2

Introduction to C++ Programming

What's in a name?
that which we call a rose
By any other name
would smell as sweet.
—William Shakespeare

When faced with a decision, I always ask, "What would be the most fun?"
—Peggy Walker

"Take some more tea," the March Hare said to Alice, very earnestly. "I've had nothing yet," Alice replied in an offended tone: "so I can't take more." "You mean you can't take less," said the Hatter: "it's very easy to take more than nothing."
—Lewis Carroll

High thoughts must have high language.
—Aristophanes

OBJECTIVES

In this chapter you'll learn:

- To write simple computer programs in C++.
- To write simple input and output statements.
- To use fundamental types.
- Basic computer memory concepts.
- To use arithmetic operators.
- The precedence of arithmetic operators.
- To write simple decision-making statements.

ISBN: 0-558-13856-X

Outline

2.1 Introduction

We now introduce C++ programming, which facilitates a disciplined approach to program design. Most of the C++ programs you'll study in this book process information and display results. In this chapter, we present five examples that demonstrate how your programs can display messages and obtain information from the user for processing. The first three examples simply display messages on the screen. The next obtains two numbers from a user, calculates their sum and displays the result. The accompanying discussion shows you how to perform various arithmetic calculations and save their results for later use. The fifth example demonstrates decision-making fundamentals by showing you how to compare two numbers, then display messages based on the comparison results. We analyze each program one line at a time to help you ease your way into C++ programming. To help you apply the skills you learn here, we provide many programming problems in the chapter's exercises.

2.2 First Program in C++: Printing a Line of Text

C++ uses notations that may appear strange to nonprogrammers. We now consider a simple program that prints a line of text (Fig. 2.1). This program illustrates several important features of the C++ language. We consider each line in detail.

Lines 1 and 2

```
// Fig. 2.1: fig02_01.cpp
// Text-printing program.
```

each begin with `//`, indicating that the remainder of each line is a **comment**. You insert comments to document your programs and to help other people read and understand them. Comments do not cause the computer to perform any action when the program is run—they are ignored by the C++ compiler and do not cause any machine-language object code to be generated. The comment `Text-printing program` describes the purpose of the program. A comment beginning with `//` is called a **single-line comment** because it terminates at the end of the current line. [*Note:* You also may use C's style in which a comment—possibly containing many lines—begins with `/*` and ends with `*/`.]

ISBN: 0-558-13856-X

```
1  // Fig. 2.1: fig02_01.cpp
2  // Text-printing program.
3  #include <iostream> // allows program to output data to the screen
4
5  // function main begins program execution
6  int main()
7  {
8     std::cout << "Welcome to C++!\n"; // display message
9
10    return 0; // indicate that program ended successfully
11
12 } // end function main
```

```
Welcome to C++!
```

Fig. 2.1 | Text-printing program.

Good Programming Practice 2.1

Every program should begin with a comment that describes the purpose of the program, author, date and time. (We are not showing the author, date and time in this book's programs because this information would be redundant.)

Line 3

```
#include <iostream> // allows program to output data to the screen
```

is a **preprocessor directive**, which is a message to the C++ preprocessor (introduced in Section 1.15). Lines that begin with `#` are processed by the preprocessor before the program is compiled. This line notifies the preprocessor to include in the program the contents of the **input/output stream header file `<iostream>`**. This file must be included for any program that outputs data to the screen or inputs data from the keyboard using C++-style stream input/output. The program in Fig. 2.1 outputs data to the screen, as we'll soon see. We discuss header files in more detail in Chapter 6 and explain the contents of `<iostream>` in Chapter 15.

Common Programming Error 2.1

Forgetting to include the `<iostream>` header file in a program that inputs data from the keyboard or outputs data to the screen causes the compiler to issue an error message, because the compiler cannot recognize references to the stream components (e.g., `cout`).

Line 4 is simply a blank line. You use blank lines, space characters and tab characters (i.e., "tabs") to make programs easier to read. Together, these characters are known as **white space**. White-space characters are normally ignored by the compiler. In this chapter and several that follow, we discuss conventions for using white-space characters to enhance program readability.

Good Programming Practice 2.2

Use blank lines, space characters and tabs to enhance program readability.

ISBN: 0-558-13856-X

Line 5

```
// function main begins program execution
```

is another single-line comment indicating that program execution begins at the next line.

