murach's
C++
Programming

(Chapter 2)

Thanks for downloading this chapter from *Murach's C++ Programming*. We hope it will show you how easy it is to learn from any Murach book, with its paired-pages presentation, its "how-to" headings, its practical coding examples, and its clear, concise style.

To view the full table of contents for this book, you can go to our **website**. From there, you can read more about this book, you can find out about any additional downloads that are available, and you can review our other books on related topics.

Thanks for your interest in our books!

Mike Murach & Associates, Inc.

4340 N. Knoll Ave. • Fresno, CA 93722
www.murach.com • murachbooks@murach.com

What to expect in a Murach book

"I had a development environment up & running within half an hour and was dabbling with code 15 minutes after that!"

Andy Bonner, Solutions Architect, New Zealand

"Every two-page spread is designed to have all the detail about a topic on the left hand page and everything that a developer would need for reference material on the right hand page. This is a fantastic approach that I have not seen before in technical books."

Jim O'Neil, Denver Visual Studio .NET

"Another thing I like is the exercises at the end of each chapter. They're a great way to reinforce the main points of each chapter and force you to get your hands dirty."

Hien Luu, SD Forum/Java SIG

"Great book. I'm hard to please, and as a programmer and engineer, I was surprised by how informative this book was."

Posted at an online bookseller

"Your books are by far the best resource I have come across for aspiring developers like me. Most programming intructional resources teach us fundamentals and concepts of OOP (which is fine), but for people who wish to learn how to develop truly professional applications, Murach bridges the gap."

Blake Biessener, Graduate Student

"I kept your book at my side throughout the entire project. It was indispensable to me. The answers were right there at every turn. All the examples made sense to me, and they all worked!"

Alan Vogt, ETL Consultant, Information Builders, Inc.

"You folks make the hard stuff seem easy."

Thomas Finn, Sr. Software Developer, Illinois

2

How to write your first programs

Once you've installed an IDE and the C++ compiler, the quickest and best way to *learn* C++ programming is to *do* C++ programming. That's why this chapter shows you how to write complete C++ programs that get input from a user, make calculations, and display output. When you finish this chapter, you should be able to write comparable programs of your own.

Basic coding skills

This chapter starts by introducing you to some basic coding skills. You'll use these skills for every C++ program you develop.

How to code statements

The *statements* in a program direct the operation of the program. When you code a statement, you can start it anywhere in a coding line, you can continue it from one line to another, and you can code one or more spaces anywhere a single space is valid.

To end most statements, you use a semicolon. In the first example in figure 2-1, for example, all of the lines that end with a semicolon are statements. But when a statement requires a set of braces (`{}`), it ends with the right brace. Then, the statements within the braces are referred to as a *block* of code. For example, the statement that defines the main() function shown in this figure contains a block of code.

To make a program easier to read, you should use indentation and spacing to align statements and blocks of code. This is illustrated by the program in this figure and by all of the programs and examples in this book.

Note that the first line of code doesn't end with a semicolon. That's because it's a *preprocessor directive*, not a C++ statement. Later in this chapter, you'll learn more about coding this directive.

How to code comments

The *comments* in a program typically document what the statements do. Since the compiler ignores comments, you can include them anywhere in a program without affecting your code. In the first example in figure 2-1, the comments are shaded.

A *single-line comment* is typically used to describe one or more lines of code. This type of comment starts with two slashes (`//`) that tell the compiler to ignore all characters until the end of the current line. In the first example in this figure, you can see four single-line comments that are used to describe groups of statements. In addition, two single-line comments are coded after the code on a line. This type of single-line comment is sometimes referred to as an *end-of-line comment*.

The second example in this figure shows how to code a *block comment*. This type of comment is typically used to document information that applies to a block of code. This information can include the author's name, program completion date, the purpose of the code, the files used by the code, and so on.

Although many programmers sprinkle their code with comments, that shouldn't be necessary if you write code that's easy to read and understand. Instead, you should use comments only to clarify code that's difficult to understand. In this figure, for example, an experienced C++ programmer wouldn't need any of the single-line comments.

A program that consists of statements and comments

```
#include <iostream>      // a preprocessor directive, not a statement

using namespace std;

int main()
{
    cout << "Welcome to the Calorie Calculator\n\n";

    // get number of servings from user
    double servings;
    cout << "Enter servings per food item: ";
    cin >> servings;

    // get number of calories from user
    double calories;
    cout << "Enter calories per serving: ";
    cin >> calories;

    // calculate total calories
    double total_calories = servings * calories;

    // display total calories
    cout << "Total calories: " << total_calories << endl << endl;

    return 0; // return a value indicating normal exit
}
```

A block comment that could be coded at the start of a program

```
/*
 * Author:  M. Delamater
 * Purpose: This program uses the console to get servings and calories from
 *          the user. It then calculates and displays the total calories.
 */
```

Description

- *Statements* direct the operations of a program, and *comments* typically document what the statements do.

- Most statements end with a semicolon.

- You can start a statement at any point in a line and continue the statement from one line to the next. To make a program easier to read, you should use indentation and extra spaces to align statements and parts of statements.

- Code within a set of braces (`{}`) can be referred to as a *block* of code.

- To code a *single-line comment*, type // followed by the comment. You can code a single-line comment on a line by itself or after a statement. A comment that's coded after a statement is sometimes called an *end-of-line comment*.

- A *block comment* typically consists of multiple lines. To code a block comment, type /* at the start of the block and */ at the end. You can also code asterisks to identify the lines in the block, but that isn't required.

Figure 2-1   How to code statements and comments

One problem with comments is that they may not accurately represent what the code does. This often happens when a programmer changes the code, but doesn't change the comments that go along with it. Then, it's even harder to understand the code because the comments are misleading. So if you change the code that you've written comments for, be sure to change the comments too.

How to code a main() function

A *function* is a named, reusable block of code that performs a task. The *main() function* is a special kind of function that's automatically executed when your program starts. All C++ programs contain a main() function.

If you use an IDE to create a project, it typically generates a main() function for you. Now, figure 2-2 presents the syntax for declaring a main() function.

The main() function starts with the C++ int keyword. This indicates that the function returns an integer value. You'll learn more about this value in a moment. After specifying the type of data that the function returns, you code the name of the function followed by a pair of parentheses. In this case, the name of the function is main.

For a main() function, you can leave the parentheses empty, as in the first example. Or, you can code two *parameters*, as in the second example. You'll learn more about function parameters later in this book. For now, all you need to know about these parameters is that they are optional, so you don't need to include them. In fact, even though some IDEs generate these parameters by default, the programs presented in this book don't use them. As a result, there's no need to include them for the examples presented in this book.

The next line of code in the main() function is the opening brace for the *body* of the function. This opening brace can be on its own line, as shown in the first and second examples. Many C++ style guides recommend this approach. As a result, this book uses this approach for most of its code. Sometimes, though, to save vertical space, this book may code the opening brace on the first line as shown in the third example. That's an acceptable style too, although coding the opening brace on its own line is more common when working with C++.

The body of the main() function contains the statements that run the program. The last statement in the main() function is a *return statement*. This statement returns an integer value that tells the operating system whether the program exited normally or abnormally. If the main() function doesn't contain a return statement, C++ returns a value of 0 to the operating system by default. Because of this, it's common to omit the return statement to save space.

When you're working with C++11 or later, you can also return the EXIT_SUCCESS and EXIT_FAILURE values. The advantage of this approach is that it makes the return statement easier to read and understand. However, since the main() function automatically returns a value that indicates a normal exit, you typically don't need to code a return statement for your main() function.

A main() function with no parameters

```
int main()
{
    // statements
    return 0;
}
```

A main() function with two parameters

```
int main(int argc, char** argv)
{
    // statements
    return 0;
}
```

A main() function with different brace placement

```
int main() {
    // statements
    return EXIT_SUCCESS;
}
```

Three standard return values for the main() function

Value	Description	C++ version
0	Normal exit	All
EXIT_SUCCESS	Normal exit	C++11 or later
EXIT_FAILURE	Abnormal exit	C++11 or later

Description

- Every C++ program contains one *main() function* that is called by the operating system.
- The statements between the braces in the main() function are run when the program is executed.
- In most cases, you can code the main() function with no parameters. To do that, type an empty set of parentheses after you type the function's name.
- If you need the execution environment to pass arguments to the main() function, you can code the parameters named argc and argv as shown in the second example. However, these parameters aren't necessary for any of the programs presented in this book.
- You can code the opening brace of the main() function on the same line as the parentheses or on the next line. Many C++ style guides recommend coding the opening brace on its own line, and this book follows that convention whenever possible.
- You can end the main() function by coding a *return statement* that returns an integer value that tells the operating system how the program exited. If you omit this return statement, the main() function automatically returns a value of 0, which indicates a normal exit.

Figure 2-2   How to code a main() function

The last line of code for the main() function is the closing brace. Unlike the opening brace, this one should always be on its own line. That's because a common syntax error is forgetting to code a closing brace. Thus, leaving the closing brace on its own line makes it easier to see whether the opening brace has a corresponding closing brace.

