CHAPTER 2

Introduction to C++

TOPICS

2.1 The Parts of a C++ Program
2.2 The `cout` Object
2.3 The `#include` Directive
2.4 Variables and Literals
2.5 Identifiers
2.6 Integer Data Types
2.7 The `char` Data Type
2.8 The C++ `string` Class
2.9 Floating-Point Data Types
2.10 The `bool` Data Type
2.11 Determining the Size of a Data Type
2.12 Variable Assignments and Initialization
2.13 Scope
2.14 Arithmetic Operators
2.15 Comments
2.16 Named Constants
2.17 Programming Style

2.1 The Parts of a C++ Program

CONCEPT: **C++ programs have parts and components that serve specific purposes.**

Every C++ program has an anatomy. Unlike human anatomy, the parts of C++ programs are not always in the same place. Nevertheless, the parts are there, and your first step in learning C++ is to learn what they are. We will begin by looking at Program 2-1.

Let's examine the program line by line. Here's the first line:

```
// A simple C++ program
```

The `//` marks the beginning of a *comment*. The compiler ignores everything from the double slash to the end of the line. That means you can type anything you want on that line and the compiler will never complain! Although comments are not required, they are very important to programmers. Most programs are much more complicated than the example in Program 2-1, and comments help explain what's going on.

Program 2-1

```
1  // A simple C++ program
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7       cout << "Programming is great fun!";
8       return 0;
9  }
```

The output of the program is shown below. This is what appears on the screen when the program runs.

Program Output

```
Programming is great fun!
```

Line 2 looks like this:

```
#include <iostream>
```

Because this line starts with a #, it is called a *preprocessor directive*. The preprocessor reads your program before it is compiled and only executes those lines beginning with a # symbol. Think of the preprocessor as a program that "sets up" your source code for the compiler.

The `#include` directive causes the preprocessor to include the contents of another file in the program. The word inside the brackets, `iostream`, is the name of the file that is to be included. The `iostream` file contains code that allows a C++ program to display output on the screen and read input from the keyboard. Because this program uses `cout` to display screen output, the `iostream` file must be included. The contents of the `iostream` file are included in the program at the point the `#include` statement appears. The `iostream` file is called a *header file,* so it should be included at the head, or top, of the program.

Line 3 reads:

```
using namespace std;
```

Programs usually contain several items with unique names. In this chapter you will learn to create variables. In Chapter 6 you will learn to create functions. In Chapter 13 you will learn to create objects. Variables, functions, and objects are examples of program entities that must have names. C++ uses *namespaces* to organize the names of program entities. The statement `using namespace std;` declares that the program will be accessing entities whose names are part of the namespace called `std`. (Yes, even namespaces have names.) The reason the program needs access to the `std` namespace is because every name created by the `iostream` file is part of that namespace. In order for a program to use the entities in `iostream`, it must have access to the `std` namespace.

Line 5 reads:

```
int main()
```

This marks the beginning of a function. A *function* can be thought of as a group of one or more programming statements that collectively has a name. The name of this function is *main,* and the set of parentheses that follows the name indicate that it is a function. The

word `int` stands for "integer." It indicates that the function sends an integer value back to the operating system when it is finished executing.

Although most C++ programs have more than one function, every C++ program must have a function called `main`. It is the starting point of the program. If you are ever reading someone else's C++ program and want to find where it starts, just look for the function named `main`.

NOTE: C++ is a case-sensitive language. That means it regards uppercase letters as being entirely different characters than their lowercase counterparts. In C++, the name of the function `main` must be written in all lowercase letters. C++ doesn't see "Main" the same as "main," or "INT" the same as "int." This is true for all the C++ key words.

Line 6 contains a single, solitary character:

```
{
```

This is called a left-brace, or an opening brace, and it is associated with the beginning of the function `main`. All the statements that make up a function are enclosed in a set of braces. If you look at the third line down from the opening brace you'll see the closing brace. Everything between the two braces is the contents of the function `main`.

WARNING! Make sure you have a closing brace for every opening brace in your program!

After the opening brace you see the following statement in line 7:

```
cout << "Programming is great fun!";
```

To put it simply, this line displays a message on the screen. You will read more about `cout` and the `<<` operator later in this chapter. The message "Programming is great fun!" is printed without the quotation marks. In programming terms, the group of characters inside the quotation marks is called a *string literal* or *string constant*.

NOTE: This is the only line in the program that causes anything to be printed on the screen. The other lines, like `#include <iostream>` and `int main()`, are necessary for the framework of your program, but they do not cause any screen output. Remember, a program is a set of instructions for the computer. If something is to be displayed on the screen, you must use a programming statement for that purpose.

At the end of the line is a semicolon. Just as a period marks the end of a sentence, a semicolon marks the end of a complete statement in C++. Comments are ignored by the compiler, so the semicolon isn't required at the end of a comment. Preprocessor directives, like `#include` statements, simply end at the end of the line and never require semicolons. The beginning of a function, like `int main()`, is not a complete statement, so you don't place a semicolon there either.

It might seem that the rules for where to put a semicolon are not clear at all. Rather than worry about it now, just concentrate on learning the parts of a program. You'll soon get a feel for where you should and should not use semicolons.

Line 8 reads:

```
return 0;
```

This sends the integer value 0 back to the operating system upon the program's completion. The value 0 usually indicates that a program executed successfully.

Line 9 contains the closing brace:

```
}
```

This brace marks the end of the `main` function. Since `main` is the only function in this program, it also marks the end of the program.

In the sample program you encountered several sets of special characters. Table 2-1 provides a short summary of how they were used.

Table 2-1 Special Characters

Character	Name	Description
`//`	Double slash	Marks the beginning of a comment.
`#`	Pound sign	Marks the beginning of a preprocessor directive.
`< >`	Opening and closing brackets	Encloses a filename when used with the `#include` directive.
`( )`	Opening and closing parentheses	Used in naming a function, as in `int main()`
`{ }`	Opening and closing braces	Encloses a group of statements, such as the contents of a function.
`" "`	Opening and closing quotation marks	Encloses a string of characters, such as a message that is to be printed on the screen.
`;`	Semicolon	Marks the end of a complete programming statement.

Checkpoint

2.1 The following C++ program will not compile because the lines have been mixed up.

```
int main()
}
// A crazy mixed up program
return 0;
#include <iostream>
cout << "In 1492 Columbus sailed the ocean blue.";
{
using namespace std;
```

When the lines are properly arranged the program should display the following on the screen:

```
In 1492 Columbus sailed the ocean blue.
```

Rearrange the lines in the correct order. Test the program by entering it on the computer, compiling it, and running it.

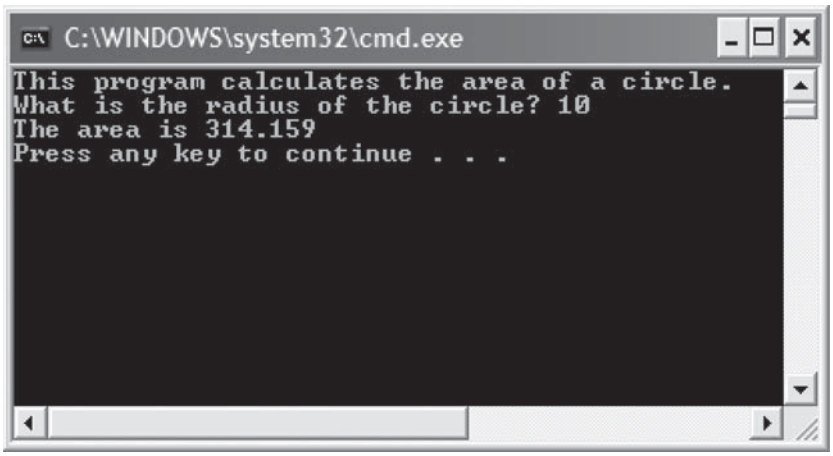
2.2 The cout Object

CONCEPT: Use the **cout** object to display information on the computer's screen.

In this section you will learn to write programs that produce output on the screen. The simplest type of screen output that a program can display is *console output*, which is merely plain text. The word *console* is an old computer term. It comes from the days when a computer operator interacted with the system by typing on a terminal. The terminal, which consisted of a simple screen and keyboard, was known as the *console*.

On modern computers, running graphical operating systems such as Windows or Mac OS X, console output is usually displayed in a window such as the one shown in Figure 2-1. In C++ you use the `cout` object to produce console output. (You can think of the word `cout` as meaning **c**onsole **out**put.)

Figure 2-1 A Console Window



`cout` is classified as a *stream object*, which means it works with streams of data. To print a message on the screen, you send a stream of characters to `cout`. Let's look at line 7 from Program 2-1:

```
cout << "Programming is great fun!";
```

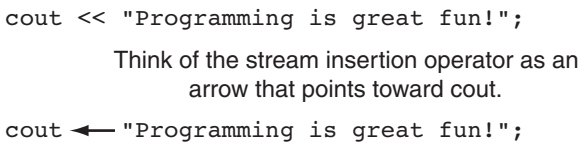
VideoNote
Using cout

Notice that the << operator is used to send the string "Programming is great fun!" to `cout`. When the << symbol is used this way, it is called the *stream insertion operator*. The item immediately to the right of the operator is sent to `cout` and then displayed on the screen.

The stream insertion operator is always written as two less-than signs with no space between them. Because you are using it to send a stream of data to the `cout` object, you can think of the stream insertion operator as an arrow that must point toward `cout`. This is illustrated in Figure 2-2.

Program 2-2 is another way to write the same program.

Figure 2-2



Program 2-2

```
1  // A simple C++ program
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7       cout << "Programming is " << "great fun!";
8       return 0;
9  }
```

Program Output

```
Programming is great fun!
```

As you can see, the stream-insertion operator can be used to send more than one item to `cout`. The output of this program is identical to that of Program 2-1. Program 2-3 shows yet another way to accomplish the same thing.