Line 6

```
int main()
```

is a part of every C++ program. The parentheses after `main` indicate that `main` is a program building block called a **function**. C++ programs typically consist of one or more functions and classes (as you'll learn in Chapter 3). Exactly one function in every program must be `main`. Figure 2.1 contains only one function. C++ programs begin executing at function `main`, even if `main` is not the first function in the program. The keyword `int` to the left of `main` indicates that `main` "returns" an integer (whole number) value. A **keyword** is a word in code that is reserved by C++ for a specific use. The complete list of C++ keywords can be found in Fig. 4.3. We'll explain what it means for a function to "return a value" when we demonstrate how to create your own functions in Section 3.5 and when we study functions in greater depth in Chapter 6. For now, simply include the keyword `int` to the left of `main` in each of your programs.

The **left brace**, **{**, (line 7) must begin the **body** of every function. A corresponding **right brace**, **}**, (line 12) must end each function's body. Line 8

```
std::cout << "Welcome to C++!\n"; // display message
```

instructs the computer to **perform an action**—namely, to print the **string** of characters contained between the double quotation marks. A string is sometimes called a **character string**, a **message** or a **string literal**. We refer to characters between double quotation marks simply as **strings**. White-space characters in strings are not ignored by the compiler.

The entire line 8, including `std::cout`, the **<< operator**, the string `"Welcome to C++!\n"` and the **semicolon** (**;**), is called a **statement**. Every C++ statement must end with a semicolon (also known as the **statement terminator**). Preprocessor directives (like `#include`) do not end with a semicolon. Output and input in C++ are accomplished with **streams** of characters. Thus, when the preceding statement is executed, it sends the stream of characters `Welcome to C++!\n` to the **standard output stream object**—**`std::cout`**—which is normally "connected" to the screen. We discuss `std::cout`'s many features in detail in Chapter 15, Stream Input/Output.

Notice that we placed `std::` before `cout`. This is required when we use names that we've brought into the program by the preprocessor directive `#include <iostream>`. The notation `std::cout` specifies that we are using a name, in this case `cout`, that belongs to "namespace" `std`. The names `cin` (the standard input stream) and `cerr` (the standard error stream)—introduced in Chapter 1—also belong to namespace `std`. Namespaces are an advanced C++ feature that we discuss in depth in Chapter 25, Other Topics. For now, you should simply remember to include `std::` before each mention of `cout`, `cin` and `cerr` in a program. This can be cumbersome—in Fig. 2.13, we introduce the `using` declaration, which will enable us to omit `std::` before each use of a name in the `std` namespace.

The << operator is referred to as the **stream insertion operator**. When this program executes, the value to the operator's right, the right **operand**, is inserted in the output stream. Notice that the operator points in the direction of where the data goes. The right operand's characters normally print exactly as they appear between the double quotes.

ISBN: 0-558-13856-X

However, the characters \n are not printed on the screen (Fig. 2.1). The backslash (\) is called an **escape character**. It indicates that a "special" character is to be output. When a backslash is encountered in a string of characters, the next character is combined with the backslash to form an **escape sequence**. The escape sequence \n means **newline**. It causes the **cursor** (i.e., the current screen-position indicator) to move to the beginning of the next line on the screen. Some common escape sequences are listed in Fig. 2.2.

Common Programming Error 2.2

*Omitting the semicolon at the end of a C++ statement is a syntax error. (Again, preprocessor directives do not end in a semicolon.) The **syntax** of a programming language specifies the rules for creating proper programs in that language. A **syntax error** occurs when the compiler encounters code that violates C++'s language rules (i.e., its syntax). The compiler normally issues an error message to help you locate and fix the incorrect code. Syntax errors are also called **compiler errors**, **compile-time errors** or **compilation errors**, because the compiler detects them during the compilation phase. You cannot execute your program until you correct all the syntax errors in it. As you'll see, some compilation errors are not syntax errors.*

Line 10

```
return 0; // indicate that program ended successfully
```

is one of several means we'll use to **exit a function**. When the `return` statement is used at the end of `main`, as shown here, the value `0` indicates that the program has terminated successfully. In Chapter 6 we discuss functions in detail, and the reasons for including this statement will become clear. For now, simply include this statement in each program, or the compiler may produce a warning on some systems. The right brace, `}`, (line 12) indicates the end of function `main`.