How to create identifiers

As you code a C++ program, you need to create and use *identifiers*. These are the names that you define in your code.

Figure 2-3 shows you how to create identifiers. In brief, you must start each identifier with a letter or an underscore. After that first character, you can use any combination of letters, underscores, or digits.

Since C++ is case-sensitive, you need to be careful when you create and use identifiers. If, for example, you define an identifier named subtotal, you can't refer to it later as Subtotal. That's a common coding error.

When you create an identifier, you should try to make the name both meaningful and easy to remember. To make a name meaningful, you should use as many characters as you need, so it's easy for other programmers to read and understand your code. For instance, net_price is more meaningful than nprice, and nprice is more meaningful than np.

Notice here that the name of this identifier is formed by separating two lower-case words with an underscore. This can be referred to as *snake case*, and it's used for most identifiers in this book. However, some developers prefer to name identifiers by capitalizing the first letter of each word. This is called *camel case*.

To make a name easy to remember, you should avoid abbreviations. If, for example, you use nwcst as an identifier, you may later have difficulty remembering whether it was ncust, nwcust, or nwcst. If you code the name as new_customer, however, you won't have any trouble remembering what it was. In addition, that's easy for other programmers to understand. Yes, you type more characters when you create identifiers that are meaningful and easy to remember, but that will be more than justified by the time you'll save when you test, debug, and maintain the program.

On the other hand, programmers often use just one or two lowercase letters for some common identifiers. For example, the letter *i* is often used to identify an integer that's used as a counter variable.

When you create identifiers, you can't use the same name as any of the C++ keywords shown in this figure. These keywords are reserved by the C++ language. To help you identify keywords in your code, most IDEs display these keywords in a different color than the rest of the code. As you progress through this book, you'll learn how to use many of these keywords.

For compatibility with programs written in the C language, you should also avoid creating identifiers that are the same as any of the C keywords. Most of the C keywords are the same as the C++ keywords, so this isn't hard to do. However, there are a few keywords shown in this figure that are specific to C. A C++ compiler might allow you to use one of these C keywords as an identifier, but you shouldn't.

Valid identifiers

```
subtotal        i               TITLE
subTotal        x               TAX_RATE
SubTotal        item1
sub_total       item2
_sub_total      June2011
```

The rules for naming an identifier

- Start each identifier with an upper- or lowercase letter or underscore.

- Use letters, underscores, or digits for subsequent characters.

- Don't use C++ or C keywords.

C++ keywords

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

Additional C keywords

_Alignas	_Bool	_Imaginary	_Thread_local
_Alignof	_Complex	_Noreturn	restrict
_Atomic	_Generic	_Static_assert	

Description

- An *identifier* is any name that you create in a C++ program. These can be the names of variables, functions, classes, and so on.

- A *keyword* is a word that's reserved by the C++ language. As a result, you can't use keywords as identifiers. For backward compatibility, you should also avoid C keywords.

- When you refer to an identifier, be sure to use the correct uppercase and lowercase letters because C++ is a case-sensitive language.

Figure 2-3   How to create identifiers

How to work with numeric variables

The topics that follow show how to work with numeric variables. They introduce you to the use of variables, assignment statements, arithmetic expressions, and two of the built-in data types that are defined by the C++ language.

How to define and initialize variables

Figure 2-4 starts by summarizing two of the built-in *data types* that are available from C++. You can use the int data type to store *integers*, which are numbers that don't contain decimal places (whole numbers), and you can use the *double* data type to store numbers that do contain decimal places.

A *variable* stores a value that can change, or *vary*, as a program executes. Before you can use a variable, you must *define* its data type, which is simply the type of data it will store, and its name. In addition, it's a good practice to *initialize* a variable by *assigning* an initial value to it. To do that, you can use either of the techniques described in this figure.

To show how this works, the first example uses one statement to define an int variable named counter. Then, it uses a second statement to initialize the variable by assigning an initial value of 1 to that variable.

However, it's often easier to define and initialize a variable in a single statement as shown by the second example. Here, the first statement defines an int variable named counter and initializes it by assigning an initial value of 1. The second statement defines a double variable named unit_price and initializes it by assigning an initial value of 14.95.

When you assign literal values to double types, it's a good coding practice to include a decimal point. If, for example, you want to assign the number 29 to the variable, you should code the number as 29.0. This isn't required, but it creates a literal value of the double type, not the int type. In addition, it makes it easy for programmers to see that the variable stores a double type, not an int type.

If you follow the naming recommendations in this figure as you name variables, it makes your programs easier to read and debug. In particular, you should use words formed with lowercase letters and separated by underscores, as in unit_price or max_quantity. This is the standard convention for naming variables when you're using C++.

When you initialize a variable, you can assign a *literal* value like 1 or 14.95 to it as illustrated by the examples in this figure. However, you can also initialize a variable to the value of another variable or to the value of an expression like the arithmetic expressions shown in the next figure.

How to code assignment statements

After you define and initialize a variable, you can assign a new value to it. To do that, you code an *assignment statement* that consists of the variable name, an equals sign, and a new value. The new value can be a literal value, the name of another variable as shown in the last statement in figure 2-4, or the result of an expression as shown in the next figure.

Two built-in data types for working with numbers

Type	Description
int	Integers (whole numbers).
double	Double-precision, floating-point numbers (decimal numbers).

How to define and initialize a variable in two statements

Syntax

```
type variable_name;
variable_name = value;
```

Example

```
int counter;              // definition statement
counter = 1;              // assignment statement
```

How to define and initialize a variable in one statement

Syntax

```
type variable_name = value;
```

Examples

```
int counter = 1;            // define and initialize an int variable
double unit_price = 14.95;  // define and initialize a double variable
```

An example that uses assignment statements

```
int quantity = 0;          // define and initialize an int variable
int max_quantity = 100;    // define and initialize another int variable

// two assignment statements
quantity = 10;             // quantity is now 10
quantity = max_quantity;   // quantity is now 100
```

Description

- A *variable* stores a value that can change, or *vary*, as a program executes.
- Before you can use a variable, you must *define* the type of data it will store (its data type) and its name. Then, you can *initialize* the variable by assigning an initial value to the variable.
- An *assignment statement* assigns a value to a variable. If the data type has already been defined, an assignment statement does not include the data type.
- A value can be a literal value, another variable, or an expression like the arithmetic expressions shown in the next figure.
- It's considered a good practice to initialize a variable before you use it since that allows you to be sure of the initial value that's assigned to the variable.

Naming recommendations for variables

- Each variable name should be a noun or a noun preceded by one or more adjectives.
- Try to use meaningful names that are easy to remember.
- Use lowercase letters and separate the words with underscores (_).

Figure 2-4   How to define and initialize variables

How to code arithmetic expressions

To code simple *arithmetic expressions*, you can use the *arithmetic operators* summarized in figure 2-5. When you use these operators, you code one *operand* on the left of the operator and one operand on the right of the operator.

As the first group of statements shows, the arithmetic operators work the way you would expect them to with one exception. If you divide one integer into another integer, the result doesn't include any decimal places. This is known as *integer division*. When you perform integer division, you can use the *modulus operator* (%), or *remainder operator*, to return the remainder.

In contrast, if you divide a double into a double, the result includes decimal places. This is known as *decimal division*.

When you code assignment statements, you can code the same variable on both sides of the equals sign. Then, you can include the variable on the right side of the equals sign in an arithmetic expression. For example, you can add 1 to the value of a variable named counter with a statement like this:

```
counter = counter + 1;
```

In this case, if counter has a value of 5 before the statement is executed, it has a value of 6 after the statement is executed. This concept is illustrated by the second group of statements.

If you mix integer and double variables in the same arithmetic expression, C++ automatically converts the int value to a double value and uses the double type for the result. However, if you store the result of the expression in an int variable, the decimal places are dropped as shown by the third group of statements. Here, the double value of 251.0 that's stored in the total variable is divided by the int value 2 that's stored in the counter variable. When the result of the division is stored in the double variable named result10, it's stored as 125.5. However, when the result of the division is stored in the int variable named result11, the decimal value is dropped and it's stored as 125.

It's important to restate that when you divide an integer by an integer, the result is always an integer. This is true even if you store the result of the division in a double variable as shown by the last statement in this figure. Here, the integer value in x (14) is divided by the integer value in y (8). This yields a result of 1, and this result is converted to a double value of 1.0 so it can be stored in the double variable.

Although it's not shown in this figure, you can also code expressions that contain two or more operators. When you do that, you need to be sure that the operations are done in the correct sequence. You'll learn more about that in chapter 3.

Five arithmetic operators

Operator	Name	Description
+	Addition	Adds two operands.
–	Subtraction	Subtracts the right operand from the left operand.
*	Multiplication	Multiplies the right operand and the left operand.
/	Division	Divides the right operand into the left operand. If both operands are integers, the result is an integer.
%	Modulus	Returns the value that is left over after dividing the right operand into the left operand.