Program 2-3

```
1  // A simple C++ program
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7       cout << "Programming is ";
8       cout << "great fun!";
9       return 0;
10 }
```

Program Output

```
Programming is great fun!
```

An important concept to understand about Program 2-3 is that, although the output is broken up into two programming statements, this program will still display the message on a single line. Unless you specify otherwise, the information you send to `cout` is displayed in a continuous stream. Sometimes this can produce less-than-desirable results. Program 2-4 is an example.

The layout of the actual output looks nothing like the arrangement of the strings in the source code. First, notice there is no space displayed between the words “sellers” and “during,” or

between "June:" and "Computer." cout displays messages exactly as they are sent. If spaces are to be displayed, they must appear in the strings.

Program 2-4

```
1  // An unruly printing program
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7       cout << "The following items were top sellers";
8       cout << "during the month of June:";
9       cout << "Computer games";
10      cout << "Coffee";
11      cout << "Aspirin";
12      return 0;
13 }
```

Program Output

```
The following items were top sellersduring the month of June:Computer
gamesCoffeeAspirin
```

Second, even though the output is broken into five lines in the source code, it comes out as one long line of output. Because the output is too long to fit on one line on the screen, it wraps around to a second line when displayed. The reason the output comes out as one long line is because cout does not start a new line unless told to do so. There are two ways to instruct cout to start a new line. The first is to send cout a *stream manipulator* called endl (which is pronounced "end-line" or "end-L"). Program 2-5 is an example.

Program 2-5

```
1  // A well-adjusted printing program
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7       cout << "The following items were top sellers" << endl;
8       cout << "during the month of June:" << endl;
9       cout << "Computer games" << endl;
10      cout << "Coffee" << endl;
11      cout << "Aspirin" << endl;
12      return 0;
13 }
```

Program Output

```
The following items were top sellers
during the month of June:
Computer games
Coffee
Aspirin
```

NOTE: The last character in `endl` is the lowercase letter L, *not* the number one.

Every time `cout` encounters an `endl` stream manipulator it advances the output to the beginning of the next line for subsequent printing. The manipulator can be inserted anywhere in the stream of characters sent to `cout`, outside the double quotes. The following statements show an example.

```
cout << "My pets are" << endl << "dog";
cout << endl << "cat" << endl << "bird" << endl;
```

Another way to cause `cout` to go to a new line is to insert an *escape sequence* in the string itself. An escape sequence starts with the backslash character (`\`) and is followed by one or more control characters. It allows you to control the way output is displayed by embedding commands within the string itself. Program 2-6 is an example.

Program 2-6

```
1  // Yet another well-adjusted printing program
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7       cout << "The following items were top sellers\n";
8       cout << "during the month of June:\n";
9       cout << "Computer games\nCoffee";
10      cout << "\nAspirin\n";
11      return 0;
12 }
```

Program Output

```
The following items were top sellers
during the month of June:
Computer games
Coffee
Aspirin
```

The *newline escape sequence* is `\n`. When `cout` encounters `\n` in a string, it doesn't print it on the screen, but interprets it as a special command to advance the output cursor to the next line. You have probably noticed inserting the escape sequence requires less typing than inserting `endl`. That's why many programmers prefer it.

A common mistake made by beginning C++ students is to use a forward slash (`/`) instead of a backslash (`\`) when trying to write an escape sequence. This will not work. For example, look at the following code.

```
// Error!
cout << "Four Score/nAnd seven/nYears ago./n";
```

In this code, the programmer accidentally wrote /n when he or she meant to write \n. The cout object will simply display the /n characters on the screen. This code will display the following output:

```
Four Score/nAnd seven/nYears ago./n
```

Another common mistake is to forget to put the \n inside quotation marks. For example, the following code will not compile.

```
// Error! This code will not compile.
cout << "Good" << \n;
cout << "Morning" << \n;
```

This code will result in an error because the \n sequences are not inside quotation marks. We can correct the code by placing the \n sequences inside the string literals, as shown here:

```
// This will work.
cout << "Good\n";
cout << "Morning\n";
```

There are many escape sequences in C++. They give you the ability to exercise greater control over the way information is output by your program. Table 2-2 lists a few of them.

Table 2-2 Common Escape Sequences

Escape Sequence	Name	Description
\n	Newline	Causes the cursor to go to the next line for subsequent printing.
\t	Horizontal tab	Causes the cursor to skip over to the next tab stop.
\a	Alarm	Causes the computer to beep.
\b	Backspace	Causes the cursor to back up, or move left one position.
\r	Return	Causes the cursor to go to the beginning of the current line, not the next line.
\\	Backslash	Causes a backslash to be printed.
\'	Single quote	Causes a single quotation mark to be printed.
\"	Double quote	Causes a double quotation mark to be printed.

WARNING! When using escape sequences, do not put a space between the backslash and the control character.

When you type an escape sequence in a string, you type two characters (a backslash followed by another character). However, an escape sequence is stored in memory as a single character. For example, consider the following string literal:

```
"One\nTwo\nThree\n"
```

The diagram in Figure 2-3 breaks this string into its individual characters. Notice how each of the \n escape sequences are considered one character.

Figure 2-3

2.3 The #include Directive

CONCEPT: **The `#include` directive causes the contents of another file to be inserted into the program.**

Now is a good time to expand our discussion of the `#include` directive. The following line has appeared near the top of every example program.

```
#include <iostream>
```

The header file `iostream` must be included in any program that uses the `cout` object. This is because `cout` is not part of the "core" of the C++ language. Specifically, it is part of the *input–output stream library*. The header file, `iostream`, contains information describing `iostream` objects. Without it, the compiler will not know how to properly compile a program that uses `cout`.

Preprocessor directives are not C++ statements. They are commands to the preprocessor, which runs prior to the compiler (hence the name "preprocessor"). The preprocessor's job is to set programs up in a way that makes life easier for the programmer.

For example, any program that uses the `cout` object must contain the extensive setup information found in `iostream`. The programmer could type all this information into the program, but it would be too time consuming. An alternative would be to use an editor to "cut and paste" the information into the program, but that would quickly become tiring as well. The solution is to let the preprocessor insert the contents of iostream automatically.

WARNING! Do not put semicolons at the end of processor directives. Because preprocessor directives are not C++ statements, they do not require semicolons. In many cases an error will result from a preprocessor directive terminated with a semicolon.

An `#include` directive must always contain the name of a file. The preprocessor inserts the entire contents of the file into the program at the point it encounters the `#include` directive. The compiler doesn't actually see the `#include` directive. Instead it sees the code that was inserted by the preprocessor, just as if the programmer had typed it there.

The code contained in header files is C++ code. Typically it describes complex objects like `cout`. Later you will learn to create your own header files.

Checkpoint

2.2 The following C++ program will not compile because the lines have been mixed up.

```
cout << "Success\n";
cout << " Success\n\n";
int main()
cout << "Success";
}
```

```
using namespace std;
// It's a mad, mad program
#include <iostream>
cout << "Success\n";
{
return 0;
```

When the lines are properly arranged the program should display the following on the screen:

Program Output

```
Success
Success Success

Success
```

Rearrange the lines in the correct order. Test the program by entering it on the computer, compiling it, and running it.

2.3 Study the following program and show what it will print on the screen.

```
// The Works of Wolfgang
#include <iostream>
using namespace std;
int main()
{
   cout << "The works of Wolfgang\ninclude the following";
   cout << "\nThe Turkish March" << endl;
   cout << "and Symphony No. 40 ";
   cout << "in G minor." << endl;
   return 0;
}
```

2.4 On paper, write a program that will display your name on the first line, your street address on the second line, your city, state, and ZIP code on the third line, and your telephone number on the fourth line. Place a comment with today's date at the top of the program. Test your program by entering, compiling, and running it.

2.4 Variables and Literals

CONCEPT: Variables represent storage locations in the computer's memory. Literals are constant values that are assigned to variables.

As you discovered in Chapter 1, variables allow you to store and work with data in the computer's memory. They provide an "interface" to RAM. Part of the job of programming is to determine how many variables a program will need and what types of information they will hold. Program 2-7 is an example of a C++ program with a variable. Take a look at line 7:

VideoNote
Variabe Definitions

```
int number;
```

This is called a *variable definition*. It tells the compiler the variable's name and the type of data it will hold. This line indicates the variable's name is `number`. The word `int` stands for integer, so `number` will only be used to hold integer numbers. Later in this chapter you will learn all the types of data that C++ allows.

Program 2-7

```
1  // This program has a variable.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7       int number;
8
9       number = 5;
10      cout << "The value in number is " << number << endl;
11      return 0;
12 }
```

Program Output

```
The value in number is 5
```

NOTE: You must have a definition for every variable you intend to use in a program. In C++, variable definitions can appear at any point in the program. Later in this chapter, and throughout the book, you will learn the best places to define variables.

Notice that variable definitions end with a semicolon. Now look at line 9:

```
number = 5;
```

This is called an *assignment*. The equal sign is an operator that copies the value on its right (5) into the variable named on its left (`number`). After this line executes, `number` will be set to 5.

NOTE: This line does not print anything on the computer's screen. It runs silently behind the scenes, storing a value in RAM.

Look at line 10.

```
cout << "The value in number is " << number << endl;
```

The second item sent to `cout` is the variable name `number`. When you send a variable name to `cout` it prints the variable's contents. Notice there are no quotation marks around `number`. Look at what happens in Program 2-8.

Program 2-8

```
1  // This program has a variable.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7       int number;
8
9       number = 5;
```

```
10       cout << "The value in number is " << "number" << endl;
11       return 0;
12   }
```

Program Output

```
The value in number is number
```

When double quotation marks are placed around the word `number` it becomes a string literal and is no longer a variable name. When string literals are sent to `cout` they are printed exactly as they appear inside the quotation marks. You've probably noticed by now that the `endl` stream manipulator has no quotation marks around it, for the same reason.

