**, then statement is executed. Otherwise, statement is skipped and the program branches to the statement after the if suite.

**Hint**

*Whenever you see a generic statement like in the preceding code example, you can replace it with*

*a single statement or a block of statements because a block is treated as a single unit.*

**if STATEMENT**

**// Score Rater**

**// Demonstrates the if statement**

**#include <iostream>**

**using namespace std;**

**int main()**

**{**

**if(true)**

**{**

**cout << "This is always displayed.\n\n";**

**}**

**if(false)**

**{**

**cout << "This is never displaye.\n\n";**

**}**

**int score = 1000;**

**if(score)**

**{**

**cout << "At least you didn't score zero.\n\n";**

**}**

**if(score >= 250)**

**{**

**cout << "You scored 250 or more. Decent.\n\n";**

**}**

**if(score >= 500)**

**{**

**cout << "You scored 500 or more. Nice.\n\n";**

**if(score >= 1000)**

**{**

**cout << "You scored 1000 or more. Impressive!\n";**

**}**

**}**

**return 0;**

**}**

**This is always displayed.**

**At least you didn't score zero.**

**You scored 250 or more. Decent.**

**You scored 500 or more. Nice.**

**You scored 1000 or more. Impressive!**

**Testing true and false**

In the first if statement I test **true**. Because **true** is, well, **true**, the program displays the message, “**This is always displayed.**”

**if (true)**

**{**

**cout << "This is always displayed.\n\n";**

**}**

In the next if statement I test **false**. Because false isn’t **true**, the program doesn’t display the message, “**This is never displayed.**”

**if (false)**

**{**

**cout << "This is never displayed.\n\n";**

**}**

**Trap**

*Notice that you don’t use a semicolon after the closing parenthesis of the expression you test in an*

**if** *statement. If you were to do this, you’d create an empty statement that would be paired with the*

**if** *statement, essentially rendering the if statement useless. Here’s an example:*

**if (false);**

**{**

**cout << "This is never displayed.\n\n";**

**}**

*By adding the semicolon after (***false**), *I create an empty statement that’s associated with the* **if** *statement. The preceding code is equivalent to:*

**if (false)**

**; // an empty statement, which does nothing**

**{**

**cout << "This is never displayed.\n\n";**

**}**

*All I’ve done is play with the whitespace, which doesn’t change the meaning of the code. Now the*

*problem should be clear. The if statement sees the false value and skips the next statement (the*

*empty statement). Then the program goes on its merry way to the statement after the* **if** *statement,*

*which displays the message,* “**This is never displayed.**”

*Be on guard for this error. It’s an easy one to make and because it’s not illegal, it won’t produce*

*a compile error.*

**Interpreting a Value as true or false**

You can interpret any value as **true** or **false**. Any **non-zero** value can be interpreted as **true** , while **0** can be interpreted as **false** . I put this to the test in the next **if** statement:

**if (score)**

**{**

**cout << "At least you didn’t score zero.\n\n";**

**}**

score is **1000**, so it’s **non-zero** and interpreted as **true**. As a result, the message, “**At least you didn’t score zero,**” is displayed.

**Using Relational Operators**

Probably the most common expression you’ll use with **if** statements involves comparing values using the relational operators. That’s just what I’ll demonstrate next. I test to see whether the score is greater than or equal to **250**.

**if (score >= 250)**

**{**

**cout << "You scored 250 or more. Decent.\n\n";**

**}**

Because **score** is **1000**, the block is executed, displaying the message that the player earned a decent score. If score had been less than **1000**, the block would have been skipped and the program would have continued with the statement following the block.

**Trap**

*The equal to relational operator is* **==** *(****two equal signs*** *in a row). Don’t confuse it with* **=** *(****one***

***equal sign****), which is the assignment operator. While it’s not illegal to use the assignment operator*

*instead of the equal to relational operator, the results might not be what you expect:*

**int score = 500;**

**if (score = 1000)**

**{**

**cout << " You scored 1000 or more. Impressive!\n";**

**}**

*As a result of this code, score is set to* **1000** *and the message,* “**You scored 1000 or more.**

**Impressive!**” *is displayed. Here’s what happens: Although score is* **500** *before the* **if** *statement,*

*that changes. When the expression of the* **if** *statement,* **(score = 1000)**, *is evaluated, score is*

*assigned* **1000**. *The assignment statement evaluates to 1000*, *and because that’s a* **non-zero** *value,*

*the expression is interpreted as* **true**. *As a result, the string is displayed.*

*Be on guard for this type of mistake. It’s easy to make, and in some cases (like this one) it won’t*

*cause a compile error.*

**Nesting if Statements**

An **if** statement can cause a program to execute a statement or block of statements, including other **if** statements. When you write one **if** statement inside another, it’s called nesting. In the following code, the if statement that begins **if (score>=1000)** is nested inside the if statement that begins if **(score>=500)**.

**if (score >= 500)**

**{**

**cout << "You scored 500 or more. Nice.\n\n";**

**if (score >= 1000)**

**{**

**cout << "You scored 1000 or more. Impressive!\n";**

**}**

**}**

Because **score** is greater than **500**, the program enters the statement block and displays the message, “**You scored 500 or more. Nice.**” Then, in the inner **if** statement, the program compares score to **1000**. Because score is greater than or equal to **1000**, the program displays the message, “**You scored 1000 or**

**more. Impressive!**”

**Hint**

*You can nest as many levels as you want. However, if you nest code too deeply, it gets hard to read.*

*In general, you should try to limit your nesting to a few levels at most.*

**USING THE ELSE CLAUSE**

You can add an **else** clause to an **if** statement to provide code that will only be executed if the tested expression is **false** . Here’s the form of an **if** statement that includes an **else** clause:

**if (expression)**

**statement1;**

**else**

**statement2;**

If expression is **true** , **statement1** is executed. Then the program skips **statement2** and executes the statement following the if suite. If expression is **false** , **statement1** is skipped and **statement2** is executed. After **statement2** completes, the program executes the statement following the if suite.

**else CLAUSE**

**// Score Rater 2.0**

**// Demonstrates an else clause**

**#include <iostream>**

**using namespace std;**

**int main()**

**{**

**int score;**

**cout << "Enter your score: ";**

**cin >> score;**

**if(score >= 1000)**

**{**

**cout << "You scored 1000 or more. Impressive!\n";**

**}**

**else**

**{**

**cout << "You scored less than 1000.\n";**

**}**

**return 0;**

**}**

**Enter your score: 700**

**You scored less than 1000.**

**Creating Two Ways to Branch**

You’ve seen the first part of the if statement already, and it works just as it did before. If score is greater than **1000**, the message, “**You scored 1000 or more. Impressive!**” is displayed.

**if (score >= 1000)**

**{**

**cout << "You scored 1000 or more. Impressive!\n";**

**}**

Here’s the twist. The **else** clause provides a statement for the program to branch to if the expression is **false**. So **if (score >= 1000)** is **false**, then the program skips the first message and instead displays the message, “**You scored less than 1000.**”

**USING A SEQUENCE OF IF STATEMENTS WITH ELSE CLAUSES**

You can chain together if statements with else clauses to create a sequence of expressions that are tested in order. The statement associated with the first expression to test true is executed; otherwise, the statement associated with the final (and optional) else clause is run. Here’s the form such a series would take:

**if (expression1)**

**statement1;**

**else if (expression2)**

**statement2;**

**…**

**else if (expressionN)**

**statementN;**

**else**

**statementN+1;**

If **expression1** is **true**, **statement1** is executed and the rest of the code in the sequence is skipped. Otherwise, **expression2** is tested and if **true**, **statement2** is executed and the rest of the code in the sequence is skipped.

The computer continues to check each expression in order (through **expressionN**) and will execute the statement associated with the first expression that is **true**. If no expression is **true**, then the statement associated with the final else clause, **statementN+1**, is executed.

**if else - if else SUIT**

**// Score Rater 3.0**

**// Demonstrates if else-if else suit**

**#include <iostream>**

**using namespace std;**

**int main()**

**{**

**int score;**

**cout << "Enter your score: ";**

**cin >> score;**

**if(score >= 1000)**

**{**

**cout << "You scored 1000 or more. Impressive!\n";**

**}**

**else if(score >= 500)**

**{**

**cout << "You scored 500 or more. Nice.\n";**

**}**

**else if(score >= 250)**

**{**

**cout << "You scored 250 or more. Decent.\n";**

**}**

**else**

**{**

**cout << "You scored less than 250. Nothing to brag about.\n";**

**}**

**return 0;**

**}**

**Enter your score: 700**

**You scored 500 or more. Nice.**

**Creating a Sequence of if Statements with else Clauses**

You’ve seen the first part of this sequence twice already, and it works just the same this time around. If **score** is greater than or equal to **1000**, the message, “**You scored 1000 or more. Impressive!**” is displayed and the computer branches to the **return** statement.

**if (score >= 1000)**

However, if the expression is **false**, then we know that score is less than **1000** and the computer evaluates the next expression in the sequence:

**else if (score >= 500)**

If score is greater than or equal to **500**, the message, “**You scored 500 or more. Nice.**” is displayed and the computer branches to the **return** statement.

However, if that expression is **false**, then we know that score is less than **500** and the computer evaluates the next expression in the sequence:

**else if (score >= 250)**

If score is greater than or equal to **250**, the message, “**You scored 250 or more. Decent.**” is displayed and the computer branches to the **return** statement. However, if that expression is **false**, then we know that score is less than **250** and the statement associated with the final else clause is executed and the message, “**You scored less than 250. Nothing to brag about.**” is displayed.

**Hint**

*While the final* **else** *clause in an* **if else-if suite** *isn’t required, you can use it as a way to*

*execute code if none of the expressions in the sequence are true.*

**USING THE SWITCH STATEMENT**

You can use a **switch** statement to create multiple branching points in your code. Here’s a generic form of the switch statement:

**switch (choice)**

**{**

**case value1:**

**statement1;**

**break;**

**case value2:**

**statement2;**

**break;**

**case value3:**

**statement3;**

**break;**

**.**

**.**

**case valueN:**

**statementN;**

**break;**

**default:**

**statementN + 1;**

**}**

The **statement** tests **choice** against the possible values - **value1**, **value2**, and **value3** - in order. If **choice** is equal to a value, then the program executes the corresponding statement. When the program hits a **break** statement, it exits the **switch** structure. If **choice** doesn’t match any value, then the statement associated with the optional **default** is executed.

The use of **break** and **default** are optional. If you leave out a **break**, however, the program will continue through the remaining statements until it hits a **break** or a **default** or until the **switch** statement ends. Usually you want one **break** statement to end each case.

**Hint**

*Although a default case isn’t required, it’s usually a good idea to have one as a catchall.*

Here’s an example to cement the ideas. Suppose **choice** is equal to **value2**. The program will first test **choice** against **value1**. Because they’re not equal, the program will continue. Next, the program will test **choice** against **value2**.

Because they are equal, the program will execute **statement2**. Then the program will hit the **break** statement and exit the **switch** structure.

**Trap**

*You can use the* **switch** *statement only to test an int (or a value that can be treated as an* **int***, such*

*as a* **char** *or an* **enumerator***). A switch statement won’t work with any other type.*

**switch STATEMENT**

**// Menu Chooser**

**// Demonstrates the switch statement**

**#include <iostream>**

**using namespace std;**

**int main()**

**{**

**cout << "Difficulty Levels\n\n";**

**cout << "1 - Easy\n";**

**cout << "2 - Normal\n";**

**cout << "3 - Hard\n\n";**

**int choice;**

**cout << "Choice: ";**

**cin >> choice;**

**switch(choice)**

**{**

**case 1:**

**cout << "You picked Easy.\n";**

**break;**

**case 2:**

**cout << "You picked Normal.\n";**

**break;**

**case 3:**

**cout << "You picked Hard.\n";**

**break;**

**default:**

**cout << "You made an illegal choice.\n";**

**}**

**return 0;**

**}**

**Difficulty Levels**

**1 - Easy**

**2 - Normal**

**3 - Hard**

**Choice: 2**

**You picked Normal.**

**Creating Multiple Ways to Branch**

The **switch** statement creates four possible branching points. If the user enters **1**, then code associated with **case** **1** is executed and “**You picked Easy**” is displayed. If the user enters **2**, then code associated with **case 2** is executed and “**You picked Normal**” is displayed. If the user enters **3**, then code associated with **case 3** is executed and “**You picked Hard**” is displayed. If the user enters any other value, then **default** kicks in and “**You made an illegal choice**” is displayed.

**Trap**

*You’ll almost always want to end each case with a* **break** *statement. Don’t forget them; otherwise,*

*your code might do things you never intended.*

**USING WHILE LOOPS**

**// Play Again**

**// Demonstrates while loops**

**#include <iostream>**

**using namespace std;**

**int main()**

**{**

**char again = 'y';**

**while(again == 'y')**

**{**

**cout << "\n\*\*Played an exciting game\*\*";**

**cout << "\nDo you want to play again? (y/n): ";**

**cin >> again;**

**}**

**cout << "\nOkay, bye.";**

**return 0;**

**}**

**\*\*Played an exciting game\*\***

**Do you want to play again? (y/n): y**

**\*\*Played an exciting game\*\***

**Do you want to play again? (y/n): n**

**Okay, bye.**

**Looping with a while Loop**

The first thing the program does in the **main()** function is declare the **char** variable named **again** and initialize it to ’**y**’ . Then the program begins the **while** loop by testing again to see whether it’s equal to ’**y**’. Because it is, the program displays the message “**\*\*Played an exciting game\*\***” asks the user whether he wants to play again, and stores the reply in again. The loop continues as long as the user enters **y**.

You’ll notice that I had to initialize again before the loop because the variable is used in the loop expression. Because a while loop evaluates its expressions before its loop body (the group of statements that repeat), you have to make sure that any variables in the expression have a value before the loop begins.

**USING DO LOOPS**

Like **while** loops, **do** loops let you repeat a section of code based on an expression. The difference is that a **do** loop tests its expression after each loop iteration. This means that the loop body is always executed at least once. Here’s a generic form of a **do** loop:

**do**

**statement;**

**while (expression)**

The program executes statement and then, as long as expression tests **true**, the loop repeats. Once expression tests **false**, the loop ends.

**do LOOPS**

**// Play Again 2.0**

**// Demonstrates do loops**

**#include <iostream>**

**using namespace std;**

**int main()**

**{**

**char again;**

**do**

**{**

**cout << "\n\*\*Played an exciting game\*\*";**

**cout << "\nDo you want to play again? (y/n): ";**

**cin >> again;**

**} while(again == 'y');**

**cout << "\nOkay, bye.";**

**return 0;**

**}**

**\*\*Played an exciting game\*\***

**Do you want to play again? (y/n): y**

**\*\*Played an exciting game\*\***

**Do you want to play again? (y/n): n**

**Okay, bye.**

**Looping with a do Loop**

Before the **do** loop begins, I declare the character **again**. However, I don’t need to initialize it because it’s not tested until after the first iteration of the loop. I get a new value for again from the user in the loop body. Then I test again in the loop expression. If again is equal to ’**y**’, the loop repeats; otherwise, the loop ends.

**In the Real World**

*Even though you can use* **while** *and* **do** *loops pretty interchangeably, most programmers use the*

**while** *loop. Although a* **do** *loop might seem more natural in some cases, the advantage of a* **while**

*loop is that its expression appears right at the top of the loop; you don’t have to go hunting to the*

*bottom of the loop to find it.*

**Trap**

*If you’ve ever had a game get stuck in the same endless cycle, you might have experienced an*

***infinite loop -*** *a loop without end. Here’s a simple example of an infinite loop:*

**int test = 10;**

**while (test == 10)**

**{**

**cout << test;**

**}**

*In this case, the loop is entered because test is* **10***. But because test never changes, the loop will*

*never stop. As a result, the user will have to kill the running program to end it. The moral of this*

*story? Make sure that the expression of a loop can eventually become* **false** *or that there’s another*

*way for the loop to end, such as described in the following section, “Using break and continue*

*Statements.”*

**USING BREAK AND CONTINUE STATEMENTS**

It’s possible to alter the behavior you’ve seen in loops. You can immediately exit a loop with the **break** statement, and you can jump directly to the top of a loop with a **continue** statement. Although you should use these powers sparingly, they do come in handy sometimes.

**break and continue STATEMENTS**

**// Finicky Counter**

**// Demonstrates break and continue statements**

**#include <iostream>**

**using namespace std;**

**int main()**

**{**

**int count = 0;**

**while(true)**

**{**

**count += 1;**

**// end loop if count is greater than 10**

**if(count > 10)**

**{**

**break;**

**}**

**// skip the number 5**

**if(count == 5)**

**{**

**continue;**

**}**

**cout << count << endl;**

**}**

**return 0;**

**}**

**1**

**2**

**3**

**4**

**6**

**7**

**8**

**9**

**10**

**Creating a while (true) Loop**

I set up the loop with the following line:

**while (true)**

Technically, this creates an infinite loop. This might seem odd coming so soon after a warning to avoid infinite loops, but this particular loop isn’t really infinite because I put an exit condition in the loop body.

**Hint**

*Although a* **while (true)** *loop sometimes can be clearer than a traditional loop, you should also*

*try to minimize your use of these loops.*

**Using the Break Statement to Exit a Loop**

This is the exit condition I put in the loop:

**//end loop if count is greater than 10**

**if (count > 10)**

**{**

**break;**

**}**

Because count is increased by **1** each time the loop body begins, it will eventually reach **11**. When it does, the **break** statement (which means “***break out of the loop***”) is executed and the loop ends.

**Using the continue Statement to Jump Back to the Top of a Loop**

Just before **count** is displayed, I included the lines:

**//skip the number 5**

**if (count == 5)**

**{**

**continue;**

**}**

The **continue** statement means “***jump back to the top of the loop.***” At the top of the loop, the **while** expression is tested and the loop is entered again if it’s **true**. So when **count** is equal to **5**, the program does not get to the **cout << count << endl;** statement. Instead, it goes right back to the top of the loop.

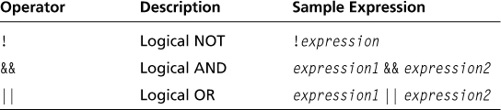
As a result, **5** is skipped and never displayed.

**Understanding When to Use break and continue**

You can use **break** and **continue** in any loop you create; they aren’t just for **while (true)** loops. But you should use them sparingly. Both **break** and **continue** can make it harder for programmers to see the flow of a loop.

**USING LOGICAL OPERATORS**

So far you’ve seen fairly simple expressions evaluated for their truth or falsity. However, you can combine simpler expressions with ***logical operators*** to create more complex expressions. Table 2.2 lists the logical operators.



***Table 2.2***

*Logical Operators*

**LOGICAL OPERATORS**

**// Designers Network**

**// Demonstrates logical operators**

**#include <iostream>**

**#include <string>**

**using namespace std;**

**int main()**

**{**

**string username;**

**string password;**

**bool success;**

**cout << "\tGame Designer's Network\n";**

**do**

**{**

**cout << "\nUsername: ";**

**cin >> username;**

**cout << "Password: ";**

**cin >> password;**

**if(username == "S.Meier" && password == "civilization")**

**{**

**cout << "\nHey, Sid.";**

**success = true;**

**}**

**else if(username == "S.Miyamoto" && password == "mariobros")**

**{**

**cout << "\nWhat's up, Shigeru?";**

**success = true;**

**}**

**else if(username == "W.Wright" && password == "thesims")**

**{**

**cout << "\nHow goes it, Will?";**

**success = true;**

**}**

**else if(username == "guest" || password == "guest")**

**{**

**cout << "\nWelcome, guest";**

**success = true;**

**}**

**else**

**{**

**cout << "\nYour login failed.";**

**success = false;**

**}**

**} while(!success);**

**return 0;**

**}**

**Game Designer's Network**

**Username: S.Meier**

**Password: civilization**

**Hey, Sid.**

**Game Designer's Network**

**Username: guest**

**Password: guest**

**Welcome, guest**

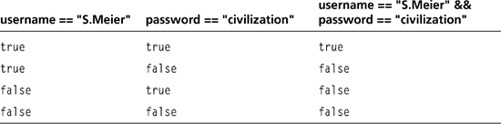
**Using the Logical AND Operator**

The logical **AND** operator, **&&**, lets you join two expressions to form a larger one, which can be evaluated to **true** or **false**. The new expression is **true** only if the two expressions it joins are **true**; otherwise, it is **false**. Just as in English, “and” means both. Both original expressions must be true for the new expression to be true. Here’s a concrete example from the Designers Network program:

**if (username == "S.Meier" && password == "civilization")**

The expression **username == "S.Meier" && password == "civilization"** is **true** only if both **username == "S.Meier"** and **password == "civilization"** are **true** . This works perfectly because I only want to grant Sid access if he enters both his username and his password. Just one or the other won’t do.

Another way to understand how **&&** works is to look at all of the possible combinations of truth and falsity (see Table 2.3).



***Table 2.3***

*Possible Login Combinations Using the AND Operator*

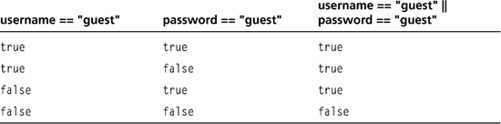
**Using the Logical OR Operator**

The logical **OR** operator, **||**, lets you join two expressions to form a larger one, which can be evaluated to **true** or **false**. The new expression is **true** if the first expression or the second expression is **true**; otherwise, it is **false**. Just as in English, “or” means either. If either the first or second expression is true, then the new expression is true. (If both are **true**, then the larger expression is still **true**.) Here’s a concrete example from the Designers Network program:

**else if (username == "guest" || password == "guest")**

The expression **username == "guest" || password == "guest"** is **true** if **username == "guest"** is **true** or if **password == "guest"** is **true**. This works perfectly because I want to grant a user access as a guest as long as he enters guest for the username or password. If the user enters guest for both, that’s fine too.

Another way to understand how **||** works is to look at all of the possible combinations of truth and falsity (see Table 2.4).



***Table 2.4***

*Possible Login Combinations Using the OR Operator*

**Using the Logical NOT Operator**

The logical **NOT** operator, **!**, lets you switch the truth or falsity of an expression. The new expression is **true** if the original is **false**; the new expression is **false** if the original is **true**. Just as in English, “not” means the opposite. The new expression has the opposite value of the original.

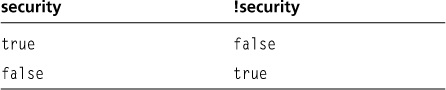
I use the NOT operator in the Boolean expression of the do loop:

**} while (!success);**

The expression **!success** is **true** when success is **false**. That works perfectly because success is **false** only when there has been a failed login. In that case, the block associated with the do loop executes again and the user is asked for his username and password once more.

The expression **!success** is **false** when success is **true** . That works perfectly because when success is **true** , the user has successfully logged in and the loop ends.

Another way to understand how **!** works is to look at all of the possible combinations of truth and falsity (see Table 2.5).



***Table 2.5***

*Possible Login Combinations Using the NOT Operator*

**Understanding Order of Operations**

Just like arithmetic operators, logical operators have precedence levels that affect the order in which an expression is evaluated. Logical **NOT**, **!** , has a higher level of precedence than logical **AND**, **&&** , which has a higher precedence than logical **OR**, **||** .

Just as with arithmetic operators, if you want an operation with lower precedence to be evaluated first, you can use parentheses. You can create complex expressions that involve arithmetic operators, relational operators, and logical operators. Operator precedence will define the exact order in which elements of the expression are evaluated. However, it’s best to try to create expressions that are clear and simple rather than expressions that require a mastery of the operator precedence list to decipher.

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

**Hint**

*Although you can use parentheses in a larger expression to change the way in which it’s evaluated,*

*you can also use redundant parentheses - parentheses that don’t change the value of the*

*expressions - to make the expression clearer. Let me give you a simple example. Check out the*

*following expression from the Designers Network program:*

**(username == "S.Meier" && password == "civilization")**

*Now, here’s the expression with some redundant parentheses:*

**( (username == "S.Meier") && (password == "civilization") )**

*While the extra parentheses don’t change the meaning of the expression, they really help the two*

*smaller expressions, joined by the* **&&** *operator, stand out.*

*Using redundant parentheses is a bit of an art form. Are they helpful or just plain redundant?*

*That’s a call you as the programmer have to make.*

**GENERATING RANDOM NUMBERS**

A sense of unpredictability can add excitement to a game. Whether it’s the sudden change in a computer opponent’s strategy in an **RTS** (***real-time strategy***) or an alien creature bursting from an arbitrary door in an **FPS** (***first-person shooter***), players thrive on a certain level of surprise. Generating random numbers is one way to achieve this kind of surprise.

**GENERATING RANDOM NUMBERS**

**// Die Roller**

**// Demonstrates generating random numbers**

**#include <iostream>**

**#include <cstdlib>**

**#include <ctime>**

**using namespace std;**

**int main()**

**{**

**// Maximum random number: cout << RAND\_MAX; --> 32767**

**srand(static\_cast<unsigned int>(time(0))); // seed a random number generator**

**int randomNumber = rand(); // generate random number**

**int die = (randomNumber % 6) + 1; // get a number between 1 and 6**

**cout << "You rolled a " << die << endl;**

**return 0;**

**}**

**You rolled a 6**

**Calling the rand() Function**

One of the first things I do in the program is include a new file:

**#include <cstdlib>**

The file **cstdlib** contains (among other things) functions that deal with generating random numbers. Because I’ve included the file, I’m free to call the functions it contains, including the function **rand()**, which is exactly what I do in **main()**:

**int randomNumber = rand(); //generate random number**

As you learned in Chapter 1, functions are pieces of code that can do some work and return a value. You call or invoke a function by using its name followed by a pair of parentheses. If a function returns a value, you can assign that value to a variable. That’s what I do here with my use of the assignment statement. I assign the value returned by **rand()** (a random number) to **randomNumber**.

**Hint**

*The* **rand()** *function generates a random number between* **0** *and at least* **32767***. The exact upper*

*limit depends on your implementation of C++. The upper limit is stored in the constant* **RAND\_MAX***,*

*which is defined in* **cstdlib***. So if you want to know the maximum random number* **rand()***can*

*generate, just send* **RAND\_MAX** *to* **cout** *.*

Functions can also take values to use in their work. You provide these values by placing them between the parentheses after the function name, separated by commas. These values are called arguments, and when you provide them, you pass them to the function. I didn’t pass any values to **rand()** because the function doesn’t take any arguments.

**Seeding the Random Number Generator**

Computers generate ***pseudorandom*** numbers - not truly random numbers - based on a formula. One way to think about this is to imagine that the computer reads from a huge book of predetermined numbers. By reading from this book, the computer can appear to produce a sequence of random numbers.