Statements that use simple arithmetic expressions

```
// integer arithmetic
int x = 14;
int y = 8;
int result1 = x + y;        // result1 = 22
int result2 = x - y;        // result2 = 6
int result3 = x * y;        // result3 = 112
int result4 = x / y;        // result4 = 1
int result5 = x % y;        // result5 = 6

// decimal arithmetic
double a = 8.5;
double b = 3.4;
double result6 = a + b;     // result6 = 11.9
double result7 = a - b;     // result7 = 5.1
double result8 = a * b;     // result8 = 28.9
double result9 = a / b;     // result9 = 2.5
```

Statements that increment a counter variable

```
int counter = 0;
counter = counter + 1;      // counter = 1
counter = counter + 1;      // counter = 2
```

Statements that mix int and double variables

```
double total = 251.0;
double result10 = total / counter;   // result10 = 125.5
int result11 = total / counter;      // result11 = 125
```

A statement that divides an integer by an integer

```
double result12 = x / y;             // result12 = 1.0 (not 1.75)
```

Description

- An *arithmetic expression* consists of one or more *operands* and *arithmetic operators*.

- When expressions mix int and double variables, the compiler converts, or *casts*, the int types to double types.

- When both operands in a division expression are integers, the result is an integer. This is known as *integer division*.

Figure 2-5   How to code arithmetic expressions

How to use the console for input and output

Most programs get input from the user and display output to the user. The easiest way to do this is with the *console*. Before you learn how to use the console, though, you need to learn how to include header files so you can use the C++ standard library to work with the console.

How to include header files

The C++ *standard library* provides code that's stored in *header files*. To use code from the standard library, you include the header file that contains the code you want to use. This allows you to include only the features of the standard library that you need, which keeps the compile time of your program manageable.

Figure 2-6 lists of some of the commonly used header files. You'll learn how to use all of these headers, and more, as you progress through this book. You can also navigate to the URL shown here to see a comprehensive list of the header files available in the standard library.

To include a header file, you use the #include *preprocessor directive*. This instructs the preprocessor you learned about in chapter 1 to include the header file's code in your program. The compiler then treats the code in the header file as if it were typed at the top of your program file.

The example in this figure shows two #include directives. Each of these directives starts with a hash symbol (#) and does *not* end with a semicolon. Typically, you code the #include directives at the top of the file before any other code, as shown in this example.

Here, the first directive includes the iostream header. This header contains code that allows you to send data from one location to another. In particular, it contains code that you can use to send output to the console and read input from the console as shown in the next two figures.

The second directive includes the cmath header. This header contains code that you can use to perform common mathematical operations such as getting a square root or rounding a number. Later in this chapter, you'll learn how to use some of this code.

After coding the #include directives, it's common to code a *using directive* that makes it easy to access the code from any headers that are included, as shown in this figure. This works because all of the code that's available from the header files of the standard library is stored in a *namespace* named std. As a result, the using directive in this figure makes all code from the iostream and cmath headers available to the statements in the main() function. Later in this chapter, you'll learn other techniques for accessing the std namespace. These techniques can improve the performance of your code and help you to avoid the naming conflicts that can occur if an object or function is available from more than one namespace.

Common header files of the C++ standard library

Header file	Description
cstdlib	General purpose utilities such as program control, sort, and search.
iostream	Input/output (IO) stream objects.
iomanip	Input/output (IO) helper functions to manipulate how IO is formatted.
cmath	Common math functions.
string	For working with a string of characters.
vector	For working with a sequence of data elements of the same type.

URL for a complete list of C++ header files

http://en.cppreference.com/w/cpp/header

A typical way to include header files and make them easily accessible

```
#include <iostream>
#include <cmath>

using namespace std;    // make code in both headers easily accessible

int main()
{
    // code that uses the headers goes here
}
```

Description

- The *standard library* for C++ provides a library of code that you can use. This library is organized into *header files*.

- To use the code that's available from the standard library, you code an #include *preprocessor directive* for the header file that contains the code you want to use.

- A *namespace* provides a way to organize code to avoid naming conflicts. All of the code that's available from the header files of the standard library is stored in a namespace named std.

- To make it easy to access the code in the std namespace, it's common to add a using directive for the entire namespace. However, to improve efficiency and avoid naming conflicts, you can use other techniques for accessing the std namespace as shown later in this chapter.

Figure 2-6   How to include header files

How to write output to the console

To write output to the console, you can use the cout (pronounced "see-out") *object* that's available from the iostream header. You'll learn more about objects later in this book. For now, all you need to know is that you can use the cout object that's available from the iostream header. To do that, you don't have to create that object or assign it to a variable. All you need to do is to include the iostream header, and the cout object is available the entire duration of the program.

The cout object represents a *stream*, which is a sequence of characters. More specifically, the cout object represents the *standard output stream* that you can use to write a stream of characters to the console.

To do that, you use the *stream insertion operator* (<<) to add data to the output stream. In figure 2-7, for instance, the first example writes a string of "Hello!" to the console output stream. Later in this chapter, you'll learn more about strings. For now, just know that you can code a string of characters by enclosing them in double quotes.

The second example begins by multiplying 4 by 4 and storing the result in an int variable. Then, it writes a string of "4 times 4 is " to the console. Note that this string contains a space at the end to separate it from the output that follows. Next, it writes the integer that's a result of the multiplication operation to the console. Since the output stream is a series of characters, the output stream automatically converts the integer value to a string of characters.

The console output for this example shows how the integer value looks when it's displayed on the console. In addition, it shows that the string and the integer values are displayed on the same line even though the two insertion operations are coded in separate statements and on separate lines.

In the second example, the second and third lines each use the << operator with the cout object as the left operand (or lvalue) and the string or int to be printed as the right operand (or rvalue). However, these insertion expressions can be *chained* as shown in the third example. This example works the same as the second example, but it only uses two lines of code. That can make this code easier to read and understand.

If you want to start a new line in your output, you can write a *stream manipulator* named endl (pronounced "end-L" or "end line") to cout. The endl manipulator adds a newline character to the stream. In this figure, the fourth example uses two endl manipulators to write three lines to the console.

The fourth example uses three statements to write three lines to the console. However, it's possible to accomplish the same task by chaining the insertion expressions as shown in the fifth example. This example only uses one statement, but it splits that statement across three lines. This works because you can use indentation and spaces to make your code easier to read. Since this example only requires you to code one cout object and one semicolon, some programmers prefer it over the technique that's used in the fourth example.

When you work with console programs, you should know that the appearance of the console may differ slightly depending on the operating system. However, even if the console looks a little different, it should work the same.

Two objects available from the iostream header

Object	Description
cout	An object that represents the output stream to the console.
endl	An object that represents the end of a line. When you insert this object into an output stream, it inserts a newline character into the output stream.

How to send a message to the console

```
cout << "Hello!";
```

The console

```
Hello!
```

How to send text and a number to the console

```
int result = 4 * 4;
cout << "4 times 4 is ";
cout << result;
```

The console

```
4 times 4 is 16
```

Another way to send the same data to the console

```
int result = 4 * 4;
cout << "4 times 4 is " << result;
```

The console

```
4 times 4 is 16
```

How to send multiple lines to the console

```
cout << "This is line 1." << endl;
cout << "This is line 2." << endl;
cout << "This is line 3.";
```

The console

```
This is line 1.
This is line 2.
This is line 3.
```

Another way to send multiple lines to the console

```
cout << "This is line 1." << endl
     << "This is line 2." << endl
     << "This is line 3.";
```

Description

- The *stream insertion operator* (<<) lets you add characters to the output stream.

Figure 2-7   How to use the cout object to write output to the console

How to read input from the console

To read input from the console, you can use the cin object (pronounced "see-in"). The cin object represents the *standard input stream* that you can use to get input from the console. Like the cout object, the cin object is available for the entire duration of the program if you include the iostream header.

You use the *stream extraction operator* (>>) to read data from an input stream. In figure 2-8, for instance, the first example reads an integer from the console. To do that, the first statement defines an int variable named value to store the data that's going to be read from the stream. The second statement prompts the user to enter a number. And the third statement uses the >> operator to extract the integer from the console input stream.

After getting the number from the user, the fourth statement calculates the result of multiplying that number by itself. Then, the fifth statement displays the result of that calculation on the console. To do that, it chains several stream insertion expressions that display the number entered by the user, a string, the number, another string, and the result of the calculation.

The second example shows how to read multiple values from the console. To start, it defines two int variables to store the data read from the stream. Then, it prompts the user to enter two numbers. Next, it extracts the values entered by the user from the input stream and stores them in the int variables. As the console output below the code shows, this works the same whether the user enters one number per line or both numbers on one line.

After getting the two numbers from the user, the second example calculates the result of multiplying those two numbers together. Then, it displays that result. Like the first example, this example creates a chain of insertion expressions.

Like insertion expressions, extraction expressions can also be chained together as shown in the third example. This example consists of a single statement that could be used instead of the two extraction statements in the second example.