Good Programming Practice 2.3

Many programmers make the last character printed by a function a newline (\n). This ensures that the function will leave the screen cursor positioned at the beginning of a new line. Conventions of this nature encourage software reusability—a key goal in software development.

Escape sequence	Description
\n	Newline. Position the screen cursor to the beginning of the next line.
\t	Horizontal tab. Move the screen cursor to the next tab stop.
\r	Carriage return. Position the screen cursor to the beginning of the current line; do not advance to the next line.
\a	Alert. Sound the system bell.
\\	Backslash. Used to print a backslash character.
\'	Single quote. Use to print a single quote character.
\"	Double quote. Used to print a double quote character.

Fig. 2.2 | Escape sequences.

ISBN: 0-558-13856-X

Good Programming Practice 2.4

Indent the entire body of each function one level within the braces that delimit the body of the function. This makes a program's functional structure stand out and makes the program easier to read.

Good Programming Practice 2.5

Set a convention for the size of indent you prefer, then apply it uniformly. The tab key may be used to create indents, but tab stops may vary. We recommend using either 1/4-inch tab stops or (preferably) three spaces to form a level of indent.

2.3 Modifying Our First C++ Program

This section continues our introduction to C++ programming with two examples, showing how to modify the program in Fig. 2.1 to print text on one line by using multiple statements, and to print text on several lines by using a single statement.

Printing a Single Line of Text with Multiple Statements

`Welcome to C++!` can be printed several ways. For example, Fig. 2.3 performs stream insertion in multiple statements (lines 8–9), yet produces the same output as the program of Fig. 2.1. [*Note:* From this point forward, we use a darker shade of gray than the code table background to highlight the key features each program introduces.] Each stream insertion resumes printing where the previous one stopped. The first stream insertion (line 8) prints `Welcome` followed by a space, and the second stream insertion (line 9) begins printing on the same line immediately following the space. In general, C++ allows you to express statements in a variety of ways.

Printing Multiple Lines of Text with a Single Statement

A single statement can print multiple lines by using newline characters, as in line 8 of Fig. 2.4. Each time the `\n` (newline) escape sequence is encountered in the output stream, the screen cursor is positioned to the beginning of the next line. To get a blank line in your output, place two newline characters back to back, as in line 8.

```
1  // Fig. 2.3: fig02_03.cpp
2  // Printing a line of text with multiple statements.
3  #include <iostream> // allows program to output data to the screen
4
5  // function main begins program execution
6  int main()
7  {
8     std::cout << "Welcome ";
9     std::cout << "to C++!\n";
10
11    return 0; // indicate that program ended successfully
12
13 } // end function main
```

```
Welcome to C++!
```

Fig. 2.3 | Printing a line of text with multiple statements.

ISBN: 0-558-13856-X

```
1  // Fig. 2.4: fig02_04.cpp
2  // Printing multiple lines of text with a single statement.
3  #include <iostream> // allows program to output data to the screen
4
5  // function main begins program execution
6  int main()
7  {
8     std::cout << "Welcome\nto\n\nC++!\n";
9
10    return 0; // indicate that program ended successfully
11
12 } // end function main
```

```
Welcome
to

C++!
```

Fig. 2.4 | Printing multiple lines of text with a single statement.

2.4 Another C++ Program: Adding Integers

Our next program uses the input stream object **`std::cin`** and the **stream extraction operator**, **`>>`**, to obtain two integers typed by a user at the keyboard, computes the sum of these values and outputs the result using `std::cout`. Figure 2.5 shows the program and sample inputs and outputs. Note that we highlight the user's input in bold.

The comments in lines 1 and 2

```
// Fig. 2.5: fig02_05.cpp
// Addition program that displays the sum of two numbers.
```

state the name of the file and the purpose of the program. The C++ preprocessor directive

```
#include <iostream> // allows program to perform input and output
```

in line 3 includes the contents of the `<iostream>` header file in the program.

The program begins execution with function `main` (line 6). The left brace (line 7) marks the beginning of `main`'s body and the corresponding right brace (line 25) marks the end of `main`.