But there’s a problem: The computer always starts reading the book from the beginning. Because of this, the computer will always produce the same series of “random” numbers in a program. In games, this isn’t something we’d want. We wouldn’t, for example, want the same series of dice rolls in a game of craps every time we played.

A solution to this problem is to tell the computer to start reading from some arbitrary place in the book when a game program begins. This process is called ***seeding*** the random number generator. Game programmers give the random number generator a number, called a ***seed***, to determine the starting place in this sequence of pseudorandom numbers.

The following code seeds the random number generator:

**srand(static\_cast<unsigned int>(time(0))); // seed a random number generator**

Wow, that’s a pretty cryptic looking line, but what it does is simple. It seeds the random number generator based on the current date and time, which is perfect since the current date and time will be different for each run of the program.

In terms of the actual code, the **srand()** function seeds the random number generator - you just have to pass it an unsigned int as a seed. What gets passed to the function here is the return value of **time(0)** - a number based on the current system date and time. The code **static\_cast<unsigned int>** just converts (or casts) this value to an unsigned **int**. Now, you don’t have to understand all the nuances of this line; the least you need to know is that if you want a program to generate a series of random numbers that are different each time the program is run, your program should execute this line once before making calls to **rand()**.

**Calculating a Number within a Range**

After generating a random number, **randomNumber** holds a value between **0** and **32767** (based on my implementation of C++). But I need a number between **1** and **6**, so next I use the modulus operator to produce a number in that range.

**int die = (randomNumber % 6) + 1; // get a number between 1 and 6**

Any positive number divided by **6** will give a remainder between **0** and **5**. In the preceding code, I take this remainder and add **1**, giving me the possible range of **1** through **6** - exactly what I wanted. You can use this technique to convert a random number to a number within a range you’re looking for.

**Trap**

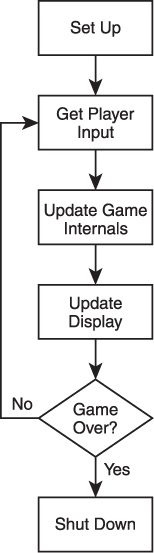
*Using the modulus operator to create a number within a range from a random number might not*

*always produce uniform results. Some numbers in the range might be more likely to appear than*

*others. However, this isn’t a problem for simple games.*

**UNDERSTANDING THE GAME LOOP**

The game loop is a generalized representation of the flow of events in a game. The core of the events repeats, which is why it’s called a loop. Although the implementation might be quite different from game to game, the fundamental structure is the same for almost all games across genres. Whether you’re talking about a simple space shooter or a complex role-playing game (RPG), you can usually break the game down into the same repeating components of the game loop. Figure 2.13 provides a visual representation of the game loop.



**Here’s an explanation of the parts of the game loop:**

* **Setup:** This often involves accepting initial settings or loading game assets, such as sound, music, and graphics. The player might also be presented with the game backstory and his objectives.
* **Getting player input:** Whether it comes from the keyboard, mouse, joystick, trackball, or some other device, input from the player is captured.
* **Updating game internals:** The game logic and rules are applied to the game world, taking into account player input. This might take the shape of a physics system determining the interaction of objects or it might involve calculations of enemy AI, for example.
* **Updating the display:** In the majority of games, this process is the most taxing on the computer hardware because it often involves drawing graphics. However, this process can be as simple as displaying a line of text.
* **Checking whether the game is over:** If the game isn’t over (if the player’s character is still alive and the player hasn’t quit, for example), control branches back to the getting player input stage. If the game is over, control falls through to the shutting down stage.
* **Shutting down:** At this point, the game is over. The player is often given some final information, such as his score. The program frees any resources, if necessary, and exits.

***Figure 2.13***

The game loop describes a basic flow of events that fits just about any game.

**INTRODUCING GUESS MY NUMBER**

**// Guess My Number**

**// The classic number guessing game**

**#include <iostream>**

**#include <cstdlib>**

**#include <ctime>**

**using namespace std;**

**int main()**

**{**

**srand(static\_cast<unsigned int>(time(0))); // seed a random number generator**

**int secretNumber = rand() % 100 + 1; // random number between 1 and 100**

**int tries = 0;**

**int guess;**

**cout << "\tWelcome to Guess My Number\n\n";**

**do**

**{**

**cout << "Enter a guess: ";**

**cin >> guess;**

**++tries;**

**if(guess > secretNumber)**

**{**

**cout << "Too High!\n\n";**

**}**

**else if(guess < secretNumber)**

**{**

**cout << "Too low!\n\n";**

**}**

**else**

**{**

**cout << "\nThat's it! You got it in " << tries << " guesses!\n";**

**}**

**} while(guess != secretNumber);**

**return 0;**

**}**

**Welcome to Guess My Number**

**Enter a guess: 77**

**Too low!**

**Enter a guess: 86**

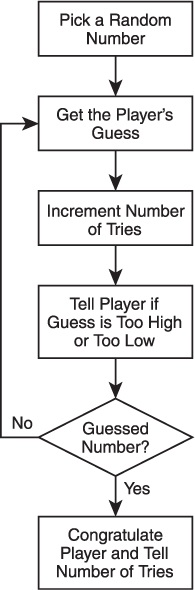
**Too High!**

**Enter a guess: 83**

**That's it! You got it in 3 guesses!**

**Applying the Game Loop**

It’s possible to examine even this simple game through the construct of the game loop. Figure 2.15 shows how nicely the game loop paradigm fits the flow of the game.



***Figure 2.15***

The game loop applied to Guess My Number.

**SUMMARY**

In this chapter, you learned the following concepts:

* You can use the truth or falsity of an expression to branch to (or skip) sections of code.
* You can represent truth or falsity with the keywords, **true** and **false**.
* You can evaluate any value or expression for truth or falsity.
* Any **non-zero** value can be interpreted as **true**, while **0** can be interpreted as **false**.
* A common way to create an expression to be evaluated as **true** or **false** is to compare values with the relational operators.
* The **if** statement tests an expression and executes a section of code only if the expression is **true**.
* The **else** clause of an **if** statement specifies code that should be executed only if the expression tested in the if statement is **false**.
* The **switch** statement tests a value that can be treated as an **int** and executes a section of code labeled with the corresponding value.
* The **default** keyword, when used in a **switch** statement, specifies code to be executed if the value tested in the **switch** statement matches no listed values.
* The **while** loop executes a section of code if an expression is **true** and repeats the code as long as the expression is **true**.
* A **do** loop executes a section of code and then repeats the code as long as the expression is **true**.
* Used in a loop, the **break** statement immediately ends the loop.
* Used in a loop, the **continue** statement immediately causes the control of the program to branch to the top of the loop.
* The **&&** (AND) operator combines two simpler expressions to create a new expression that is **true** only if both simpler expressions are **true**.
* The **||** (OR) operator combines two simpler expressions to create a new expression that is **true** if either simpler expression is **true**.
* The **!** (NOT) operator creates a new expression that is the opposite truth value of the original.
* The game loop is a generalized representation of the flow of events in a game, the core of which repeats.
* The file **cstdlib** contains functions that deal with generating random numbers.
* The function **srand()**,defined in **cstdlib** , seeds the random number generator.
* The function **rand()**, defined in **cstdlib** , returns a random number.

**CHAPTER 3**

**FOR LOOPS, STRINGS, AND ARRAYS: WORD JUMBLE**

**USING FOR LOOPS**

You met one type of loop in Chapter 2, “Truth, Branching, and the Game Loop: Guess My Number,” - the **while** loop. Well, it’s time to meet another - the **for** loop. Like its cousin the while loop, the **for** loop lets you repeat a section of code, but **for** loops are particularly suited for counting and moving through a sequence of things (like the items in an RPG character’s inventory).

Here’s the generic form of **for** loop:

**for (initialization; test; action)**

**statement;**

**initialization** is a statement that sets up some initial condition **for** the loop. (For example, it might set a counter variable to **0**.) The expression test is tested each time before the loop body executes, just as in a while loop. If test is **false**, the program moves on to the statement after the loop. If test is **true**, the program executes **statement**. Next, action is executed (which often involves incrementing a counter variable). The cycle repeats until test is **false**, at which point the loop ends.

**for LOOPS**

**// Counter**

**// Demonstrates for loops**

**#include <iostream>**

**using namespace std;**

**int main()**

**{**

**cout << "Counting forward:\n";**

**for(int i = 0; i < 10; ++i)**

**{**

**cout << i << " ";**

**}**

**cout << "\n\nCounting backward:\n";**

**for(int i = 9; i >= 0; --i)**

**{**

**cout << i << " ";**

**}**

**cout << "\n\nCounting by fives:\n";**

**for(int i = 0; i <= 50; i += 5)**

**{**

**cout << i << " ";**

**}**

**cout << "\n\nCounting with null statements:\n";**

**int count = 0;**

**for( ; count < 10; )**

**{**

**cout << count << " ";**

**++count;**

**}**

**cout << "\n\nCounting with nested for loops:\n";**

**const int ROWS = 5;**

**const int COLUMNS = 3;**

**for(int i = 0; i < ROWS; ++i)**

**{**

**for( int j = 0; j < COLUMNS; ++j)**

**{**

**cout << i << "," << j << " ";**

**}**

**cout << endl;**

**}**

**return 0;**

**}**

**Counting forward:**

**0 1 2 3 4 5 6 7 8 9**

**Counting backward:**

**9 8 7 6 5 4 3 2 1 0**

**Counting by fives:**

**0 5 10 15 20 25 30 35 40 45 50**

**Counting with null statements:**

**0 1 2 3 4 5 6 7 8 9**

**Counting with nested for loops:**

**0,0 0,1 0,2**

**1,0 1,1 1,2**

**2,0 2,1 2,2**

**3,0 3,1 3,2**

**4,0 4,1 4,2**

**Counting with for Loops**

The first for loop counts from **0** to **9**. The loop begins:

**for (int i = 0; i < 10; ++i)**

The initialization statement, **int i = 0**, declares **i** and initializes it to **0**. The expression **i < 10** says that the loop will continue as long as **i** is less than **10**. Lastly, the action statement, **++i**, says **i** is to be incremented each time the loop body finishes. As a result, the loop iterates **10** times - once for each of the values **0** through **9**. And during each iteration, the loop body displays the value of **i**.

The next for loop counts from **9** down to **0**. The loop begins:

**for (int i = 9; i >= 0; --i)**

Here, **i** is initialized to **9**, and the loop continues as long as **i** is greater than or equal to **0**. Each time the loop body finishes, **i** is decremented. As a result, the loop displays the values **9** through **0**.

The next loop counts from **0** to **50**, by fives. The loop begins:

**for (int i = 0; i <= 50; i += 5)**

Here, **i** is initialized to **0**, and the loop continues as long as **i** is less than or equal to **50**. But notice the action statement, **i += 5**. This statement increases **i** by five each time the loop body finishes. As a result, the loop displays the values **0**, **5**, **10**, **15**, and so on. The expression **i <= 50** says to execute the loop body as long as **i** is less than or equal to **50**.

You can initialize a **counter** variable, create a test condition, and update the **counter** variable with any values you want. However, the most common thing to do is to start the counter at **0** and increment it by **1** after each loop iteration.

Finally, the caveats regarding infinite loops that you learned about while studying while loops apply equally well to for loops. Make sure you create loops that can end; otherwise, you’ll have a very unhappy gamer on your hands.

**Using Empty Statements in for Loops**

You can use empty statements in creating your for loop, as I did in the following loop:

**for ( ; count < 10; )**

I used an empty statement for the initialization and action statements. That’s fine because I declared and initialized **count** before the loop and incremented it inside the loop body. This loop displays the same sequence of integers as the very first loop in the program. Although the loop might look odd, it’s perfectly legal.

**Hint**

*Different game programmers have different traditions. In the last chapter, you saw that you can*

*create a loop that continues until it reaches an exit statement - such as a* **break** *- using* **while**

**(true)***. Well, some programmers prefer to create these kinds of loops using a for statement that*

*begins with* **for (;;)***. Because the test expression in this loop is the empty statement, the loop will*

*continue until it encounters some exit statement.*

**Nesting for Loops**

You can nest for loops by putting one inside the other. That’s what I did in the following section of code, which counts out the elements of a grid. The outer loop, which begins:

**for (int i = 0; i < ROWS; ++i)**

simply executes its loop body **ROWS** (five) times. But it just so happens that there’s another for loop inside this loop, which begins:

**for (int j = 0; j < COLUMNS; ++j)**

As a result, the inner loop executes in full for each iteration of the outer loop.

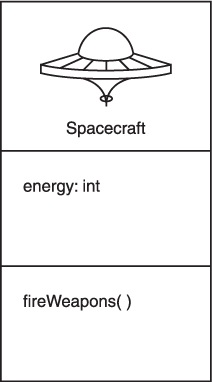
In this case, that means the inner loop executes **COLUMNS** (three) times, for the **ROWS** (five) times the outer loop iterates, for a total of **15** times. Specifically, here’s what happens:

1. The outer for loop declares **i** and initializes it to **0**. Since **i** is less than **ROWS** (**5**), the program enters the outer loop’s body.
2. The inner loop declares **j** and initializes it to **0**. Since **j** is less than **COLUMNS** (**3**), the program enters its loop body, sending the values of **i** and **j** to **cout** , which displays **0, 0**.
3. The program reaches the end of the body of the inner loop and increments j to 1. Since j is still less than **COLUMNS** (**3**), the program executes the inner loop’s body again, displaying **0, 1**.
4. The program reaches the end of the inner loop’s body and increments **j** to 2. Since **j** is still less than **COLUMNS** (**3**), the program executes the inner loop’s body again, displaying **0, 2**.
5. The program reaches the end of the inner loop’s body and increments **j** to **3**. This time, however, **j** is not less than **COLUMNS** (**3**) and the inner loop ends.
6. The program finishes the first iteration of the outer loop by sending **endl** to **cout** , ending the first row.
7. The program reaches the end of the outer loop’s body and increments **i** to **1**. Since **i** is less than **ROWS** (**5**), the program enters the outer loop’s body again.
8. The program reaches the inner loop, which starts from the beginning once again, by declaring and initializing **j** to **0**. The program goes through the process described in Steps 2 through 7, displaying the second row of the grid. This process continues until all five rows have been displayed.

Again, the important thing to remember is that the inner loop is executed in full for each iteration of the outer loop.

**UNDERSTANDING OBJECTS**

So far, you’ve seen how to store individual pieces of information in variables and how to manipulate those variables using operators and functions. But most of the things you want to represent in games - such as, say, an alien spacecraft—are objects. They’re encapsulated, cohesive things that combine qualities (such as an energy level) and abilities (for example, firing weapons). Often it makes no sense to talk about the individual qualities and abilities in isolation from each other.

Fortunately, most modern programming languages let you work with software objects (often just called **objects**) that combine ***data*** and ***functions***. A data element of an object is called a ***data member***, while a function of an object is called a ***member function***. As a concrete example, think about that alien spacecraft. An alien spacecraft object might be of a new type called Spacecraft, defined by a game programmer, and might have a data member for its energy level and a member function to fire its weapons. In practice, an object’s energy level might be stored in its data member **energy** as an **int**, and its ability to fire its weapons might be defined in a member function called **fireWeapons()**.

Every object of the same type has the same basic structure, so each object will have the same set of data members and member functions. However, as an individual, each object will have its own values for its data members. If you had a squadron of five alien spacecrafts, each would have its own energy level. One might have an energy level of 75, while another might have an energy level of only 10, and so on. Even if two crafts have the same energy level, each would belong to a unique spacecraft. Each craft could also fire its own weapons with a call to its member function, **fireWeapons()**. Figure 3.2 illustrates the concept of an alien spacecraft.

***Figure 3.2***

*This representation of the definition of an alien spacecraft says that each object will have a data member called* **energy** *and a member function called* **fireWeapons()**.

The cool thing about objects is that you don’t need to know the implementation details to use them - just as you don’t need to know how to build a car in order to drive one. You only have to know the object’s data members and member functions - just as you only need to know where a car’s steering wheel, gas pedal, and brake pedal are located.

You can store objects in variables, just like with built-in types. Therefore, you could store an alien spacecraft object in a variable of the Spacecraft type. You can access data members and member functions using the ***member selection operator*** (**.**), by placing the operator after the variable name of the object. So if you want your alien spacecraft, ship, to fire its weapons only if its energy level is greater than **10**, you could write:

**// ship is an object of Spacecraft type**

**if (ship.energy > 10) // ship.energy** accesses the object’s energy data member

**{**

**ship.fireWeapons() //** calls the object’s **fireWeapons()** member function.

**}**

**USING STRING OBJECTS**

**string** objects, which you met briefly in Chapter 1, “Types, Variables, and Standard I/O: Lost Fortune,” are the perfect way to work with sequences of characters, whether you’re writing a complete word puzzle game or simply storing a player’s name. A **string** is actually an **object**, and it provides its own set of member functions that allow you to do a range of things with the **string** **object -** everything from simply getting its length to performing complex character substitutions. In addition, strings are defined so that they

work intuitively with a few of the operators you already know.

**string OBJECTS**

**// String Tester**

**// Demonstrate string objects**

**#include <iostream>**

**#include <string>**

**using namespace std;**

**int main()**

**{**

**string word1 = "Game";**

**string word2("Over");**

**string word3(3, '!');**

**string phrase = word1 + " " + word2 + word3;**

**cout << "The phrase has: " << phrase.size() << " characters in it\n\n";**

**cout << "The character at position 0 is: " << phrase[0] << "\n\n";**

**cout << "Changing the character at position 0.\n";**

**phrase[0] = 'L';**

**cout << "The phrase is now: " << phrase << "\n\n";**

**for(unsigned int i = 0; i < phrase.size(); ++i)**

**{**

**cout << "Character at position " << i << " is: " << phrase[i] << endl;**

**}**

**cout << "\nThe sequence 'Over' begins at location ";**

**cout << phrase.find("Over") << endl;**

**if(phrase.find("eggplant") == string::npos)**

**{**

**cout << "'eggplant' is not in the phrase.\n\n";**

**}**

**phrase.erase(4, 5);**

**cout << "The phrase is now: " << phrase << endl;**

**phrase.erase(4);**

**cout << "The phrase is now: " << phrase << endl;**

**phrase.erase();**

**cout << "The phrase is now: " << phrase << endl;**

**if(phrase.empty())**

**{**

**cout << "\nThe phrase is no more.\n";**

**}**

**return 0;**

**}**

**The phrase has: 12 characters in it**

**The character at position 0 is: G**

**Changing the character at position 0.**

**The phrase is now: Lame Over!!!**

**Character at position 0 is: L**

**Character at position 1 is: a**

**Character at position 2 is: m**

**Character at position 3 is: e**

**Character at position 4 is:**

**Character at position 5 is: O**

**Character at position 6 is: v**

**Character at position 7 is: e**

**Character at position 8 is: r**

**Character at position 9 is: !**

**Character at position 10 is: !**

**Character at position 11 is: !**

**The sequence 'Over' begins at location 5**

**'eggplant' is not in the phrase.**

**The phrase is now: Lame!!!**

**The phrase is now: Lame**

**The phrase is now:**

**The phrase is no more.**

**Creating string Objects**

The first thing I do in **main()** is create three strings in three different ways:

**string word1 = "Game";**

**string word2("Over");**

**string word3(3, ’!’);**

In the first line of this group, I simply create the string object word1 using the assignment operator in the same way you’ve seen for other variables. As a result, **word1** is "**Game**".

Next, I create **word2** by placing the string object to which I want the variable set between a pair of parentheses. As a result, **word2** is "**Over**".

Finally, I create **word3** by supplying between a pair of parentheses a number followed by a single character. This produces a string object made up of the provided character, which has a length equal to the number. As a result, **word3** is "**!!!**".

**Concatenating string Objects**

Next, I create a new string object, **phrase**, by concatenating the first three string objects:

**string phrase = word1 + " " + word2 + word3;**

As a result, **phrase** is “**Game Over!!!**”.

Notice that the **+** operator, which you’ve seen work only with numbers, also concatenates string objects. That’s because the **+** operator has been overloaded. Now, when you first hear the term overloaded, you might think it’s a bad thing—the operator is about to blow! But it’s a good thing. Operator overloading redefines a familiar operator so it works differently when used in a new, previously undefined context. In this case, I use the **+** operator not to add numbers but to join string objects. I’m able to do this only because the string type specifically overloads the **+** operator and defines it so the operator means string object concatenation when used with strings.

**Using the size() Member Function**

Okay, it’s time to take a look at a string member function. Next, I use the member function **size()**:

**cout << "The phrase has: " << phrase.size() << " characters in it\n\n";**

**phrase.size()** calls the member function **size()** of the string object **phrase** through the member selection operator **.** (the dot). The **size()** member function simply returns an unsigned integer value of the size of the string object - its number of characters. Because the string object is "**Game Over!!!**", the member function returns **12**. (Every character counts, including spaces.) Of course, calling **size()** for another string object might return a different result based on the number of characters in the string object.

**Hint**

**string** *objects also have a member function* **length()***, which, just like* **size()***, returns the number*

*of characters in the string object.*

**Indexing a string Object**

A **string** object stores a sequence of char values. You can access any individual **char** value by providing an index number with the subscripting operator ( **[]** ). That’s what I do next:

**cout << "The character at position 0 is: " << phrase[0] << "\n\n";**

The first element in a sequence is at position **0**. In the previous statement, **phrase[0]** is the character **G**. And because counting begins at **0**, the last character in the string object is **phrase[11]**, even though the string object has **12** characters in it.

**Trap**

*It’s a common mistake to forget that indexing begins at position* **0***. Remember, a* **string** *object*

*with* **n** *characters in it can be indexed from position* **0** *to position* **n−1***.*

Not only can you access characters in a **string** object with the subscripting operator, but you can also reassign them. That’s what I do next:

**phrase[0] = 'L';**

I change the first character of phrase to the character **L**, which means phrase becomes "**Lame Over!!!**"

**Trap**

*C++ compilers do not perform bounds checking when working with* **string** *objects and the*

*subscripting operator. This means that the compiler doesn’t check to see whether you’re attempting*

*to access an element that doesn’t exist. Accessing an invalid sequence element can lead to*

*disastrous results because it’s possible to write over critical data in your computer’s memory. By*

*doing this, you can crash your program, so take care when using the subscripting operator.*

**Iterating through string Objects**

Given your new knowledge of **for** loops and string objects, it’s a snap to iterate through the individual characters of a string object. That’s what I do next:

**for(unsigned int i = 0; i < phrase.size(); ++i)**

**{**

**std::cout << "Character at position " << i << " is: " << phrase[i] << std::endl;**

**}**

The loop iterates through all of the valid positions of **phrase**. It starts with **0** and goes through **11**. During each iteration, a character of the string object is displayed with **phrase[i]**. Note that I made the loop variable, **i**, an unsigned **int** because the value returned by **size()** is an unsigned integral type.

**In the Real World**

*Iterating through a sequence is a powerful and often-used technique in games. You might, for*

*example, iterate through hundreds of individual units in a strategy game, updating their status and*

*order. Or you might iterate through the list of vertices of a 3D model to apply some geometric*

*transformation. Using the* **find()** *Member Function Next, I use the member function* **find()** *to*

*check whether either of two string*

**Using the find( ) Member Function**

Next, I use the member function **find()** to check whether either of two string literals are contained in phrase . First, I check for the string literal "**Over**":

**cout << "\nThe sequence ’Over’ begins at location ";**

**cout << phrase.find("Over") << endl;**

The **find()** member function searches the calling string object for the string supplied as an argument. The member function returns the position number of the first occurrence where the **string** object for which you are searching begins in the calling string object. This means that **phrase.find("Over")** returns the position number where the first occurrence of "**Over**" begins in **phrase** . Since phrase is "**Lame Over!!!**", **find()** returns **5**. (Remember, position numbers begin at **0**, so **5** means the sixth character.)

But what if the string for which you are searching doesn’t exist in the calling string? I tackle that situation next:

**if (phrase.find("eggplant") == string::npos)**

**{**

**cout << "’eggplant’ is not in the phrase.\n\n";**

**}**

Because "**eggplant**" does not exist in **phrase**, **find()** returns a special constant defined in the file string, which I access with **string::npos** . As a result, the screen displays the message, “**’eggplant’ is not in the phrase.**”

The constant I access through **string::npos** represents the largest possible size of a string object, so it is greater than any possible valid position number in a **string** object. Informally, it means “a position number that can’t exist.” It’s the perfect return value to indicate that one string couldn’t be found in another.

**Hint**

*When using* **find()***, you can supply an optional argument that specifies a character number for the program to start looking for the substring. The following line will start looking for the string literal "***eggplant***" beginning at position***5** *in the string object* **phrase***.*

**location = phrase.find("eggplant", 5);**

**Using the erase() Member Function**

The **erase()** member function removes a specified substring from a **string** object. One way to call the member function is to specify the beginning position and the length of the substring, as I did in this code:

**phrase.erase(4, 5);**

The previous line removes the five-character substring starting at position **4**. Because phrase is "**Lame Over!!!**", the member function removes the substring **Over** and, as a result, phrase becomes "**Lame!!!**".

Another way to call **erase()** is to supply just the beginning position of the substring. This removes all of the characters starting at that position number to the end of the **string** object. That’s what I do next:

**phrase.erase(4);**

This line removes all of the characters of the string object starting at position **4**. Since phrase is "**Lame!!!**", the member function removes the substring **!!!** and, as a result, phrase becomes "**Lame**".

Yet another way to call **erase()** is to supply no arguments, as I did in this code:

**phrase.erase();**

The previous line erases every character in **phrase**. As a result, phrase becomes the empty string, which is equal to **""**.

**Using the empty() Member Function**

The **empty()** member function returns a **bool** value— **true** if the **string** object is empty and **false** otherwise. I use **empty()** in the following code:

**if (phrase.empty())**

**{**

**cout << "\nThe phrase is no more.\n";**

**}**

Because phrase is equal to the empty string, **phrase().empty** returns **true**, and the screen displays the message, “**The phrase is no more.**”

**USING ARRAYS**

While **string** objects provide a great way to work with a sequence of characters, arrays provide a way to work with elements of any type. That means you can use an array to store a sequence of integers for, say, a high-score list. But it also means that you can use arrays to store elements of programmer-defined types, such as a sequence of items that an RPG character might carry.