Since the insertion and extraction operators are similar in appearance, they can be easy to confuse, especially when you're first getting started. One way to keep them straight is to think of them as indicating the direction that the stream is flowing. For example, you can think of the output stream as flowing out toward the cout object. So, the insertion operator points at cout (cout << val). Conversely, you can think of the input stream as flowing in from the cin object. So, the extraction operator points away from cin (cin >> val).

Another object that's available from the iostream header

Object	Description
cin	An object that represents the input stream from the console.

How to read one value from the console

```
// read input from console
int value;
cout << "Please enter a number: ";
cin >> value;

// make a calculation and display output to the console
int result = value * value;
cout << value << " times " << value << " equals " << result;
```

The console

```
Please enter a number: 5
5 times 5 equals 25
```

How to read multiple values from the console

```
// read input from console
int value1;
int value2;
cout << "Please enter two numbers: ";
cin >> value1;
cin >> value2;

// make a calculation and display output to the console
int result = value1 * value2;
cout << value1 << " times " << value2 << " equals " << result;
```

The console when the user separates the numbers with a space

```
Please enter two numbers: 4 5
4 times 5 equals 20
```

The console when the user presses Enter after the first number

```
Please enter two numbers: 4
5
4 times 5 equals 20
```

How to chain extraction expressions

```
cin >> value1 >> value2;
```

Description

- The *stream extraction operator* (**>>**) gets characters from the input stream.
- When you use the cin object, it doesn't matter if the user enters one value per line or all the values on one line.
- When working with strings and characters, you may need to use some other techniques for reading input that are described later in this chapter.

Figure 2-8   How to use the cin object to read input from the console

The Gallons to Liters program

Figure 2-9 presents a program that reads input from the console and writes output to the console. This program starts with a block comment indicating the author and purpose of the program. Then, it includes the iostream header, so the program can work with the cout and cin objects, as well as the endl stream manipulator. Next, a using directive for the std namespace makes it easy to access the objects in the iostream header.

The main() function starts by writing the name of the program to the console. This line of code ends with two endl manipulators. The first endl manipulator starts a new line after the program title, and the second endl manipulator produces the blank line between the title and the first prompt.
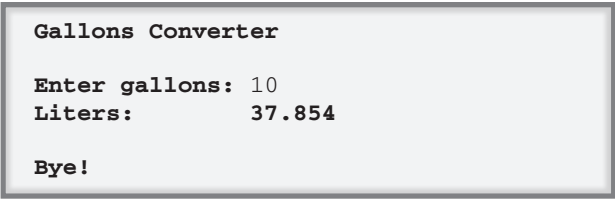
The next group of statements gets the number of gallons from the user. To do that, it starts by defining a double variable named gallons to store the data. Then, it prompts the user to enter the number of gallons, and it uses an extraction expression to read the value the user entered from the console and store it in the gallons variable.

After getting the number of gallons entered by the user, this code converts the gallons to liters by multiplying the number of gallons by 3.7854, and it stores the result of the calculation in a double variable named liters. Then, the next statement displays the number of liters and a message that indicates that the program is ending. To do that, this statement writes the string "Liters: " to the console, followed by the calculated number of liters, two endl manipulators, and the string "Bye!".

As you review this code, note that it uses spaces to format the input and output so it aligns in two columns. Later in this chapter, you'll learn how to use tab characters to align output in columns. Then, in chapter 5, you'll learn how to use stream manipulators to use an even more sophisticated technique for aligning output in columns.

***

If you're new to programming, we recommend that you take a break now and complete exercise 2-1 that's presented at the end of this chapter. This should give you some hands-on practice working with the skills presented so far in this chapter, and it should help you absorb this information. After that, you'll be ready to continue with the rest of the chapter.

The console

```
Gallons Converter

Enter gallons: 10
Liters:         37.854

Bye!
```

The code

```cpp
/*
 * Author:  M. Delamater
 * Purpose: This program uses the console to get gallons
 *          from the user. Then, it converts gallons to liters
 *          and displays the result of the conversion.
 */

#include <iostream>

using namespace std;

int main()
{
    // print name of program
    cout << "Gallons Converter" << endl << endl;

    // get gallons from user
    double gallons;
    cout << "Enter gallons: ";
    cin >> gallons;

    // convert gallons to liters
    double liters = gallons * 3.7854;

    // write output to console
    cout << "Liters:         " << liters << endl << endl
         << "Bye!" << endl << endl;

    // return value that indicates normal program exit
    return 0;
}
```

Figure 2-9   The Gallons to Liters program

How to work with the standard library

So far, you've learned how to do simple arithmetic operations with built-in data types, you've learned how to include headers, and you've learned how to use the cout and cin objects that are available from the iostream header. As you develop more complex programs, though, you'll need to use dozens of different objects and functions that are available from the C++ *standard library*. In addition, you'll need to learn more about how to work with the std namespace.

How to call a function

The C++ standard library provides a large collection of functions that you can use in your programs. As mentioned earlier in this chapter, a *function* is a reusable block of code that performs a specified task. Each function has a name and other code can call the function to execute it.

Figure 2-10 begins by showing five of the many functions that are available from the cmath header. If you include the cmath header as shown earlier in this chapter, these five functions and many other mathematical functions are available to your code.

Once you include the header and the using directive for the std namespace, you can call the functions in the header by using the syntax shown in this figure. To start, you code the function name followed by a set of parentheses. Within the parentheses, you code the *arguments* that are required by the function, separating the arguments with commas. If the function doesn't require any arguments, you leave the parentheses empty.

If the function returns a value, you define a variable to store the return value. You define this variable just like you would any other variable. Then, you assign the result of calling the function to the variable. When you do this, the data type of the variable that stores the return value should be the same data type that's returned by the function. For example, all of the functions in this figure return a double value. As a result, you should assign the values returned by these functions to a variable of the double type.

The first example shows how to use the sqrt() function to calculate the square root of a number. This function requires a single argument. As a result, you can call it by coding the name of the function, a set of parentheses, and a number within those parentheses. Then, you can assign the value that this function returns to a variable. In this example, the first statement assigns the square root of 16 to a double variable named root1, and the second statement assigns the square root of 6.25 to a double variable named root2.

The second example shows how to use the pow() function to raise a number to the specified exponent. This function requires two arguments. The first argument specifies the base number, and the second argument specifies the power of the exponent.

The third, fourth, and fifth examples show how to use the round(), floor(), and ceil() functions to round double values to a specified number of decimal places. These functions all round to the nearest whole number and return a value of the double type. However, a common coding trick for rounding to a specified number of decimal places is to multiply the number by a multiple of ten, round

Five functions available from the cmath header

Function	Description
sqrt(number)	Calculates the square root of the specified number and returns the resulting value as a double.
pow(base, exponent)	Raises the base to the power of the exponent and returns the resulting value as a double.
round(number)	Rounds the number up or down to the nearest whole number and returns the resulting value as a double.
floor(number)	Rounds the number down (towards the floor) to the nearest whole number and returns the resulting value as a double.
ceil(number)	Rounds the number up (towards the ceiling) to the nearest whole number and returns the resulting value as a double.

How to call a function

Syntax

```
function_name(arguments);
```

The sqrt() function

```
double root1 = sqrt(16);   // root1 is 4
double root2 = sqrt(6.25); // root2 is 2.5
```

The pow() function

```
double pow1 = pow(2, 4);   // pow1 is 16
double pow2 = pow(2.5, 2); // pow2 is 6.25
```

The round() function

```
double x = 10.315;
double round1 = round(x);            // round1 is 10.0
double round2 = round(x * 10) / 10;  // round2 is 10.3
double round3 = round(x * 100) / 100; // round3 is 10.32
```

The floor() function

```
double floor1 = floor(x);            // floor1 is 10.0
double floor2 = floor(x * 10) / 10;  // floor2 is 10.3
double floor3 = floor(x * 100) / 100; // floor3 is 10.31
```

The ceil() function

```
double ceil1 = ceil(x);            // ceil1  is 11.0
double ceil2 = ceil(x * 10) / 10;     // ceil2 is 10.4
double ceil3 = ceil(x * 100) / 100;   // ceil3 is 10.32
```

Description

- When you call a *function*, you code its name followed by a pair of parentheses. If the function requires *arguments*, you code the arguments within the parentheses. These arguments must be of the correct data type, and they must be coded in the correct sequence separated by commas.

- A function can return a value to the code that calls it. If a function returns a value, you can store the return value in a variable.

Figure 2-10   How to call a function

the number, and then divide by the same multiple of ten. For example, to get two decimal places, you can multiply the number by 100, round the number to the nearest whole number, and then divide the number by 100. If you review these three examples, you should get a feel for how this works.

As you progress through this book, you'll learn how to use dozens of functions and objects. You'll also learn how to create your own.

How to work with the std namespace

A *namespace* provides a way to organize objects and functions. This helps to avoid naming conflicts. As mentioned earlier, the standard library stores its code in a namespace named std. So far in this chapter, you have been coding a using directive to automatically make all objects and functions in the std namespace available to your code. This is shown by the first example in figure 2-11. The advantage of this approach is that it's an easy way to make all objects and functions in a namespace available to your code.