Sometimes a Number Isn't a Number

As shown in Program 2-8, just placing quotation marks around a variable name changes the program's results. In fact, placing double quotation marks around anything that is not intended to be a string literal will create an error of some type. For example, in Program 2-8 the number 5 was assigned to the variable `number`. It would have been incorrect to perform the assignment this way:

```
number = "5";
```

In this line, 5 is no longer an integer, but a string literal. Because `number` was defined as an integer variable, you can only store integers in it. The integer 5 and the string literal "5" are not the same thing.

The fact that numbers can be represented as strings frequently confuses students who are new to programming. Just remember that strings are intended for humans to read. They are to be printed on computer screens or paper. Numbers, however, are intended primarily for mathematical operations. You cannot perform math on strings. Before numbers can be displayed on the screen, they must first be converted to strings. (Fortunately, `cout` handles the conversion automatically when you send a number to it.)

Literals

A literal is a piece of data that is written directly into a program's code. One of the most common uses of literals is to assign a value to a variable. For example, in the following statement assume that `number` is an `int` variable. The statement assigns the literal value 100 to the variable `number`:

```
number = 100;
```

Another common use of literals is to display something on the screen. For example, the following statement displays the string literal "Welcome to my program."

```
cout << "Welcome to my program." << endl;
```

Program 2-9 shows an example that uses a variable and several literals.

Program 2-9

```
1  // This program has literals and a variable.
2  #include <iostream>
3  using namespace std;
```

(program continues)

Program 2-9 *(continued)*

```
4
5  int main()
6  {
7       int apples;
8
9       apples = 20;
10      cout << "Today we sold " << apples << " bushels of apples.\n";
11      return 0;
12 }
```

Program Output

```
Today we sold 20 bushels of apples.
```

Of course, the variable is `apples`. It is defined as an integer. Table 2-3 lists the literals found in the program.

Table 2-3

Literal	Type of Literal
`20`	Integer literal
`"Today we sold "`	String literal
`"bushels of apples.\n"`	String literal
`0`	Integer literal

NOTE: Literals are also called constants.

Checkpoint

2.5 Examine the following program.

```
// This program uses variables and literals.
#include <iostream>
using namespace std;
int main()
{
     int little;
     int big;
     little = 2;
     big = 2000;
     cout << "The little number is " << little << endl;
     cout << "The big number is " << big << endl;
     return 0;
}
```

List all the variables and literals that appear in the program.

2.6 What will the following program display on the screen?

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

```
int main()
{
     int number;
     number = 712;
     cout << "The value is " << "number" << endl;
     return 0;
}
```

2.5 Identifiers

CONCEPT: Choose variable names that indicate what the variables are used for.

An *identifier* is a programmer-defined name that represents some element of a program. Variable names are examples of identifiers. You may choose your own variable names in C++, as long as you do not use any of the C++ *key words*. The key words make up the "core" of the language and have specific purposes. Table 2-4 shows a complete list of the C++ key words. Note that they are all lowercase.

Table 2-4 The C++ Key Words

alignas	const	for	private	throw
alignof	constexpr	friend	protected	true
and	const_cast	goto	public	try
and_eq	continue	if	register	typedef
asm	decltype	inline	reinterpret_cast	typeid
auto	default	int	return	typename
bitand	delete	long	short	union
bitor	do	mutable	signed	unsigned
bool	double	namespace	sizeof	using
break	dynamic_cast	new	static	virtual
case	else	noexcept	static_assert	void
catch	enum	not	static_cast	volatile
char	explicit	not_eq	struct	wchar_t
char16_t	export	nullptr	switch	while
char32_t	extern	operator	template	xor
class	false	or	this	xor_eq
compl	float	or_eq	thread_local	

You should always choose names for your variables that give an indication of what the variables are used for. You may be tempted to define variables with names like this:

```
int x;
```

The rather nondescript name, `x`, gives no clue as to the variable's purpose. Here is a better example.

```
int itemsOrdered;
```

The name `itemsOrdered` gives anyone reading the program an idea of the variable's use. This way of coding helps produce self-documenting programs, which means you get an understanding of what the program is doing just by reading its code. Because real-world programs usually have thousands of lines, it is important that they be as self-documenting as possible.

You probably have noticed the mixture of uppercase and lowercase letters in the name `itemsOrdered`. Although all of C++'s key words must be written in lowercase, you may use uppercase letters in variable names.

The reason the `O` in `itemsOrdered` is capitalized is to improve readability. Normally "items ordered" is two words. Unfortunately you cannot have spaces in a variable name, so the two words must be combined into one. When "items" and "ordered" are stuck together you get a variable definition like this:

```
int itemsordered;
```

Capitalization of the first letter of the second word and succeeding words makes `itemsOrdered` easier to read. It should be mentioned that this style of coding is not required. You are free to use all lowercase letters, all uppercase letters, or any combination of both. In fact, some programmers use the underscore character to separate words in a variable name, as in the following.

```
int items_ordered;
```

Legal Identifiers

Regardless of which style you adopt, be consistent and make your variable names as sensible as possible. Here are some specific rules that must be followed with all identifiers.

- The first character must be one of the letters a through z, A through Z, or an underscore character (_).
- After the first character you may use the letters a through z or A through Z, the digits 0 through 9, or underscores.
- Uppercase and lowercase characters are distinct. This means `ItemsOrdered` is not the same as `itemsordered`.

Table 2-5 lists variable names and tells whether each is legal or illegal in C++.

Table 2-5 Some Variable Names

Variable Name	Legal or Illegal?
`dayOfWeek`	Legal.
`3dGraph`	Illegal. Variable names cannot begin with a digit.
`_employee_num`	Legal.
`June1997`	Legal.
`Mixture#3`	Illegal. Variable names may only use letters, digits, or underscores.

2.6 Integer Data Types

CONCEPT: **There are many different types of data. Variables are classified according to their data type, which determines the kind of information that may be stored in them. Integer variables can only hold whole numbers.**

Computer programs collect pieces of data from the real world and manipulate them in various ways. There are many different types of data. In the realm of numeric information, for example, there are whole numbers and fractional numbers. There are negative numbers and positive numbers. And there are numbers so large, and others so small, that they don't even have a name. Then there is textual information. Names and addresses, for instance, are stored as groups of characters. When you write a program you must determine what types of information it will be likely to encounter.

If you are writing a program to calculate the number of miles to a distant star, you'll need variables that can hold very large numbers. If you are designing software to record microscopic dimensions, you'll need to store very small and precise numbers. Additionally, if you are writing a program that must perform thousands of intensive calculations, you'll want variables that can be processed quickly. The data type of a variable determines all of these factors.

Although C++ offers many data types, in the very broadest sense there are only two: numeric and character. Numeric data types are broken into two additional categories: integer and floating point. Integers are whole numbers like 12, 157, –34, and 2. Floating point numbers have a decimal point, like 23.7, 189.0231, and 0.987. Additionally, the integer and floating point data types are broken into even more classifications. Before we discuss the character data type, let's carefully examine the variations of numeric data.

Your primary considerations for selecting a numeric data type are

- The largest and smallest numbers that may be stored in the variable
- How much memory the variable uses
- Whether the variable stores signed or unsigned numbers
- The number of decimal places of precision the variable has

The size of a variable is the number of bytes of memory it uses. Typically, the larger a variable is, the greater the range it can hold.

Table 2-6 shows the C++ integer data types with their typical sizes and ranges.

NOTE: The data type sizes and ranges shown in Table 2-6 are typical on many systems. Depending on your operating system, the sizes and ranges may be different.

Table 2-6 Integer Data Types

Data Type	Typical Size	Typical Range
`short int`	2 bytes	−32,768 to +32,767
`unsigned short int`	2 bytes	0 to +65,535
`int`	4 bytes	−2,147,483,648 to +2,147,483,647
`unsigned int`	4 bytes	0 to 4,294,967,295
`long int`	4 bytes	−2,147,483,648 to +2,147,483,647
`unsigned long int`	4 bytes	0 to 4,294,967,295
`long long int`	8 bytes	−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`unsigned long long int`	8 bytes	0 to 18,446,744,073,709,551,615

Here are some examples of variable definitions:

```
int days;
unsigned int speed;
short int month;
unsigned short int amount;
long int deficit;
unsigned long int insects;
```

Each of the data types in Table 2-6, except `int`, can be abbreviated as follows:

- `short int` can be abbreviated as `short`
- `unsigned short int` can be abbreviated as `unsigned short`
- `unsigned int` can be abbreviated as `unsigned`
- `long int` can be abbreviated as `long`
- `unsigned long int` can be abbreviated as `unsigned long`
- `long long int` can be abbreviated as `long long`
- `unsigned long long int` can be abbreviated as `unsigned long long`

Because they simplify definition statements, programmers commonly use the abbreviated data type names. Here are some examples:

```
unsigned speed;
short month;
unsigned short amount;
long deficit;
unsigned long insects;
long long grandTotal;
unsigned long long lightYearDistance;
```

Unsigned data types can only store nonnegative values. They can be used when you know your program will not encounter negative values. For example, variables that hold ages or weights would rarely hold numbers less than 0.

Notice in Table 2-6 that the `int` and `long` data types have the same sizes and ranges, and that the `unsigned int` data type has the same size and range as the `unsigned long` data type. This is not always true because the size of integers is dependent on the type of system you are using. Here are the only guarantees:

- Integers are at least as big as short integers.
- Long integers are at least as big as integers.
- Unsigned short integers are the same size as short integers.
- Unsigned integers are the same size as integers.
- Unsigned long integers are the same size as long integers.
- The `long long int` and the `unsigned long long int` data types are guaranteed to be at least 8 bytes (64 bits) in size.

Later in this chapter you will learn to use the `sizeof` operator to determine how large all the data types are on your computer.

NOTE: The `long long int` and the `unsigned long long int` data types were introduced in C++ 11.

As mentioned before, variables are defined by stating the data type key word followed by the name of the variable. In Program 2-10 an integer, an unsigned integer, and a long integer have been defined.