**ARRAYS**

**// Hero's Inventory**

**// Demonstrates arrays**

**#include <iostream>**

**#include <string>**

**using namespace std;**

**int main()**

**{**

**const int MAX\_ITEMS = 10;**

**string inventory[MAX\_ITEMS];**

**int numItems = 0;**

**inventory[numItems++] = "sword";**

**inventory[numItems++] = "armor";**

**inventory[numItems++] = "shield";**

**cout << "Your items:\n";**

**for(int i = 0; i < numItems; ++i)**

**{**

**cout << inventory[i] << endl;**

**}**

**cout << "\nYou trade your sword for a battle axe.";**

**inventory[0] = "battle axe";**

**cout<< "\nYour items:\n";**

**for(int i = 0; i < numItems; ++i)**

**{**

**cout << inventory[i] << endl;**

**}**

**cout<< "\nThe item name '" << inventory[0] << "' has ";**

**cout<< inventory[0].size() << " letters in it.\n";**

**cout<< "\nYou find a healing potion.";**

**if(numItems < MAX\_ITEMS)**

**{**

**inventory[numItems++] = "healing potion";**

**}**

**else**

**{**

**cout<< "You have too many items and can't carry another.";**

**}**

**cout<< "\nYour items:\n";**

**for(int i = 0; i < numItems; ++i)**

**{**

**cout << inventory[i] << endl;**

**}**

**return 0;**

**}**

**Your items:**

**sword**

**armor**

**shield**

**You trade your sword for a battle axe.**

**Your items:**

**battle axe**

**armor**

**shield**

**The item name 'battle axe' has 10 letters in it.**

**You find a healing potion.**

**Your items:**

**battle axe**

**armor**

**shield**

**healing potion**

**Creating Arrays**

It’s often a good idea to define a constant for the number of elements in an array. That’s what I did with **MAX\_ITEMS**, which represents the maximum number of items the hero can carry:

**const int MAX\_ITEMS = 10;**

You declare an array much the same way you would declare any variable you’ve seen so far: You provide a type followed by a name. In addition, your compiler must know the size of the array so it can reserve the necessary memory space. You can provide that information following the array name, surrounded by square brackets. Here’s how I declare the array for the hero’s inventory:

**string inventory[MAX\_ITEMS];**

The preceding code declares an array inventory of **MAX\_ITEMS** string objects. (Because **MAX\_ITEMS** is **10**, that means **10** string objects.)

**Trick**

*You can initialize an array with values when you declare it by providing an initializer list - a*

*sequence of elements separated by commas and surrounded by curly braces. Here’s an example:*

**string inventory[MAX\_ITEMS] = {"sword", "armor", "shield"};**

*The preceding code declares an array of string objects, inventory, that has a size of* **MAX\_ITEMS***.*

*The first three elements of the array are initialized to "***sword***", "***armor***", and "***shield***".*

*If you omit the number of elements when using an initializer list, the array will be created with a*

*size equal to the number of elements in the list. Here’s an example:*

**string inventory[] = {"sword", "armor", "shield"};**

*Because there are three elements in the initializer list, the preceding line creates an array,*

**inventory***, that is three elements in size. Its elements are "***sword***", "***armor***", and "***shield***".*

**Indexing Arrays**

You index arrays much like you index string objects. You can access any individual element by providing an index number with the subscripting operator ( **[]** ).

Next, I add three items to the hero’s inventory using the subscripting operator:

**int numItems = 0;**

**inventory[numItems++] = "sword";**

**inventory[numItems++] = "armor";**

**inventory[numItems++] = "shield";**

I start by defining **numItems** for the number of items the hero is carrying at the moment. Next, I assign "**sword**" to position **0** of the array. Because I use the postfix increment operator, **numItems** is incremented after the assignment to the array. The next two lines add "**armor**" and "**shield**" to the array, leaving **numItems** at the correct value of **3** when the code finishes.

Now that the hero is stocked with some items, I display his inventory:

**cout << "Your items:\n";**

**for (int i = 0; i < numItems; ++i)**

**{**

**cout << inventory[i] << endl;**

**}**

This should remind you of string indexing. The code loops through the first three elements of **inventory**, displaying each string object in order.

Next, the hero trades his sword for a battle axe. I accomplish this through the following line:

**inventory[0] = "battle axe";**

The previous code reassigns the element at position **0** in inventory the string object "**battle** **axe**". Now the first three elements of inventory are "**battle** **axe**", "**armor**", and "**shield**".

**Trap**

*Array indexing begins at* **0***, just as you saw with string objects. This means that the following code*

*defines a five-element array.*

**int highScores[5];**

*Valid position numbers are* **0** *through* **4***, inclusive. There is no element* **highScores[5]** *! An attempt*

*to access* **highScores[5]** *could lead to disastrous results, including a program crash.*

**Accessing Member Functions of an Array Element**

You can access the member functions of an array element by writing the array element, followed by the member selection operator, followed by the member function name. This sounds a bit complicated, but it’s not. Here’s an example:

**cout << inventory[0].size() << " letters in it.\n";**

The code **inventory[0].size()** means the program should call the **size()** member function of the element **inventory[0]** . In this case, because **inventory[0]** is "**battle** **axe**", the call returns **10**, the number of characters in the string object.

**Being Aware of Array Bounds**

As you learned, you have to be careful when you index an array. Because an array has a ***fixed size***, you can create an integer constant to store the size of an array. Again, that’s just what I did in the beginning of the program:

**const int MAX\_ITEMS = 10;**

In the following lines, I use **MAX\_ITEMS** to protect myself before adding another item to the hero’s inventory:

**if (numItems < MAX\_ITEMS)**

**{**

**inventory[numItems++] = "healing potion";**

**}**

**else**

**{**

**cout << "You have too many items and can’t carry another.";**

**}**

In the preceding code, I first checked to see whether **numItems** is less than **MAX\_ITEMS**. If it is, then I can safely use **numItems** as an index and assign a new string object to the array. In this case, **numItems** is **3**, so I assign the string "**healing** **potion**" to array position **3**. If this hadn’t been the case, then I would have displayed the message, “**You have too many items and can’t carry another.**”

So what happens if you do attempt to access an array element outside the bounds of the array? It depends, because you’d be accessing some unknown part of the computer’s memory. At worst, if you attempt to assign some value to an element outside the bounds of an array, you could cause your program to do unpredictable things, and it might even crash.

Testing to make sure that an index number is a valid array position before using it is called bounds checking. It’s critical for you to perform bounds checking when there’s a chance that an index you want to use might not be valid.

**UNDERSTANDING C-STYLE STRINGS**

Before **string** objects came along, C++ programmers represented strings with ***arrays of characters*** terminated by a ***null character***. These arrays of characters are now called **C-style strings** because the practice began in C programs. You can declare and initialize a C-style string as you would any other array:

**char phrase[] = "Game Over!!!";**

C-style strings terminate with a character called the ***null character*** to signify their end. You can write the null character as ’**\0**’. I didn’t need to use the null character in the previous code because it is stored at the end of the string for me. So technically, phrase has **13** elements. (However, functions that work with C-style strings will say that phrase has a length of **12** , which makes sense and is in line with how string objects work.)

As with any other type of array, you can specify the array size when you define it. So another way to declare and initialize a C-style string is

**char phrase[81] = "Game Over!!!";**

The previous code creates a C-style string that can hold **80** printable characters (plus its terminating null character).

C-style strings don’t have member functions. But the **cstring** file, which is part of the standard library, contains a variety of functions for working with C-style strings.

A nice thing about **string** objects is that they’re designed to work seamlessly with C-style strings. For example, all of the following are completely valid uses of C-style strings with string objects:

**string word1 = "Game";**

**char word2[] = " Over";**

**string phrase = word1 + word2;**

**if (word1 != word2)**

**{**

**cout << "word1 and word2 are not equal.\n";**

**}**

**if (phrase.find(word2) != string::npos)**

**{**

**cout << "word2 is contained in phrase.\n";**

**}**

You can concatenate string objects and C-style strings, but the result is always a **string** object (so the code **char** **phrase2[] = word1 + word2;** would produce an ***error***). You can compare string objects and C-style strings using the relational operators. And you can even use C-style strings as arguments in **string** object member functions.

C-style strings have the same shortcomings as arrays. One of the biggest is that their lengths are ***fixed***. So the moral is: Use **string** objects whenever possible, but be prepared to work with C-style strings if necessary.

**USING MULTIDIMENSIONAL ARRAYS**

As you’ve seen, sequences are great for games. You can use them in the form of a string to store a player’s name, or you can use them in the form of any array to store a list of items in an RPG. But sometimes part of a game cries out for more than a linear list of things. Sometimes part of a game literally requires more dimension. For example, while you could represent a chessboard with a **64**-element array, it really is much more intuitive to work with it as a two-dimensional entity of **8 × 8** elements. Fortunately, you can create an array of two or three (or even more dimensions) to best fit your game’s needs.

**Introducing the Tic-Tac-Toe Board Program**

The Tic-Tac-Toe Board program displays a tic-tac-toe board. The program displays the board and declares X the winner. Although the program could have been written using a one-dimensional array, it uses a two-dimensional array to represent the board.

**MULTIDIMESIONAL ARRAYS**

**// Tic-Tac-Toe Board**

**// Demonstrates multidimensional arrays**

**#include <iostream>**

**using namespace std;**

**int main()**

**{**

**const int ROWS = 3;**

**const int COLUMNS = 3;**

**char board[ROWS][COLUMNS] = { {'O', 'X', 'O'},**

**{' ', 'X', 'X'},**

**{'X', 'O', 'O'} };**

**cout<< "Here's the tic-tac-toe board:\n";**

**for(int i = 0; i <ROWS; ++i)**

**{**

**for(int j = 0; j < COLUMNS; ++j)**

**{**

**cout<< board[i][j];**

**}**

**cout<< endl;**

**}**

**cout<< "\n'X' moves to the empty location.\n\n";**

**board[1][0] = 'X';**

**cout<< "Now the tic-tac-toe board is:\n";**

**for(int i = 0; i <ROWS; ++i)**

**{**

**for(int j = 0; j < COLUMNS; ++j)**

**{**

**cout<< board[i][j];**

**}**

**cout<< endl;**

**}**

**cout<< "\n'X' wins!";**

**return 0;**

**}**

**Here's the tic-tac-toe board:**

**OXO**

**XX**

**XOO**

**'X' moves to the empty location.**

**Now the tic-tac-toe board is:**

**OXO**

**XXX**

**XOO**

**'X' wins!**

**Creating Multidimensional Arrays**

One of the first things I do in the program is declare and initialize an array for the tictac-toe board.

**char board[ROWS][COLUMNS] = { {’O’, ’X’, ’O’},**

**{’ ’, ’X’, ’X’},**

**{’X’, ’O’, ’O’} };**

The preceding code declares a **3 × 3** (since **ROWS** and **COLUMNS** are both **3**) two-dimensional character array. It also initializes all of the elements.

**Hint**

*It’s possible to simply declare a multidimensional array without initializing it. Here’s an example:*

**char chessBoard[8][8];**

*The preceding code declares an* **8 × 8***, two-dimensional character array,* **chessBoard***. By the way, multidimensional arrays aren’t required to have the same size for each dimension. The following is a perfectly valid declaration for a game map represented by individual characters:*

**char map[12][20];**

**Indexing Multidimensional Arrays**

The next thing I do in the program is display the tic-tac-toe board. But before I get into the details of that, I want to explain how to index an individual array element. You index an individual element of a multidimensional array by supplying a value for each dimension of the array. That’s what I do to place an **X** in the array where a space was:

**board[1][0] = ’X’;**

The previous code assigns the character to the element at **board[1][0]** (which was **’ ’**). Then I display the tic-tac-toe board after the move the same way I displayed it before the move.

**for (int i = 0; i < ROWS; ++i)**

**{**

**for (int j = 0; j < COLUMNS; ++j)**

**{**

**cout << board[i][j];**

**}**

**cout << endl;**

**}**

By using a pair of nested for loops, I move through the two-dimensional array and display the character elements as I go, forming a tic-tac-toe board.

**INTRODUCING WORD JUMBLE**

**// Word Jumble**

**// The classic word jumble game where the player can ask for a hint**

**#include <iostream>**

**#include <string>**

**#include <cstdlib>**

**#include <ctime>**

**using namespace std;**

**int main()**

**{**

**enum fields {WORD, HINT, NUM\_FIELDS};**

**const int NUM\_WORDS = 5;**

**const string WORDS[NUM\_WORDS][NUM\_FIELDS]**

**{**

**{"wall", "Do you feel you're banging your head against something?"},**

**{"glasses", "These might help you see the answer."},**

**{"labored", "Going slowly, is it?"},**

**{"persistent", "Keep at it."},**

**{"jumble", "It's what the game is all about."},**

**};**

**srand(static\_cast<unsigned int>(time(0)));**

**int choice = (rand() % NUM\_WORDS);**

**string theWord = WORDS[choice][WORD]; // word to guess**

**string theHint = WORDS[choice][HINT]; // hint for word**

**string jumble = theWord; // jumbled version of word**

**int lenght = jumble.size();**

**for(int i = 0; i < lenght; ++i)**

**{**

**int index1 = (rand() % lenght);**

**int index2 = (rand() % lenght);**

**char temp = jumble[index1];**

**jumble[index1] = jumble[index2];**

**jumble[index2] = temp;**

**}**

**cout << "\t\tWelcome to Word Jumble!\n\n";**

**cout << "Unscramble the letters to make a word.\n";**

**cout << "Enter 'hint' for a hint.\n";**

**cout << "Enter 'quit' to quit the game.\n\n";**

**cout << "The jumble is: " << jumble;**

**string guess;**

**cout << "\n\nYour guess: ";**

**cin >> guess;**

**while((guess != theWord) && (guess != "quit"))**

**{**

**if(guess == "hint")**

**{**

**cout << theHint;**

**}**

**else**

**{**

**cout << "Sorry, that's not it.";**

**}**

**cout << "\n\nYour guess: ";**

**cin >> guess;**

**}**

**if(guess == theWord)**

**{**

**cout << "\nThat's it! You guessed it!\n";**

**}**

**cout << "\nThanks for playing.\n";**

**return 0;**

**}**

**Welcome to Word Jumble!**

**Unscramble the letters to make a word.**

**Enter 'hint' for a hint.**

**Enter 'quit' to quit the game.**

**The jumble is: tpesstnrei**

**Your guess: hint**

**Keep at it.**

**Your guess: persistent**

**That's it! You guessed it!**

**Thanks for playing.**

**Welcome to Word Jumble!**

**Unscramble the letters to make a word.**

**Enter 'hint' for a hint.**

**Enter 'quit' to quit the game.**

**The jumble is: uljbme**

**Your guess: jumleb**

**Sorry, that's not it.**

**Your guess: jumble**

**That's it! You guessed it!**

**Thanks for playing.**

**Picking a Word to Jumble**

First, I create a list of words and hints and I declare and initialize a two-dimensional array with words and corresponding hints. The enumeration defines enumerators for accessing the array. For example, **WORDS[x][WORD]** is always a string object that is one of the words, while **WORDS[x][HINT]** is the corresponding hint.

**Trick**

*You can list a final enumerator in an enumeration as a convenient way to store the number of*

*elements. Here’s an example:*

**enum difficulty {EASY, MEDIUM, HARD, NUM\_DIFF\_LEVELS};**

**cout << “There are “ << NUM\_DIFF\_LEVELS << “ difficulty levels.”;**

*In the previous code,* **NUM\_DIFF\_LEVELS** *is* **3***, the exact number of difficulty levels in the*

*enumeration. As a result, the second line of code displays the message, “***There are 3 difficulty**

**levels.***”*

Next, I pick a random word from my choices.

**srand(static\_cast<unsigned int>(time(0)));**

**int choice = (rand() % NUM\_WORDS);**

**string theWord = WORDS[choice][WORD]; //word to guess**

**string theHint = WORDS[choice][HINT]; //hint for word**

I generate a random index based on the number of words in the array. Then I assign both the random word at that index and its corresponding hint to the variables **theWord** and **theHint**.

**Jumbling the Word**

Now that I have the word for the player to guess, I need to create a jumbled version of it.

**string jumble = theWord; //jumbled version of word**

**int length = jumble.size();**

**for (int i = 0; i < length; ++i)**

**{**

**int index1 = (rand() % length);**

**int index2 = (rand() % length);**

**char temp = jumble[index1];**

**jumble[index1] = jumble[index2];**

**jumble[index2] = temp;**

**}**

In the preceding code, I created a copy of the word jumble to...well, jumble. I generated two random positions in the string object and swapped the characters at those positions. I did this a number of times equal to the length of the word.

**Welcoming the Player**

Now it’s time to welcome the player, which is what I do next.

**cout << "\t\t\tWelcome to Word Jumble!\n\n";**

**cout << "Unscramble the letters to make a word.\n";**

**cout << "Enter ’hint’ for a hint.\n";**

**cout << "Enter ’quit’ to quit the game.\n\n";**

**cout << "The jumble is: " << jumble;**

**string guess;**

**cout << "\n\nYour guess: ";**

**cin >> guess;**

I gave the player instructions on how to play, including how to quit and how to ask for a hint.

**Hint**

*As enthralling as you think your game is, you should always provide a way for the player to exit it.*

**Entering the Game Loop**

Next, I enter the game loop.

**while ((guess != theWord) && (guess != "quit"))**

**{**

**if (guess == "hint")**

**{**

**cout << theHint;**

**}**

**else**

**{**

**cout << "Sorry, that’s not it.";**

**}**

**cout <<"\n\nYour guess: ";**

**cin >> guess;**

**}**

The loop continues to ask the player for a guess until the player either guesses the word or asks to quit.

**Saying Goodbye**

When the loop ends, the player has either won or quit, so it’s time to say goodbye.

**if (guess == theWord)**

**{**

**cout << "\nThat’s it! You guessed it!\n";**

**}**

**cout << "\nThanks for playing.\n";**

**return 0;**

**}**

If the player has guessed the word, I congratulate him or her. Finally, I thank the player for playing.

**SUMMARY**

In this chapter, you learned the following concepts:

* The **for** loop lets you repeat a section of code. In a **for** loop, you can provide an initialization statement, an expression to test, and an action to take after each loop iteration.
* **for** loops are often used for counting or looping through a sequence.
* Objects are encapsulated, cohesive entities that combine data (called ***data members***) and functions (called ***member functions***).
* **string** objects (often just called ***strings***) are defined in the file string, which is part of the standard library. **string** objects allow you to store a sequence of characters and also have member functions.
* **string** objects are defined so that they work intuitively with familiar operators, such as the concatenation operator and the relational operators.
* All **string** objects have member functions, including those for determining a **string** object’s length, determining whether a **string** object is empty, finding substrings, and removing substrings.
* Arrays provide a way to store and access sequences of any type.
* A limitation of arrays is that they have a fixed length.
* You can access individual elements of **string** objects and arrays through the subscripting operator.
* Bounds checking is not enforced when attempts are made to access individual elements of **string** objects or arrays. Therefore, bounds checking is up to the programmer.
* C-style strings are character arrays terminated with the null character. They are the standard way to represent strings in the C language. And even though C-style **strings** are perfectly legal in C++, string objects are the preferred way to work with sequences of characters.
* Multidimensional arrays allow for access to array elements using multiple subscripts. For example, a chessboard can be represented as a two-dimensional array, **8 × 8** elements.

**CHAPTER 4**

**THE STANDARD TEMPLATE LIBRARY: HANGMAN**

**INTRODUCING THE STANDARD TEMPLATE LIBRARY**

Good game programmers are lazy. It’s not that they don’t want to work; it’s just that they don’t want to redo work that’s already been done - especially if it has been done well. The **STL** (***Standard Template Library***) represents a powerful collection of programming work that’s been done well. It provides a group of ***containers***, ***algorithms***, and ***iterators***, among other things.

So what’s a **container** and how can it help you write games? Well, containers let you store and access collections of values of the same type. Yes, arrays let you do the same thing, but the STL containers offer more flexibility and power than a simple but trusty array. The STL defines a variety of container types; each works in a different way to meet different needs.

The **algorithms** defined in the STL work with its containers. The algorithms are common functions that game programmers find themselves repeatedly applying to groups of values. They include algorithms for ***sorting***, ***searching***, ***copying***, ***merging***, ***inserting***, and ***removing*** container elements. The cool thing is that the same algorithm can work its magic on many different container types.

**Iterators** are objects that identify elements in containers and can be manipulated to move among elements. They’re great for, well, iterating through containers. In addition, iterators are required by the STL algorithms.

All of this makes a lot more sense when you see an actual implementation of one of the container types, so that’s up next.

**USING VECTORS**

The vector class defines one kind of container provided by the STL. It meets the general description of a dynamic array - an array that can grow and shrink in size as needed. In addition, vector defines member functions to manipulate vector elements. This means that the vector has all of the functionality of the array plus more.

At this point, you may be thinking to yourself: Why learn to use these fancy new vectors when I can already use arrays? Well, vectors have certain advantages over arrays, including:

* Vectors can grow as needed while arrays cannot. This means that if you use a vector to store objects for enemies in a game, the vector will grow to accommodate the number of enemies that are created. If you use an array, you have to create one that can store some maximum number of enemies. And if, during play, you need more room in the array than you thought, you’re out of luck.
* Vectors can be used with the STL algorithms while arrays cannot. This means that by using vectors you get complex functionality like searching and sorting, built-in. If you use arrays, you have to write your own code to achieve this same functionality.

There are a few disadvantages to vectors when compared to arrays, including:

* Vectors require a bit of extra memory as overhead.
* There can be a performance cost when a vector grows in size.,
* Vectors may not be available on some game console systems.

Overall, vectors (and the STL) can be a welcome tool in most any project.

**VECTORS**

**// Hero's Inventory 2.0**

**// Demonstrates vectors**

**#include <iostream>**

**#include <string>**

**#include <vector>**

**using namespace std;**

**int main()**

**{**

**vector<string> inventory;**

**inventory.push\_back("sword");**

**inventory.push\_back("armor");**

**inventory.push\_back("shield");**

**cout << "You have " << inventory.size() << " items.\n";**

**cout << "\nYour items:\n";**

**for(unsigned int i = 0; i < inventory.size(); ++i)**

**{**

**cout << inventory[i] << endl;**

**}**

**cout << "\nThe item name '" << inventory[0] << "' has ";**

**cout << inventory[0].size() << " letters in it.\n";**

**cout << "\nYour shield is destroyed in a fierce battle.";**

**inventory.pop\_back();**

**cout << "\nYour items:\n";**

**for(unsigned int i = 0; i < inventory.size(); ++i)**

**{**

**cout << inventory[i] << endl;**

**}**

**cout << "\nYou were robbed of all your possessions by a thief.";**

**inventory.clear();**

**if(inventory.empty())**

**{**

**cout << "\nYou have nothing.\n";**

**}**

**else**

**{**

**cout << "\nYou have at least one item.\n";**

**}**

**return 0;**

**}**

**You have 3 items.**

**Your items:**

**sword**

**armor**

**shield**

**The item name 'sword' has 5 letters in it.**

**Your shield is destroyed in a fierce battle.**

**Your items:**

**sword**

**armor**

**You were robbed of all your possessions by a thief.**

**You have nothing.**

**Preparing to Use Vectors**

Before I can declare a vector, I have to include the file that contains its definition:

**#include <vector>**

All **STL** components live in the **std** namespace, so by using the following code (as I typically do) I can refer to vector without having to precede it with **std::** .

**using namespace std;**

**Declaring a Vector**

Okay, the first thing I do in **main()** is declare a new vector.

**vector<string> inventory;**

The preceding line declared an empty vector named **inventory**, which can contain **string** object elements. Declaring an empty vector is fine because it grows in size when you add new elements.

To declare a vector of your own, write vector followed by the type of objects you want to use with the **vector** (surrounded by the **<** and **>** symbols), followed by the ***vector name***.

**Hint**

*There are additional ways to declare a vector. You can declare one with a starting size by specifying*

*a number in parentheses after the vector name.*

**vector<string> inventory(10);**

*The preceding code declared a vector to hold* **string** *object elements with a starting size of* **10***.*

*You can also initialize all of a vector’s elements to the same value when you declare it. You simply*

*supply the number of elements followed by the starting value, as in:*

**vector<string> inventory(10, "nothing");**

*The preceding code declared a vector with a size of* **10** *and initialized all* **10** *elements to “***nothing***”.*

*Finally, you can declare a vector and initialize it with the contents of another vector.*

**vector<string> inventory(myStuff);**

*The preceding code created a new vector with the same contents as the vector* **myStuff***.*

**Using the push\_back() Member Function**

Next, I give the hero the same three starting items as in the previous version of the program.

**inventory.push\_back("sword");**

**inventory.push\_back("armor");**

**inventory.push\_back("shield");**

The **push\_back()** member function adds a new element to the end of a vector. In the preceding lines, I added “**sword**”, “**armor**”, and “**shield**” to **inventory**. As a result, **inventory[0]** is equal to “**sword**”, **inventory[1]** is equal to “**armor**”, and **inventory[2]** is equal to “**shield**”.

**Using the size() Member Function**

Next, I display the number of items the hero has in his possession.

**cout << "You have " << inventory.size() << " items.\n";**

I get the size of inventory by calling the **size()** member function with **inventory.size()**. The **size()** member function simply returns the size of a vector. In this case, it returns **3**.

**Indexing Vectors**

Next, I display all of the hero’s items.

**cout << "\nYour items:\n";**

**for (unsigned int i = 0; i < inventory.size(); ++i)**

**{**

**cout << inventory[i] << endl;**

**}**

Just as with arrays, you can index vectors by using the subscripting operator. In fact, the preceding code is nearly identical to the same section of code from the original Hero’s Inventory program. The only difference is that I used **inventory.size()** to specify when the loop should end. Note that I made the loop variable **i** an **unsigned** **int** because the value returned by **size()** is an ***unsigned integer*** type.

Next, I replace the hero’s first item.

**inventory[0] = "battle axe";**

Again, just as with arrays, I use the subscripting operator to assign a new value to an existing element position.

**Trap**

*Although vectors are* **dynamic***, you can’t increase a vector’s size by applying the subscripting*

*operator. For example, the following highly dangerous code snippet does not increase the size of*

*the vector* **inventory***:*

**vector<string> inventory; // creating an empty vector**

**inventory[0] = "sword"; // may cause your program to crash!**

*Just as with arrays, you can attempt to access a nonexistent element position - but with potentially*

*disastrous results. The preceding code changed some unknown section of your computer’s memory*

*and could cause your program to crash. To add a new element at the end of a vector, use the*

**push\_back()** *member function.*

**Calling Member Functions of an Element**

Next, I show the number of letters in the name of the first item in the hero’s inventory.

**cout << inventory[0].size() << " letters in it.\n";**

Just as with arrays, you can access the member functions of a vector element by writing the element, followed by the member selection operator, followed by the member function name. Because **inventory[0]** is equal to “ **battle axe** ”, **inventory[0].size()** returns **10**.