However, it's considered a better practice to specify individual members with *using declarations* as shown in the second example. This example uses three declarations to specify the objects and functions that are used in the main() function. To do that, these declarations use the *scope resolution operator* (`::`) to create a fully qualified name. A *fully qualified name* consists of the namespace, the scope resolution operator, and the member of the namespace. For example, the fully qualified name for the cout object is

```
std::cout
```

and the fully qualified name for the sqrt() function is

```
std::sqrt
```

If you don't code a using directive or declaration for an object or function, you can still use it in your code as shown in the third example. In that case, however, you must use a fully qualified name to access the object or function.

There are several advantages to the approaches shown in the second and third examples. First, they should help your code compile more quickly. Second, they should reduce the chance of encountering a naming conflict. Third, they clearly show the objects and functions that your code needs, which makes it easier to read and maintain your code.

Many programmers prefer to use using declarations, since they make it easy to access the specified objects and functions. This is especially true if your code needs to use a namespace member repeatedly. Then, you don't have to qualify the member each time you use it, which saves you typing and can make your code easier to read and maintain. If you're only going to use a member once or twice, though, you might want to use a fully qualified name.

To save space, most of the examples shown in this book use a using directive for the std namespace. However, you should know that using declarations are generally considered a better practice.

A using directive for the std namespace

```
#include <iostream>
#include <cmath>

using namespace std;    // use all elements of the std namespace

int main() {
    int num = 7;
    double root = sqrt(num);
    root = round(root * 1000) / 1000;    // round to 3 decimal places
    cout << "The square root of " << num << " is " << root << endl;
}
```

Using declarations for three members of the std namespace

```
#include <iostream>
#include <cmath>

using std::cout;        // use the cout object
using std::sqrt;        // use the sqrt function
using std::round;       // use the round function

int main()
{
    int num = 7;
    double root = sqrt(num);
    root = round(root * 1000) / 1000;    // round to 3 decimal places
    cout << "The square root of " << num << " is " << root << endl;
}
```

Fully qualified names

```
#include <iostream>
#include <cmath>

int main()
{
    int num = 7;
    double root = std::sqrt(num);
    root = std::round(root * 1000) / 1000;    // round to 3 decimal places
    std::cout << "The square root of " << num << " is " << root << std::endl;
}
```

The console for all three examples

```
The square root of 7 is 2.646
```

Description

- The members of the standard library are grouped in a *namespace* named std.
- You can identify members of namespaces by using the *scope resolution operator* (::).
- You can code a *using directive* for a namespace to make it easy to access all members of that namespace. However, this can lead to naming conflicts.
- You can code a *using declaration* for a specific member of a namespace to make it easy to access that member of the namespace. This significantly reduces the chance of a naming conflict.

Figure 2-11   How to work with the std namespace

The Circle Calculator program

Figure 2-12 presents an enhanced version of the Circle Calculator program that you saw in chapter 1. This program prompts the user to enter the radius for a circle. Then, it uses two functions from the cmath header to calculate the diameter, circumference, and area of the circle with that radius.

This program starts by including the iostream and cmath headers. Then, the first three using declarations make it easy for the program to work with the cout and cin objects, as well as the endl stream manipulator. After that, the last two using declarations make it easy to work with the pow() and round() functions of the cmath header.

The main() function starts by writing the name of the program to the console. This line of code ends with two endl manipulators. The first endl manipulator starts a new line after the program title, and the second endl manipulator produces the blank line between the title and the first prompt.

The next group of statements gets the radius from the user. To do that, it defines a double variable named radius to store the value, it prompts the user to enter the radius, and it uses an extraction expression to read the value the user entered from the console and store it in the radius variable.

After getting the radius entered by the user, this code uses four statements to calculate the diameter, circumference, and area of the circle. Here, the first statement defines a double variable named pi and assigns it an initial value of 3.14159. The second statement calculates the diameter by multiplying the radius by 2. The third statement calculates the circumference by multiplying the diameter by pi. And the fourth statement calculates the area by using the standard geometric formula of pi times radius squared (πr^2).

As you review this code, note that both the third and fourth statements use pi. As a result, assigning the value of pi to a variable named pi makes your code easier for others to read and follow. In addition, if you wanted to use a more or less precise approximation of pi, you could just modify the value that's assigned by the first statement.

After making the calculations, this code uses the round() function to round the circumference and area to 1 decimal place. Then, it displays the results of the calculations and a message that indicates that the program is ending. Note that after it displays the area, this code uses two endl manipulators to print a blank line between the result of the calculations and the end message.

The console

```
Circle Calculator

Enter radius:   9
Diameter:       18
Circumference: 56.5
Area:           254.5

Bye!
```

The code

```cpp
#include <iostream>
#include <cmath>

using std::cout;
using std::cin;
using std::endl;
using std::pow;
using std::round;

int main()
{
    // print name of program
    cout << "Circle Calculator" << endl << endl;

    // get radius from user
    double radius;
    cout << "Enter radius:  ";
    cin >> radius;

    // make calculations
    double pi = 3.14159;
    double diameter = 2 * radius;
    double circumference = diameter * pi;
    double area = pi * pow(radius, 2.0);

    // round to 1 decimal place
    circumference = round(circumference * 10) / 10;
    area = round(area * 10) / 10;

    // write output to console
    cout << "Diameter:      " << diameter << endl
         << "Circumference: " << circumference << endl
         << "Area:          " << area << endl << endl
         << "Bye! ";

    // return value that indicates normal program exit
    return 0;
}
```

Figure 2-12   The Circle Calculator program

How to generate random numbers

When learning how to program, it's often helpful to be able to generate random numbers. They're useful if you want to develop games that involve dice, cards, or other elements of random chance. They're also useful for simulations such as testing a function with a range of random numbers.

To generate random numbers with C++, you can use the functions of the cstdlib and ctime headers presented in figure 2-13. First, you use the time() function to get a seed value for the srand() function. To do that, you can use the nullptr keyword to pass a null pointer argument to the time() function. This returns the number of seconds that have elapsed since Jan 1, 1970. As a result, the time() function returns a different number each time it's called.

Once you have a seed value, you use the srand() function to specify the seed value for the rand() function. Next, you use the rand() function to get a random integer between 0 and RAND_MAX, which is a large integer that varies from compiler to compiler.

After you get the random integer, you can use the modulus operator to specify the upper limit of the random number. In this figure, for example, the code uses the modulus operator to return the remainder after dividing by 6. That way, the result of the operation is from 0 to 5 inclusive. Then, you can add 1 to the result to get the value of a six-sided die (1 to 6 inclusive).

So, why do you need to provide a seed value for the rand() function? Because the rand() function actually generates *pseudorandom numbers*, which are series of numbers that appear to be random but are actually the same for each seed value. As a result, if you don't change the seed value each time you run the program, the program will use the same series of numbers each time, which isn't typically what you want.

The functions presented in this figure provide a simple way to generate a series of pseudorandom numbers. This is adequate for creating programs like the one shown here. However, random numbers that are generated using this technique are not cryptographically secure. As a result, if you need to generate cryptographically secure numbers, you'll have to use a different technique. For example, if you're using C++ 11 or later, you can use the random header to generate random numbers that are cryptographically secure. By the time you finish this book, you should have the skills you need to learn to use the random header on your own.

Two functions of the cstdlib header for generating random numbers

Function	Description
rand()	Returns a pseudorandom integer between 0 and RAND_MAX. This function uses a seed to generate the series of integers, and the seed should be initialized to a distinctive value using the srand() function.
srand(seed**)**	Specifies the seed value for the rand() function.

One function of the ctime header

Function	Description
time(timer**)**	If the timer argument is a null pointer, it returns the current calendar time, generally as the number of seconds since Jan 1, 1970 00:00 UTC.