Lines 9–11

```
int number1; // first integer to add
int number2; // second integer to add
int sum; // sum of number1 and number2
```

are **declarations**. The identifiers `number1`, `number2` and `sum` are the names of **variables**. A variable is a location in the computer's memory where a value can be stored for use by a program. These declarations specify that the variables `number1`, `number2` and `sum` are data of type **`int`**, meaning that these variables will hold **integer** values, i.e., whole numbers such as 7, –11, 0 and 31914. All variables must be declared with a name and a data type before they can be used in a program. Several variables of the same type may be declared in one

ISBN: 0-558-13856-X

```
1  // Fig. 2.5: fig02_05.cpp
2  // Addition program that displays the sum of two integers.
3  #include <iostream> // allows program to perform input and output
4
5  // function main begins program execution
6  int main()
7  {
8     // variable declarations
9     int number1; // first integer to add
10    int number2; // second integer to add
11    int sum; // sum of number1 and number2
12
13    std::cout << "Enter first integer: "; // prompt user for data
14    std::cin >> number1; // read first integer from user into number1
15
16    std::cout << "Enter second integer: "; // prompt user for data
17    std::cin >> number2; // read second integer from user into number2
18
19    sum = number1 + number2; // add the numbers; store result in sum
20
21    std::cout << "Sum is " << sum << std::endl; // display sum; end line
22
23    return 0; // indicate that program ended successfully
24
25 } // end function main
```

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

Fig. 2.5 | Addition program that displays the sum of two integers entered at the keyboard.

declaration or in multiple declarations. We could have declared all three variables in one declaration as follows:

```
int number1, number2, sum;
```

This makes the program less readable and prevents us from providing comments that describe each variable's purpose. If more than one name is declared in a declaration (as shown here), the names are separated by commas (,); this is referred to as a **comma-separated list**.

Good Programming Practice 2.6

Place a space after each comma (,) to make programs more readable.

Good Programming Practice 2.7

Some programmers prefer to declare each variable on a separate line. This format allows for easy insertion of a descriptive comment next to each declaration.

We'll soon discuss the data type `double` for specifying real numbers, and the data type `char` for specifying character data. Real numbers are numbers with decimal points, such as 3.4, 0.0 and –11.19. A `char` variable may hold only a single lowercase letter, a single uppercase letter, a single digit or a single special character (e.g., $ or *). Types such as `int`,

ISBN: 0-558-13856-X

`double` and `char` are often called **fundamental types**, **primitive types** or **built-in types**. Fundamental-type names are keywords and therefore must appear in all lowercase letters. Appendix C contains the complete list of fundamental types.

A variable name (such as `number1`) is any valid **identifier** that is not a keyword. An identifier is a series of characters consisting of letters, digits and underscores (_) that does not begin with a digit. C++ is **case sensitive**—uppercase and lowercase letters are different, so `a1` and `A1` are different identifiers.

Portability Tip 2.1

C++ allows identifiers of any length, but your C++ implementation may impose some restrictions on the length of identifiers. Use identifiers of 31 characters or fewer to ensure portability.

Good Programming Practice 2.8

Choosing meaningful identifiers makes a program self-documenting—a person can understand the program simply by reading it rather than having to refer to manuals or comments.

Good Programming Practice 2.9

Avoid using abbreviations in identifiers. This promotes program readability.

Good Programming Practice 2.10

Avoid identifiers that begin with underscores and double underscores, because C++ compilers may use names like that for their own purposes internally. This will prevent names you choose from being confused with names the compilers choose.

Error-Prevention Tip 2.1

Languages like C++ are "moving targets." As they evolve, more keywords could be added to the language. Avoid using "loaded" words like "object" as identifiers. Even though "object" is not currently a keyword in C++, it could become one; therefore, future compiling with new compilers could break existing code.

Declarations of variables can be placed almost anywhere in a program, but they must appear before their corresponding variables are used in the program. For example, in the program of Fig. 2.5, the declaration in line 9

```
int number1; // first integer to add
```

could have been placed immediately before line 14

```
std::cin >> number1; // read first integer from user into number1
```

the declaration in line 10

```
int number2; // second integer to add
```

could have been placed immediately before line 17

```
std::cin >> number2; // read second integer from user into number2
```

and the declaration in line 11

```
int sum; // sum of number1 and number2
```

could have been placed immediately before line 19

```
sum = number1 + number2; // add the numbers; store result in sum
```

ISBN: 0-558-13856-X

Good Programming Practice 2.11

Always place a blank line between a declaration and adjacent executable statements. This makes the declarations stand out in the program and contributes to program clarity.