**Using the pop\_back() Member Function**

I remove the hero’s shield using

**inventory.pop\_back();**

The **pop\_back()** member function removes the last element of a vector and reduces the vector size by one. In this case, **inventory.pop\_back()** removes “**shield**” from inventory because that was the last element in the vector. Also, the size of inventory is reduced from **3** to **2**.

**Using the clear() Member Function**

Next, I simulate the act of a thief robbing the hero of all of his items.

**inventory.clear();**

The **clear()** member function removes all of the items of a vector and sets its size to **0** . After the previous line of code executes, inventory is an empty vector.

**Using the empty() Member Function**

Finally, I check to see whether the hero has any items in his inventory.

**if (inventory.empty())**

**{**

**cout << "\nYou have nothing.\n";**

**}**

**else**

**{**

**cout << "\nYou have at least one item.\n";**

**}**

The vector member function **empty()** works just like the string member function **empty()**. It returns **true** if the vector object is empty; otherwise, it returns **false**. Because inventory is empty in this case, the program displays the message, “**You have nothing.**”

**USING ITERATORS**

Iterators are the key to using containers to their fullest potential. With iterators you can, well, iterate through a sequence container. In addition, important parts of the STL require iterators. Many container member functions and STL algorithms take iterators as arguments. So if you want to reap the benefits of these member functions and algorithms, you must use iterators.

**Introducing the Hero’s Inventory 3.0 Program**

The Hero’s Inventory 3.0 program acts like its two predecessors, at least at the start. The program shows off a list of items, replaces the first item, and displays the number of letters in the name of an item. But then the program does something new: It inserts an item at the beginning of the group, and then it removes an item from the middle of the group

**ITERATORS**

**// Hero's Inventory 3.0**

**// Demonstrates iterators**

**#include <iostream>**

**#include <string>**

**#include <vector>**

**using namespace std;**

**int main()**

**{**

**vector<string> inventory;**

**inventory.push\_back("sword");**

**inventory.push\_back("armor");**

**inventory.push\_back("shield");**

**vector<string>::iterator myIterator;**

**vector<string>::const\_iterator iter;**

**cout << "\nYour items:\n";**

**for(iter = inventory.begin(); iter != inventory.end(); ++iter)**

**{**

**cout << \*iter << endl;**

**}**

**cout << "\nYou trade your sword for a battle axe.";**

**myIterator = inventory.begin();**

**\*myIterator = "battle axe";**

**cout << "\nYour items:\n";**

**for(iter = inventory.begin(); iter != inventory.end(); ++iter)**

**{**

**cout << \*iter << endl;**

**}**

**cout << "\nThe item name '" << \*myIterator << "' has ";**

**cout << (\*myIterator).size() << " letters in it.\n";**

**cout << "\nThe item name '" << \*myIterator << "' has ";**

**cout << myIterator->size() << " letters in it.\n";**

**cout << "\nYou recover a crossbow from a slain enemy.";**

**inventory.insert(inventory.begin(), "crossbow");**

**cout << "\nYour items:\n";**

**for(iter = inventory.begin(); iter != inventory.end(); ++iter)**

**{**

**cout << \*iter << endl;**

**}**

**cout << "\nYour armor is destroyed in a fierce battle.";**

**inventory.erase((inventory.begin() + 2));**

**cout << "\nYour items:\n";**

**for(iter = inventory.begin(); iter != inventory.end(); ++iter)**

**{**

**cout << \*iter << endl;**

**}**

**return 0;**

**}**

**Your items:**

**sword**

**armor**

**shield**

**You trade your sword for a battle axe.**

**Your items:**

**battle axe**

**armor**

**shield**

**The item name 'battle axe' has 10 letters in it.**

**The item name 'battle axe' has 10 letters in it.**

**You recover a crossbow from a slain enemy.**

**Your items:**

**crossbow**

**battle axe**

**armor**

**shield**

**Your armor is destroyed in a fierce battle.**

**Your items:**

**crossbow**

**battle axe**

**shield**

**Declaring Iterators**

After I declare a vector for the hero’s inventory and add the same three string objects from the previous incarnations of the program, I declare an iterator.

**vector<string>::iterator myIterator;**

The preceding line declares an iterator named **myIterator** for a vector that contains **string** objects. To declare an iterator of your own, follow the same pattern. Write the container type, followed by the ***type*** of objects the container will hold (surrounded by the **<** and **>** symbols), followed by the ***scope resolution operator*** (the **::** symbol), followed by ***iterator*** , followed by a ***name*** for your new iterator.

So what are iterators? Iterators are values that identify a particular element in a container. Given an iterator, you can access the value of the element. Given the right kind of iterator, you can change the value. Iterators can also move among elements via familiar arithmetic operators.

A way to think about iterators is to imagine them as Post-it notes that you can stick on a specific element in a container. An iterator is not one of the elements, but a way to refer to one. Specifically, I can use **myIterator** to refer to a particular element of the vector **inventory**. That is, I can stick the **myIterator** Post-it note on a specific element in **inventory**. Once I’ve done that, I can access the element or even change it through the iterator.

Next, I declare another iterator.

**vector<string>::const\_iterator iter;**

The preceding line of code creates a ***constant*** iterator named **iter** for a **vector** that contains **string** objects. A **constant** iterator is just like a regular iterator except that you can’t use it to change the element to which it refers; the element must remain constant. You can think of a constant iterator as providing ***read-only*** access. However, the iterator itself can change. This means you can move **iter** all around the vector inventory as you see fit. You can’t, however, change the value of any of the elements through **iter**. With a constant iterator the Post-It can change, but the thing it’s stuck to can’t.

Why would you want to use a constant iterator if it’s a limited version of a regular iterator? First, it makes your intentions clearer. When you use a constant iterator, it’s clear that you won’t be changing any element to which it refers. Second, it’s safer. You can use a constant iterator to avoid accidentally changing a container element. (If you attempt to change an element through a constant iterator, you’ll generate a compile error.)

**Trap**

*Using* **push\_back()** *might invalidate all iterators referencing the vector.*

Is all of this iterator talk a little too abstract for you? Are you tired of analogies about Post-it notes? Fear not - next, I put an actual iterator to work.

**Looping through a Vector**

Next, I loop through the contents of the vector and display the hero’s inventory.

**cout << "Your items:\n";**

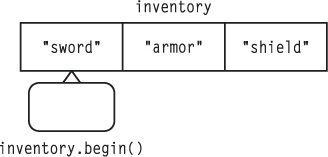
**for (iter = inventory.begin(); iter != inventory.end(); ++iter)**

**cout << \*iter << endl;**

In the preceding code, I use a for loop to move from the first to the last element of **inventory**. At this general level, this is exactly how I looped through the contents of the vector in Hero’s Inventory 2.0. But instead of using an integer and the subscripting operator to access each element, I used an iterator. Basically, I moved the Post-it note through the entire sequence of elements and displayed the value of each element to which the note was stuck. There are a lot of new ideas in this little loop, so I’ll tackle them one at a time.

**Calling the begin() Vector Member Function**

In the initialization statement of the loop, I assign the return value of **inventory.begin()** to **iter**. The **begin()** member function returns an iterator that refers to a container’s first element. So in this case, the statement assigns an iterator that refers to the first element of inventory (the **string** object equal to “**sword**”) to **iter** . Figure 4.3 shows an abstract view of the iterator returned by a call to **inventory.begin()** . (Note that the figure is abstract because the vector **inventory** doesn’t contain the string literals “**sword**”, “**armor**”, and “**shield**”; it contains **string** objects.)



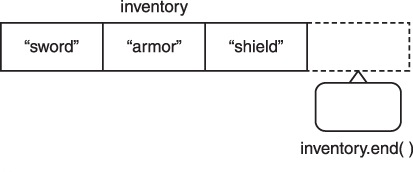
***Figure 4.3***

*A call to* **inventory.begin()** *returns an iterator that refers to the first element in the vector.*

**Calling the end() Vector Member Function**

In the test statement of the loop, I test the return value of **inventory.end()** against **iter** to make sure the two are not equal. The **end()** member function returns an **iterator** one past the last element in a container. This means the loop will continue until **iter** has moved through all of the elements in **inventory**. Figure 4.4 shows an abstract view of the iterator returned by a call to this member function. (Note that the figure is abstract because the vector inventory doesn’t contain the string literals “**sword**”, “**armor**”, **and** “**shield**”;

it contains **string** objects.)



***Figure 4.4***

*A call to* **inventory.end()** *returns an iterator one past the last element of the vector.*

**Trap**

*The* **end()** *vector member function returns an* **iterator** *that’s* ***one past the last element*** *in the*

*vector -* ***not the last element****. Therefore, you can’t get a value from the iterator returned by* **end()***.*

*This might seem counterintuitive, but it works well for loops that move through a container.*

**Altering an Iterator**

The action statement in the loop, **++iter**, increments **iter**, which moves it to the next element in the vector. Depending upon the **iterator**, you can perform other mathematical operations on iterators to move them around a container. Most often, though, you’ll find that you simply want to increment an iterator.

**Dereferencing an Iterator**

In the loop body, I send **\*iter** to **cout**. By placing the dereference operator ( **\*** ) in front of **iter**, I display the value of the element to which the iterator refers (not the iterator itself). By placing the dereference operator in front of an iterator, you’re saying, “Treat this as the thing that the iterator references, not as the iterator itself.”

**Changing the Value of a Vector Element**

Next, I change the first element in the vector from the **string** object equal to “**sword**” to the string object equal to “**battle** **axe**”. First, I set **myIterator** to reference the first element of **inventory**.

**myIterator = inventory.begin();**

Then I change the value of the first element.

**\*myIterator = "battle axe";**

Remember, by dereferencing **myIterator** with **\***, the preceding assignment statement says, “Assign “**battle** **axe**” to the element that **myIterator** references.” It does not change **myIterator**. After the assignment statement, **myIterator** still refers to the first element in the vector. Just to prove that the assignment worked, I then display all of the elements in **inventory**.

**Accessing Member Functions of a Vector Element**

Next, I display the number of characters in the name of the first item in the hero’s inventory.

**cout << "\nThe item name ’" << \*myIterator << "’ has ";**

**cout << (\*myIterator).size() << " letters in it.\n";**

The code **(\*myIterator).size()** says, “Take the result of dereferencing **myIterator** and call that object’s **size()** member function.” Because **myIterator** refers to the **string** object equal to “**battle** **axe**”, the code returns **10**.

**Hint**

*Whenever you dereference an iterator to access a data member or member function, surround the*

*dereferenced iterator by a pair of parentheses. This ensures that the dot operator will be applied*

*to the object the iterator references.*

The code **(\*myIterator).size()** is not the prettiest, so C++ offers an alternative, more intuitive way to express the same thing, which I demonstrate in the next two lines of the program.

**cout << "\nThe item name ’" << \*myIterator << "’ has ";**

**cout << myIterator->size() << " letters in it.\n";**

The preceding code does exactly the same thing the first pair of lines I presented in this section do; it displays the number of characters in “**battle axe**”. However, notice that I substitute **myIterator->size()** for **(\*myIterator).size()** . You can see that this version (with the **->** symbol) is more readable. The two pieces of code mean exactly the same thing to the computer, but this new version is easier for humans to use. In general, you can use the indirect member selection operator, **->**, to access the member functions or data members of an object that an iterator references.

**Hint**

*Syntactic sugar is a nicer, alternative syntax. It replaces harsh syntax with something that’s a bit*

*easier to swallow. As an example, instead of writing the code* **(\*myIterator).size()** *, I can use*

*the syntactic sugar provided by the* **->** *operator and write* **myIterator->size()***.*

**Using the insert() Vector Member Function**

Next, I add a new item to the hero’s inventory. This time, though, I don’t add the item to the end of the sequence; instead, I insert it at the beginning.

**inventory.insert(inventory.begin(), "crossbow");**

One form of the **insert()** member function inserts a new element into a vector just before the element referred to by a given iterator. You supply two arguments to this version of **insert()** - the first is an iterator, and the second is the element to be inserted. In this case, I inserted “**crossbow**” into inventory just before the first element. As a result, all of the other elements will move down by one. This version of the **insert()** member function returns an iterator that references the newly inserted element. In this case, I don’t assign the returned iterator to a variable.

**Trap**

*Calling the* **insert()** *member function on a vector invalidates all of the iterators that reference*

*elements after the insertion point because all of the elements after the insertion point are shifted*

*down by one.*

Next, I show the contents of the vector to prove the insertion worked.

**Using the erase() Vector Member Function**

Next, I remove an item from the hero’s inventory. However, this time I don’t remove the item at the end of the sequence; instead, I remove one from the middle.

**inventory.erase((inventory.begin() + 2));**

One form of the **erase()** member function removes an element from a vector. You supply one argument to this version of **erase()** - the iterator that references the element you want to remove. In this case, I passed **(inventory.begin() + 2)** , which is equal to the iterator that references the third element in inventory . This removes the **string** object equal to “**armor**”.

As a result, all of the following elements will move up by one. This version of the **erase()** member function returns an iterator that references the element after the element that was removed. In this case, I don’t assign the returned iterator to a variable.

**Trap**

*Calling the* **erase()** *member function on a vector invalidates all of the iterators that reference*

*elements after the removal point because all of the elements after the removal point are shifted up*

*by one.*

Next, I show the contents of the vector to prove the removal worked.

**USING ALGORITHMS**

The **STL** defines a group of algorithms that allow you to manipulate elements in containers through iterators. Algorithms exist for common tasks such as ***searching***, ***randomizing***, and ***sorting***. These algorithms are your built-in arsenal of flexible and efficient weapons. By using them, you can leave the mundane task of manipulating container elements in common ways to the STL so you can concentrate on writing your game. The powerful thing about these algorithms is that they are generic - the same algorithm can work with elements of different container types.

**Introducing the High Scores Program**

The High Scores program creates a vector of high scores. It uses **STL** algorithms to search, shuffle, and sort the scores.

**// High Scores**

**// Demonstrates algorithms**

**#include <iostream>**

**#include <vector>**

**#include <algorithm>**

**#include <ctime>**

**#include <cstdlib>**

**using namespace std;**

**int main()**

**{**

**vector<int>::const\_iterator iter;**

**cout << "Creating a list of scores.";**

**vector<int> scores;**

**scores.push\_back(1500);**

**scores.push\_back(3500);**

**scores.push\_back(7500);**

**cout <<"\nHigh Scores:\n";**

**for(iter = scores.begin(); iter != scores.end(); ++iter)**

**{**

**cout << \*iter << endl;**

**}**

**cout << "\nFinding a score.";**

**int score;**

**cout << "\nEnter a score to find: ";**

**cin >> score;**

**iter = find(scores.begin(), scores.end(), score);**

**if(iter != scores.end())**

**{**

**cout << "Score found.\n";**

**}**

**else**

**{**

**cout << "Score not found.\n";**

**}**

**cout << "\nRandomizing scores.";**

**srand(static\_cast<unsigned int>(time(0)));**

**random\_shuffle(scores.begin(), scores.end());**

**cout << "\nHigh Scores:\n";**

**for(iter = scores.begin(); iter != scores.end(); ++iter)**

**{**

**cout << \*iter << endl;**

**}**

**cout << "\nSorting scores.";**

**sort(scores.begin(), scores.end());**

**cout << "\nHigh Scores:\n";**

**for(iter = scores.begin(); iter != scores.end(); ++iter)**

**{**

**cout << \*iter << endl;**

**}**

**return 0;**

**}**

**Creating a list of scores.**

**High Scores:**

**1500**

**3500**

**7500**

**Finding a score.**

**Enter a score to find: 3500**

**Score found.**

**Randomizing scores.**

**High Scores:**

**3500**

**1500**

**7500**

**Sorting scores.**

**High Scores:**

**1500**

**3500**

**7500**

**Preparing to Use Algorithms**

In order to use the STL algorithms, I include the file with their definitions.

**#include <algorithm>**

As you know, all **STL** components live in the **std** namespace. By using the following code (as I typically do), I can refer to algorithms without having to precede them with **std::**

**using namespace std;**

**Using the find() Algorithm**

After I display the contents of the vector scores, I get a value from the user to find and store it in the variable **score**. Then I use the **find()** algorithm to search the vector for the value:

**iter = find(scores.begin(), scores.end(), score);**

The **find()** STL algorithm searches a specified range of a container’s elements for a value. It returns an iterator that references the first matching element. If no match is found, it returns an iterator to the end of the range.

You must pass the starting point as an iterator, the ending point as an iterator, and a value to find. The algorithm searches from the starting iterator up to but not including the ending iterator. In this case, I passed **scores.begin()** and **scores.end()** as the first and second arguments to search the entire vector. I passed score as the third argument to search for the value the user entered.

Next, I check to see if the value score was found:

**if (iter != scores.end())**

**{**

**cout << "Score found.\n";**

**}**

**else**

**{**

**cout << "Score not found.\n";**

**}**

Remember, **iter** will reference the first occurrence of **score** in the vector, if the value was found. So, as long as iter is not equal to **scores.end()**, I know that **score** was found and I display a message saying so. Otherwise, iter will be equal to **scores.end()** and I know **score** was not found.

**Using the random\_shuffle( ) Algorithm**

Next, I prepare to randomize the scores using the **random\_shuffle()** algorithm. Just as when I generate a single random number, I seed the random number generator before I call **random\_shuffle()**, so the order of the scores might be different each time I run the program.

**srand(static\_cast<unsigned int>(time(0)));**

Then I reorder the scores in a random way.

**random\_shuffle(scores.begin(), scores.end());**

The **random\_shuffle()** algorithm randomizes the elements of a sequence. You must supply as iterators the starting and ending points of the sequence to shuffle. In this case, I passed the iterators returned by **scores.begin()** and **scores.end()**. These two iterators indicate that I want to shuffle all of the elements in scores. As a result, scores contains the same scores, but in some random order.

Then I display the scores to prove the randomization worked.

**Trick**

*Although you might not want to randomize a list of high scores,* **random\_shuffle()***is a valuable algorithm for games. You can use it for everything from shuffling a deck of cards to mixing up the order of the enemies a player will encounter in a game level.*

**Using the sort( ) Algorithm**

Next, I sort the **scores**.

**sort(scores.begin(), scores.end());**

The **sort()** algorithm sorts the elements of a sequence in ascending order. You must supply as iterators the starting and ending points of the sequence to sort. In this particular case, I passed the iterators returned by **scores.begin()** and **scores.end()**. These two iterators indicate that I want to sort all of the elements in scores. As a result, scores contains all of the scores in ascending order.

Finally, I display the **scores** to prove the sorting worked.

**Trick**

*A very cool property of STL algorithms is that they can work with containers defined outside of the STL. These containers only have to meet certain requirements. For example, even though string objects are not part of the STL, you can use appropriate STL algorithms on them. The following code snippet demonstrates this:*

**string word = "High Scores";**

**random\_shuffle(word.begin(), word.end());**

*The preceding code randomly shuffles the characters in word. As you can see, string objects have both* **begin()** *and* **end()** *member functions, which return iterators to the first character and one past the last character, respectively. That’s part of the reason why STL algorithms work with strings—because they’re designed to.*

**UNDERSTANDING VECTOR PERFORMANCE**

Like all STL containers, vectors provide game programmers with sophisticated ways to work with information, but this level of sophistication can come at a performance cost. And if there’s one thing game programmers obsess about, it’s performance. But fear not, vectors and other STL containers are incredibly efficient. In fact, they’ve already been used in published PC and console games. However, these containers have their strengths and weaknesses; a game programmer needs to understand the performance characteristics of the various container types so that he can choose the right one for the job.

**Examining Vector Growth**

Although vectors grow dynamically as needed, every vector has a specific size. When a new element added to a vector pushes the vector beyond its current size, the computer reallocates memory and might even copy all of the vector elements to this newly seized chunk of memory real estate. This can cause a performance hit.

The most important thing to keep in mind about program performance is whether you need to care. For example, vector memory reallocation might not occur at a performance-critical part of your program. In that case, you can safely ignore the cost of reallocation. Also, with small vectors, the reallocation cost might be insignificant so, again, you can safely ignore it. However, if you need greater control over when these memory reallocations occur, you have it.

**Using the capacity() Member Function**

The **capacity()** **vector** member function returns the capacity of a vector - in other words, the number of elements that a vector can hold before a program must reallocate more memory for it. A vector’s capacity is not the same thing as its size (the number of elements a vector currently holds). Here’s a code snippet to help drive this point home:

**cout << “Creating a 10 element vector to hold scores.\n”;**

**vector<int> scores(10, 0); //** initialize all 10 elements to 0

**cout << “Vector size is :” << scores.size() << endl;**

**cout << “Vector capacity is:” << scores.capacity() << endl;**

**cout << “Adding a score.\n”;**

**scores.push\_back(0); //** memory is reallocated to accommodate growth

**cout << “Vector size is :” << scores.size() << endl;**

**cout << “Vector capacity is:” << scores.capacity() << endl;**

**Creating a 10 element vector to hold scores.**

**Vector size is :10**

**Vector capacity is:10**

**Adding a score.**

**Vector size is :11**

**Vector capacity is:20**

Right after I declare and initialize the vector, this code reports that its size and capacity are both **10**. However, after an element is added, the code reports that the vector’s size is **11** while its capacity is **20**. That’s because the capacity of a vector doubles every time a program reallocates additional memory for it. In this case, when a new score was added, memory was reallocated, and the capacity of the vector doubled from **10** to **20**.

**Using the reserve() Member Function**

The **reserve()** member function increases the capacity of a vector to the number supplied as an argument. Using **reserve()** gives you control over when a reallocation of additional memory occurs. Here’s an example:

**cout << "Creating a list of scores.\n";**

**vector<int> scores(10, 0); //** initialize all 10 elements to 0

**cout << "vector size is :" << scores.size() << endl;**

**cout << "Vector capacity is:" << scores.capacity() << endl;**

**cout << "Reserving more memory.\n";**

**scores.reserve(20); //** reserve memory for 10 additional elements

**cout << "Vector size is :" << scores.size() << endl;**

**cout << "Vector capacity is:" << scores.capacity() << endl;**

**Creating a list of scores.**

**vector size is :10**

**Vector capacity is:10**

**Reserving more memory.**

**Vector size is :10**

**Vector capacity is:20**

Right after I declare and initialize the vector, this code reports that its size and capacity are both **10**. However, after I reserve memory for **10** additional elements, the code reports that the vector’s size is still **10** while its capacity is **20**.

By using **reserve()** to keep a vector’s capacity large enough for your purposes, you can delay memory reallocation to a time of your choosing.

**Hint**

*As a beginning game programmer, it’s good to be aware of how vector memory allocation works; however, don’t obsess over it. The first game programs you’ll write probably won’t benefit from a more manual process of vector memory allocation.*

**Examining Element Insertion and Deletion**

Adding or removing an element from the end of a vector using the **push\_back()** or **pop\_back()** member functions is extremely efficient. However, adding or removing an element at any other point in a vector (for

example, using **insert()** or **erase()**) can require more work because you might have to move multiple elements to accommodate the insertion or deletion. With small vectors the overhead is usually insignificant, but with larger vectors (with, say, thousands of elements), inserting or erasing elements from the middle of a vector can cause a performance hit.

Fortunately, the STL offers another sequence container type, **list**, which allows for efficient insertion and deletion regardless of the sequence size. The important thing to remember is that one container type isn’t the solution for every problem. Although vector is versatile and perhaps the most popular STL container type, there are times when another container type might make more sense.

**Trap**

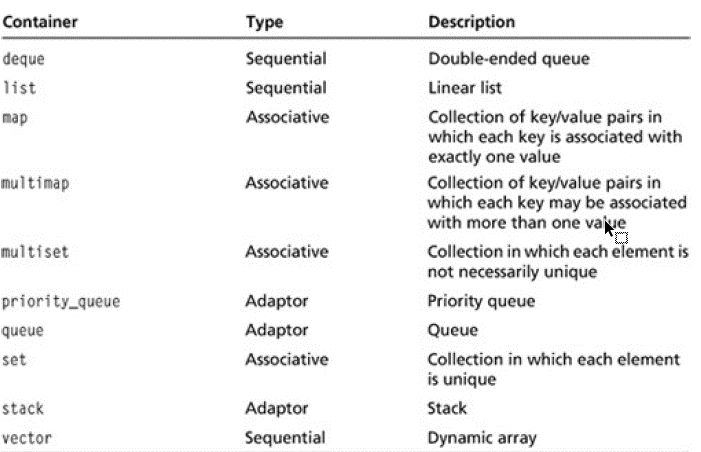
*Just because you want to insert or delete elements from the middle of a sequence, that doesn’t mean you should abandon the vector. It might still be a good choice for your game program. It really depends on how you use the sequence. If your sequence is small or there are only a few insertions and deletions, then a vector might still be your best bet.*

**EXAMINING OTHER STL CONTAINERS**

The STL defines a variety of container types that fall into two basic categories: sequential and associative. With a sequential container, you can retrieve values in sequence, while an associative container lets you retrieve values based on keys. **vector** is an example of a sequential container.

How might you use these different container types? Consider an online, turnbased strategy game. You could use a sequential container to store a group of players that you want to cycle through in, well, sequence. On the other hand, you could use an associative container to retrieve player information in a random-access fashion by looking up a unique identifier, such as a player’s IP address.

Finally, the STL defines container adaptors that adapt one of the sequence containers. ***Container adaptors*** represent standard computer science data structures. Although they are not official containers, they look and feel just like them. Table 4.1 lists the container types offered by the STL.



**PLANNING YOUR PROGRAMS**

So far, all the programs you’ve seen have been pretty simple. The idea of formally planning any of them on paper probably seems like overkill. It’s not. Planning your programs (even the small ones) will almost always result in time (and frustration) saved.

Programming is a lot like construction. Imagine a contractor building a house for you without a blueprint. Yikes! You might end up with a house that has 12 bathrooms, no windows, and a front door on the second floor. Plus, it probably would cost you 10 times the estimated price. Programming is the same way. Without a plan, you’ll likely struggle through the process and waste time. You might even end up with a program that doesn’t quite work.

**Using Pseudocode**

Many programmers sketch out their programs using pseudocode - a language that falls somewhere between English and a formal programming language. Anyone who understands English should be able to follow pseudocode. Here’s an example: Suppose I want to make a million dollars. A worthy goal, but what do I do to achieve it? I need a plan. So I come up with one and put it in pseudocode.

***If you can think of a new and useful product***

***Then that’s your product***

***Otherwise***

***Repackage an existing product as your product***

***Make an infomercial about your product***

***Show the infomercial on TV***

***Charge $100 per unit of your product***

***Sell 10,000 units of your product***

Even though anyone, even a non-programmer, can understand my plan, my pseudocode feels vaguely like a program. The first four lines resemble an if statement with an else clause, and that’s intentional. When you write your plan, you should try to incorporate the feel of the code that you’re representing with pseudocode.