Code that simulates the roll of a pair of dice

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

using namespace std;

int main()
{
    // use time() to get an int value
    int elapsed_seconds = time(nullptr);

    // seed the random number generator
    srand(elapsed_seconds);

    // roll the first die
    int die1 = rand() % 6;   // die1 is >= 0 and < 6
    die1 = die1 + 1;         // die2 is >= 1 and <= 6

    // roll the second die
    int die2 = rand() % 6;   // die2 is >= 0 and < 6
    die2 = die2 + 1;         // die2 is >= 1 and <= 6

    // write output to console
    cout << "Your roll:  " << die1 << " " << die2;
}
```

The console

```
Your roll:  6 3
```

Description

- The rand() and srand() functions provide a simple way to generate a series of *pseudorandom numbers*, which is a series of numbers that appears to be random but is actually the same if you specify the same seed value for the series.

- If you're using C++ 11 or later, you can use the random header to generate random numbers that are cryptographically secure. However, this requires advanced programming techniques that aren't presented until later in this book.

Figure 2-13   How to generate random numbers

How to work with char and string variables

As you've already seen, programs often need to work with character data. In the next few topics, you'll learn about char and string variables, you'll learn how to work with special characters, and you'll learn about some functions for reading char and string data from the console.

How to assign values to char and string variables

A *char* is a built-in data type defined by the C++ core language. A char contains a single character, which can be any letter, number, punctuation mark, or special character in the *ASCII character set*. Technically, the char type is an integer type since it actually contains an integer that maps to one of the ASCII characters.

The first example in figure 2-14 shows how to define and initialize a char variable. This is similar to defining a numeric variable. However, the *character literal* that's assigned to a char variable is enclosed in single quotation marks.

A *string*, conversely, is not a built-in data type. Rather, it's an *object* that consists of a sequence of zero or more characters. This object type is defined by the string *class* that's available from the string header file and that's a member of the std namespace. When you include the iostream header, most compilers also include the string header. However, to make sure your code is portable across all compilers, it's a good practice to include the string header file as shown in the second example.

The third example shows how to define and initialize a string variable. Here, the first statement creates a *string literal* of "Invalid data entry" by coding multiple characters within double quotation marks, and it assigns that string literal to a string object named message1. A string variable can also contain an *empty string*. To assign an empty string to a string variable, you code a set of double quotation marks with nothing between them as shown in the second statement in this example.

If you want to *join*, or *concatenate*, two or more strings into one, you can use the **+** operator. The fourth example shows how this works. Here, two string variables are initialized with first name and last name values. Then, these two values are concatenated with a string literal that contains a comma and a space. The resulting string is assigned to a third string variable called name. When concatenating strings, you can use string variables or string literals.

You can also join a string with a char. This is illustrated in the last statement in the fourth example. Here, the first_name and last_name variables are concatenated with two character literals and a char variable that contains a middle initial. As before, the resulting string is assigned to the name variable.

You can use the **+** and **+=** operators to *append* a string or character to the end of a string as shown by the fifth example. Here, the second statement uses the **+=** operator to append the last name to the first name followed by a space. If you use

The built-in data type for characters

Type	Description
char	A built-in data type that stores an integer value that maps to a single character in the ASCII character set. This character set includes letters, numbers, punctuation marks, and special characters like *, &, and #.

A class that's available from the string header

Class	Description
string	A class that defines a type of object that consists of a sequence of zero or more characters.

How to define and initialize a char variable

```
char middle_initial = 'M';
```

How to include the string header file

```
#include <string>
```

How to define and initialize a string variable

```
string message1 = "Invalid data entry";
string message2 = "";                               // empty string
```

How to concatenate strings and chars

```
string first_name = "Bob";                      // first_name is Bob
string last_name = "Smith";                     // last_name is Smith

string name = last_name + ", " + first_name;  // name is Smith, Bob
name = first_name + ' ' + middle_initial + ' ' + last_name;
                                                // name is Bob M Smith
```

How to append one string to another with the += operator

```
name = first_name + ' ';            // name is Bob followed by a space
name += last_name;                  // name is Bob Smith
```

Description

- To specify a character value, you can enclose the character in single quotation marks. This is known as a *character literal*.

- A class defines a type of *object*. When you create an object from a class, you can assign it to a variable of the class type.

- To create a string object, you enclose text in double quotation marks. This is known as a *string literal*.

- To assign an *empty string* to a string variable, you can code a set of quotation marks with nothing between them. This means that the string doesn't contain any characters.

- To *concatenate* a string with another string or a char, use a plus sign.

- The **+=** operator is a shortcut for appending a string or char to a string variable.

- If you only need to store a single character, the char type works more efficiently than the string type.

Figure 2-14   How to assign values to char and string variables

the **+** operator, you need to include the string variable on both sides of the **=** operator. Otherwise, the assignment statement replaces the old value with the new value.

When working with char and string variables, a string uses slightly more overhead than a char. As a result, if you're working with a single character, you should use a char variable or literal, not a string variable or literal.

How to work with special characters

Figure 2-15 shows how to work with special characters. To do that, you can use the *escape sequences* shown in this figure.

Each escape sequence starts with a backslash. If you code a backslash followed by the letter n, for example, the compiler includes a newline character as shown in the first string example. If you omit the backslash, the compiler just includes the letter n in the string value. The escape sequence for the tab character works similarly as shown in the second string example.

To code a string literal, you enclose it in double quotes. As a result, if you want to include a double quote within a string literal, you must use an escape sequence as shown in the third string example. Here, the \" escape sequence is used to include two double quotes within the string literal.

Finally, you need to use an escape sequence if you want to include a backslash in a string literal. To do that, you code two backslashes as shown in the fourth string example. If you forget to do that and code a single backslash, the compiler uses the backslash and the next character to create an escape sequence. That can yield unexpected results.

When you work with escape sequences, remember that they define special characters, not special strings. As a result, there are times when you may want to assign a special character to a variable of the char data type as shown in the char examples. To do that, you can create a char literal by enclosing the escape sequence in single quotes. This defines a single character that can be assigned to a char variable. Here, the first statement assigns a newline character to a char variable named newline. The second statement assigns a tab character to a char variable named tab. And the third statement assigns a backslash character to a char variable named sep, which is short for separator.

Common escape sequences for special characters

Sequence	Character
\n	Newline
\t	Tab
\r	Return
\"	Quotation mark
\\	Backslash

String examples

A string that uses a newline character

```
"Code: CPP\nPrice: $49.50"
```

Result

```
Code: CPP
Price: $49.50
```

A string that uses tabs

```
"Joe\tSmith\nKate\tLewis"
```

Result

```
Joe       Smith
Kate      Lewis
```

A string that uses double quotes

```
"Type \"x\" to exit"
```

Result

```
Type "x" to exit
```

A string that uses backslashes

```
"C:\\murach\\cpp\\files"
```

Result

```
C:\murach\cpp\files
```

Char examples

```
char newline = '\n';
char tab = '\t';
char sep = '\\';          // use backslash as a separator
```

Description

- You can use *escape sequences* to identify certain types of special characters.
- Within a string, you can code the escape sequence anywhere you want to use the special character.
- A special character is a single character of the char type. As a result, to code a char literal, you code single quotes around the two characters of the escape sequence.

Figure 2-15   How to work with special characters

How to read strings and chars from the console

When you use the extraction operator (`>>`) to read data from the cin object, it uses *whitespace* (spaces, tabs, or new lines) to separate the data in the input stream into one or more *tokens*. Then, it reads one token per extraction expression. So, if a user enters "Grace M. Hopper",

```
cin >> name;
```

only stores "Grace" in the name variable.

You could fix this by declaring variables for the first name, middle initial, and last name and then chaining the extraction expressions like this:

```
cin >> first_name >> middle_initial >> last_name;
```

However, what if you want to give your users the flexibility to enter only a first name or a first and last name? One way to do that is to use the getline() function that's presented in figure 2-16 to read the entire line of user input.

Unlike an extraction expression, the getline() function only uses line breaks to separate the data in the stream. As a result, it reads all data up to the end of the current line of input. This function takes two arguments. The first is the cin object, and the second is the string variable where the extracted data is stored.

The first example in this figure shows how this works. Here, the first statement defines the name variable, the second statement prompts the user to enter a full name, and the third statement passes the cin object and the name variable to the getline() function. As a result, this code stores whatever the user enters in the name variable, whether it's "Grace" or "Grace Hopper" or "Grace M. Hopper".

When reading strings and chars from the console, you sometimes need to ignore characters. To do that, you can use a special kind of function known as a *member function*, which is a function that's available from an object. For example, this figure summarizes the get() function and two versions of the ignore() function. All of these methods are members of the cin object. As a result, to call them, you start by coding the object name and the *dot operator* (`.`). Then, you code the function name and its arguments just as you would for any other function.

The examples below the syntax summary show how this works. Here, the first statement calls the get() member function of the cin object. This gets the next character in the stream. The second statement calls the ignore() member function of the cin object. Since this statement doesn't pass any arguments to the function, it just extracts and discards the next char in the input stream. However, the third statement calls the ignore() function and passes two arguments to it. These arguments indicate that the function should discard the next 100 characters or all characters until the next space character, whichever comes first.

The last example shows how to pause program execution until the user presses the Enter key. This is useful if the console for a program closes before the program is done. Here, the first statement discards all data that the user may have entered earlier, up to 1000 characters. This statement isn't always necessary, but it makes sure that this code will work even if there are some extra characters remaining in the input stream. Then, the second statement prompts the user to press Enter. Finally, the third statement extracts and ignores the newline character that's inserted into the input stream when the user presses Enter.

The getline() function of the iostream header

Function	Description
getline(cin, var**)**	Extracts an entire line of console input, including spaces, and assigns it to the specified variable.

How to use the getline() function to read a full name

```
string name;
cout << "Enter full name: ";
getline(cin, name);
cout << "Your name is " << name;
```

The console

```
Enter full name: Grace M. Hopper
Your name is Grace M. Hopper
```

Member functions of the cin object

Member function	Description
get()	Gets the next character in the input stream.
ignore()	Extracts and discards the next character in the input stream.
ignore(n, delim**)**	Extracts and discards characters in the input stream until either the number of characters discarded is n, or the character in delim is found.

How to call a member function of an object

Syntax