Good Programming Practice 2.12

If you prefer to place declarations at the beginning of a function, separate them from the executable statements in that function with one blank line to highlight where the declarations end and the executable statements begin.

Line 13

```
std::cout << "Enter first integer: "; // prompt user for data
```

prints the string `Enter first integer:` on the screen. This message is called a **prompt** because it directs the user to take a specific action. We like to pronounce the preceding statement as "`std::cout` *gets* the character string `"Enter first integer: "`." Line 14

```
std::cin >> number1; // read first integer from user into number1
```

uses the **input stream object `cin`** (of namespace `std`) and the **stream extraction operator**, `>>`, to obtain a value from the keyboard. Using the stream extraction operator with `std::cin` takes character input from the standard input stream, which is usually the keyboard. We like to pronounce the preceding statement as, "`std::cin` *gives* a value to `number1`" or simply "`std::cin` *gives* `number1`."

Error-Prevention Tip 2.2

Programs should validate the correctness of all input values to prevent erroneous information from affecting a program's calculations.

When the computer executes the preceding statement, it waits for the user to enter a value for variable `number1`. The user responds by typing an integer (as characters), then pressing the *Enter* key (sometimes called the *Return* key) to send the characters to the computer. The computer converts the character representation of the number to an integer and assigns (i.e., copies) this number (or **value**) to the variable `number1`. Any subsequent references to `number1` in this program will use this same value.

The `std::cout` and `std::cin` stream objects facilitate interaction between the user and the computer. Because this interaction resembles a dialog, it is often called **conversational computing** or **interactive computing.**

Line 16

```
std::cout << "Enter second integer: "; // prompt user for data
```

prints `Enter second integer:` on the screen, prompting the user to take action. Line 17

```
std::cin >> number2; // read second integer from user into number2
```

obtains a value for variable `number2` from the user.

The assignment statement in line 19

```
sum = number1 + number2; // add the numbers; store result in sum
```

ISBN: 0-558-13856-X

calculates the sum of the variables `number1` and `number2` and assigns the result to variable `sum` using the **assignment operator** `=`. The statement is read as, "`sum` *gets* the value of `number1 + number2`." Most calculations are performed in assignment statements. The `=` operator and the `+` operator are called **binary operators** because each has two operands. In the case of the `+` operator, the two operands are `number1` and `number2`. In the case of the preceding `=` operator, the two operands are `sum` and the value of the expression `number1 + number2`.

Good Programming Practice 2.13

Place spaces on either side of a binary operator. This makes the operator stand out and makes the program more readable.

Line 21

```
std::cout << "Sum is " << sum << std::endl; // display sum; end line
```

displays the character string `Sum is` followed by the numerical value of variable `sum` followed by `std::endl`—a so-called **stream manipulator**. The name `endl` is an abbreviation for "end line" and belongs to namespace `std`. The `std::endl` stream manipulator outputs a newline, then "flushes the output buffer." This simply means that, on some systems where outputs accumulate in the machine until there are enough to "make it worthwhile" to display them on the screen, `std::endl` forces any accumulated outputs to be displayed at that moment. This can be important when the outputs are prompting the user for an action, such as entering data.

Note that the preceding statement outputs multiple values of different types. The stream insertion operator "knows" how to output each type of data. Using multiple stream insertion operators (`<<`) in a single statement is referred to as **concatenating**, **chaining** or **cascading stream insertion operations**. It is unnecessary to have multiple statements to output multiple pieces of data.

Calculations can also be performed in output statements. We could have combined the statements in lines 19 and 21 into the statement

```
std::cout << "Sum is " << number1 + number2 << std::endl;
```

thus eliminating the need for the variable `sum`.

A powerful feature of C++ is that users can create their own data types called classes (we introduce this capability in Chapter 3 and explore it in depth in Chapters 9 and 10). Users can then "teach" C++ how to input and output values of these new data types using the `>>` and `<<` operators (this is called **operator overloading**—a topic we explore in Chapter 11).

2.5 Memory Concepts

Variable names such as `number1`, `number2` and `sum` actually correspond to **locations** in the computer's memory. Every variable has a name, a type, a size and a value.

In the addition program of Fig. 2.5, when the statement

```
std::cin >> number1; // read first integer from user into number1
```

in line 14 is executed, the characters typed by the user are converted to an integer that is placed into a memory location to which the name `number1` has been assigned by the C++

ISBN: 0-558-13856-X

compiler. Suppose the user enters the number 45 as the value for number1. The computer will place 45 into location number1, as shown in Fig. 2.6.