**Using Stepwise Refinement**

Your programming plan might not be finished after only one draft. Often pseudocode needs multiple passes before it can be implemented in programming code. Stepwise refinement is one process used to rewrite pseudocode to make it ready for implementation. ***Stepwise refinemen***t is pretty simple. Basically, it means, “Make it more detailed.” By taking each step described in pseudocode and breaking it down into a series of simpler steps, the plan becomes closer to programming code. Using stepwise refinement, you keep breaking down each step until you feel the entire plan could be fairly easily translated into a program. As an example, take a step from my master plan to make a million dollars:

***Create an infomercial about your product***

This might seem like too vague of a task. How do you create an infomercial?

Using stepwise refinement, you can break down the single step into several others so it becomes:

***Write a script for an infomercial about your product***

***Rent a TV studio for a day***

***Hire a production crew***

***Hire an enthusiastic audience***

***Film the infomercial***

If you feel these five steps are clear and achievable, then that part of the pseudocode has been thoroughly refined. If you’re still unclear about a step, refine it some more. Continue with this process and you will have a complete plan - and a million dollars.

**INTRODUCING HANGMAN**

In the Hangman program, the computer picks a secret word and the player tries to guess it one letter at a time. The player is allowed eight incorrect guesses. If he or she fails to guess the word in time, the player is hanged and the game is over.

**Planning the Game**

Before I write a single line in C++, I plan the game program using pseudocode.

***Create a group of words***

***Pick a random word from the group as the secret word***

***While player hasn’t made too many incorrect guesses and hasn’t guesses the secret word***

***Tell player how many incorrect guesses he or she has left***

***Show player the letters he or she has guessed***

***Show player how much of the secret word he or she has guessed***

***Get player’s next guess***

***While player has entered a letter that he or she has already guessed***

***Get player’s guess***

***Add the new guess to the group of used letters***

***If the guess is in the secret word***

***Tell the player the guess is correct***

***Update the word guessed so far with new letter***

***Otherwise***

***Tell the player the guess is incorrect***

***Increment the number of incorrect guesses the player has made***

***If the player has made too many incorrect guesses***

***Tell the player that he or she has been hanged***

***Otherwise***

***Congratulate the player on guessing the secret word***

Although the pseudocode doesn’t account for every line of C++ I’ll write, I think it does a good job describing what I need to do. Then I begin writing the program.

**Setting Up the Program**

As usual, I start with some comments and include the files I need.

**// Hangman**

**// The classic game of hangman**

**#include <iostream>**

**#include <string>**

**#include <vector>**

**#include <algorithm>**

**#include <ctime>**

**#include <cctype>**

**using namespace std;**

Notice that I include a new file: **cctype**. It’s part of the standard library, and it includes functions for converting characters to uppercase, which I use so I can compare apples to apples (uppercase to uppercase) when I compare individual characters.

**Initializing Variables and Constants**

Next, I start the **main()** function and initialize variables and constants for the game.

**int main()**

**{**

**//** setup

**const int MAX\_WRONG = 8; //** maximum number of incorrect guesses allowed

**vector<string> words; //** collection of possible words to guess

**words.push\_back("GUESS");**

**words.push\_back("HANGMAN");**

**words.push\_back("DIFFICULT");**

**srand(static\_cast<unsigned int>(time(0)));**

**random\_shuffle(words.begin(), words.end());**

**const string THE\_WORD = words[0]; //** words to guess

**int wrong = 0; //** number of incorrect guesses

**string soFar(THE\_WORD.size(), '-'); //** word guesses so far guessed

**string used = ""; //** letters already guessed

**cout << "Welcome to Hangman. Good luck!\n";**

**return 0;**

**}**

**Entering the Main Loop**

Next, I enter the main loop, which continues until the player has made too many incorrect guesses or has guessed the word.

**//** main loop

**while ((wrong < MAX\_WRONG) && (soFar != THE\_WORD))**

**{**

**cout << "\n\nYou have " << (MAX\_WRONG - wrong);**

**cout << " incorrect guesses left.\n";**

**cout << "\nYou’ve used the following letters:\n" << used << endl;**

**cout << "\nSo far, the word is:\n" << soFar << endl;**

**Getting the Player’s Guess**

Next, I get the player’s guess.

**char guess;**

**cout << "\n\nEnter your guess: ";**

**cin >> guess;**

**guess = toupper(guess); //** make uppercase since secret word in upppercase

**while (used.find(guess) != string::npos)**

**{**

**cout << "\nYou've already guessed " << guess << endl;**

**cout << "Enter your guess: ";**

**cin >> guess;**

**guess = toupper(guess);**

**}**

**used + = guess;**

**if(THE\_WORD.find(guess) != string::npos)**

**{**

**cout << "That's right! " << guess << " is in the word.\n";**

**//** update soFar to include newly guessed letter

**for(int i = 0; i < THE\_WORD.length(); ++i)**

**{**

**if(THE\_WORD[i] == guess)**

**{**

**soFar[i] = guess;**

**}**

**}**

**}**

**else**

**{**

**cout << "Sorry, " << guess << " isn't in the word.\n";**

**++wrong;**

**}**

**}**

I convert the **guess** to uppercase using the function **uppercase()**, which is defined in the file **cctype**. I do this so I can compare uppercase letters to uppercase letters when I’m checking a guess against the letters of the secret word.

If the player guesses a letter that he or she has already guessed, I make the player guess again. If the player guesses a letter correctly, I update the word guessed so far. Otherwise, I tell the player the guess is not in the secret word and I increase the number of incorrect guesses the player has made.

**Ending the Game**

At this point, the player has guessed the word or has made one too many incorrect guesses. Either way, the game is over.

**// shut down**

**if(wrong == MAX\_WRONG)**

**{**

**cout << "\nYou have been hanged!";**

**}**

**else**

**{**

**cout << "\nYou guessed it !";**

**}**

**cout << "\The word was " << THE\_WORD << endl;**

**return 0;**

**}**

I congratulate the player or break the bad news that he or she has been hanged. Then I reveal the secret word.

Welcome to Hangman. Good luck!

You have 8 incorrect guesses left.

You've used the following letters:

So far, the word is:

-----

Enter your guess: F

Sorry, F isn't in the word.

You have 7 incorrect guesses left.

You've used the following letters:

F

So far, the word is:

-----

Enter your guess: S

That's right! S is in the word.

You have 7 incorrect guesses left.

You've used the following letters:

FS

So far, the word is:

---SS

Enter your guess: E

That's right! E is in the word.

You have 7 incorrect guesses left.

You've used the following letters:

FSE

So far, the word is:

--ESS

Enter your guess: G

That's right! G is in the word.

You have 7 incorrect guesses left.

You've used the following letters:

FSEG

So far, the word is:

G-ESS

Enter your guess: E

You've already guessed E

Enter your guess: U

That's right! U is in the word.

You guessed it!

The word was GUESS

**SUMMARY**

In this chapter, you learned the following concepts:

* ***The Standard Template Library*** (**STL**) is a powerful collection of programming code that provides ***containers***, ***algorithms***, and ***iterators***.
* ***Containers*** are objects that let you store and access collections of values of the same type.
* ***Algorithms*** defined in the **STL** can be used with their containers and provide common functions for working with groups of objects.
* ***Iterators*** are objects that identify elements in containers and can be manipulated to move among elements.
* ***Iterators*** are the key to using containers to their fullest. Many of the container member functions require iterators, and the **STL** ***algorithms*** require them too.
* To get the value referenced by an iterator, you must dereference the iterator using the dereference operator (**\***).
* A ***vector*** is one kind of sequential container provided by the **STL**. It acts like a dynamic array.
* It’s very efficient to iterate through a vector. It’s also very efficient to insert or remove an element from the end of a vector.
* It can be inefficient to insert or delete elements from the middle of a vector, especially if the vector is large.
* ***Pseudocode***, which falls somewhere between English and a programming language, is used to plan programs.
* ***Stepwise refinement*** is a process used to rewrite pseudocode to make it ready for implementation.

**CHAPTER 5**

**FUNCTIONS: MAD LIB**

**CREATING FUNCTIONS**

C++ lets you write programs with multiple functions. Your new functions work just like the ones that are part of the standard language - they go off and perform a task and then return control to your program. A big advantage of writing new functions is that doing so allows you to break up your code into manageable pieces. Just like the functions you’ve already learned about from the standard library, your new functions should do one job well.

**Introducing the Instructions Program**

The results of the Instructions program are pretty basic - a few lines of text that are the beginning of some game instructions. From the looks of the output, Instructions seems like a program you could have written back in Chapter 1, “Types, Variables, and Standard I/O: Lost Fortune.” But this program has a fresh element working behind the scenes - a new function.

**FUNCTIONS**

// Instructions

// Demonstrates writing new functions

**#include <iostream>**

**using namespace std;**

// function prototype (declaration)

**void instructions();**

**int main()**

**{**

**instructions();**

**return 0;**

**}**

// function definition

**void instructions()**

**{**

**cout << "Welcome to the most fun you've ever had with text!\n\n";**

**cout << "Here's how to play the game...\n";**

**}**

**Welcome to the most fun you've ever had with text!**

**Here's how to play the game...**

**Declaring Functions**

Before you can call a function you’ve written, you have to declare it. One way to declare a function is to write a ***function prototype*** - code that describes the function. You write a prototype by listing the return value of the function (or **void** if the function returns no value), followed by the name of the function, followed by a list of parameters between a set of parentheses. Parameters receive the values sent as arguments in a function call.

Just before the **main()** function, I write a function prototype:

**void instructions();**

In the preceding code, I declared a function named **instructions** that doesn’t return a value. (You can tell this because I used **void** as the return type.) The function also takes no values, so it has no parameters. (You can tell this because there’s nothing between the parentheses.)

Prototypes are not the only way to declare a function. Another way to accomplish the same thing is to let the function definition act as its own declaration. To do that, you simply have to put your function definition before the call to the function.

**Hint**

*Although you don’t have to use prototypes, they offer a lot of benefits - not the least of which is making*

*your code clearer.*

**Defining Functions**

Defining functions means writing all the code that makes the function tick. You define a function by listing the ***return value*** of the function (or **void** if the function returns no value), followed by the na***me of the function***, followed by ***a list of parameters*** between a set of parentheses - just like a function prototype (except you don’t end the line with a semicolon). This is called the ***function header***. Then you create a block with curly braces that contains the instructions to be executed when the function is executed. This is called the ***function body***.

At the end of the Instructions program, I define my simple **instructions()** function, which displays some game instructions. Because the function doesn’t return any value, I don’t need to use a return statement like I do in **main()**. I simply end the function definition with a closing curly brace.

**void instructions()**

**{**

**cout << "Welcome to the most fun you’ve ever had with text!\n\n";**

**cout << "Here’s how to play the game...\n";**

**}**

**Trap**

*A function definition must match its prototype on return type and function name; otherwise, you’ll*

*generate a compile error.*

**Calling Functions**

You call your own functions the same way you call any other function – by writing the function’s name followed by a pair of parentheses that encloses a valid list of arguments. In **main()** , I call my newly minted function simply with:

**instructions();**

This line invokes **instructions()** . Whenever you call a function, control of the program jumps to that function. In this case, it means control jumps to **instructions()** and the program executes the function’s code, which displays the game instructions. When a function finishes, control returns to the calling code. In this case, it means control returns to **main()**. The next statement in **main() ( return 0; )** is executed and the program ends.

**Understanding Abstraction**

By writing and calling functions, you practice what’s known as ***abstraction***. Abstraction lets you think about the big picture without worrying about the details. In this program, I can simply use the function **instructions()** without worrying about the details of displaying the text. All I have to do is call the function with one line of code, and it gets the job done.

You might be surprised where you find abstraction, but people use it all the time. For example, consider two employees at a fast-food restaurant. If one tells the other that he just filled a Number 3 and “sized it,” the other employee knows that the first employee took a customer’s order, went to the heat lamps, grabbed a burger, went over to the deep fryer, filled their biggest cardboard container with french fries, went to the soda fountain, grabbed their biggest cup, filled it with soda, gave it all to the customer, took the customer’s money, and gave the customer change. Not only would this level of detail make for a boring conversation, but also it’s unnecessary. Both employees understand what it means to fill a Number 3 and “size it.” They don’t have to concern themselves with all the details because they’re using abstraction.

**Introducing the Yes or No Program**

The Yes or No program asks the user typical questions a gamer might have to answer. First, the program asks the user to indicate yes or no. Then the program gets more specific and asks whether the user wants to save his game. Again, the results of the program are not remarkable; it’s their implementation that’s interesting. Each question is posed by a different function that communicates with **main()**.

**Return Values and parameters**

// Yes or No

// Demonstrates return values and parameters

**#include <iostream>**

**#include <string>**

**using namespace std;**

**char askYesNo1();**

**char askYesNo2(string question);**

**int main()**

**{**

**char answer1 = askYesNo1();**

**cout << "Thanks for answering: " << answer1 << "\n\n";**

**char answer2 = askYesNo2("Do you wish to save your game?");**

**cout << "Thanks for answering: " << answer2 << "\n";**

**return 0;**

**}**

**char askYesNo1()**

**{**

**char response1;**

**do**

**{**

**cout << "Please enter 'y' or 'n': ";**

**cin >> response1;**

**} while (response1 != 'y' && response1 != 'n');**

**return response1;**

**}**

**char askYesNo2(string question)**

**{**

**char response2;**

**do**

**{**

**cout << question << " (y/n): ";**

**cin >> response2;**

**} while (response2 != 'y' && response2 != 'n');**

**return response2;**

**}**

**Please enter 'y' or 'n': y**

**Thanks for answering: y**

**Do you wish to save your game? (y/n): y**

**Thanks for answering: y**

**Returning a Value**

You can return a value from a function to send information back to the calling code. To return a value, you need to specify a return type and then return a value of that type from the function.

**Specifying a Return Type**

The first function I declare, **askYesNo1()** , returns a **char** value. You can tell this from the function prototype before **main()** :

**char askYesNo1();**

You can also see this from the function definition after **main()**:

**char askYesNo1()**

**Using the return Statement**

**askYesNo1()** asks the user to enter **y** or **n** and keeps asking until he does. Once the user enters a valid character, the function wraps up with the following line, which returns the value of **response1** .

**return response1;**

Notice that **response1** is a **char** value. It has to be because that’s what I promised to return in both the function prototype and function definition.

A function ends whenever it hits a **return** statement. It’s perfectly acceptable for a function to have more than one **return** . This just means that the function has several points at which it can end.

**Trick**

*You don’t have to return a value with a* **return** *statement. You can use* **return** *by itself in a function that returns no value (one that indicates* **void** *as its return type) to end the function.*

**Using a Returned Value**

**In main()** , I call the function with the following line, which assigns the return value of the function to **answer1**.

**char answer1 = askYesNo1();**

This means that **answer1** is assigned either ’ **y** ’ or ’ **n** ’ - whichever character the user entered when prompted by **askYesNo1()**.

Next, in **main()**, I display the value of answer1 for all to see.

**Accepting Values into Parameters**

You can send a function values that it accepts into its parameters. This is the most common way to get information into a function.

**Specifying Parameters**

The second function I declare, **askYesNo2()**, accepts a value into a parameter. Specifically, it accepts a value of type **string** . You can tell this from the function prototype before **main()**:

**char askYesNo2(string question);**

**Hint**

*You don’t have to use parameter names in a prototype; all you have to include are the parameter types. For example, the following is a perfectly valid prototype which declares* **askYesNo2()** *, a function with one* **string** *parameter that returns a* **char***.*

**char askYesNo2(string);**

*Even though you don’t have to use parameter names in prototypes, it’s a good idea to do so. It makes your code clearer, and it’s worth the minor effort.*

From the header of **askYesNo2()**, you can see that the function accepts a **string** object as a parameter and names that parameter question .

**char askYesNo2(string question)**

Unlike prototypes, you must specify parameter names in a function definition. You use a parameter name inside a function to access the parameter value.

**Trap**

*The parameter types specified in a function prototype must match the parameter types listed in the function definition. If they don’t, you’ll generate a nasty compile error.*

**Passing Values to Parameters**

The **askYesNo2()** function is an improvement over **askYesNo1()**. The new function allows you to ask your own personalized question by passing a string prompt to the function. In **main()** , I call **askYesNo2()** with:

**char answer2 = askYesNo2("Do you wish to save your game?");**

This statement calls **askYesNo2()** and passes the string literal argument "**Do you wish to save your game?**" to the function.

**Using Parameter Values**

**askYesNo2()** accepts "**Do you wish to save your game?**" into its parameter question, which acts like any other variable in the function. In fact, I display question with:

**cout << question << " (y/n): ";**

**Hint**

*Actually, there’s a little more going on behind the scenes here. When the string literal "***Do you wish to save your game?***" is passed to question , a* **string** *object equal to the string literal is created and the string object is assigned to question .*

Just like **askYesNo1(),** **askYesNo2()** continues to prompt the user until he enters **y** or **n**. Then the function returns that value and ends.

Back in **main()**, the returned char value is assigned to **answer2**, which I then display.

**Understanding Encapsulation**

You might not see the need for return values when using your own functions. Why not just use the variables **response1** and **response2** back in the **main()**? Because you can’t; **response1** and **response2** don’t exist outside of the functions in which they were defined. In fact, no variable you create in a function, including its parameters, can be directly accessed outside its function. This is a good thing, and it is called encapsulation. Encapsulation helps keep independent code truly separate by hiding or encapsulating the details. That’s why you use parameters and return values - to communicate only the information that needs to be exchanged. Plus, you don’t have to keep track of variables you create within a function in the rest of your program. As your programs get large, this is a great benefit.

Encapsulation might sound a lot like abstraction. That’s because they’re closely related. Encapsulation is a principle of abstraction. Abstraction saves you from worrying about the details, while encapsulation hides the details from you. As an example, consider a television remote control with volume up and down buttons. When you use a TV remote to change the volume, you’re employing abstraction because you don’t need to know what happens inside the TV for it to work. Now suppose the TV remote has 10 volume levels. You can get to them all through the remote, but you can’t directly access them. That is, you can’t get a specific volume number directly. You can only press the up and down volume buttons to eventually get to the level you want. The actual volume number is encapsulated and not directly available to you.

**UNDERSTANDING SOFTWARE REUSE**

You can reuse functions in other programs. For example, since asking the user a yes or no question is such a common thing to do in a game, you could create an **askYesNo()** function and use it in all of your future game programs. So writing good functions not only saves you time and energy in your current game project, but it can save you effort in future ones, too.

**In the Real World**

*It’s always a waste of time to reinvent the wheel, so software reuse - employing existing software and other elements in new projects - is a technique that game companies take to heart. The benefits of software reuse include:*

* ***Increased company productivity****. By reusing code and other elements that already exist, such as a graphics engine, game companies can get their projects done with less effort.*
* ***Improved software quality****. If a game company already has a tested piece of code, such as a networking module, then the company can reuse the code with the knowledge that it’s bug-free.*
* ***Improved software performance****. Once a game company has a high-performance piece of code, using it again not only saves the company the trouble of reinventing the wheel, it saves them from reinventing a less efficient one.*

You can reuse code you’ve written by copying from one program and pasting it into another, but there is a better way. You can divide a big game project into multiple files. You’ll learn about this technique in Chapter 10, “Inheritance and Polymorphism: Blackjack.”

**WORKING WITH SCOPES**

A variable’s ***scope*** determines where the variable can be seen in your program. Scopes allow you to limit the accessibility of variables and are the key to encapsulation, helping keep separate parts of your program, such as functions, apart from each other.

***Introducing the Scoping Program***

The Scoping program demonstrates scopes. The program creates three variables with the same name in three separate scopes. The program displays the values of these variables, and you can see that even though they all have the same name, the variables are completely separate entities.

// Scoping

// Demonstrates scopes

**#include <iostream>**

**using namespace std;**

**void func();**

**int main()**

**{**

**int var = 5;** // local variable in main()

**cout << "In main() var is: " << var << "\n\n";**

**func();**

**cout << "Back in main() var is: " << var << "\n\n";**

**{**

**cout << "In main() in a new scope var is: " << var << "\n\n";**

**cout << "Creating new var in new scope.\n";**

**int var = 10;** // variable in new scope, hides other variable named var

**cout << "In main() in a new scope var is: " << var << "\n\n";**

**}**

**cout << "At end of main() var created in new scope no longer exists.\n";**

**cout << "At end of main() var is: " << var << "\n";**

**return 0;**

**}**

**void func()**

**{**

**int var = -5;** // local variable in func()

**cout << "In func() var is: " << var << "\n\n";**

**}**

**In main() var is: 5**

**In func() var is: -5**

**Back in main() var is: 5**

**In main() in a new scope var is: 5**

**Creating new var in new scope.**

**In main() in a new scope var is: 10**

**At end of main() var created in new scope no longer exists.**

**At end of main() var is: 5**

***Working with Separate Scopes***

Every time you use curly braces to create a block, you create a scope. Functions are one example of this. Variables declared in a scope aren’t visible outside of that scope. This means that variables declared in a function aren’t visible outside of that function.

Variables declared inside a function are considered ***local variables*** - they’re local to the function. This is what makes functions encapsulated.

You’ve seen many local variables in action already. I define yet another local variable in **main()** with:

**int var = 5;** // local variable in main()

This line declares and initializes a local variable named **var**. I send the variable to **cout** in the next line of code:

**cout << "In main() var is: " << var << "\n\n";**

This works just as you’d expect, **5** is displayed.

Next, I call **func()**. Once I enter the function, I’m in a separate scope outside of the scope defined by **main()**. As a result, I can’t access the variable **var** that I defined in **main()** . This means that when I next define a variable named **var** in **func()** with the following line, this new variable is completely separate from the variable named var in **main(** .

**int var = -5;** // local variable in func()

The two have no effect on each other, and that’s the beauty of scopes. When you write a function, you don’t have to worry if another function uses the same variable names.

Then, when I display the value of **var** in **func()** with the following line, the computer displays **-5**.

**cout << "In func() var is: " << var << "\n\n";**

That’s because, as far as the computer can see in this scope, there’s only one variable named **var** - the local variable I declared in this function.

Once a scope ends, all of the variables declared in that scope cease to exist. They’re said to go out of scope. So next, when **func()** ends, its scope ends. This means all of the variables declared in **func()** are destroyed. As a result, the **var** I declared in **func()** with a value of **-5** is destroyed.

After **func()** ends, control returns to **main()** and picks up right where it left off. Next, the following line is executed, which sends var to cout.

**cout << "Back in main() var is: " << var << "\n\n";**

The value of the var local to **main() (5)** is displayed again. You might be wondering what happened to the var I created in **main()** while I was in **func()**. Well, the variable wasn’t destroyed because **main()** hadn’t yet ended. (Program control simply took a small detour to **func()** .) When a program momentarily exits one function to enter another, the computer saves its place in the first function, keeping safe the values of all of its local variables, which are reinstated when control returns to the first function.

**Hint**

*Parameters act just like local variables in functions.*

***Working with Nested Scopes***

You can create a nested scope with a pair of curly braces in an existing scope. That’s what I do next in **main()**, with:

**{**

**cout << "In main() in a new scope var is: " << var << "\n\n";**

**cout << "Creating new var in new scope.\n";**

**int var = 10;** // variable in new scope, hides other variable named var

**cout << "In main() in a new scope var is: " << var << "\n\n";**

**}**

This new scope is a nested scope in **main()**. The first thing I do in this nested scope is display **var**. If a variable hasn’t been declared in a scope, the computer looks up the levels of nested scopes one at a time to find the variable you requested. In this case, because **var** hasn’t been declared in this nested scope, the computer looks one level up to the scope that defines **main()** and finds **var**. As a result, the program displays that variable’s value, **5**.

However, the next thing I do in this nested scope is declare a new variable named **var** and initialize it to **10**. Now when I send **var** to **cout** , **10** is displayed. This time the computer doesn’t have to look up any levels of nested scopes to find **var**; there’s a **var** local to this scope. And don’t worry, the **var** I first declared in **main()** still exists; it’s simply hidden in this nested scope by the new **var**.

**Trap**

*Although you can declare variables with the same name in a series of nested scopes, it’s not a good idea because it can lead to confusion.*

Next, when the nested scope ends, the **var** that was equal to **10** goes out of scope and ceases to exist. However, the first **var** I created is still around, so when I display **var** for the last time in **main()** with the following line, the program displays **5**.

**cout << "At end of main() var is: " << var << "\n";**

**Hint**

*When you define variables inside for loops, while loops, if statements, and switch statements, these variables don’t exist outside their structures. They act like variables declared in a nested scope. For example, in the following code, the variable* ***i*** *doesn’t exist outside the loop.*

**for(int i = 0; i < 10; ++i)**

**{**

**cout << i;**

**}**

// i doesn’t exist outside the loop

*But beware - some older compilers don’t properly implement this functionality of standard C++. I recommend that you use an IDE with a modern compiler, such as Microsoft Visual Studio Express 2013 for Windows Desktop. For step-by-step instructions on how to create your first project with this IDE, check out Appendix A, “Creating Your First* C++ Program.”

**USING GLOBAL VARIABLES**

Through the magic of encapsulation, the functions you’ve seen are all totally sealed off and independent from each other. The only way to get information into them is through their parameters, and the only way to get information out of them is from their return values. Well, that’s not completely true. There is another way to share information among parts of your program – through ***global variables*** (variables that are accessible from any part of your program).

***Introducing the Global Reach Program***

The Global Reach program demonstrates global variables. The program shows how you can access a global variable from anywhere in your program. It also shows how you can hide a global variable in a scope. Finally, it shows that you can change a global variable from anywhere in your program.

**Global Variables**

// Global Reach

// Demonstrates global variables

In main() glob is: 10

In access\_global() glob is: 10

In hide\_global() glob is: 0

In main() glob is: 10

In change\_global() glob is: -10

In main() glob is: -10

**#include <iostream>**

**using namespace std;**

**int glob = 10;** // global variable

**void access\_global();**

**void hide\_global();**

**void change\_global();**

**int main()**

**{**

**cout << "In main() glob is: " << glob << "\n\n";**

**access\_global();**

**hide\_global();**

**cout << "In main() glob is: " << glob << "\n\n";**

**change\_global();**

**cout << "In main() glob is: " << glob << "\n\n";**

**return 0;**

**}**

**void access\_global()**

**{**

**cout << "In access\_global() glob is: " << glob << "\n\n";**

**}**

**void hide\_global()**

**{**

**int glob = 0;** // hide global variable glob

**cout << "In hide\_global() glob is: " << glob << "\n\n";**

**}**

**void change\_global()**

**{**

**glob = -10;** // change global variable glob

**cout << "In change\_global() glob is: " << glob << "\n\n";**

**}**

***Declaring Global Variables***

You declare global variables outside of any function in your program file. That’s what I do in the following line, which creates a global variable named **glob** initialized to **10**.