```
object_name.function_name(arguments);
```

Examples

```
char initial = cin.get(); // extract and return the next char
cin.ignore();             // extract and discard the next char
cin.ignore(100, ' ');     // extract and discard the next 100 chars
                          // or all characters up to the next space
```

Code that pauses until the user presses Enter to continue

```
cin.ignore(1000, '\n');   // discard any extra characters
                          // on the current line
cout << "\nPress [Enter] to close the terminal ...\n";
cin.ignore();
```

The console

```
Press [Enter] to close the terminal ...
```

Description

- Many objects in the standard library have *member functions*. To call a member function, code the name of the object, the dot operator (**.**), and the name of the member function.

- You pass arguments to and get return values from a member function just like you do a regular function.

Figure 2-16   How to read strings and chars from the console

How to fix a common problem with reading strings

In most cases, the getline() function works the way you want it to. Sometimes, though, you can run into a problem if you use getline() after code that uses an extraction operator to extract data from an input stream. Figure 2-17 begins by showing this problem. Here, the first example starts by using an extraction expression to read an account number. Then, it uses the getline() function to read a full name. However, the console doesn't give the user a chance to enter a full name.

What's causing this problem? Well, when a user types a value and presses the Enter key, C++ adds the value and the newline character to the input stream. But, the extraction expression only extracts the value from the stream, not the newline character. That's fine if you're using the extraction operator because this operator ignores a leading newline character. Unfortunately, the getline() function doesn't ignore this character. As a result, getline() reads the leading newline character into the name variable, and program execution continues to the statement that writes the output to the console.

To fix this problem, you can use the ignore() function of the cin object to extract and discard the newline character that's causing the problem as shown in the second example. Once you extract and discard this newline character, the getline() function works correctly.

Some programmers consider it a best practice to call the ignore() function after every extraction expression. That way, the leading newline character is always discarded. But, it's also common to only call the ignore() function when necessary.

A common problem with reading strings

The code

```
int account_num;
cout << "Enter account number: ";
cin >> account_num;         // extracts data but leaves the newline character

string name;
cout << "Enter full name: ";
getline(cin, name);         // reads the newline character left in the stream

cout << "Name: " << name << " | Account: " << account_num;
```

The console

```
Enter account number: 1234
Enter full name:
Name:  | Account: 1234
```

The data in the cin object after the user enters "1234"

1	2	3	4	\n

How to fix the problem

The code

```
int account_num;
cout << "Enter account number: ";
cin >> account_num;

string name;
cout << "Enter full name: ";
cin.ignore();            // discards the newline character left in the stream
getline(cin, name);      // reads the next line

cout << "Name: " << name << " | Account: " << account_num;
```

The console

```
Enter account number: 1234
Enter full name: Mary Delamater
Name: Mary Delamater | Account: 1234
```

Description

- When the user presses the Enter key at the console, C++ inserts a newline character into the input stream.

- When you use the extraction operator to extract data, it skips over any leading whitespace characters (such as spaces and newline characters) and extracts the data up to the next whitespace character, leaving the whitespace character in the stream.

- When you use the getline() function to read data, it doesn't skip over a leading newline character. Instead, it returns an empty string. To skip over a leading newline character, you can call the ignore() member function of the cin object to ignore the leading newline character.

Figure 2-17   How to fix a common problem with reading strings

The Guest Book program

Figure 2-18 presents another program that reads input from the console and writes output to the console. This program accepts string input from the user and displays formatted output.

Like the Gallon Converter program, the Guest Book program starts by including the iostream header. It also includes the string header. In addition, it specifies a using directive for the std namespace so the code doesn't need to use fully qualified names for the objects in the iostream and string headers.

The main() function starts by writing the title of the program to the console. Note that this statement uses the newline character rather than the endl manipulator to add line breaks. Here, the first newline character (\n) starts a new line and the second adds a blank line after the program title.

After the title, this code prompts the user to enter a first name. Then, it uses an extraction expression to get a first name from the user. Next, it calls the ignore() function of the cin object to discard the leading newline character left in the stream by the previous extraction expression as well as any other characters left in the stream, up to 100 characters.

After the first name, the code prompts the user to enter a middle initial. Then, it uses the get() member function to get the middle initial. Next, it calls the ignore() function to discard any characters left in the stream, including the leading newline character.

After the middle initial, the code prompts the user for a last name. Here, the code uses the getline() function to get the last name, so the user can enter a last name that contains spaces. This code continues by using the getline() function to get the user's city and country.

After getting all of the input from the user, the code formats the data and writes it to the console. To do that, it uses the newline character (\n) to start new lines. In addition, it uses the **+** operator to create a full name by concatenating the first name, a space character, the middle initial, a period, a space character, and the last name. It also uses the **+** operator to create the line that contains the city and country.

As you review this code, note that each prompt uses a tab character (\t) to align the user's entry. In addition, this code encloses special characters such as the tab character in double quotes if the special character is part of a string. Otherwise, this code encloses the special character in single quotes so it's treated as a char type. That's true for other single characters as well, such as a space. As you learned earlier in this chapter, that makes your code slightly more efficient.

Also, note that this program only extracts the first name that's entered before any whitespace. As a result, if a user enters "Emmy Lou" as the first name, the program only stores "Emmy" as the first name. Similarly, this program only extracts the first character that's entered for the middle initial. As a result, if a user enters "Wiliford" for the middle initial, the program only stores "W". If that's not what you want, you can use the getline() function instead of the extraction operator to get the first name and middle initial. And if you do that, you don't need to use the ignore() function.

The console

```
Guest Book

First name:     Dave
Middle initial: Williford
Last name:      Von Ronk
City:           New York
Country:        United States

ENTRY
Dave W. Von Ronk
New York, United States
```

The code

```cpp
#include <iostream>
#include <string>

using namespace std;

int main()
{
    cout << "Guest Book\n\n";

    string first_name;
    cout << "First name:\t";
    cin >> first_name;              // get first string only
    cin.ignore(100, '\n');          // discard leftover chars and newline

    char middle_initial;
    cout << "Middle initial:\t";
    middle_initial = cin.get();     // get first char only
    cin.ignore(100, '\n');          // discard leftover chars and newline

    string last_name;
    cout << "Last name:\t";
    getline(cin, last_name);        // get entire line

    string city;
    cout << "City:\t\t";
    getline(cin, city);             // get entire line

    string country;
    cout << "Country:\t";
    getline(cin, country);          // get entire line

    cout << "\nENTRY\n"             // display the entry
        << first_name + ' ' + middle_initial + ". " + last_name + '\n'
        << city + ", " + country + "\n\n";
}
```

Figure 2-18   The Guest Book program

How to test and debug a program

In chapter 1, you were introduced to *syntax errors* that are detected by an IDE when you enter code. Because syntax errors prevent a program from compiling, they are also commonly referred to as *compile-time errors*. For example, if you try to compile the code shown in figure 2-19, the compiler returns an error like the one shown in this figure.

This error information is typically displayed by the IDE, and it can help you determine the location of the error. In this figure, for example, the message indicates that a semicolon was expected before the string keyword on line 10. That's because a semicolon was missing from the end of line 8. Once you've fixed the compile-time errors, you're ready to test and debug the program.

How to test a program

When you test a program, you run it to make sure it works correctly. As you test, you should try every possible combination of valid and invalid data to be certain that the program works correctly under every set of conditions. Remember that the goal of testing is to find errors, or *bugs*, so they're not encountered by users when they run the program.

As you test, you will inevitably encounter two types of bugs. The first type of bug causes a *runtime error* (also known as a *runtime exception*). A runtime error can cause a program to crash or to behave erratically.

The second type of bug produces inaccurate results when a program runs. These bugs occur due to *logic errors* in the source code. For example, these types of bugs can cause a program to enter an infinite loop or to display incorrect data. For instance, the console in this figure shows output for the Circle Calculator program that displays incorrect data. Here, one of the calculations is not correct. This type of bug can be more difficult to find and correct than a runtime error.

How to debug a program

When you *debug* a program, you find the cause of the bugs, fix them, and test again. As your programs become more complex, debugging can be one of the most time-consuming aspects of programming. That's why it's important to write your code in a way that makes it easy to read, understand, and debug.

To find the cause of runtime errors, you can start by finding the source statement that was running when the program crashed. To do that, you can start by studying any error messages that you get.

To find the cause of incorrect output, you can start by figuring out why the program produced the output that it did. For instance, you can start by asking why the Circle Calculator program in this figure didn't perform all of the calculations correctly. Once you figure that out, you're well on your way to fixing the bug.

The beginning of a file that contains an error

```
1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6  int main()
7  {
8      cout << "Guest Book\n\n"
9
10     string first_name;
11     cout << "First name:\t";
...
...
```

The error that's returned by the compiler

```
main.cpp: In function 'int main()':
main.cpp:10:5: error: expected ';' before 'string'
    string first_name;
    ^
```

The Circle Calculator program with incorrect output