Program 2-10

```
1  // This program has variables of several of the integer types.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7       int checking;
8       unsigned int miles;
9       long days;
10
11      checking = -20;
12      miles = 4276;
13      days = 189000;
14      cout << "We have made a long journey of " << miles;
15      cout << " miles.\n";
16      cout << "Our checking account balance is " << checking;
17      cout << "\nAbout " << days << " days ago Columbus ";
18      cout << "stood on this spot.\n";
19      return 0;
20 }
```

Program Output

```
We have made a long journey of 4276 miles.
Our checking account balance is -20
About 189000 days ago Columbus stood on this spot.
```

In most programs you will need more than one variable of any given data type. If a program uses two integers, `length` and `width`, they could be defined separately, like this:

```
int length;
int width;
```

It is easier, however, to combine both variable definitions on one line:

```
int length, width;
```

You can define several variables of the same type like this, simply separating their names with commas. Program 2-11 illustrates this.

Program 2-11

```
1   // This program shows three variables defined on the same line.
2  #include <iostream>
3  using namespace std;
4
5  int main()
```

(program continues)

Program 2-11 *(continued)*

```
 6  {
 7       int floors, rooms, suites;
 8
 9       floors = 15;
10       rooms = 300;
11       suites = 30;
12       cout << "The Grande Hotel has " << floors << " floors\n";
13       cout << "with " << rooms << " rooms and " << suites;
14       cout << " suites.\n";
15       return 0;
16  }
```

Program Output

```
The Grande Hotel has 15 floors
with 300 rooms and 30 suites.
```

Integer and Long Integer Literals

In C++, if a numeric literal is an integer (not written with a decimal point) and it fits within the range of an `int` (see Table 2-6 for the minimum and maximum values), then the numeric literal is treated as an `int`. A numeric literal that is treated as an `int` is called an *integer literal*. For example, look at lines 9, 10, and 11 in Program 2-11:

```
floors = 15;
rooms = 300;
suites = 30;
```

Each of these statements assigns an integer literal to a variable.

One of the pleasing characteristics of the C++ language is that it allows you to control almost every aspect of your program. If you need to change the way something is stored in memory, the tools are provided to do that. For example, what if you are in a situation where you have an integer literal, but you need it to be stored in memory as a long integer? (Rest assured, this is a situation that does arise.) C++ allows you to force an integer literal to be stored as a long integer by placing the letter L at the end of the number. Here is an example:

```
long amount;
amount = 32L;
```

The first statement defines a `long` variable named `amount`. The second statement assigns the literal value 32 to the `amount` variable. In the second statement, the literal is written as `32L`, which makes it a *long integer literal*. This means the literal is treated as a `long`.

If you want an integer literal to be treated as a `long long int`, you can append `LL` at the end of the number. Here is an example:

```
long long amount;
amount = 32LL;
```

The first statement defines a `long long` variable named `amount`. The second statement assigns the literal value 32 to the `amount` variable. In the second statement, the literal is written as `32LL`, which makes it a *long long integer literal*. This means the literal is treated as a `long long int`.

TIP: When writing long integer literals or long long integer literals, you can use either an uppercase or lowercase L. Because the lowercase l looks like the number 1, you should always use the uppercase L.

If You Plan to Continue in Computer Science: Hexadecimal and Octal Literals

Programmers commonly express values in numbering systems other than decimal (or base 10). Hexadecimal (base 16) and octal (base 8) are popular because they make certain programming tasks more convenient than decimal numbers do.

By default, C++ assumes that all integer literals are expressed in decimal. You express hexadecimal numbers by placing 0x in front of them. (This is zero-x, not oh-x.) Here is how the hexadecimal number F4 would be expressed in C++:

```
0xF4
```

Octal numbers must be preceded by a 0 (zero, not oh). For example, the octal 31 would be written

```
031
```

NOTE: You will not be writing programs for some time that require this type of manipulation. It is important, however, that you understand this material. Good programmers should develop the skills for reading other people's source code. You may find yourself reading programs that use items like long integer, hexadecimal, or octal literals.

Checkpoint

2.7 Which of the following are illegal variable names, and why?

```
X
99bottles
july97
theSalesFigureForFiscalYear98
r&d
grade_report
```

2.8 Is the variable name `Sales` the same as `sales`? Why or why not?

2.9 Refer to the data types listed in Table 2-6 for these questions.

A) If a variable needs to hold numbers in the range 32 to 6,000, what data type would be best?

B) If a variable needs to hold numbers in the range −40,000 to +40,000, what data type would be best?

C) Which of the following literals uses more memory? `20` or `20L`

2.10 On any computer, which data type uses more memory, an integer or an unsigned integer?

2.7 The char Data Type

The char data type is used to store individual characters. A variable of the char data type can hold only one character at a time. Here is an example of how you might declare a char variable:

```
char letter;
```

This statement declares a char variable named letter, which can store one character. In C++, *character literals* are enclosed in single quotation marks. Here is an example showing how we would assign a character to the letter variable:

```
letter = 'g';
```

This statement assigns the character 'g' to the letter variable. Because char variables can hold only one character, they are not compatible with strings. For example, you cannot assign a string to a char variable, even if the string contains only one character. The following statement, for example, will not compile because it attempts to assign a string literal to a char variable.

```
letter = "g"; // ERROR! Cannot assign a string to a char
```

It is important that you do not confuse character literals, which are enclosed in single quotation marks, with string literals, which are enclosed in double quotation marks.

Program 2-12 shows an example program that works with characters.

Program 2-12

```
1  // This program works with characters.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7       char letter;
8
9       letter = 'A';
10      cout << letter << endl;
11      letter = 'B';
12      cout << letter << endl;
13      return 0;
14 }
```

Program Output

```
A
B
```

Although the char data type is used for storing characters, it is actually an integer data type that typically uses 1 byte of memory. (The size is system dependent. On some systems, the char data type is larger than 1 byte.)

The reason an integer data type is used to store characters is because characters are internally represented by numbers. Each printable character, as well as many nonprintable characters, is assigned a unique number. The most commonly used method for encoding characters is ASCII, which stands for the American Standard Code for Information Interchange. (There are other codes, such as EBCDIC, which is used by many IBM mainframes.)

When a character is stored in memory, it is actually the numeric code that is stored. When the computer is instructed to print the value on the screen, it displays the character that corresponds with the numeric code.

You may want to refer to Appendix B, which shows the ASCII character set. Notice that the number 65 is the code for A, 66 is the code for B, and so on. Program 2-13 demonstrates that when you work with characters, you are actually working with numbers.

Program 2-13

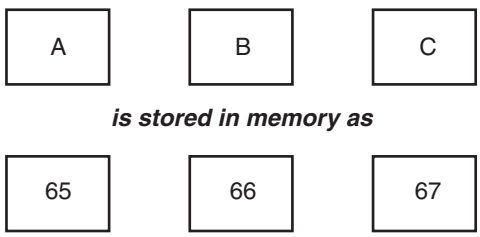
```
1  // This program demonstrates the close relationship between
2  // characters and integers.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8       char letter;
9
10      letter = 65;
11      cout << letter << endl;
12      letter = 66;
13      cout << letter << endl;
14      return 0;
15 }
```

Program Output

```
A
B
```

Figure 2-4 illustrates that when characters, such as A, B, and C, are stored in memory, it is really the numbers 65, 66, and 67 that are stored.

Figure 2-4



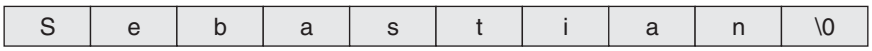
The Difference Between String Literals and Character Literals

It is important that you do not confuse character literals with string literals. Strings, which are a series of characters stored in consecutive memory locations, can be virtually any length. This means that there must be some way for the program to know how long a string is. In C++ an extra byte is appended to the end of string literals when they are stored in memory. In this last byte, the number 0 is stored. It is called the *null terminator* or *null character*, and it marks the end of the string.

Don't confuse the null terminator with the character `'0'`. If you look at Appendix B, you will see that ASCII code 48 corresponds to the character `'0'`, whereas the null terminator is the same as the ASCII code 0. If you want to print the character 0 on the screen, you use ASCII code 48. If you want to mark the end of a string, however, you use ASCII code 0.

Let's look at an example of how a string literal is stored in memory. Figure 2-5 depicts the way the string literal `"Sebastian"` would be stored.

Figure 2-5



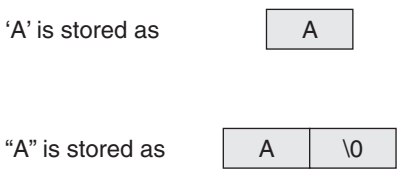
First, notice the quotation marks are not stored with the string. They are simply a way of marking the beginning and end of the string in your source code. Second, notice the very last byte of the string. It contains the null terminator, which is represented by the `\0` character. The addition of this last byte means that although the string `"Sebastian"` is 9 characters long, it occupies 10 bytes of memory.

The null terminator is another example of something that sits quietly in the background. It doesn't print on the screen when you display a string, but nevertheless, it is there silently doing its job.

NOTE: C++ automatically places the null terminator at the end of string literals.

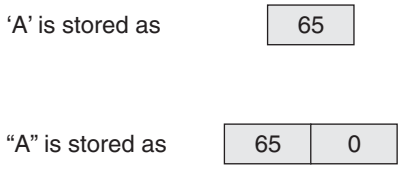
Now let's compare the way a string and a `char` are stored. Suppose you have the literals `'A'` and `"A"` in a program. Figure 2-6 depicts their internal storage.

Figure 2-6



As you can see, `'A'` is a 1-byte element and `"A"` is a 2-byte element. Since characters are really stored as ASCII codes, Figure 2-7 shows what is actually being stored in memory.