**int glob = 10; // global variable**

***Accessing Global Variables***

You can access a global variable from anywhere in your program. To prove it, I display **glob** in **main()** with:

**cout << "In main() glob is: " << glob << "\n\n";**

The program displays **10** because as a global variable, **glob** is available to any part of the program. To show this again, I next call **access\_global()**, and the computer executes the following code in that function:

**cout << "In access\_global() glob is: " << glob << "\n\n";**

Again, **10** is displayed. That makes sense because I’m displaying the exact same variable in each function.

***Hiding Global Variables***

You can hide a global variable like any other variable in a scope; you simply declare a new variable with the same name. That’s exactly what I do next, when I call **hide\_global()**. The key line in that function doesn’t change the global variable **glob**; instead, it creates a new variable named **glob**, local to **hide\_global()**, that hides the global variable.

**int glob = 0; // hide global variable glob**

As a result, when I send **glob** to **cout** next in **hide\_global()** with the following line, **0** is displayed.

**cout << "In hide\_global() glob is: " << glob << "\n\n";**

The global variable **glob** remains hidden in the scope of **hide\_global()** until the function ends.

To prove that the global variable was only hidden and not changed, next I display **glob** back in **main()** with:

**cout << "In main() glob is: " << glob << "\n\n";**

Once again, 10 is displayed.

**Trap**

*Although you can declare variables in a function with the same name as a global variable, it’s not a good idea because it can lead to confusion.*

***Altering Global Variables***

Just as you can access a global variable from anywhere in your program, you can alter one from anywhere in your program, too. That’s what I do next, when I call the **change\_global()** function. The key line of the function assigns **-10** to the global variable glob.

**glob = -10; // change global variable glob**

To show that it worked, I display the variable in **change\_global()** with:

**cout << "In change\_global() glob is: " << glob << "\n\n";**

Then, back in **main()**, I send **glob** to **cout** with:

**cout << "In main() glob is: " << glob << "\n\n";**

Because the global variable **glob** was changed, **-10** is displayed.

***Minimizing the Use of Global Variables***

Just because you can doesn’t mean you should. This is a good programming motto. Sometimes things are technically possible but not a good idea. Using global variables is an example of this. In general, global variables make programs confusing because it can be difficult to keep track of their changing values. You should limit your use of global variables as much as possible.

**USING GLOBAL CONSTANTS**

Unlike global variables, which can make your programs confusing, global constants - constants that can be accessed from anywhere in your program - can help make programs clearer. You declare a global constant much like you declare a global variable - by declaring it outside of any function. And because you’re declaring a constant, you need to use the **const** keyword. For example, the following line defines a global constant (assuming the declaration is outside of any function) named **MAX\_ENEMIES** with a value of **10** that can be accessed anywhere in the program.

**const int MAX\_ENEMIES = 10;**

**Trap**

*Just like with global variables, you can hide a global constant by declaring a local constant with the same name. However, you should avoid this because it can lead to confusion.*

How exactly can global constants make game programming code clearer? Well, suppose you’re writing an action game in which you want to limit the total number of enemies that can blast the poor player at once. Instead of using a numeric literal everywhere, such as **10**, you could define a global constant **MAX\_ENEMIES** that’s equal to **10**. Then whenever you see that global constant name, you know exactly what it stands for.

One caveat: You should only use global constants if you need a constant value in more than one part of your program. If you only need a constant value in a specific scope (such as in a single function), use a local constant instead.

**USING DEFAULT ARGUMENTS**

When you write a function in which a parameter almost always gets passed the same value, you can save the caller the effort of constantly specifying this value by using a default argument - a value assigned to a parameter if none is specified. Here’s a concrete example. Suppose you have a function that sets the graphics display. One of your parameters might be **bool** **fullScreen**, which tells the function whether to display the game in full screen or windowed mode. Now, if you think the function will often be called with

true for **fullScreen** , you could give that parameter a default argument of **true**, saving the caller the effort of passing **true** to **fullScreen** whenever the caller invokes this display-setting function.

***Introducing the Give Me a Number Program***

The Give Me a Number program asks the user for two different numbers in two different ranges. The same function is called each time the user is prompted for a number. However, each call to this function uses a different number of arguments because this function has a default argument for the lower limit. This means the caller can omit an argument for the lower limit, and the function will use a default value automatically.

**Default Function Arguments**

// Give Me a Number

// Demonstrates default function arguments

**#include <iostream>**

**#include <string>**

**using namespace std;**

**int askNumber(int high, int low = 1);**

**int main()**

**{**

**int number = askNumber(5);**

**cout << "Thanks for entering: " << number << "\n\n";**

**number = askNumber(10, 5);**

**cout << "Thanks for entering: " << number << "\n\n";**

**return 0;**

**}**

**int askNumber(int high, int low)**

**{**

**int num;**

**do**

**{**

**cout << "Please enter a number" << " (" << low << " - " << high << "): ";**

**cin >> num;**

**} while (num > high || num < low);**

**return num;**

**}**

Please enter a number (1 - 5): 4

Thanks for entering: 4

Please enter a number (5 - 10): 12

Please enter a number (5 - 10): 6

Thanks for entering: 6

**Specifying Default Arguments**

The function **askNumber()** has two parameters: **high** and **low**. You can tell this from the function prototype:

**int askNumber(int high, int low = 1);**

Notice that the second parameter, **low**, looks like it’s assigned a value. In a way, it is. The **1** is a default argument, meaning that if a value isn’t passed to **low** when the function is called, low is assigned **1**. You specify default arguments by using **=** followed by a value after a parameter name.

**Trap**

*Once you specify a default argument in a list of parameters, you must specify default arguments for all remaining parameters. So the following prototype is valid:*

**void setDisplay(int height, int width, int depth = 32, bool fullScreen = true);**

*while this one is illegal:*

**void setDisplay(int width, int height, int depth = 32, bool fullScreen);**

By the way, you don’t repeat the default argument in the function definition, as you can see in the function definition of **askNumber()**.

**int askNumber(int high, int low)**

**Assigning Default Arguments to Parameters**

The **askNumber()** function asks the user for a number between an upper and a lower limit. The function keeps asking until the user enters a number within the range, and then it returns the number. I first call the function in **main()** with:

**int number = askNumber(5);**

As a result of this code, the parameter high in **askNumber()** is assigned **5**. Because I don’t provide any value for the second parameter, **low** , it is assigned the default value of **1**. This means the function prompts the user for a number between **1** and **5**.

**Trap**

*When you are calling a function with default arguments, once you omit an argument, you must omit arguments for all remaining parameters. For example, given the prototype*

**void setDisplay(int height, int width, int depth = 32, bool fullScreen = true);**

*a valid call to the function would be*

**setDisplay(1680, 1050);**

*while an illegal call would be*

**setDisplay(1680, 1050, false);**

Once the user enters a valid number, **askNumber()** returns that value and ends. Back in **main()**, the value is assigned to number and displayed.

**Overriding Default Arguments**

Next, I call **askNumber()** again with:

**number = askNumber(10, 5);**

This time I pass a value for low - **5**. This is perfectly fine; you can pass an argument for any parameter with a default argument, and the value you pass will override the default. In this case, it means that **low** is assigned **5**.

As a result, the user is prompted for a number between **5** and **10**. Once the user enters a valid number, **askNumber()** returns that value and ends. Back in **main()**, the value is assigned to number and displayed.

**OVERLOADING FUNCTIONS**

You’ve seen how you must specify a parameter list and a single return type for each function you write. But what if you want a function that’s more versatile - one that can accept different sets of arguments? For example, suppose you want to write a function that performs a 3D transformation on a set of vertices that are represented as **float**s, but you want the function to work with **int**s as well. Instead of writing two separate functions with two different names, you could use function overloading so that a single function could handle the different parameter lists. This way, you could call one function and pass vertices as either float s or **int**s.

**Introducing the Triple Program**

The Triple program triples the value **5** and “**gamer**”. The program triples these values using a single function that’s been overloaded to work with an argument of two different types: **int** and **string**.

**Function Overloading**

// Triple

// Demonstrates function overloading

**#include <iostream>**

**#include <string>**

**using namespace std;**

**int triple(int number);**

**string triple(string text);**

**int main()**

**{**

**cout << "Tripling 5: " << triple(5) << "\n\n";**

**cout << "Tripling 'gamer': " << triple("gamer");**

**return 0;**

**}**

**int triple(int number)**

**{**

**return (number \* 3);**

**}**

**Tripling 5: 15**

**Tripling 'gamer': gamergamergamer**

**string triple(string text)**

**{**

**return (text + text + text);**

**}**

**Creating Overloaded Functions**

To create an overloaded function, you simply need to write multiple function definitions with the ***same name*** and ***different parameter lists***. In the Triple program, I write two definitions for the function **triple()**, each of which specifies a different type as its single argument. Here are the function prototypes:

**int triple(int number);**

**string triple(string text);**

The first takes an **int** argument and returns an **int**. The second takes a **string** object and returns a **string** object.

In each function definition, you can see that I return triple the value sent. In the first function, I return the **int** sent, tripled. In the second function, I return the **string** sent, repeated three times.

**Trap**

*To implement function overloading, you need to write multiple definitions for the same function with different parameter lists. Notice that I didn’t mention anything about return types. That’s because if you write two function definitions in which only the return type is different, you’ll generate a compile error. For example, you cannot have both of the following prototypes in a program:*

**int Bonus(int);**

**float Bonus(int);**

**Calling Overloaded Functions**

You can call an overloaded function the same way you call any other function, by using its name with a set of valid arguments. But with overloaded functions, the compiler (based on the argument values) determines which definition to invoke. For example, when I call **triple()** with the following line and use an **int** as the argument, the compiler knows to invoke the definition that takes an **int**. As a result, the function returns the int **15**.

**cout << "Tripling 5: " << triple(5) << "\n\n";**

I call **triple()** again with:

**cout << "Tripling ’gamer’: " << triple("gamer");**

Because I use a **string** literal as the argument, the compiler knows to invoke the definition of the function that takes a **string** object. As a result, the function returns the string object equal to **gamergamergamer**.

**INLINING FUNCTIONS**

There’s a small performance cost associated with calling a function. Normally this isn’t a big deal because the cost is relatively minor. However, for tiny functions (such as one or two lines), it’s sometimes possible to speed up program performance by inlining them. By inlining a function, you ask the compiler to make a copy of the function everywhere it’s called. As a result, program control doesn’t have to jump to a different location each time the function is called.

**Introducing the Taking Damage Program**

The Taking Damage program simulates what happens to a character’s health as the character takes radiation damage. The character loses half of his health each round. Fortunately, the program runs only three rounds, so we’re spared the sad end of the character. The program inlines the tiny function that calculates the character’s new health.

**Function Inlining**

// Taking Damage

// Demonstrates function inlining

**#include <iostream>**

**int radiation(int health);**

**using namespace std;**

**int main()**

**{**

**int health = 80;**

**cout << "Your health is " << health << "\n\n";**

**health = radiation(health);**

**cout << "After radiation exposure your health is " << health << "\n\n";**

**health = radiation(health);**

**cout << "After radiation exposure your health is " << health << "\n\n";**

**health = radiation(health);**

**cout << "After radiation exposure your health is " << health << "\n\n";**

**return 0;**

**}**

**inline int radiation(int health)**

**{**

**return (health / 2);**

**}**

Your health is 80

After radiation exposure your health is 40

After radiation exposure your health is 20

After radiation exposure your health is 10

**Specifying Functions for Inlining**

To mark a function for inlining, simply put **inline** before the function definition. That’s what I do when I define the following function:

**inline int radiation(int health)**

Note that you don’t use inline in the function declaration:

**int radiation(int health);**

By flagging the function with **inline**, you ask the compiler to ***copy the function directly into the calling code***. This saves the overhead of making the function call. That is, program control doesn’t have to jump to another part of your code. For small functions, this can result in a performance boost.

However, inlining is not a silver bullet for performance. In fact, indiscriminate inlining can lead to worse performance because inlining a function creates extra copies of it, which can dramatically increase memory consumption.

**Hint**

*When you inline a function, you really make a request to the compiler, which has the ultimate decision on whether to inline the function. If your compiler thinks that inlining won’t boost performance, it won’t inline the function.*

**Calling Inlined Functions**

Calling an inlined function is no different than calling a non-inlined function, as you see with my first call to **radiation()**.

**health = radiation(health);**

This line of code assigns health one-half of its original value. Assuming that the compiler grants my request for inlining, this code doesn’t result in a function call. Instead, the compiler places the code to halve health right at this place in the program. In fact, the compiler does this for all three calls to the function.

**In the Real World**

*Although obsessing about performance is a game programmer’s favorite hobby, there’s a danger in focusing too much on speed. In fact, the approach many developers take is to first get their game programs working well before they tweak for small performance gains. At that point, programmers will* ***profile*** *their code by running a utility (a profiler) that analyzes where the game program spends its time. If a programmer sees bottlenecks, he or she might consider hand optimizations such as function inlining.*

**INTRODUCING THE MAD LIB GAME**

The Mad Lib game asks for the user’s help in creating a story. The user supplies the name of a person, a plural noun, a number, a body part, and a verb. The program takes all of this information and uses it to create a personalized story.

// Mad-Lib

// Creates a story based on user input

**#include <iostream>**

**#include <string>**

**using namespace std;**

**string askText(string prompt);**

**int askNumber(string prompt);**

**void tellStory(string name, string noun, int number, string bodyPart, string verb);**

**int main()**

**{**

**cout << "Welcome to Mad Lib.\n\n";**

**cout << "Answer the following questions to help create a new story.\n";**

**string name = askText("Please enter a name: ");**

**string noun = askText("Please enter a plural noun: ");**

**int number = askNumber("Please enter a number: ");**

**string bodyPart = askText("Please enter a body part: ");**

**string verb = askText("Please enter a verb: ");**

**tellStory(name, noun, number, bodyPart, verb);**

**return 0;**

**}**

**string askText(string prompt)**

**{**

**string text;**

**cout << prompt;**

**cin >> text;**

**return text;**

**}**

**int askNumber(string prompt)**

**{**

**int num;**

**cout << prompt;**

**cin >> num;**

**return num;**

**}**

**void tellStory(string name, string noun, int number, string bodyPart, string verb)**

**{**

**cout << "\nHere's your story:\n";**

**cout << "The famous explorer ";**

**cout << name;**

**cout << " had nearly given up a life-long quest to find\n";**

**cout << "The Lost City of ";**

**cout << noun;**

**cout << " when one day, the ";**

**cout << noun;**

**cout << " found the explorer.\n";**

**cout << "Surrounded by ";**

**cout << number;**

**cout << " " << noun;**

**cout << ", a tear came to ";**

**cout << name << "'s ";**

**cout << bodyPart << ".\n";**

**cout << "After all this time, the quest was finally over. ";**

**cout << "And then, the ";**

**cout << noun << "\n";**

**cout << "promptly devoured ";**

**cout << name << ". ";**

**cout << "The moral of the story? Be careful what you ";**

**cout << verb;**

**cout << " for.";**

**}**

Welcome to Mad Lib.

Answer the following questions to help create a new story.

Please enter a name: Ozzy

Please enter a plural noun: papers

Please enter a number: 7

Please enter a body part: eye

Please enter a verb: walk

Here's your story:

The famous explorer Ozzy had nearly given up a life-long quest to find

The Lost City of papers when one day, the papers found the explorer.

Surrounded by 7 papers, a tear came to Ozzy's eye.

After all this time, the quest was finally over. And then, the papers

promptly devoured Ozzy. The moral of the story? Be careful what you walk for.

**The main() Function**

The **main()** function calls all of the other functions. It calls the function **askText()** to get a **name**, **plural** **noun**, **body part**, and **verb** from the user. It calls **askNumber()** to get a number from the user. It calls **tellStory()** with all of the user-supplied information to generate and display the story.

**The askText() Function**

The **askText ()** functiongets a **string** from the user. The function is versatile and takes a parameter of type **string** , which it uses to prompt the user. Because of this, I’m able to call this single function to ask the user for a variety of different pieces of information, including a **name**, **plural** **noun**, **body** **part**, and **verb**.

**Trap**

*Remember that this simple use of* **cin** *works only with strings that have no white space in them (such as tabs or spaces). So when a user is prompted for a body part, he can enter* **bellybutton** *, but* **medulla****oblongata** *will cause a problem for the program.*

*There are ways to compensate for this, but that really requires a discussion of something called streams, which is beyond the scope of this book. So use* **cin** *in this way, but just be aware of its limitations.*

**The askNumber() Function**

The **askNumber()** function gets an integer from the user. Although I only call it once in the program, it’s versatile because it takes a parameter of type string that it uses to prompt the user.

**The tellStory() Function**

The **tellStory()** function takes all of the information entered by the user and uses it to display a personalized story.

**SUMMARY**

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

* Functions allow you to break up your programs into manageable chunks.
* One way to declare a function is to write a function prototype - code that lists the return value, name, and parameter types of a function.
* Defining a function means writing all the code that makes the function tick.
* You can use the **return** statement to return a value from a function. You can also use return to end a function that has void as its return type.
* A variable’s scope determines where the variable can be seen in your program.
* Global variables are accessible from any part of your program. In general, you should try to limit your use of global variables.
* Global constants are accessible from any part of your program. Using global constants can make your program code clearer.
* Default arguments are assigned to a parameter if no value for the parameter is specified in the function call.
* Function overloading is the process of creating multiple definitions for the same function, each of which has a different set of parameters.
* Function inlining is the process of asking the compiler to inline a function - meaning that the compiler should make a copy of the function everywhere in the code where the function is called. Inlining very small functions can sometimes yield a performance boost.

**CHAPTER 6**

**REFERENCES: TIC-TAC-TOE**

**USING REFERENCES**

A reference provides ***another name*** for a variable. Whatever you do to a reference is done to the variable to which it refers. You can think of a reference as a nickname for a variable - another name that the variable goes by. In the first program in this chapter, I’ll show you how to create references. Then, in the next few programs, I’ll show you why you’d want to use references and how they can improve your game programs.

**Introducing the Referencing Program**

The Referencing program demonstrates references. The program declares and initializes a variable to hold a score and then creates a reference that refers to the variable. The program displays the score using the variable and the reference to show that they access the same single value. Next, the program shows that this single value can be altered through either the variable or the reference.

**References**

// Referencing

// Demonstrates using references

**#include <iostream>**

**using namespace std;**

**int main()**

**{**

**int myScore = 1000;**

**int& mikesScore = myScore;** // create a reference

**cout << "myScore is: " << myScore << "\n";**

**cout << "mikesScore is: " << mikesScore << "\n\n";**

**cout << "Adding 500 to myScore\n";**

**myScore += 500;**

myScore is: 1000

mikesScore is: 1000

Adding 500 to myScore

myScore is: 1500

mikesScore is: 1500

Adding 500 to mikesScore

myScore is: 2000

mikesScore is: 2000

**cout << "myScore is: " << myScore << "\n";**

**cout << "mikesScore is: " << mikesScore << "\n\n";**

**cout << "Adding 500 to mikesScore\n";**

**mikesScore += 500;**

**cout << "myScore is: " << myScore << "\n";**

**cout << "mikesScore is: " << mikesScore << "\n\n";**

**return 0;**

**}**

**Creating References**

The first thing I do in **main()** is create a variable to hold my score.

**int myScore = 1000;**

Then I create a reference that refers to **myScore**.

**int& mikesScore = myScore;** //create a reference

The preceding line declares and initializes **mikesScore** , a reference that refers to **myScore**. **mikesScore** is an alias for **myScore**. **mikesScore** does not hold its own **int** value; it’s simply another way to get at the **int** value that **myScore** holds.

To declare and initialize a reference, start with the type of value to which the reference will refer, followed by the reference operator ( **&** ), followed by the reference name, followed by **=** , followed by the variable to which the reference will refer.

**Trick**

*Sometimes programmers prefix a reference name with the letter “****r****” to remind them that they’re working with a reference. A programmer might include the following lines:*

**int playerScore = 1000;**

**int& rScore = playerScore;**

One way to understand references is to think of them as nicknames. For example, suppose you’ve got a friend named ***Eugene***, and he (understandably) asks to be called by a nickname - ***Gibby*** (not much of an improvement, but it’s what Eugene wants). So when you’re at a party with your friend, you can call him over using either ***Eugene*** or ***Gibby***. Your friend is only one person, but you can call him using either his name or a nickname. This is the same as how a variable and a reference to that variable work. You can get to a single value stored in a variable by using its variable name or the name of a reference to that variable. Finally, whatever you do, try not to name your variables ***Eugene*** - for their sakes.

**Trap**

*Because a reference must always refer to another value,* ***you must initialize the reference when you***

***declare it****. If you don’t, you’ll get a compile error. The following line is quite illegal:*

**int& mikesScore;** //don’t try this at home!

**Accessing Referenced Values**

Next, I send both **myScore** and **mikesScore** to **cout**.

**cout << "myScore is: " << myScore << "\n";**

**cout << "mikesScore is: " << mikesScore << "\n\n";**

Both lines of code display **1000** because they each ***access the same single chunk of memory*** that stores the number **1000**. Remember, there is only one value, and it is stored in the variable **myScore**. **mikesScore** simply provides another way to get to that value.

**Altering Referenced Values**

Next, I increase the value of **myScore** by **500**.

**myScore += 500;**

When I send **myScore** to **cout**, **1500** is displayed, just as you’d expect. When I send **mikesScore** to **cout**, **1500** is also displayed. Again, that’s because **mikesScore** is just another name for the variable **myScore**. In essence, I’m sending the same variable to cout both times.

Next, I increase **mikesScore** by **500**.

**mikesScore += 500;**

Because **mikesScore** is just another name for myScore, the preceding line of code increases the value of **myScore** by **500**. So when I next send **myScore** to **cout**, **2000** is displayed. When I send **mikesScore** to **cout**, **2000** is displayed again.

**Trap**

*A reference always refers to the variable with which it was initialized****. You can’t reassign a reference to refer to another variable*** *so, for example, the results of the following code might not be obvious.*

**int myScore = 1000;**

**int& mikesScore = myScore;**

**int larrysScore = 2500;**

**mikesScore = larrysScore;** // may not do what you think!

*The line* **mikesScore = larrysScore;** *does not reassign the reference* **mikesScore** *so it refers to* **larrysScore** *because a reference can’t be reassigned. However, because* **mikesScore** *is just another name for* **myScore** *, the code* **mikesScore = larrysScore;** *is equivalent to* **myScore = larrysScore;** *, which assigns* **2500** *to* **myScore** *. And after all is said and done,* **myScore** *becomes* **2500** *and* **mikesScore** *still refers to* **myScore***.*

**PASSING REFERENCES TO ALTER ARGUMENTS**

Now that you’ve seen how references work, you might be wondering why you’d ever use them. Well, references come in quite handy when you are passing variables to functions because when you pass a variable to a function, the function gets a copy of the variable. This means that the original variable you passed (called the ***argument variable***) can’t be changed. Sometimes this might be exactly what you want because it keeps the argument variable safe and unalterable. But other times you might want to change an argument variable from inside the function to which it was passed. You can accomplish this by using references.

**Introducing the Swap Program**

The Swap program defines two variables - one that holds my pitifully low score and another that holds your impressively high score. After displaying the scores, the program calls a function meant to swap the scores. But because only copies of the score values are sent to the function, the argument variables that hold the scores are unchanged. Next, the program calls another swap function. This time, through the use of references, the argument variables’ values are successfully exchanged—giving me the great big score and leaving you with the small one.

**Passing references to alter argument variables**

// Swap

// Demonstrates passing references to alter argument variables

**#include <iostream>**

**using namespace std;**

**void badSwap(int x, int y);**

**void goodSwap(int& x, int& y);**

**int main()**

**{**

**int myScore = 150;**

**int yourScore = 1000;**

**cout << "Original values\n";**

**cout << "myScore: " << myScore << "\n";**

**cout << "yourScore: " << yourScore << "\n\n";**

**cout << "Calling badSwap()\n";**

**badSwap(myScore, yourScore);**

**cout << "myScore: " << myScore << "\n";**

**cout << "yourScore: " << yourScore << "\n\n";**

**cout << "Calling goodSwap()\n";**

**goodSwap(myScore, yourScore);**

**cout << "myScore: " << myScore << "\n";**

**cout << "yourScore: " << yourScore << "\n";**

**return 0;**

**}**

**void badSwap(int x, int y)**

**{**

**int temp = x;**

**x = y;**

**y = temp;**

**}**

**void goodSwap(int& x, int& y)**

**{**

**int temp = x;**

**x = y;**

**y = temp;**

**}**

Original values

myScore: 150

yourScore: 1000

Calling badSwap()

myScore: 150

yourScore: 1000

Calling goodSwap()

myScore: 1000

yourScore: 150

**Passing by Value**

After declaring and initializing **myScore** and **yourScore**, I send them to **cout**. As you’d expect, **150** and **1000** are displayed. Next, I call **badSwap()**.

When you specify a parameter the way you’ve seen so far (as an ordinary variable, not as a reference), you’re indicating that the argument for that parameter will be ***passed by value***, meaning that the parameter will get a ***copy of the argument* *variable*** and ***not access to the argument variable itself***. By looking at the function header of **badSwap()**, you can tell that a call to the function passes both arguments by value.

**void badSwap(int x, int y)**

This means that when I call **badSwap()** with the following line, ***copies*** of **myScore** and **yourScore** are sent to the parameters, **x** and **y**.

**badSwap(myScore, yourScore);**

Specifically, **x** is assigned **150** and **y** is assigned **1000**. As a result, nothing I do with **x** and **y** in the function **badSwap()** will have any effect on **myScore** and **yourScore**.

When the guts of **badSwap()** execute, **x** and **y** do exchange values – **x** becomes **1000** and **y** becomes **150**. However, when the function ends, both **x** and **y** go out of scope and cease to exist. Control then returns to **main()**, where **myScore** and **yourScore** ***haven’t changed***. Then, when I send **myScore** and **yourScore** to **cout**, **150** and **1000** are displayed again. Sadly, I still have the small score and you still have the large one.

**Passing by Reference**

It’s possible to give a function access to an argument variable by passing a parameter a reference to the argument variable. As a result, anything done to the parameter will be done to the argument variable. To pass by reference, you must first declare the parameter as a reference.

You can tell that a call to **goodSwap()** passes both arguments by reference by looking at the function header.

**void goodSwap(int& x, int& y)**

This means that when I call **goodSwap()** with the following line, the parameter **x** will refer to **myScore**, and the parameter **y** will refer to **yourScore**.