```
Circle Calculator

Enter radius:  9
Diameter:      18
Circumference: 28.3
Area:          254.5
```

Description

- A *syntax* or *compile-time error* occurs when a statement can't be compiled. Before you can test a program, you must fix the syntax errors.

- To *test* a program, you run it to make sure that it works properly no matter what combinations of valid and invalid data you enter. The goal of testing is to find the errors (or *bugs*) in the program.

- To *debug* a program, you find the causes of the bugs and fix them.

- One type of bug leads to a *runtime error* (also known as a *runtime exception*) that causes the program to end prematurely. This type of bug must be fixed before testing can continue.

- Even if a program runs to completion, the results may be incorrect due to *logic errors*. These bugs must also be fixed.

Debugging tips

- For a runtime error, go to the line in the source code that was running when the program crashed. In most IDEs, you can do that by clicking on the link to the line of source code. That should give you a strong indication of what caused the error.

- For a logical error, first figure out how the source code produced that output. Then, fix the code and test the program again.

Figure 2-19   How to test and debug a program

Perspective

The goal of this chapter has been to get you off to a fast start with C++ programming. Now, if you understand how the programs presented in this chapter work, you've come a long way. You should also be able to write comparable programs of your own.

Keep in mind, though, that this chapter is just an introduction to C++ programming. The next few chapters will build upon what you learned in this chapter to expand your knowledge of C++ programming. This will include learning some new skills as well as learning more about data types, arithmetic expressions, input and output streams, strings, and debugging.

Terms

statement	stream insertion operator
block of code	chained expressions
comment	stream manipulator
single-line comment	standard input stream
end-of-line comment	stream extraction operator
block comment	function
main() function	argument
parameter	using directive
body of a function	using declaration
return statement	scope resolution operator
identifier	fully qualified name
snake case	pseudorandom number
camel case	char
keyword	ASCII character set
data type	character literal
variable	string
define a variable	class
initialize a variable	object
literal value	string literal
assignment statement	empty string
arithmetic expression	concatenate strings
arithmetic operator	append to a string
operand	escape sequence
cast a data type	whitespace
integer division	token
modulus operator	member function
remainder operator	dot operator
decimal division	syntax error
console	compile-time error
preprocessor directive	test a program
standard library	bug
header file	runtime error
namespace	logic error
stream	debug a program
standard output stream	

Summary

- *Statements* direct the operations of a program, and *comments* typically document what the statements do.
- The *main( ) function* is a special type of function that's called by the operating system when it starts your program.
- *Variables* are used to store data that changes, or varies, as a program runs.
- When you *define* a variable, you must code its data type and its name. Two of the most common data types for numeric variables are the int and double types.
- After you define a variable, you can *initialize* it by assigning an initial value to it.
- You can use *assignment statements* to assign new values to variables.
- You can use *arithmetic operators* to code *arithmetic expressions* that perform operations on one or more *operands*.
- The *standard library* uses *header files* to organize and store its *classes*, *objects*, and *functions*.
- The standard library groups its members in the std *namespace*. You can use a *using directive*, *using declarations*, or *fully qualified* names to work with the members of a namespace.
- You *call* a function by typing the function name and a set of parentheses. A function may require one or more *arguments*, and it may *return* a value.
- You can use objects from the iostream header file to work with the *standard input and output streams* of the *console*. The cout object writes output to the console, and the cin object reads input from the console.
- A *char* is a built-in data type that contains a single character.
- A *string* is an object that contains zero or more characters. A string object is created from the string class that's available from the string header file.
- You can use the **+** operator to *concatenate* one string with another string or a char, and you can use the **+** or **+=** operator to *append* one string to another.
- To include special characters in strings, you can use *escape sequences*.
- *Testing* is the process of finding the errors or *bugs* in a program. *Debugging* is the process of locating and fixing the bugs.

Exercise 2-1   Create a Rectangle Calculator program

In this exercise, you'll code a program that accepts the height and width of a rectangle and then displays the area of the rectangle. When you're done, a test run should look something like this:

```
Rectangle Calculator

Enter height and width: 4.25 5.75
Area:                   24.4375

Bye!
```

Open the project, review the code, and add a comment

1. Use your IDE to open the Rectangle Calculator project or solution in this folder:

    `ex_starts\ch02_ex1_rectangle_calculator`

2. Review the code in the main.cpp file to see that it contains an #include directive for the iostream header file, a using directive for the std namespace, and a main() function with a return statement.

3. Add a block comment at the beginning of the file that includes your name, the current date, and the purpose of the program.

Code the main() function

4. Add a statement at the beginning of the main() function that writes the name of the program to the console. The name should be followed by two newline characters.

5. Define two variables with the double data type to store the height and width the user enters. Then, add a statement that prompts the user for a height and width.

6. Add a statement that uses chained extraction expressions to get the height and width values that the user enters at the console.

7. Define a variable with the double data type to store the area. Then, calculate the area of the rectangle (height times width), and assign it to this variable.

8. Add a statement that displays the area on the console, followed by two newline characters, the "Bye!" message, and two more newline characters.

Run the program

9. Run the program to be sure it works correctly. If any errors are detected, fix them and then run the program again.

Exercise 2-2   Enhance the Gallons Converter program

In this exercise, you'll enhance the Gallons Converter program presented in this chapter. When you're done, a test run should look something like this:

```
Gallons Converter

Enter gallons: 2.335
Liters:        8.84
Quarts:        9.34
Ounces:        298.88


Bye!
```

Open the project, review the code, and run the program

1. Use your IDE to open the Gallons Converter project or solution in this folder:
    `ex_starts\ch02_ex2_gallons_converter`

2. Review the code in the main.cpp file. Then, run the program to see how it works.

Add code to round the number of liters

3. Add an include directive for the cmath header file.

4. Add a statement that rounds the number of liters to two decimal places.

5. Run the program to be sure this works correctly. If any errors occur, debug the program, fix the errors, and then run the program again.

Add code to calculate the quarts and ounces

6. Define a variable with the double data type to store the number of quarts.

7. Calculate the number of quarts, and assign it to this variable. (One gallon is equal to four quarts.) The number of quarts should be rounded to two decimal places.

8. Modify the statement that writes the number of liters to the console so it also writes the number of quarts.

9. Run the program, and fix any errors that occur.

10. Repeat steps 6 through 9 to calculate and display the number of ounces in a gallon. (One gallon is equal to 128 ounces.)

Replace the using directive with using declarations

11. Replace the using directive for the std namespace with the using declarations needed by this program.

12. Run the program one more time to be sure it still works.

Exercise 2-3   Enhance the Guest Book program

In this exercise, you'll enhance the Guest Book program presented in this chapter so it accepts additional information. When you're done, a test run should look something like this:

```
Guest Book

First name:      Dave
Middle initial:  Williford
Last name:       Von Ronk
Address:         123 Wall St.
City:            New York
State:           NY
Postal code:     10001
Country:         United States

ENTRY
Dave W. Von Ronk
123 Wall St.
New York, NY 10001
United States
```

Open the project, review the code, and run the program

1.   Use your IDE to open the Guest Book project or solution in this folder:
     `ex_starts\ch02_ex3_guest_book`

2.   Review the code in the main.cpp file. Then, run the program to see how it works.

Add code to accept and display the address, state code, and zip code

3.   Define three string variables to store an address, a state code, and a zip code.

4.   Add statements that prompt the user for the address, state code, and zip code.

5.   Add statements that get the data entered by the user for the address, state code, and zip code and assign the values to the appropriate variables. The entire line of data should be retrieved for the address and zip code, but only the first two characters should be retrieved for the state code.

6.   Modify the statement that writes the entry to the console so it displays the data as shown above.

7.   Run the program and test it with the data shown above. If the program doesn't pause for you to enter a value for the zip code, that's because the newline character after the state code hasn't been retrieved from the console. In that case, you can fix this problem by adding a statement that extracts and discards any additional characters in the input stream.

8.   Run the program again to be sure it works correctly.

How to become a C++ programmer

The easiest way is to let **Murach's C++ Programming** be your guide! So if you've enjoyed this chapter, I hope you'll get your own copy of the book today. You can use it to:

Mike Murach, Publisher

- Teach yourself how to code, test, and debug C++ programs by using modern features of the language that help you learn more quickly and easily than ever before

- Master professional skills like how to work with I/O streams, files, strings, vectors, functions, the STL, arrays, C strings, exceptions, structures, classes, pointers, templates, and generic programming

- Learn classic C++ techniques for working with legacy code and embedded systems

- Develop C++ programs that combine the best procedural practices with the best object-oriented practices

- Pick up new skills whenever you want or need to by focusing on material that's new to you

- Look up coding details or refresh your memory on forgotten details when you're in the middle of developing a C++ program

- Loan to your colleagues who will be asking you more and more questions about C++

To get your copy, you can order online at **www.murach.com** or call us at 1-800-221-5528 (toll-free in the U.S. and Canada). And remember, when you order directly from us, this book comes with my personal guarantee:

100% Guarantee

You must be satisfied. Each book you buy directly from us must outperform any competing book or course you've ever tried, or send it back within 60 days for a full refund…no questions asked.

Thanks for your interest in Murach books!