Figure 2-7



Because `char` variables are only large enough to hold one character, you cannot assign string literals to them. For example, the following code defines a `char` variable named `letter`. The character literal `'A'` can be assigned to the variable, but the string literal `"A"` cannot.

```
char letter;
letter = 'A';  // This will work.
letter = "A";  // This will not work!
```

One final topic about characters should be discussed. You have learned that some strings look like a single character but really aren't. It is also possible to have a character that looks like a string. A good example is the newline character, `\n`. Although it is represented by two characters, a slash and an `n`, it is internally represented as one character. In fact, all escape sequences, internally, are just 1 byte.

Program 2-14 shows the use of `\n` as a character literal, enclosed in single quotation marks. If you refer to the ASCII chart in Appendix B, you will see that ASCII code 10 is the linefeed character. This is the code C++ uses for the newline character.

Program 2-14

```
1  // This program uses character literals.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7      char letter;
8
9      letter = 'A';
10     cout << letter << '\n';
11     letter = 'B';
12     cout << letter << '\n';
13     return 0;
14 }
```

Program Output

```
A
B
```

Let's review some important points regarding characters and strings:

- Printable characters are internally represented by numeric codes. Most computers use ASCII codes for this purpose.
- Characters normally occupy a single byte of memory.

- Strings are consecutive sequences of characters that occupy consecutive bytes of memory.
- String literals are always stored in memory with a null terminator at the end. This marks the end of the string.
- Character literals are enclosed in single quotation marks.
- String literals are enclosed in double quotation marks.
- Escape sequences such as `'\n'` are stored internally as a single character.

2.8 The C++ `string` Class

CONCEPT: Standard C++ provides a special data type for storing and working with strings.

Because a `char` variable can store only one character in its memory location, another data type is needed for a variable able to hold an entire string. Although C++ does not have a built-in data type able to do this, standard C++ provides something called the `string` class that allows the programmer to create a `string` type variable.

Using the `string` Class

The first step in using the `string` class is to `#include` the `string` header file. This is accomplished with the following preprocessor directive:

```
#include <string>
```

The next step is to define a `string` type variable, called a `string` object. Defining a `string` object is similar to defining a variable of a primitive type. For example, the following statement defines a `string` object named `movieTitle`.

```
string movieTitle;
```

You can assign a string literal to `movieTitle` with the assignment operator:

```
movieTitle = "Wheels of Fury";
```

You can use `cout` to display the value of the `movieTitle` object, as shown in the next statement:

```
cout << "My favorite movie is " << movieTitle << endl;
```

Program 2-15 is a complete program that demonstrates the preceding statements.

Program 2-15

```
1 // This program demonstrates the string class.
2 #include <iostream>
3 #include <string> // Required for the string class.
4 using namespace std;
```

```
 5
 6  int main()
 7  {
 8       string movieTitle;
 9
10       movieTitle = "Wheels of Fury";
11       cout << "My favorite movie is " << movieTitle << endl;
12       return 0;
13  }
```

Program Output

```
My favorite movie is Wheels of Fury
```

As you can see, working with `string` objects is similar to working with variables of other types. Throughout this text we will continue to discuss `string` class features and capabilities.

Checkpoint

2.11 What are the ASCII codes for the following characters? (Refer to Appendix B)

```
C
F
W
```

2.12 Which of the following is a character literal?

```
'B'
"B"
```

2.13 Assuming the `char` data type uses 1 byte of memory, how many bytes do the following literals use?

```
'Q'
"Q"
"Sales"
'\n'
```

2.14 Write a program that has the following character variables: `first`, `middle`, and `last`. Store your initials in these variables and then display them on the screen.

2.15 What is wrong with the following program statement?

```
char letter = "Z";
```

2.16 What header file must you include in order to use `string` objects?

2.17 Write a program that stores your name, address, and phone number in three separate `string` objects. Display the contents of the `string` objects on the screen.

2.9 Floating-Point Data Types

CONCEPT: Floating-point data types are used to define variables that can hold real numbers.

Whole numbers are not adequate for many jobs. If you are writing a program that works with dollar amounts or precise measurements, you need a data type that allows fractional values. In programming terms, these are called *floating-point* numbers.

Internally, floating-point numbers are stored in a manner similar to *scientific notation*. Take the number 47,281.97. In scientific notation this number is 4.728197×10^4. (10^4 is equal to 10,000, and $4.728197 \times 10{,}000$ is 47,281.97.) The first part of the number, 4.728197, is called the *mantissa*. The mantissa is multiplied by a power of ten.

Computers typically use *E notation* to represent floating-point values. In E notation, the number 47,281.97 would be 4.728197E4. The part of the number before the E is the mantissa, and the part after the E is the power of 10. When a floating point number is stored in memory, it is stored as the mantissa and the power of 10.

Table 2-7 shows other numbers represented in scientific and E notation.

Table 2-7 Floating Point Representations

Decimal Notation	Scientific Notation	E Notation
247.91	2.4791×10^2	`2.4791E2`
0.00072	7.2×10^{-4}	`7.2E-4`
2,900,000	2.9×10^6	`2.9E6`

In C++ there are three data types that can represent floating-point numbers. They are

```
float
double
long double
```

The `float` data type is considered *single precision*. The `double` data type is usually twice as big as `float`, so it is considered *double precision*. As you've probably guessed, the `long double` is intended to be larger than the `double`. Of course, the exact sizes of these data types are dependent on the computer you are using. The only guarantees are

- A `double` is at least as big as a `float`.
- A `long double` is at least as big as a `double`.

Table 2-8 shows the sizes and ranges of floating-point data types usually found on PCs.

Table 2-8 Floating Point Data Types on PCs

Data Type	Key Word	Description
Single precision	`float`	4 bytes. Numbers between ±3.4E-38 and ±3.4E38
Double precision	`double`	8 bytes. Numbers between ±1.7E-308 and ±1.7E308
Long double precision	`long double`	8 bytes*. Numbers between ±1.7E-308 and ±1.7E308

*Some compilers use 10 bytes for long doubles. This allows a range of ±3.4E-4932 to ±1.1E4832

You will notice there are no unsigned floating point data types. On all machines, variables of the `float`, `double`, and `long double` data types can store positive or negative numbers.

Floating Point Literals

Floating point literals may be expressed in a variety of ways. As shown in Program 2-16, E notation is one method. When you are writing numbers that are extremely large or extremely small, this will probably be the easiest way. E notation numbers may be expressed with an uppercase E or a lowercase e. Notice that in the source code the literals were written as 1.495979E11 and 1.989E30, but the program printed them as 1.49598e+ 011 and 1.989e+30. The two sets of numbers are equivalent. (The plus sign in front of the exponent is also optional.) In Chapter 3 you will learn to control the way `cout` displays E notation numbers.

Program 2-16

```
1  // This program uses floating point data types.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7       float distance;
8       double mass;
9
10      distance = 1.495979E11;
11      mass = 1.989E30;
12      cout << "The Sun is " << distance << " meters away.\n";
13      cout << "The Sun\'s mass is " << mass << " kilograms.\n";
14      return 0;
15 }
```

Program Output

```
The Sun is 1.49598e+011 meters away.
The Sun's mass is 1.989e+030 kilograms.
```

You can also express floating-point literals in decimal notation. The literal 1.495979E11 could have been written as

```
149597900000.00
```

Obviously the E notation is more convenient for lengthy numbers, but for numbers like 47.39, decimal notation is preferable to 4.739E1.

All of the following floating-point literals are equivalent:

```
1.4959E11
1.4959e11
1.4959E+11
1.4959e+11
149590000000.00
```

Floating-point literals are normally stored in memory as `doubles`. But remember, C++ provides tools for handling just about any situation. Just in case you need to force a literal to be stored as a `float`, you can append the letter F or f to the end of it. For example, the following literals would be stored as `floats`:

```
1.2F
45.907f
```

NOTE: Because floating-point literals are normally stored in memory as `doubles`, many compilers issue a warning message when you assign a floating-point literal to a `float` variable. For example, assuming `num` is a `float`, the following statement might cause the compiler to generate a warning message:

```
num = 14.725;
```

You can suppress the warning message by appending the f suffix to the floating-point literal, as shown below:

```
num = 14.725f;
```

If you want to force a value to be stored as a `long double`, append an `L` or `l` to it, as in the following examples:

```
1034.56L
89.2l
```

The compiler won't confuse these with long integers because they have decimal points. (Remember, the lowercase `L` looks so much like the number 1 that you should always use the uppercase `L` when suffixing literals.)

Assigning Floating-Point Values to Integer Variables

When a floating-point value is assigned to an integer variable, the fractional part of the value (the part after the decimal point) is discarded. For example, look at the following code.

```
int number;
number = 7.5;    // Assigns 7 to number
```

This code attempts to assign the floating-point value 7.5 to the integer variable `number`. As a result, the value 7 will be assigned to `number`, with the fractional part discarded. When part of a value is discarded, it is said to be *truncated*.

Assigning a floating-point variable to an integer variable has the same effect. For example, look at the following code.

```
int i;
float f;
f = 7.5;
i = f;              // Assigns 7 to i.
```

When the float variable f is assigned to the int variable i, the value being assigned (7.5) is truncated. After this code executes i will hold the value 7 and f will hold the value 7.5.

NOTE: When a floating-point value is truncated, it is not rounded. Assigning the value 7.9 to an int variable will result in the value 7 being stored in the variable.

WARNING! Floating-point variables can hold a much larger range of values than integer variables can. If a floating-point value is being stored in an integer variable, and the whole part of the value (the part before the decimal point) is too large for the integer variable, an invalid value will be stored in the integer variable.

2.10 The bool Data Type

CONCEPT: Boolean variables are set to either true or false.

Expressions that have a true or false value are called *Boolean* expressions, named in honor of English mathematician George Boole (1815–1864).

The bool data type allows you to create small integer variables that are suitable for holding true or false values. Program 2-17 demonstrates the definition and assignment of a bool variable.