**goodSwap(myScore, yourScore);**

This means that **x** is just another name for **myScore**, and **y** is just another name for **yourScore**. When **goodSwap()** executes and **x** and **y** exchange values, what really happens is that **myScore** and **yourScore** exchange values.

After the function ends, control returns to **main()**, where I send **myScore** and **yourScore** to **cout**. This time **1000** and **150** are displayed. The variables have exchanged values. I’ve taken the large score and left you with the small one. Success at last!

**PASSING REFERENCES FOR EFFICIENCY**

Passing a variable by value creates some overhead because you must copy the variable before you assign it to a parameter. When we’re talking about variables of simple, builtin types, such as an **int** or a **float**, the overhead is negligible. But a large object, such as one that represents an entire 3D world, could be expensive to copy. Passing by reference, on the other hand, is efficient because you don’t make a copy of an argument variable. Instead, you simply provide access to the existing object through a reference.

**Introducing the Inventory Displayer Program**

The Inventory Displayer program creates a vector of strings that represents a hero’s inventory. The program then calls a function that displays the inventory. The program passes the displayer function the vector of items as a reference, so it’s an efficient call; the vector isn’t copied. However, there’s a new wrinkle. The program passes the vector as a special kind of reference that prohibits the displayer function from changing the vector.

**Constant References**

// Inventory Displayer

// Demonstrates constant references

**#include <iostream>**

**#include <string>**

**#include <vector>**

**using namespace std;**

//parameter vec is a constant reference to a vector of strings

**void display(const vector<string>& inventory);**

**int main()**

**{**

**vector<string> inventory;**

**inventory.push\_back("sword");**

**inventory.push\_back("armor");**

**inventory.push\_back("shield");**

**display(inventory);**

**return 0;**

**}**

//parameter vec is a constant reference to a vector of strings

**void display(const vector<string>& vec)**

**{**

**cout << "Your items:\n";**

**for (vector<string>::const\_iterator iter = vec.begin();**

**iter != vec.end(); ++iter)**

**{**

**cout << \*iter << endl;**

**}**

**}**

Your items:

sword

armor

shield

**Understanding the Pitfalls of Reference Passing**

One way to efficiently give a function access to a large object is to pass it by reference. However, this introduces a potential problem. As you saw in the Swap program, it opens up an argument variable to being changed. But what if you don’t want to change the argument variable? Is there a way to take advantage of the efficiency of passing by reference while protecting an argument variable’s integrity? Yes, there is. The answer is to pass a constant reference.

**Hint**

*In general, you should avoid changing an argument variable. Try to write functions that send back new information to the calling code through a return value.*

**Declaring Parameters as Constant References**

The function **display()** shows the contents of the hero’s inventory. In the function’s header I specify one parameter - a constant reference to a vector of **string** objects named **vec**.

**void display(const vector<string>& vec)**

A ***constant reference***is a restricted reference. It acts like any other reference, except you can’t use it to change the value to which it refers. To create a constant reference, simply put the keyword **const** before the type in the reference declaration.

What does this all mean for the function **display()**? Because the parameter **vec** is a constant reference, it means **display()** can’t change **vec**. In turn, this means that inventory is safe; it can’t be changed by **display()**. In general, you can efficiently pass an argument to a function as a constant reference so it’s accessible, but not changeable. It’s like providing the function ***read-only*** access to the argument. Although constant references are very useful for specifying function parameters, you can use them anywhere in your program.

**Hint**

*A constant reference comes in handy in another way. If you need to assign a constant value to a reference, you have to assign it to a constant reference. (A non-constant reference won’t do.)*

**Passing a Constant Reference**

Back in **main()**, I create inventory and then call **display()** with the following line, which passes the vector as a constant reference.

**display(inventory);**

This results in an efficient and safe function call. It’s efficient because only a reference is passed; the vector is not copied. It’s safe because the reference to the vector is a constant reference; inventory can’t be changed by **display()**.

**Trap**

*You can’t modify a parameter marked as a constant reference. If you try, you’ll generate a compile error.*

Next, **display()** lists the elements in the vector using a constant reference to inventory. Then control returns to **main()** and the program ends.

**DECIDING HOW TO PASS ARGUMENTS**

At this point you’ve seen three different ways to pass arguments - ***by value***, as a ***reference***, and as a ***constant*** ***reference***. So how do you decide which method to use? Here are some guidelines:

* **By value.** Pass by value when an argument variable is one of the fundamental built-in types, such as **bool**, **int**, or **float**. Objects of these types are so small that passing by reference doesn’t result in any gain in efficiency. You should also pass by value when you want the computer to make a copy of a variable. You might want to use a copy if you plan to alter a parameter in a function, but you don’t want the actual argument variable to be affected.
* **As a constant reference.** Pass a constant reference when you want to efficiently pass a value that you don’t need to change.
* **As a reference.** Pass a reference only when you want to alter the value of the argument variable. However, you should try to avoid changing argument variables whenever possible.

**RETURNING REFERENCES**

Just like when you pass a value, when you return a value from a function, you’re really returning a copy of the value. Again, for values of the basic built-in types, this isn’t a big deal. However, it can be an expensive operation if you’re returning a large object. Returning a reference is an efficient alternative.

**Introducing the Inventory Referencer Program**

The Inventory Referencer program demonstrates returning references. The program displays the elements of a vector that holds a hero’s inventory by using returned references. Then the program changes one of the items through a returned reference.

**Returning a Reference**

// Inventory Referencer

// Demonstrates returning a reference

**#include <iostream>**

**#include <string>**

**#include <vector>**

**using namespace std;**

//returns a reference to a string

**string& refToElement(vector<string>& inventory, int i);**

**int main()**

**{**

**vector<string> inventory;**

**inventory.push\_back("sword");**

**inventory.push\_back("armor");**

**inventory.push\_back("shield");**

//displays string that the returned reference refers to

**cout << "Sending the returned reference to cout:\n";**

**cout << refToElement(inventory, 0) << "\n\n";**

//assigns one reference to another -- inexpensive assignment

**cout << "Assigning the returned reference to another reference.\n";**

**string& rStr = refToElement(inventory, 1);**

**cout << "Sending the new reference to cout:\n";**

**cout << rStr << "\n\n";**

//copies a string object -- expensive assignment

**cout << "Assigning the returned reference to a string object.\n";**

**string str = refToElement(inventory, 2);**

**cout << "Sending the new string object to cout:\n";**

**cout << str << "\n\n";**

//altering the string object through a returned reference

**cout << "Altering an object through a returned reference.\n";**

**rStr = "Healing Potion";**

**cout << "Sending the altered object to cout:\n";**

**cout << inventory[1] << endl;**

**return 0;**

**}**

//returns a reference to a string

**string& refToElement(vector<string>& vec, int i)**

**{**

**return vec[i];**

**}**

Sending the returned reference to cout:

sword

Assigning the returned reference to another reference.

Sending the new reference to cout:

armor

Assigning the returned reference to a string object.

Sending the new string object to cout:

shield

Altering an object through a returned reference.

Sending the altered object to cout:

Healing Potion

**Returning a Reference**

Before you can return a reference from a function, you must specify that you’re returning one. That’s what I do in the **refToElement()** function header.

**string& refToElement(vector<string>& inventory, int i)**

By using the reference operator in **string&** when I specify the return type, I’m saying that the function will return a reference to a string object (not a string object itself). You can use the reference operator as I did to specify that a function returns a reference to an object of a particular type. Simply put the reference operator after the type name.

The body of the function **refToElement()** contains only one statement, which returns a reference to the element at position **i** in the vector.

**return vec[i];**

Notice that there’s nothing in the return statement to indicate that the function returns a reference. The function header and prototype determine whether a function returns an object or a reference to an object.

**Trap**

*Although returning a reference can be an efficient way to send information back to a calling function, you have to be careful not to return a reference to an out-of-scope object - an object that ceases to exist. For example, the following function returns a reference to a string object that no longer exists after the function ends - and that’s illegal.*

**string& badReference()**

**{**

**string local = "This string will cease to exist once the function ends.";**

**return local;**

**}**

*One way to avoid this type of problem is to never return a reference to a local variable.*

**Displaying the Value of a Returned Reference**

After creating inventory, a vector of items, I display the first item through a returned reference.

**cout << refToElement(inventory, 0) << "\n\n";**

The preceding code calls **refToElement()**, which returns a reference to the element at position **0** of inventory and then sends that reference to **cout**. As a result, **sword** is displayed.

**Assigning a Returned Reference to a Reference**

Next, I assign a returned reference to another reference with the following line, which takes a reference to the element in position **1** of inventory and assigns it to **rStr**.

**string& rStr = refToElement(inventory, 1);**

This is an efficient assignment because assigning a reference to a reference does not involve the copying of an object. Then I send **rStr** to cout, and **armor** is displayed.

**Assigning a Returned Reference to a Variable**

Next, I assign a returned reference to a variable.

**string str = refToElement(inventory, 2);**

The preceding code doesn’t assign a reference to **str**. It can’t, because **str** is a **string** object. Instead, the code copies the element to which the returned reference refers (the element in position **2** of inventory) and assigns that new copy of the string object to **str**. Because this kind of assignment involves copying an object, it’s more expensive than assigning one reference to another. Sometimes the cost of copying an object this way is perfectly acceptable, but you should be aware of the extra overhead associated with this kind of assignment and avoid it when necessary.

Next, I send the new string object, **str**, to **cout**, and **shield** is displayed.

**Altering an Object through a Returned Reference**

You can also alter the object to which a returned reference refers. This means you can change the hero’s inventory through **rStr**, as in the following line of code.

**rStr = "Healing Potion";**

Because **rStr** refers to the element in position **1** of inventory, this code changes **inventory[1]** so it’s equal to “**Healing** **Potion**”. To prove it, I display the element using the following line, which does indeed show

**Healing** **Potion**.

**cout << inventory[1] << endl;**

If I want to protect inventory so a reference returned by **refToElement()** can’t be used to change the vector, I should specify the return type of the function as a constant reference.

**INTRODUCING THE TIC-TAC-TOE GAME**

In this chapter project, you’ll learn how to create a computer opponent using a dash of ***AI* (*Artificial Intelligence*)**. In the game, the player and computer square off in a high-stakes, man-versus-machine showdown of Tic-Tac-Toe. The computer plays a formidable (although not perfect) game and comes with

enough attitude to make any match fun.

**Planning the Game**

This game is your most ambitious project yet. You certainly have all the skills you need to create it, but I’m going to go through a longer planning section to help you get the big picture and understand how to create a larger program. Remember, the most important part of programming is planning to program. Without a roadmap, you’ll never get to where you want to go (or it’ll take you a lot longer as you travel the scenic route).

**In the Real World**

*Game designers work countless hours on concept papers, design documents, and prototypes before programmers write any game code. Once the design work is done, the programmers start their work – more planning. It’s only after programmers write their own technical designs that they then begin coding in earnest. The moral of this story? Plan. It’s easier to scrap a blueprint than a 50-story building.*

**Tic-Tac-Toe Game**

// Tic-Tac-Toe

// Plays the game of tic-tac-toe against a human opponent

**#include <iostream>**

**#include <string>**

**#include <vector>**

**#include <algorithm>**

**using namespace std;**

// global constants

**const char X = 'X';**

**const char O = 'O';**

**const char EMPTY = ' ';**

**const char TIE = 'T';**

**const char NO\_ONE = 'N';**

// function prototypes

**void instructions();**

**char askYesNo(string question);**

**int askNumber(string question, int high, int low = 0);**

**char humanPiece();**

**char opponent(char piece);**

**void displayBoard(const vector<char>& board);**

**char winner(const vector<char>& board);**

**bool isLegal(const vector<char>& board, int move);**

**int humanMove(const vector<char>& board, char human);**

**int computerMove(vector<char> board, char computer);**

**void announceWinner(char winner, char computer, char human);**

// main function

**int main()**

**{**

**int move;**

**const int NUM\_SQUARES = 9;**

**vector<char> board(NUM\_SQUARES, EMPTY);**

**instructions();**

**char human = humanPiece();**

**char computer = opponent(human);**

**char turn = X;**

**displayBoard(board);**

**while (winner(board) == NO\_ONE)**

**{**

**if (turn == human)**

**{**

**move = humanMove(board, human);**

**board[move] = human;**

**}**

**else**

**{**

**move = computerMove(board, computer);**

**board[move] = computer;**

**}**

**displayBoard(board);**

**turn = opponent(turn);**

**}**

**announceWinner(winner(board), computer, human);**

**return 0;**

**}**

// functions

**void instructions()**

**{**

**cout << "Welcome to the ultimate man-machine showdown: Tic-Tac-Toe.\n";**

**cout << "--where human brain is pit against silicon processor\n\n";**

**cout << "Make your move known by entering a number, 0 - 8. The number\n";**

**cout << "corresponds to the desired board position, as illustrated:\n\n";**

**cout << " 0 | 1 | 2\n";**

**cout << " ---------\n";**

**cout << " 3 | 4 | 5\n";**

**cout << " ---------\n";**

**cout << " 6 | 7 | 8\n\n";**

**cout << "Prepare yourself, human. The battle is about to begin.\n\n";**

**}**

**char askYesNo(string question)**

**{**

**char response;**

**do**

**{**

**cout << question << " (y/n): ";**

**cin >> response;**

**} while (response != 'y' && response != 'n');**

**return response;**

**}**

**int askNumber(string question, int high, int low)**

**{**

**int number;**

**do**

**{**

**cout << question << " (" << low << " - " << high << "): ";**

**cin >> number;**

**} while (number > high || number < low);**

**return number;**

**}**

**char humanPiece()**

**{**

**char go\_first = askYesNo("Do you require the first move?");**

**if (go\_first == 'y')**

**{**

**cout << "\nThen take the first move. You will need it.\n";**

**return X;**

**}**

**else**

**{**

**cout << "\nYour bravery will be your undoing... I will go first.\n";**

**return O;**

**}**

**}**

**char opponent(char piece)**

**{**

**if (piece == X)**

**return O;**

**else**

**return X;**

**}**

**void displayBoard(const vector<char>& board)**

**{**

**cout << "\n\t" << board[0] << " | " << board[1] << " | " << board[2];**

**cout << "\n\t" << "---------";**

**cout << "\n\t" << board[3] << " | " << board[4] << " | " << board[5];**

**cout << "\n\t" << "---------";**

**cout << "\n\t" << board[6] << " | " << board[7] << " | " << board[8];**

**cout << "\n\n";**

**}**

**char winner(const vector<char>& board)**

**{**

**const int WINNING\_ROWS[8][3] = { {0, 1, 2},** // all possible winning rows

**{3, 4, 5},**

**{6, 7, 8},**

**{0, 3, 6},**

**{1, 4, 7},**

**{2, 5, 8},**

**{0, 4, 8},**

**{2, 4, 6} };**

**const int TOTAL\_ROWS = 8;**

// if any winning row has three values that are the same (and not EMPTY), then we have a winner

**for(int row = 0; row < TOTAL\_ROWS; ++row)**

**{**

**if ( (board[WINNING\_ROWS[row][0]] != EMPTY) &&**

**(board[WINNING\_ROWS[row][0]] == board[WINNING\_ROWS[row][1]]) &&**

**(board[WINNING\_ROWS[row][1]] == board[WINNING\_ROWS[row][2]]) )**

**{**

**return board[WINNING\_ROWS[row][0]];**

**}**

**}**

// since nobody has won, check for a tie (no empty squares left)

**if (count(board.begin(), board.end(), EMPTY) == 0)**

**return TIE;**

**return NO\_ONE;** // since nobody has won and it isn't a tie, the game ain't over

**}**

**inline bool isLegal(int move, const vector<char>& board)**

**{**

**return (board[move] == EMPTY);**

**}**

**int humanMove(const vector<char>& board, char human)**

**{**

**int move = askNumber("Where will you move?", (board.size() - 1));**

**while (!isLegal(move, board))**

**{**

**cout << "\nThat square is already occupied, foolish human.\n";**

**move = askNumber("Where will you move?", (board.size() - 1));**

**}**

**cout << "Fine...\n";**

**return move;**

**}**

**int computerMove(vector<char> board, char computer)**

**{**

**unsigned int move = 0;**

**bool found = false;**

//if computer can win on next move, that’s the move to make

**while (!found && move < board.size())**

**{**

**if (isLegal(move, board))**

**{**

**board[move] = computer;** //try move

**found = winner(board) == computer;** //test for winner

**board[move] = EMPTY;** //undo move

**}**

**if (!found)**

**++move;**

**}**

//otherwise, if opponent can win on next move, that's the move to make

**if (!found)**

**{**

**move = 0;**

**char human = opponent(computer);**

**while (!found && move < board.size())**

**{**

**if (isLegal(move, board))**

**{**

**board[move] = human;** //try move

**found = winner(board) == human;** //test for winner

**board[move] = EMPTY;** //undo move

**}**

**if (!found)**

**++move;**

**}**

**}**

//otherwise, moving to the best open square is the move to make

**if (!found)**

**{**

**move = 0;**

**unsigned int i = 0;**

**const int BEST\_MOVES[] = {4, 0, 2, 6, 8, 1, 3, 5, 7};**

//pick best open square

**while (!found && i < board.size())**

**{**

**move = BEST\_MOVES[i];**

**if (isLegal(move, board))**

**{**

**found = true;**

**}**

**++i;**

**}**

**}**

**cout << "I shall take square number " << move << endl;**

**return move;**

**}**

**void announceWinner(char winner, char computer, char human)**

**{**

**if (winner == computer)**

**{**

**cout << winner << "'s won!\n";**

**cout << "As I predicted, human, I am triumphant once more -- proof\n";**

**cout << "that computers are superior to humans in all regards.\n";**

**}**

**else if (winner == human)**

**{**

**cout << winner << "'s won!\n";**

**cout << "No, no! It cannot be! Somehow you tricked me, human.\n";**

**cout << "But never again! I, the computer, so swear it!\n";**

**}**

**else**

**{**

**cout << "It's a tie.\n";**

**cout << "You were most lucky, human, and somehow managed to tie me.\n";**

**cout << "Celebrate... for this is the best you will ever achieve.\n";**

**}**

**}**

**Writing the Pseudocode**

It’s back to your favorite language that’s not really a language - pseudocode. Because I’ll use functions for most of the tasks in the program, I can afford to think about the code at a rather abstract level. Each line of pseudocode should feel like one function call. Later, all I’ll have to do is write the functions that the plan implies. Here’s the pseudocode:

Create an empty Tic-Tac-Toe board

Display the game instructions

Determine who goes first

Display the board

While nobody has won and it’s not a tie

If it’s the human’s turn

Get the human’s move

Update the board with the human’s move

Otherwise

Calculate the computer’s move

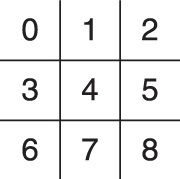
Update the board with the computer’s move

Display the board

Switch turns

Congratulate the winner or declare a tie

**Representing the Data**

All right, I’ve got a good plan, but it is rather abstract and talks about throwing around different elements that aren’t really defined in my mind yet. I see the idea of making a move as placing a piece on a game board. But how exactly am I going to represent the game board? Or a piece? Or a move?

Since I’m going to display the game board on the screen, why not just represent a piece as a single character - an X or an O? An empty piece could just be a space. Therefore, the board itself could be a vector of **char**s. There are nine squares on a Tic-Tac-Toe board, so the vector should have nine elements. Each square on the board will correspond to an element in the vector. Figure 6.6 illustrates what I mean.

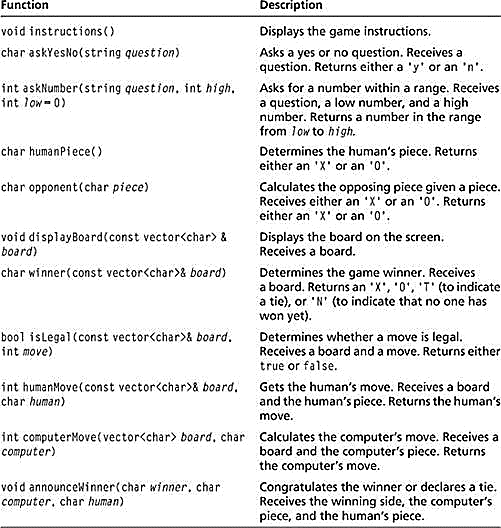
**Figure 6.6:** *Each square number corresponds to a position in the vector that represents the board.*

Each square or position on the board is represented by a number, **0–8**. That means the vector will have ***nine elements***, giving it position numbers **0–8**. Because each move indicates a square where a piece should be placed, a move is also just a number, **0–8**. That means a move could be represented as an **int**.

The side the player and computer play could also be represented by a **char** - either an ’ **X** ’ or an ’ **O** ’, just like a game piece. A variable to represent the side of the current turn would also be a **char**, either an ’ **X** ’ or an ’ **O** ’.

**Creating a List of Functions**

The pseudocode inspires the different functions I’ll need. I created a list of them, thinking about what each will do, what parameters they’ll have, and what values they’ll return. Table 6.1 shows the results of my efforts.



**Table 6.1** *Tic-Tac-Toe Functions*

**Setting Up the Program**

The first thing I do in the program is include the files I need, define some global constants, and write my function prototypes.

// Tic-Tac-Toe

// Plays the game of tic-tac-toe against a human opponent

**#include <iostream>**

**#include <string>**

**#include <vector>**

**#include <algorithm>**

**using namespace std;**

// global constants

**const char X = ‘X’;**

**const char O = ‘O’;**

**const char EMPTY = ‘ ‘;**

**const char TIE = ‘T’;**

**const char NO\_ONE = ‘N’;**

// function prototypes

**void instructions();**

**char askYesNo(string question);**

**int askNumber(string question, int high, int low = 0);**

**char humanPiece();**

**char opponent(char piece);**

**void displayBoard(const vector<char>& board);**

**char winner(const vector<char>& board);**

**bool isLegal(const vector<char>& board, int move);**

**int humanMove(const vector<char>& board, char human);**

**int computerMove(vector<char> board, char computer);**

**void announceWinner(char winner, char computer, char human);**

In the global constants section, **X** is shorthand for the char ‘**X**’, one of the two pieces in the game. **O** represents the char ‘**O**’, the other piece in the game. **EMPTY**, also a **char**, represents an empty square on the board. It’s a space because when it’s displayed, it will look like an empty square. **TIE** is a **char** that represents a tie game. And **NO\_ONE** is a **char** used to represent neither side of the game, which I use to indicate that no one has won yet.

**The main() Function**

As you can see, **the main()** function is almost exactly the pseudocode I created earlier.

// main function

**int main()**

**{**

**int move;**

**const int NUM\_SQUARES = 9;**

**vector<char> board(NUM\_SQUARES, EMPTY);**

**instructions();**

**char human = humanPiece();**

**char computer = opponent(human);**

**char turn = X;**

**displayBoard(board);**

**while (winner(board) == NO\_ONE)**

**{**

**if (turn == human)**

**{**

**move = humanMove(board, human);**

**board[move] = human;**

**}**

**else**

**{**

**move = computerMove(board, computer);**

**board[move] = computer;**

**}**

**displayBoard(board);**

**turn = opponent(turn);**

**}**

**announceWinner(winner(board), computer, human);**

**return 0;**

**}**

**The instructions() Function**

This function displays the game instructions and gives the computer opponent a little attitude.

**void instructions()**

**{**

**cout << "Welcome to the ultimate man-machine showdown: Tic-Tac-Toe.\n";**

**cout << "--where human brain is pit against silicon processor\n\n";**

**cout << "Make your move known by entering a number, 0 - 8. The number\n";**

**cout << "corresponds to the desired board position, as illustrated:\n\n";**

**cout << " 0 | 1 | 2\n";**

**cout << " ---------\n";**

**cout << " 3 | 4 | 5\n";**

**cout << " ---------\n";**

**cout << " 6 | 7 | 8\n\n";**

**cout << "Prepare yourself, human. The battle is about to begin.\n\n";**

**}**

**The askYesNo() Function**

This function asks a **yes** or **no** question. It keeps asking the question until the player enters either a **y** or an **n**. It receives a question and returns either a ‘**y**’ or an ‘**n**’.

**char askYesNo(string question)**

**{**

**char response;**

**do**

**{**

**cout << question << " (y/n): ";**

**cin >> response;**

**} while (response != 'y' && response != 'n');**

**return response;**

**}**

**The askNumber( ) Function**

This function asks for a number within a range and keeps asking until the player enters a valid number. It receives a question, a high number, and a low number. It returns a number within the range specified.

**int askNumber(string question, int high, int low)**

**{**

**int number;**

**do**

**{**

**cout << question << " (" << low << " - " << high << "): ";**

**cin >> number;**

**} while (number > high || number < low);**

**return number;**

**}**

If you take a look at this function’s prototype, you can see that the low number has a default value of **0** . I take advantage of this fact when I call the function later in the program.

**The** **humanPiece( )** **Function**

This function asks the player if he wants to go first, and returns the human’s piece based on that choice. As the great tradition of Tic–Tac–Toe dictates, the **X** goes first.

**char humanPiece()**

**{**

**char go\_first = askYesNo("Do you require the first move?");**

**if (go\_first == 'y')**

**{**

**cout << "\nThen take the first move. You will need it.\n";**

**return X;**

**}**

**else**

**{**

**cout << "\nYour bravery will be your undoing... I will go first.\n";**

**return O;**

**}**

**}**

**The opponent() Function**

This function gets a piece (either an ’**X**’ or an ’**0**’) and returns the opponent’s piece (either an ’**X**’ or an ’**0**’).

**char opponent(char piece)**

**{**

**if (piece == X)**

**{**

**return O;**

**}**

**else**

**{**

**return X;**

**}**

**}**

**The displayBoard() Function**

This function displays the board passed to it. Because each element in the board is a space, an ’**X**’, or an ’**0**’, the function can display each one. I use a few other characters on my keyboard to draw a decent–looking Tic–Tac–Toe board.

**void displayBoard(const vector<char>& board)**

**{**

**cout << "\n\t" << board[0] << " | " << board[1] << " | " << board[2];**

**cout << "\n\t" << "---------";**

**cout << "\n\t" << board[3] << " | " << board[4] << " | " << board[5];**

**cout << "\n\t" << "---------";**

**cout << "\n\t" << board[6] << " | " << board[7] << " | " << board[8];**

**cout << "\n\n";**

**}**

Notice that the vector that represents the board is passed through a constant reference. This means that the vector is passed efficiently; it is not copied. It also means that the vector is safeguarded against any changes. Since I plan to simply display the board and not change it in this function, this is perfect.