Program 2-17

```
1  // This program demonstrates boolean variables.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7       bool boolValue;
8
9       boolValue = true;
10      cout << boolValue << endl;
11      boolValue = false;
12      cout << boolValue << endl;
13      return 0;
14 }
```

Program Output

```
1
0
```

As you can see from the program output, the value true is represented in memory by the number 1, and false is represented by 0. You will not be using bool variables until Chapter 4, however, so just remember they are useful for evaluating conditions that are either true or false.

2.11 Determining the Size of a Data Type

CONCEPT: **The `sizeof` operator may be used to determine the size of a data type on any system.**

Chapter 1 discussed the portability of the C++ language. As you have seen in this chapter, one of the problems of portability is the lack of common sizes of data types on all machines. If you are not sure what the sizes of data types are on your computer, C++ provides a way to find out.

A special operator called `sizeof` will report the number of bytes of memory used by any data type or variable. Program 2-18 illustrates its use. The first line that uses the operator is line 10:

```
cout << "The size of an integer is " << sizeof(int);
```

The name of the data type or variable is placed inside the parentheses that follow the operator. The operator "returns" the number of bytes used by that item. This operator can be invoked anywhere you can use an unsigned integer, including in mathematical operations.

Program 2-18

```
1  // This program determines the size of integers, long
2  // integers, and long doubles.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8       long double apple;
9
10      cout << "The size of an integer is " << sizeof(int);
11      cout << " bytes.\n";
12      cout << "The size of a long integer is " << sizeof(long);
13      cout << " bytes.\n";
14      cout << "An apple can be eaten in " << sizeof(apple);
15      cout << " bytes!\n";
16      return 0;
17 }
```

Program Output

```
The size of an integer is 4 bytes.
The size of a long integer is 4 bytes.
An apple can be eaten in 8 bytes!
```

Checkpoint

2.18 Yes or No: Is there an unsigned floating point data type? If so, what is it?

2.19 How would the following number in scientific notation be represented in E notation?

$$6.31 \times 10^{17}$$

2.20 Write a program that defines an integer variable named `age` and a `float` variable named `weight`. Store your age and weight, as literals, in the variables. The program should display these values on the screen in a manner similar to the following:

Program Output

My age is 26 and my weight is 180 pounds.

(Feel free to lie to the computer about your age and your weight—it'll never know!)

2.12 Variable Assignments and Initialization

CONCEPT: An assignment operation assigns, or copies, a value into a variable. When a value is assigned to a variable as part of the variable's definition, it is called an initialization.

As you have already seen in several examples, a value is stored in a variable with an *assignment statement*. For example, the following statement copies the value 12 into the variable `unitsSold`.

```
unitsSold = 12;
```

The = symbol is called the *assignment operator*. Operators perform operations on data. The data that operators work with are called *operands*. The assignment operator has two operands. In the previous statement, the operands are `unitsSold` and `12`.

In an assignment statement, C++ requires the name of the variable receiving the assignment to appear on the left side of the operator. The following statement is incorrect.

```
12 = unitsSold;     // Incorrect!
```

In C++ terminology, the operand on the left side of the = symbol must be an *lvalue*. It is called an lvalue because it is a value that may appear on the left side of an assignment operator. An lvalue is something that identifies a place in memory whose contents may be changed. Most of the time this will be a variable name. The operand on the right side of the = symbol must be an *rvalue*. An rvalue is any expression that has a value. The assignment statement takes the value of the rvalue and puts it in the memory location of the object identified by the lvalue.

You may also assign values to variables as part of the definition. This is called *initialization*. Program 2-19 shows how it is done.

Program 2-19

```
1  // This program shows variable initialization.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7       int month = 2, days = 28;
8
9       cout << "Month " << month << " has " << days << " days.\n";
10      return 0;
11 }
```

Program Output

```
Month 2 has 28 days.
```

As you can see, this simplifies the program and reduces the number of statements that must be typed by the programmer. Here are examples of other definition statements that perform initialization.

```
double interestRate = 12.9;
char stockCode = 'D';
long customerNum = 459L;
```

Of course, there are always variations on a theme. C++ allows you to define several variables and only initialize some of them. Here is an example of such a definition:

```
int flightNum = 89, travelTime, departure = 10, distance;
```

The variable `flightNum` is initialized to 89 and `departure` is initialized to 10. The variables `travelTime` and `distance` remain uninitialized.

Declaring Variables With the `auto` Key Word

C++ 11 introduces an alternative way to define variables, using the `auto` key word and an initialization value. Here is an example:

```
auto amount = 100;
```

Notice that this statement uses the word `auto` instead of a data type. The `auto` key word tells the compiler to determine the variable's data type from the initialization value. In this example the initialization value, 100, is an `int`, so `amount` will be an `int` variable. Here are other examples:

```
auto interestRate = 12.0;
auto stockCode = 'D';
auto customerNum = 459L;
```

In this code, the `interestRate` variable will be a `double` because the initialization value, 12.0, is a `double`. The `stockCode` variable will be a `char` because the initialization value, `'D'`, is a `char`. The `customerNum` variable will be a `long` because the initialization value, `459L`, is a `long`.

These examples show how to use the `auto` key word, but they don't really show its usefulness. The `auto` key word is intended to simplify the syntax of declarations that are more complex than the ones shown here. Later in the book, you will see examples of how the `auto` key word can improve the readability of complex definition statements.

2.13 Scope

CONCEPT: A variable's scope is the part of the program that has access to the variable.

Every variable has a *scope*. The scope of a variable is the part of the program where the variable may be used. The rules that define a variable's scope are complex, and you will only be introduced to the concept here. In other sections of the book we will revisit this topic and expand on it.

The first rule of scope you should learn is that a variable cannot be used in any part of the program before the definition. Program 2-20 illustrates this.

Program 2-20

```
1  // This program can't find its variable.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7       cout << value; // ERROR! value not defined yet!
8
9       int value = 100;
10      return 0;
11 }
```

The program will not work because line 7 attempts to send the contents of the variable `value` to `cout` before the variable is defined. The compiler reads your program from top to bottom. If it encounters a statement that uses a variable before the variable is defined, an error will result. To correct the program, the variable definition must be put before any statement that uses it.

2.14 Arithmetic Operators

CONCEPT: There are many operators for manipulating numeric values and performing arithmetic operations.

C++ offers a multitude of operators for manipulating data. Generally, there are three types of operators: *unary*, *binary*, and *ternary*. These terms reflect the number of operands an operator requires.

VideoNote
Assignment Statements and Simple Math Expressions

Unary operators only require a single operand. For example, consider the following expression:

```
-5
```

Of course, we understand this represents the value negative five. The literal 5 is preceded by the minus sign. The minus sign, when used this way, is called the *negation operator*. Since it only requires one operand, it is a unary operator.

Binary operators work with two operands. The assignment operator is in this category. Ternary operators, as you may have guessed, require three operands. C++ only has one ternary operator, which will be discussed in Chapter 4.

Arithmetic operations are very common in programming. Table 2-9 shows the common arithmetic operators in C++.

Table 2-9 Fundamental Arithmetic Operators

Operator	Meaning	Type	Example
`+`	Addition	Binary	`total = cost + tax;`
`-`	Subtraction	Binary	`cost = total - tax;`
`*`	Multiplication	Binary	`tax = cost * rate;`
`/`	Division	Binary	`salePrice = original / 2;`
`%`	Modulus	Binary	`remainder = value % 3;`

Each of these operators works as you probably expect. The addition operator returns the sum of its two operands. In the following assignment statement, the variable `amount` will be assigned the value 12:

```
amount = 4 + 8;
```

The subtraction operator returns the value of its right operand subtracted from its left operand. This statement will assign the value 98 to `temperature`:

```
temperature = 112 - 14;
```

The multiplication operator returns the product of its two operands. In the following statement, `markUp` is assigned the value 3:

```
markUp = 12 * 0.25;
```

The division operator returns the quotient of its left operand divided by its right operand. In the next statement, `points` is assigned the value 5:

```
points = 100 / 20;
```

It is important to note that when both of the division operator's operands are integers, the result of the division will also be an integer. If the result has a fractional part, it will be thrown away. We will discuss this behavior, which is known as *integer division*, in greater detail later in this section.

The modulus operator, which only works with integer operands, returns the remainder of an integer division. The following statement assigns 2 to `leftOver`:

```
leftOver = 17 % 3;
```

In Chapter 3 you will learn how to use these operators in more complex mathematical formulas. For now we will concentrate on their basic usage. For example, suppose we need to write a program that calculates and displays an employee's total wages for the week. The regular hours for the work week are 40, and any hours worked over 40 are considered overtime. The employee earns $18.25 per hour for regular hours and $27.78 per hour for overtime hours. The employee has worked 50 hours this week. The following pseudocode algorithm shows the program's logic.

Regular wages = base pay rate × regular hours
Overtime wages = overtime pay rate × overtime hours
Total wages = regular wages + overtime wages
Display the total wages

Program 2-21 shows the C++ code for the program.

Program 2-21

```
1  // This program calculates hourly wages, including overtime.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7      double regularWages,             // To hold regular wages
8             basePayRate = 18.25,      // Base pay rate
9             regularHours = 40.0,      // Hours worked less overtime
10            overtimeWages,            // To hold overtime wages
11            overtimePayRate = 27.78,  // Overtime pay rate
12            overtimeHours = 10,       // Overtime hours worked
13            totalWages;               // To hold total wages
14
15     // Calculate the regular wages.
16     regularWages = basePayRate * regularHours;
17
18     // Calculate the overtime wages.
19     overtimeWages = overtimePayRate * overtimeHours;
20
21     // Calculate the total wages.
22     totalWages = regularWages + overtimeWages;
23
24     // Display the total wages.
25     cout << "Wages for this week are $" << totalWages << endl;
26     return 0;
27 }
```

Program Output

```
Wages for this week are $1007.8
```

Let's take a closer look at the program. As mentioned in the comments, there are variables for regular wages, base pay rate, regular hours worked, overtime wages, overtime pay rate, overtime hours worked, and total wages.

Here is line 16, which multiplies `basePayRate` times `regularHours` and stores the result in `regularWages`:

```
regularWages = basePayRate * regularHours;
```

Here is line 19, which multiplies `overtimePayRate` times `overtimeHours` and stores the result in `overtimeWages`:

```
overtimeWages = overtimePayRate * overtimeHours;
```

Line 22 adds the regular wages and the overtime wages and stores the result in `totalWages`:

```
totalWages = regularWages + overtimeWages;
```

Line 25 displays the message on the screen reporting the week's wages.

Integer Division

When both operands of a division statement are integers, the statement will result in *integer division*. This means the result of the division will be an integer as well. If there is a remainder, it will be discarded. For example, look at the following code:

```
double number;
number = 5 / 2;
```

This code divides 5 by 2 and assigns the result to the `number` variable. What will be stored in `number`? You would probably assume that 2.5 would be stored in `number` because that is the result your calculator shows when you divide 5 by 2. However, that is not what happens when the previous C++ code is executed. Because the numbers 5 and 2 are both integers, the fractional part of the result will be thrown away, or truncated. As a result, the value 2 will be assigned to the `number` variable.

In the previous code, it doesn't matter that the `number` variable is declared as a `double` because the fractional part of the result is discarded before the assignment takes place. In order for a division operation to return a floating-point value, one of the operands must be of a floating-point data type. For example, the previous code could be written as follows:

```
double number;
number = 5.0 / 2;
```

In this code, 5.0 is treated as a floating-point number, so the division operation will return a floating-point number. The result of the division is 2.5.

In the Spotlight:

Calculating Percentages and Discounts

Determining percentages is a common calculation in computer programming. Although the % symbol is used in general mathematics to indicate a percentage, most programming languages (including C++) do not use the % symbol for this purpose. In a program, you have to convert a percentage to a floating-point number, just as you would if you were using a calculator. For example, 50 percent would be written as 0.5 and 2 percent would be written as 0.02.

Let's look at an example. Suppose you earn $6,000 per month and you are allowed to contribute a portion of your gross monthly pay to a retirement plan. You want to determine the amount of your pay that will go into the plan if you contribute 5 percent, 7 percent, or 10 percent of your gross wages. To make this determination you write the program shown in Program 2-22.

Program 2-22

```
1  // This program calculates the amount of pay that
2  // will be contributed to a retirement plan if 5%,
3  // 7%, or 10% of monthly pay is withheld.
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9       // Variables to hold the monthly pay and the
10      // amount of contribution.
11      double monthlyPay = 6000.0, contribution;
12
13      // Calculate and display a 5% contribution.
14      contribution = monthlyPay * 0.05;
15      cout << "5 percent is $" << contribution
16           << " per month.\n";
17
18      // Calculate and display a 7% contribution.
19      contribution = monthlyPay * 0.07;
20      cout << "7 percent is $" << contribution
21           << " per month.\n";
22
23      // Calculate and display a 10% contribution.
24      contribution = monthlyPay * 0.1;
25      cout << "10 percent is $" << contribution
26           << " per month.\n";
27
28      return 0;
29 }
```

Program Output

```
5 percent is $300 per month.
7 percent is $420 per month.
10 percent is $600 per month.
```

Line 11 defines two variables: `monthlyPay` and `contribution`. The `monthlyPay` variable, which is initialized with the value 6000.0, holds the amount of your monthly pay. The `contribution` variable will hold the amount of a contribution to the retirement plan.

The statements in lines 14 through 16 calculate and display 5 percent of the monthly pay. The calculation is done in line 14, where the `monthlyPay` variable is multiplied by 0.05. The result is assigned to the `contribution` variable, which is then displayed in line 15.

Similar steps are taken in Lines 18 through 21, which calculate and display 7 percent of the monthly pay, and lines 24 through 26, which calculate and display 10 percent of the monthly pay.

Calculating a Percentage Discount

Another common calculation is determining a percentage discount. For example, suppose a retail business sells an item that is regularly priced at $59.95 and is planning to have a sale where the item's price will be reduced by 20 percent. You have been asked to write a program to calculate the sale price of the item.

To determine the sale price you perform two calculations:

- First, you get the amount of the discount, which is 20 percent of the item's regular price.
- Second, you subtract the discount amount from the item's regular price. This gives you the sale price.

Program 2-23 shows how this is done in C++.

Program 2-23

```
1  // This program calculates the sale price of an item
2  // that is regularly priced at $59.95, with a 20 percent
3  // discount subtracted.
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9       // Variables to hold the regular price, the
10      // amount of a discount, and the sale price.
11      double regularPrice = 59.95, discount, salePrice;
12
13      // Calculate the amount of a 20% discount.
14      discount = regularPrice * 0.2;
15
16      // Calculate the sale price by subtracting the
17      // discount from the regular price.
18      salePrice = regularPrice - discount;
19
20      // Display the results.
21      cout << "Regular price: $" << regularPrice << endl;
22      cout << "Discount amount: $" << discount << endl;
23      cout << "Sale price: $" << salePrice << endl;
24      return 0;
25 }
```

Program Output

```
Regular price: $59.95
Discount amount: $11.99
Sale price: $47.96
```

Line 11 defines three variables. The `regularPrice` variable holds the item's regular price, and is initialized with the value 59.95. The `discount` variable will hold the amount of the discount once it is calculated. The `salePrice` variable will hold the item's sale price.

Line 14 calculates the amount of the 20 percent discount by multiplying `regularPrice` by 0.2. The result is stored in the `discount` variable. Line 18 calculates the sale price by subtracting `discount` from `regularPrice`. The result is stored in the `salePrice` variable. The `cout` statements in lines 21 through 23 display the item's regular price, the amount of the discount, and the sale price.

In the Spotlight:

Using the Modulus Operator and Integer Division

The modulus operator (`%`) is surprisingly useful. For example, suppose you need to extract the rightmost digit of a number. If you divide the number by 10, the remainder will be the rightmost digit. For instance, 123 ÷ 10 = 12 with a remainder of 3. In a computer program you would use the modulus operator to perform this operation. Recall that the modulus operator divides an integer by another integer, and gives the remainder. This is demonstrated in Program 2-24. The program extracts the rightmost digit of the number 12345.

Program 2-24

```
1  // This program extracts the rightmost digit of a number.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7      int number = 12345;
8      int rightMost = number % 10;
9
10     cout << "The rightmost digit in "
11          << number << " is "
12          << rightMost << endl;
13
14     return 0;
15 }
```

Program Output

```
The rightmost digit in 12345 is 5
```

Interestingly, the expression `number % 100` will give you the rightmost two digits in `number`, the expression `number % 1000` will give you the rightmost three digits in `number`, etc.

The modulus operator (`%`) is useful in many other situations. For example, Program 2-25 converts 125 seconds to an equivalent number of minutes, and seconds.

Program 2-25

```
1  // This program converts seconds to minutes and seconds.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7      // The total seconds is 125.
8      int totalSeconds = 125;
9
10     // Variables for the minutes and seconds
11     int minutes, seconds;
12
13     // Get the number of minutes.
14     minutes = totalSeconds / 60;
15
16     // Get the remaining seconds.
17     seconds = totalSeconds % 60;
18
19     // Display the results.
20     cout << totalSeconds << " seconds is equivalent to:\n";
21     cout << "Minutes: " << minutes << endl;
22     cout << "Seconds: " << seconds << endl;
23     return 0;
24 }
```

Program Output

```
125 seconds is equivalent to:
Minutes: 2
Seconds: 5
```

Let's take a closer look at the code:

- Line 8 defines an `int` variable named `totalSeconds`, initialized with the value 125.
- Line 11 declares the `int` variables `minutes` and `seconds`.
- Line 14 calculates the number of minutes in the specified number of seconds. There are 60 seconds in a minute, so this statement divides `totalSeconds` by 60. Notice that we are performing integer division in this statement. Both `totalSeconds` and the numeric literal 60 are integers, so the division operator will return an integer result. This is intentional because we want the number of minutes with no fractional part.
- Line 17 calculates the number of remaining seconds. There are 60 seconds in a minute, so this statement uses the `%` operator to divide the `totalSeconds` by 60, and get the remainder of the division. The result is the number of remaining seconds.
- Lines 20 through 22 display the number of minutes and seconds.

Checkpoint

2.21 Is the following assignment statement valid or invalid? If it is invalid, why?

```
72 = amount;
```

2.22 How would you consolidate the following definitions into one statement?

```
int x = 7;
int y = 16;
int z = 28;
```

2.23 What is wrong with the following program? How would you correct it?

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

int main()
{
  number = 62.7;
  double number;
  cout << number << endl;
  return 0;
}
```

2.24 Is the following an example of integer division or floating-point division? What value will be stored in `portion`?