**The** **winner()** **Function**

This function receives a board and returns the winner. There are four possible values for a winner. The function will return either **X** or **O** if one of the players has won. If every square is filled and no one has won, it returns **TIE** . Finally, if no one has won and there is at least one empty square, the function returns **NO\_ONE**`

**char winner(const vector<char>& board)**

**{**

// all possible winning rows

**const int WINNING\_ROWS[8][3] = { {0, 1, 2},**

**{3, 4, 5},**

**{6, 7, 8},**

**{0, 3, 6},**

**{1, 4, 7},**

**{2, 5, 8},**

**{0, 4, 8},**

**{2, 4, 6} };**

The first thing to notice is that the vector that represents the board is passed through a constant reference. This means that the vector is passed efficiently; it is not copied. It also means that the vector is safeguarded against any change.

In this initial section of the function, I define a constant, two–dimensional array of **int**s called **WINNING\_ROWS**, which represents all eight ways to get three in a row and win the game. Each winning row is represented by a group of three numbers - three board positions that form a winning row. For example, the group **{0, 1, 2}** represents the top row - board positions **0**, **1**, and **2**. The next group, **{3, 4, 5}**, represents the middle row – board positions **3**, **4**, and **5**. And so on....

Next, I check to see whether either player has won.

**const int TOTAL\_ROWS = 8;**

// if any winning row has three values that are the same (and not EMPTY),

// then we have a winner

**for(int row = 0; row < TOTAL\_ROWS; ++row)**

**{**

**if ( (board[WINNING\_ROWS[row][0]] != EMPTY) &&**

**(board[WINNING\_ROWS[row][0]] == board[WINNING\_ROWS[row][1]]) &&**

**(board[WINNING\_ROWS[row][1]] == board[WINNING\_ROWS[row][2]]) )**

**{**

**return board[WINNING\_ROWS[row][0]];**

**}**

**}**

I loop through each possible way a player can win to see whether either player has three in a row. The **if** statement checks to see whether the three squares in question all contain the same value and are not **EMPTY**. If so, it means that the row has either three **X**s or three **O**s in it, and one side has won. The function then returns the piece in the first position of this winning row.

If neither player has won, I check for a tie game.

// since nobody has won, check for a tie (no empty squares left)

**if (count(board.begin(), board.end(), EMPTY) == 0)**

**return TIE;**

If there are no empty squares on the board, then the game is a tie. I use the **STL** **count()** algorithm, which counts the number of times a given value appears in a group of container elements, to count the number of **EMPTY** elements in board. If the number is equal to **0**, the function returns **TIE**.

Finally, if neither player has won and the game isn’t a tie, then there is no winner yet. Thus, the function returns **NO\_ONE**.

// since nobody has won and it isn't a tie, the game ain't over

**return NO\_ONE;**

**}**

**The isLegal() Function**

This function receives a board and a move. It returns **true** if the move is a legal one on the board or **false** if the move is not legal. A legal move is represented by the number of an empty square.

**inline bool isLegal(int move, const vector<char>& board)**

**{**

**return (board[move] == EMPTY);**

**}**

Again, notice that the vector that represents the board is passed through a constant reference. This means that the vector is passed efficiently; it is not copied. It also means that the vector is safeguarded against any change.

You can see that I inlined **isLegal()**. Modern compilers are quite good at optimizing on their own; however, since this function is just one line, it’s a good candidate for inlining.

**The** **humanMove()** **Function**

This next function receives a board and the human’s piece. It returns the square number for where the player wants to move. The function asks the player for the square number to which he wants to move until the response is a legal move. Then the function returns the move.

**int humanMove(const vector<char>& board, char human)**

**{**

**int move = askNumber("Where will you move?", (board.size() - 1));**

**while (!isLegal(move, board))**

**{**

**cout << "\nThat square is already occupied, foolish human.\n";**

**move = askNumber("Where will you move?", (board.size() - 1));**

**}**

**cout << "Fine...\n";**

**return move;**

**}**

Again, notice that the vector that represents the board is passed through a constant reference. This means that the vector is passed efficiently; it is not copied. It also means that the vector is safeguarded against any change.

**The computerMove() Function**

This function receives the board and the computer’s piece. It returns the computer’s move. The first thing to notice is that I do not pass the board by reference.

**int computerMove(vector<char> board, char computer)**

Instead, I choose to pass by value, even though it’s not as efficient as passing by reference. I pass by value because I need to work with and modify a copy of the board as I place pieces in empty squares to determine the best computer move. By working with a copy, I keep the original vector that represents the board safe.

Now on to the guts of the function. Okay, how do I program a bit of AI so the computer puts up a decent fight? Well, I came up with a basic three–step strategy for choosing a move.

1. If the computer can win on this move, make that move.
2. Otherwise, if the human can win on his next move, block him.
3. Otherwise, take the best remaining open square. The best square is the center. The next best squares are the corners, and then the rest of the squares.

The next section of the function implements Step 1.

**{**

**unsigned int move = 0;**

**bool found = false;**

//if computer can win on next move, that’s the move to make

**while (!found && move < board.size())**

**{**

**if (isLegal(move, board))**

**{**

**board[move] = computer;** //try move

**found = winner(board) == computer;** //test for winner

**board[move] = EMPTY;** //undo move

**}**

**if (!found)**

**++move;**

**}**

I begin to loop through all of the possible moves, **0–8.** For each move, I test to see whether the move is legal. If it is, I put the computer’s piece in the corresponding square and check to see whether the move gives the computer a win. Then I undo the move by making that square empty again. If the move didn’t result in a win for the computer, I go on to the next empty square. However, if the move did give the computer a win, then the loop ends – and I’ve found the move ( found is **true** ) that I want the computer to make (square number move ) to win the game.

Next, I check to see if I need to go on to Step 2 of my AI strategy. If I haven’t found a move yet ( found is **false** ), then I check to see whether the human can win on his next move.

//otherwise, if opponent can win on next move, that's the move to make

**if (!found)**

**{**

**move = 0;**

**char human = opponent(computer);**

**while (!found && move < board.size())**

**{**

**if (isLegal(move, board))**

**{**

**board[move] = human;** //try move

**found = winner(board) == human;** //test for winner

**board[move] = EMPTY;** //undo move

**}**

**if (!found)**

**++move;**

**}**

**}**

I begin to loop through all of the possible moves, **0–8**. For each move, I test to see whether the move is legal. If it is, I put the human’s piece in the corresponding square and check to see whether the move gives the human a win. Then I undo the move by making that square empty again. If the move didn’t result in a win for the human, I go on to the next empty square. However, if the move did give the human a win, then the loop ends - and I’ve found the move ( found is **true** ) that I want the computer to make (square number move) to block the human from winning on his next move.

Next, I check to see if I need to go on to Step 3 of my AI strategy. If I haven’t found a move yet ( found is **false** ) then I look through the list of best moves, in order of desirability, and take the first legal one.

//otherwise, moving to the best open square is the move to make

**if (!found)**

**{**

**move = 0;**

**unsigned int i = 0;**

**const int BEST\_MOVES[] = {4, 0, 2, 6, 8, 1, 3, 5, 7};**

**while (!found && i < board.size())** //pick best open square

**{**

**move = BEST\_MOVES[i];**

**if (isLegal(move, board))**

**found = true;**

**++i;**

**}**

**}**

At this point in the function, I’ve found the move I want the computer to make - whether that’s a move that gives the computer a win, blocks a winning move for the human, or is simply the best empty square available. So, I have the computer announce the move and return the corresponding square number.

**cout << "I shall take square number " << move << endl;**

**return move;**

**}**

**In the Real World**

*The Tic–Tac–Toe game considers only the next possible move. Programs that play serious games of strategy, such as chess, look far deeper into the consequences of individual moves and consider many levels of moves and countermoves. In fact, good computer chess programs can consider literally millions of board positions before making a move.*

**The announceWinner() Function**

This function receives the winner of the game, the computer’s piece, and the human’s piece. The function announces the winner or declares a tie.

**void announceWinner(char winner, char computer, char human)**

**{**

**if (winner == computer)**

**{**

**cout << winner << "'s won!\n";**

**cout << "As I predicted, human, I am triumphant once more -- proof\n";**

**cout << "that computers are superior to humans in all regards.\n";**

**}**

**else if (winner == human)**

**{**

**cout << winner << "'s won!\n";**

**cout << "No, no! It cannot be! Somehow you tricked me, human.\n";**

**cout << "But never again! I, the computer, so swear it!\n";**

**}**

**else**

**{**

**cout << "It's a tie.\n";**

**cout << "You were most lucky, human, and somehow managed to tie me.\n";**

**cout << "Celebrate... for this is the best you will ever achieve.\n";**

**}**

**}**

**SUMMARY**

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

* A reference is an alias; it’s another name for a variable.
* You create a reference using **&** - the referencing operator.
* A reference must be initialized when it’s defined.
* A reference can’t be changed to refer to a different variable.
* Whatever you do to a reference is done to the variable to which the reference refers.
* When you assign a reference to a variable, you create a new copy of the referenced value.
* When you pass a variable to a function by value, you pass a copy of the variable to the function.
* When you pass a variable to a function by reference, you pass access to the variable.
* Passing by reference can be more efficient than passing by value, especially when you are passing large objects.
* Passing a reference provides direct access to the argument variable passed to a function. As a result, the function can make changes to the argument variable.
* A constant reference can’t be used to change the value to which it refers. You declare a constant reference by using the keyword const .
* You can’t assign a constant reference or a constant value to a non-constant reference.
* Passing a constant reference to a function protects the argument variable from being changed by that function.
* Changing the value of an argument variable passed to a function can lead to confusion, so game programmers consider passing a constant reference before passing a non–constant reference.
* Returning a reference can be more efficient than returning a copy of a value, especially when you are returning large objects.
* You can return a constant reference to an object so the object can’t be changed through the returned reference.
* A basic technique of game AI is to have the computer consider all of its legal moves and all of its opponent’s legal replies before deciding which move to take next.

**CHAPTER 7**

**POINTERS: TIC-TAC-TOE 2.0**

Pointers are a powerful part of C++. In some ways, they behave like iterators from the STL. Often you can use them in place of references. But pointers offer functionality that no other part of the language can. In this chapter, you’ll learn the basic mechanics of pointers and get an idea of what they’re good for. Specifically, you’ll learn to:

* Declare and initialize pointers
* Dereference pointers
* Use constants and pointers
* Pass and return pointers
* Work with pointers and arrays

**UNDERSTANDING POINTER BASICS**

Pointers have a reputation for being difficult to understand. In reality, the essence of pointers is quite simple—a ***pointer***is a variable that can contain a memory address. Pointers give you the ability to work directly and efficiently with computer memory. Like iterators from the STL, they’re often used to access the contents of other variables. But before you can put pointers to good use in your game programs, you have to understand the basics of how they work.

**Hint**

*Computer memory is a lot like a neighborhood, but instead of houses in which people store their stuff, you have memory locations where you can store data. Just like a neighborhood where houses sit side by side, labeled with addresses, chunks of computer memory sit side by side, labeled with addresses. In a neighborhood, you can use a slip of paper with a street address on it to get to a particular house (and to the stuff stored inside it). In a computer, you can use a pointer with a memory address in it to get to a particular memory location (and to the stuff stored inside it).*

**Introducing the Pointing Program**

The Pointing program demonstrates the mechanics of pointers. The program creates a variable for a score and then creates a pointer to store the address of that variable. The program shows that you can change the value of a variable directly, and the pointer will reflect the change. It also shows that you can change the value of a variable through a pointer. It then demonstrates that you can change a pointer to point to another variable entirely. Finally, the program shows that pointers can work just as easily with objects.

**Using Pointers**

// Pointing

// Demonstrates using pointers

**#include <iostream>**

**#include <string>**

**using namespace std;**

**int main()**

**{**

**int\* pAPointer;** //declare a pointer

**int\* pScore = 0;** //declare and initialize a pointer

**int score = 1000;**

**pScore = &score;** //assign pointer pScore address of a variable score

**cout << "Assigning &score to pScore\n";**

**cout << "&score is: " << &score << "\n";** //address of score variable

**cout << "pScore is: " << pScore << "\n";** //address stored in pointer

**cout << "score is: " << score << "\n";**

**cout << "\*pScore is: " << \*pScore << "\n\n";** //value pointed to by pointer

**Assigning &score to pScore**

**&score is: 0x28ff20**

**pScore is: 0x28ff20**

**score is: 1000**

**\*pScore is: 1000**

**Adding 500 to score**

**score is: 1500**

**\*pScore is: 1500**

**Adding 500 to \*pScore**

**score is: 2000**

**\*pScore is: 2000**

**Assigning &newScore to pScore**

**&newScore is: 0x28ff1c**

**pScore is: 0x28ff1c**

**newScore is: 5000**

**\*pScore is: 5000**

**Assigning &str to pStr**

**str is: score**

**\*pStr is: score**

**(\*pStr).size() is: 5**

**pStr->size() is: 5**

**cout << "Adding 500 to score\n";**

**score += 500;**

**cout << "score is: " << score << "\n";**

**cout << "\*pScore is: " << \*pScore << "\n\n";**

**cout << "Adding 500 to \*pScore\n";**

**\*pScore += 500;**

**cout << "score is: " << score << "\n";**

**cout << "\*pScore is: " << \*pScore << "\n\n";**

**cout << "Assigning &newScore to pScore\n";**

**int newScore = 5000;**

**pScore = &newScore;**

**cout << "&newScore is: " << &newScore << "\n";**

**cout << "pScore is: " << pScore << "\n";**

**cout << "newScore is: " << newScore << "\n";**

**cout << "\*pScore is: " << \*pScore << "\n\n";**

**cout << "Assigning &str to pStr\n";**

**string str = "score";**

**string\* pStr = &str;** //pointer to string object

**cout << "str is: " << str << "\n";**

**cout << "\*pStr is: " << \*pStr << "\n";**

**cout << "(\*pStr).size() is: " << (\*pStr).size() << "\n";**

**cout << "pStr->size() is: " << pStr->size() << "\n";**

**return 0;**

**}**

**Declaring Pointers**

With the first statement in **main()** I declare a pointer named **pAPointer**.

**int\* pAPointer;** //declare a pointer

Because pointers work in such a unique way, programmers often prefix pointer variable names with the letter “**p**” to remind them that the variable is indeed a pointer.

Just like an iterator, a pointer is declared to point to a specific type of value. **pAPointer** is a pointer to **int**, which means that it can only point to an int value. **pAPointer** can’t point to a **float** or a **char**, for example. Another way to say this is that **pAPointer** can only store the address of an **int**.

To declare a pointer of your own, begin with the type of object to which the pointer will point, followed by an asterisk, followed by the pointer name. When you declare a pointer, you can put whitespace on either side of the asterisk. So **int\*** **pAPointer;**, **int** **\*pAPointer;**, and **int \* pAPointer**; all declare a pointer named **pAPointer**.

**Trap**

*When you declare a pointer, the asterisk only applies to the single variable name that immediately follows it. So the following statement declares* ***pScore*** *as a pointer to* ***int*** *and* ***score*** *as an* ***int****.*

***int\* pScore****,* ***score;***

***score*** *is not a pointer! It’s a variable of type* ***int****. One way to make this clearer is to play with the*

*whitespace and rewrite the statement as:*

***int \*pScore****,* ***score;***

*However, the clearest way to declare a pointer is to declare it in its own statement, as in the following*

*lines.*

***int\* pScore;***

***int score;***

**Initializing Pointers**

As with other variables, you can initialize a pointer in the same statement you declare it. That’s what I do next with the following line, which assigns **0** to **pScore**.

**int\* pScore = 0;** //declare and initialize a pointer

Assigning **0** to a pointer has special meaning. Loosely translated, it means, “***Point to nothing***.” Programmers call a pointer with the value of zero a ***null pointer***. You should always initialize a pointer with some value when you declare it, even if that value is zero.

**Hint**

*Many programmers assign* **NULL** *to a pointer instead of* **0** *to make the pointer a null pointer.* **NULL** *is a constant defined in multiple library files, including* **iostream***.*

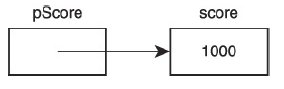
**Assigning Addresses to Pointers**

Because pointers store addresses of objects, you need a way to get addresses into the pointers. One way to do that is to get the memory address of an existing variable and assign it to a pointer. That’s what I do in the following line, which gets the address of the variable score and assigns it to **pScore**.

**pScore = &score;** //assign pointer address of variable score

I get the address of **score** by preceding the variable name with **&**, the ***address*** *of* operator. (Yes, you’ve seen the **&** symbol before, when it was used as the reference operator. However, in this context, the **&** symbol gets the address of an object.)

As a result of the preceding line of code, pS**c**ore contains the address of **score**. It’s as if **pScore** knows exactly where score is located in the computer’s memory. This means you can use **pScore** to get to **score** and manipulate the value stored in **score**. Figure 7.2 serves as a visual illustration of the relationship between **pScore** and **score**.



***Figure 7.2***

*The pointer* ***pScore*** *points to* ***score****, which stores the value* ***1000****.*

To prove that **pScore** contains the address of **score**, I display the address of the variable and the value of the pointer with the following lines.

**cout << "&score is: " << &score << "\n";** //address of score variable

**cout << "pScore is: " << pScore << "\n";** //address stored in pointer

As you can see from the program output, **pScore** contains **0x28ff20**, which is the address of **score**. (The specific addresses displayed by the Pointing program might be different on your system. The important thing is that the values for **pScore** and &**score** are the same.)

**Dereferencing Pointers**

Just as you dereference an iterator to access the object to which it refers, you dereference a pointer to access the object to which it points. You accomplish the dereferencing the same way—with **\***, the dereference operator. I put the dereference operator to work with the following line, which displays **1000** because **\*pScore** accesses the value stored in score.

**cout << "\*pScore is: " << \*pScore << "\n\n";** //value pointed to by pointer

Remember, **\*pScore** means, “the object to which **pScore** points.”

**Trap**

*Don’t dereference a* ***null*** *pointer because it could lead to disastrous results.*

Next, I add **500** to **score** with the following line.

**score += 500;**

When I send score to **cout**, 1500 is displayed, as you’d expect. When I send **\*pScore** to **cout**, the contents of score are again sent to **cout**, and **1500** is displayed once more.

Next, I add **500** to the value to which **pScore** points with the following line.

**\*pScore += 500;**

Because **pScore** points to **score**, the preceding line of code adds **500** to **score**. Therefore, when I next send **score** to co**u**t, **2000** is displayed. Then, when I send **\*pScore** to **cout**… you guessed it, **2000** is displayed again.

**Trap**

*Don’t change the value of a pointer when you want to change the value of the object to which the pointer points. For example, if I want to add* ***500*** *to the* ***int*** *that* ***pScore*** *points to, then the following line would be a big mistake.*

***pScore += 500;***

*The preceding code adds* ***500*** *to the address stored in* ***pScore****, not to the value to which* ***pScore*** *originally pointed. As a result,* ***pScore*** *now points to some address that might contain anything. Dereferencing a pointer like this can lead to disastrous* results.

**Reassigning Pointers**

Unlike references, pointers can point to different objects at different times during the life of a program. Reassigning a pointer works like reassigning any other variable. Next, I reassign **pScore** with the following line.

**pScore = &newScore;**

As the result, **pScore** now points to **newScore**. To prove this, I display the address of **newScore** by sending **&newScore** to **cout**, followed by the address stored in **pScore**. Both statements display the same address. Then I send **newScore** and **\*pScore** to **cout**. Both display **5000** because they both access the same chunk of memory that stores this value.

**Trap**

*Don’t change the value to which a pointer points when you want to change the pointer itself. For example, if I want to change* ***pScore*** *to point to* ***newScore****, then the following line would be a big mistake.*

***\*pScore = newScore;***

*This code simply changes the value to which* ***pScore*** *currently points; it doesn’t change* ***pScore*** *itself. If* ***newScore*** *is equal to* ***5000****, then the previous code is equivalent to* ***\*pScore = 5000;*** *and* ***pScore*** *still points to the same variable it pointed to before the assignment.*

**Using Pointers to Objects**

So far, the Pointing program has worked only with values of a built-in type, **int**. But you can use pointers with objects just as easily. I demonstrate this next with the following lines, which create **str**, a string object equal to "**score**", and **pStr**, a pointer that points to that object.

**string str = "score";**

**string\* pStr = &str;** //pointer to string object

**pStr** is a pointer to string, meaning that it can point to any string object. Another way to say this is to say that **pStr** can store the address of any string object.

You can access an object through a pointer using the dereference operator. That’s what I do next with the following line.

**cout << "\*pStr is: " << \*pStr << "\n";**

By using the dereference operator with **\*pStr**, I send the object to which **pStr** points (**str**) to **cout**. As a result, the text score is displayed.

You can call the member functions of an object through a pointer the same way you can call the member functions of an object through an iterator. One way to do this is by using the dereference operator and the member access operator, which is what I do next with the following line.

**cout << "(\*pStr).size() is: " << (\*pStr).size() << "\n";**

The code **(\*pStr).size()** says, “Take the result of dereferencing pStr and call that object’s **size()** member function.” Because **pStr** refers to the string object equal to "**score**", the code returns **5**.

**Hint**

Whenever you dereference a pointer to access a data member or member function, surround the dereferenced pointer with a pair of parentheses. This ensures that the dot operator will be applied to the object to which the pointer points.

Just as with iterators, you can use the **->** operator with pointers for a more readable way to access object members. That’s what I demonstrate next with the following line.

**cout << "pStr->size() is: " << pStr->size() << "\n";**

The preceding statement again displays the number of characters in the string object equal to "**score**"; however, I’m able to substitute **pStr->size()** for **(\*pStr).size()** this time, making the code more readable.

**UNDERSTANDING POINTERS AND CONSTANTS**

There are still some pointer mechanics you need to understand before you can start to use pointers effectively in your game programs. You can use the keyword **const** to restrict the way a pointer works. These restrictions can act as safeguards and can make your programming intentions clearer. Since pointers are quite versatile, restricting how a pointer can be used is in line with the programming mantra of asking only for what you need.

**Using a Constant Pointer**

As you’ve seen, pointers can point to different objects at different times in a program. However, by using the **const** keyword when you declare and initialize a pointer, you can restrict the pointer so it can only point to the object to which it was initialized to point. A pointer like this is called a ***constant pointer***. Another way to say this is to say that ***the address stored in a constant pointer can never change*** - it’s constant. Here’s an example of creating a constant pointer:

**int score = 100;**

**int\* const pScore = &score;** //a constant pointer

The preceding code creates a constant pointer, **pScore**, which points to **score**. You create a constant pointer by putting const right before the name of the pointer when you declare it.

Like all constants, you must initialize a constant pointer when you first declare it. The following line is illegal and will produce a big, fat compile error.

**int\* const pScore;** // illegal – you must initialize a constant

Because **pScore** is a constant pointer, it can’t ever point to any other memory location. The following code is also quite illegal.

**pScore = &anotherScore;** // illegal – pScore can’t point to a different object

Although you can’t change **pScore** itself, you can use **pScore** to change the value to which it points. The following line is completely legal.

**\*pScore = 500;**

Confused? Don’t be. It’s perfectly fine to use a constant pointer to change the value to which it points. Remember, the restriction on a constant pointer is that its value - the address that the pointer stores - can’t change.

The way a constant pointer works should remind you of something - a reference. Like a reference, a constant pointer can refer only to the object to which it was initialized to refer.

**Hint**

*Although you can use a constant pointer instead of a reference in your programs, you should stick with references when possible. References have a cleaner syntax than pointers and can make your code easier to read.*

**Using a Pointer to a Constant**

As you’ve seen, you can use pointers to change the values to which they point. However, by using the **const** keyword when you declare a pointer, you can restrict a pointer so it can’t be used to change the value to which it points.

A pointer like this is called a ***pointer to a constant***. Here’s an example of declaring such a pointer:

**const int\* pNumber;** //a pointer to a constant

The preceding code declares a pointer to a constant, **pNumber**. You declare a pointer to a constant by putting const right before the type of value to which the pointer will point.

You assign an address to a pointer to a constant as you did before.

**int lives = 3;**

**pNumber = &lives;**

However, you can’t use the pointer to change the value to which it points. The following line is illegal.

**\*pNumber -= 1;** // Illegal – can’t use pointer to a constant to change value

// that pointer points to

Although you can’t use a pointer to a constant to change the value to which it points, the pointer itself can change. This means that a pointer to a constant can point to different objects in a program. The following code is perfectly legal.

**const int MAX\_LIVES = 5;**

**pNumber = &MAX\_LIVES;** //pointer itself can change

**Using a Constant Pointer to a Constant**

A constant pointer to a constant combines the restrictions of a constant pointer and a pointer to a constant. This means that a constant pointer to a constant can only point to the object to which it was initialized to point. In addition, it can’t be used to change the value of the object to which it points. Here’s the declaration and initialization of such a pointer:

**const int\* const pBONUS = &BONUS;** //a constant pointer to a constant

The preceding code creates a constant pointer to a constant named **pBONUS** that points to the constant **BONUS**.

**Hint**

*Like a pointer to a constant, a constant pointer to a constant can point to either a non-constant or a*

*constant value.*

You can’t reassign a constant pointer to a constant.

**pBonus = &MAX\_LINES;** // Illegal – pBonus can’t point to another

You can’t use a constant pointer to a constant to change the value to which it points.

**\*pBonus = MAX\_LINES;** // Illegal – can’t change value through pointer

In many ways, a constant pointer to a constant acts like a constant reference, which can only refer to the value to which it was initialized to refer and which can’t be used to change that value.

**Hint**

*Although you can use a constant pointer to a constant instead of a constant reference in your programs,*

*you should stick with constant references when possible. References have a cleaner syntax than*

*pointers and can make your code easier to read.*

**Summarizing Constants and Pointers**

I’ve presented a lot of information on constants and pointers, so I want to provide a summary to help crystallize the new concepts. Here are three examples of the different ways in which you can use the keyword const when declaring pointers:

1. **int\* const p = &i;**
2. **const int\* p;**
3. **const int\* const p = &I;**
4. The first example declares and initializes a **constant** **pointer**. A constant pointer can only point to the object to which it was initialized to point. The value - the memory address - stored in the pointer itself is constant and can’t change. A constant pointer can only point to a non-constant value; it can’t point to a constant.
5. The second example declares a **pointer** **to a constant**. A pointer to a constant can’t be used to change the value to which it points. A pointer to a constant can point to different objects during the life of a program. A pointer to a constant can point to a constant or non-constant value.
6. The third example declares a **constant pointer to a constant**. A constant pointer to a constant can only point to the value to which it was initialized to point. In addition, it can’t be used to change the value to which it points. A constant pointer to a constant can be initialized to point to a constant or a non-constant value.