```
portion = 70 / 3;
```

2.15 Comments

CONCEPT: Comments are notes of explanation that document lines or sections of a program. Comments are part of the program, but the compiler ignores them. They are intended for people who may be reading the source code.

It may surprise you that one of the most important parts of a program has absolutely no impact on the way it runs. In fact, the compiler ignores this part of a program. Of course, I'm speaking of the comments.

As a beginning programmer, you might be resistant to the idea of liberally writing comments in your programs. After all, it can seem more productive to write code that actually does something! It is crucial, however, that you develop the habit of thoroughly annotating your code with descriptive comments. It might take extra time now, but it will almost certainly save time in the future.

Imagine writing a program of medium complexity with about 8,000 to 10,000 lines of C++ code. Once you have written the code and satisfactorily debugged it, you happily put it away and move on to the next project. Ten months later you are asked to make a modification to the program (or worse, track down and fix an elusive bug). You open the file that contains your source code and stare at thousands of statements that now make no sense at all. If only you had left some notes to yourself explaining the program's code. Of course it's too late now. All that's left to do is decide what will take less time: figuring out the old program or completely rewriting it!

This scenario might sound extreme, but it's one you don't want to happen to you. Real-world programs are big and complex. Thoroughly documented code will make your life easier, not to mention the other programmers who may have to read your code in the future.

Single-Line Comments

You have already seen one way to place comments in a C++ program. You simply place two forward slashes (`//`) where you want the comment to begin. The compiler ignores everything from that point to the end of the line. Program 2-26 shows that comments may be placed liberally throughout a program.

Program 2-26

```
1  // PROGRAM: PAYROLL.CPP
2  // Written by Herbert Dorfmann
3  // This program calculates company payroll
4  // Last modification: 8/20/2014
5  #include <iostream>
6  using namespace std;
7
8  int main()
9  {
10   double payRate;   // Holds the hourly pay rate
11   double hours;     // Holds the hours worked
12   int employNumber; // Holds the employee number
```

(The remainder of this program is left out.)

In addition to telling who wrote the program and describing the purpose of variables, comments can also be used to explain complex procedures in your code.

Multi-Line Comments

The second type of comment in C++ is the multi-line comment. *Multi-line comments* start with `/*` (a forward slash followed by an asterisk) and end with `*/` (an asterisk followed by a forward slash). Everything between these markers is ignored. Program 2-27 illustrates how multi-line comments may be used. Notice that a multi-line comment starts in line 1 with the `/*` symbol, and it ends in line 6 with the `*/` symbol.

Program 2-27

```
1  /*
2       PROGRAM: PAYROLL.CPP
3       Written by Herbert Dorfmann
4       This program calculates company payroll
5       Last modification: 8/20/2014
6  */
7
8  #include <iostream>
```

```
 9  using namespace std;
10
11  int main()
12  {
13      double payRate;   // Holds the hourly pay rate
14      double hours;     // Holds the hours worked
15      int employNumber; // Holds the employee number
```

(The remainder of this program is left out.)

Unlike a comment started with `//`, a multi-line comment can span several lines. This makes it more convenient to write large blocks of comments because you do not have to mark every line. Consequently, the multi-line comment is inconvenient for writing single-line comments because you must type both a beginning and ending comment symbol.

NOTE: Many programmers use a combination of single-line comments and multi-line comments in their programs. Convenience usually dictates which style to use.

Remember the following advice when using multi-line comments:

- Be careful not to reverse the beginning symbol with the ending symbol.
- Be sure not to forget the ending symbol.

Both of these mistakes can be difficult to track down and will prevent the program from compiling correctly.

2.16 Named Constants

CONCEPT: Literals may be given names that symbolically represent them in a program.

Assume the following statement appears in a banking program that calculates data pertaining to loans:

```
amount = balance * 0.069;
```

In such a program, two potential problems arise. First, it is not clear to anyone other than the original programmer what 0.069 is. It appears to be an interest rate, but in some situations there are fees associated with loan payments. How can the purpose of this statement be determined without painstakingly checking the rest of the program?

The second problem occurs if this number is used in other calculations throughout the program and must be changed periodically. Assuming the number is an interest rate, what if the rate changes from 6.9 percent to 7.2 percent? The programmer will have to search through the source code for every occurrence of the number.

Both of these problems can be addressed by using named constants. A *named constant* is like a variable, but its content is read-only and cannot be changed while the program is running. Here is a definition of a named constant:

```
const double INTEREST_RATE = 0.069;
```

It looks just like a regular variable definition except that the word `const` appears before the data type name, and the name of the variable is written in all uppercase characters. The key word `const` is a qualifier that tells the compiler to make the variable read-only. Its value will remain constant throughout the program's execution. It is not required that the variable name be written in all uppercase characters, but many programmers prefer to write them this way so they are easily distinguishable from regular variable names.

An initialization value must be given when defining a constant with the `const` qualifier, or an error will result when the program is compiled. A compiler error will also result if there are any statements in the program that attempt to change the value of a named constant.

An advantage of using named constants is that they make programs more self-documenting. The following statement

```
amount = balance * 0.069;
```

can be changed to read

```
amount = balance * INTEREST_RATE;
```

A new programmer can read the second statement and know what is happening. It is evident that `balance` is being multiplied by the interest rate. Another advantage to this approach is that widespread changes can easily be made to the program. Let's say the interest rate appears in a dozen different statements throughout the program. When the rate changes, the initialization value in the definition of the named constant is the only value that needs to be modified. If the rate increases to 7.2%, the definition is changed to the following:

```
const double INTEREST_RATE = 0.072;
```

The program is then ready to be recompiled. Every statement that uses `INTEREST_RATE` will then use the new value.

Named constants can also help prevent typographical errors in a program's code. For example, suppose you use the number 3.14159 as the value of *pi* in a program that performs various geometric calculations. Each time you type the number 3.14159 in the program's code, there is a chance that you will make a mistake with one or more of the digits. As a result, the program will not produce the correct results. To help prevent a mistake such as this, you can define a named constant for *pi*, initialized with the correct value, and then use that constant in all of the formulas that require its value. Program 2-28 shows an example. It calculates the circumference of a circle that has a diameter of 10.

Program 2-28

```
1  // This program calculates the circumference of a circle.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7      // Constants
8      const double PI = 3.14159;
9      const double DIAMETER = 10.0;
10
11     // Variable to hold the circumference
```

```
12      double circumference;
13
14      // Calculate the circumference.
15      circumference = PI * DIAMETER;
16
17      // Display the circumference.
18      cout << "The circumference is: " << circumference << endl;
19      return 0;
20  }
```

Program Output

```
The circumference is: 31.4159
```

Let's take a closer look at the program. Line 8 defines a constant `double` named `PI`, initialized with the value 3.14159. This constant will be used for the value of *pi* in the program's calculation. Line 9 defines a constant `double` named `DIAMETER`, initialized with the value 10. This will be used for the circle's diameter. Line 12 defines a `double` variable named `circumference`, which will be used to hold the circle's circumference. Line 15 calculates the circle's circumference by multiplying `PI` by `DIAMETER`. The result of the calculation is assigned to the `circumference` variable. Line 18 displays the circle's circumference.

Checkpoint

2.25 Write statements using the `const` qualifier to create named constants for the following literal values:

Literal Value	Description
2.71828	Euler's number (known in mathematics as *e*)
5.256E5	Number of minutes in a year
32.2	The gravitational acceleration constant (in feet per second^2)
9.8	The gravitational acceleration constant (in meters per second^2)
1609	Number of meters in a mile

2.17 Programming Style

CONCEPT: Programming style refers to the way a programmer uses identifiers, spaces, tabs, blank lines, and punctuation characters to visually arrange a program's source code. These are some, but not all, of the elements of programming style.

In Chapter 1 you learned that syntax rules govern the way a language may be used. The syntax rules of C++ dictate how and where to place key words, semicolons, commas, braces, and other components of the language. The compiler's job is to check for syntax errors and, if there are none, generate object code.

When the compiler reads a program it processes it as one long stream of characters. The compiler doesn't care that each statement is on a separate line, or that spaces separate operators from operands. Humans, on the other hand, find it difficult to read programs that aren't written in a visually pleasing manner. Consider Program 2-29 for example.

Program 2-29

```
1  #include <iostream>
2  using namespace std;int main(){double shares=220.0;
3  double avgPrice=14.67;cout<<"There were "<<shares
4  <<" shares sold at $"<<avgPrice<<" per share.\n";
5  return 0;}
```

Program Output

```
There were 220 shares sold at $14.67 per share.
```

Although the program is syntactically correct (it doesn't violate any rules of C++), it is very difficult to read. The same program is shown in Program 2-30, written in a more reasonable style.

Program 2-30

```
1  // This example is much more readable than Program 2-29.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7       double shares = 220.0;
8       double avgPrice = 14.67;
9
10      cout << "There were " << shares << " shares sold at $";
11      cout << avgPrice << " per share.\n";
12      return 0;
13 }
```

Program Output

```
There were 220 shares sold at $14.67 per share.
```

Programming style refers to the way source code is visually arranged. Ideally, it is a consistent method of putting spaces and indentions in a program so visual cues are created. These cues quickly tell a programmer important information about a program.

For example, notice in Program 2-30 that inside the function `main`'s braces each line is indented. It is a common C++ style to indent all the lines inside a set of braces. You will also notice the blank line between the variable definitions and the `cout` statements. This is intended to visually separate the definitions from the executable statements.

NOTE: Although you are free to develop your own style, you should adhere to common programming practices. By doing so, you will write programs that visually make sense to other programmers.

Another aspect of programming style is how to handle statements that are too long to fit on one line. Because C++ is a free-flowing language, it is usually possible to spread a statement over several lines. For example, here is a `cout` statement that uses five lines:

```
cout << "The Fahrenheit temperature is "
     << fahrenheit
     << " and the Celsius temperature is "
     << celsius
     << endl;
```

This statement will work just as if it were typed on one line. Here is an example of variable definitions treated similarly:

```
int fahrenheit,
    celsius,
    kelvin;
```

There are many other issues related to programming style. They will be presented throughout the book.

Review Questions and Exercises

Short Answer

1. How many operands does each of the following types of operators require?

   _____ `Unary`

   _____ `Binary`

   _____ `Ternary`

2. How may the `double` variables `temp`, `weight`, and `age` be defined in one statement?
3. How may the `int` variables `months`, `days`, and `years` be defined in one statement, with `months` initialized to 2 and `years` initialized to 3?
4. Write assignment statements that perform the following operations with the variables `a`, `b`, and `c`.

   A) Adds 2 to `a` and stores the result in `b`.

   B) Multiplies `b` times 4 and stores the result in `a`.

   C) Divides `a` by 3.14 and stores the result in `b`.

   D) Subtracts 8 from `b` and stores the result in `a`.

   E) Stores the value 27 in a.

   F) Stores the character 'K' in `c`.

   G) Stores the ASCII code for 'B' in `c`.

5. Is the following comment written using single-line or multi-line comment symbols?

   ```
   /* This program was written by M. A. Codewriter*/
   ```

6. Is the following comment written using single-line or multi-line comment symbols?

   ```
   // This program was written by M. A. Codewriter
   ```