Whenever a value is placed in a memory location, the value overwrites the previous value in that location; thus, placing a new value into a memory location is said to be **destructive**.

Returning to our addition program, when the statement

```
std::cin >> number2; // read second integer from user into number2
```

in line 17 is executed, suppose the user enters the value 72. This value is placed into location number2, and memory appears as in Fig. 2.7. Note that these locations are not necessarily adjacent in memory.

Once the program has obtained values for number1 and number2, it adds these values and places the sum into variable sum. The statement

```
sum = number1 + number2; // add the numbers; store result in sum
```

that performs the addition also replaces whatever value was stored in sum. This occurs when the calculated sum of number1 and number2 is placed into location sum (without regard to what value may already be in sum; that value is lost). After sum is calculated, memory appears as in Fig. 2.8. Note that the values of number1 and number2 appear exactly as they did before they were used in the calculation of sum. These values were used, but not destroyed, as the computer performed the calculation. Thus, when a value is read out of a memory location, the process is **nondestructive**.

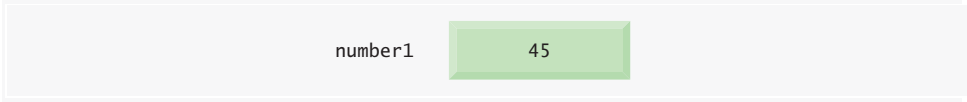

Fig. 2.6 | Memory location showing the name and value of variable number1.

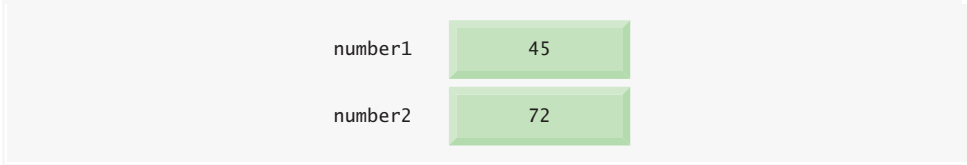

Fig. 2.7 | Memory locations after storing values for number1 and number2.

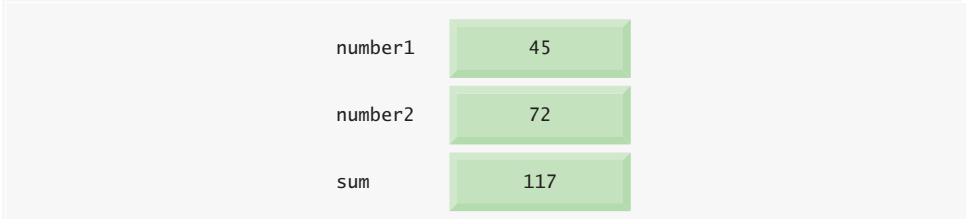

Fig. 2.8 | Memory locations after calculating and storing the sum of number1 and number2.

ISBN: 0-558-13856-X

2.6 Arithmetic

Most programs perform arithmetic calculations. Figure 2.9 summarizes the C++ **arithmetic operators**. Note the use of various special symbols not used in algebra. The **asterisk** (`*`) indicates multiplication and the **percent sign** (`%`) is the **modulus** operator that will be discussed shortly. The arithmetic operators in Fig. 2.9 are all binary operators, i.e., operators that take two operands. For example, the expression `number1 + number2` contains the binary operator `+` and the two operands `number1` and `number2`.

Integer division (i.e., where both the numerator and the denominator are integers) yields an integer quotient; for example, the expression `7 / 4` evaluates to `1` and the expression `17 / 5` evaluates to `3`. Note that any fractional part in integer division is discarded (i.e., **truncated**)—no rounding occurs.

C++ provides the **modulus operator**, `%`, that yields the remainder after integer division. The modulus operator can be used only with integer operands. The expression `x % y` yields the remainder after `x` is divided by `y`. Thus, `7 % 4` yields `3` and `17 % 5` yields `2`. In later chapters, we discuss many interesting applications of the modulus operator, such as determining whether one number is a multiple of another (a special case of this is determining whether a number is odd or even).

Common Programming Error 2.3

Attempting to use the modulus operator (%) with noninteger operands is a compilation error.

Arithmetic Expressions in Straight-Line Form

Arithmetic expressions in C++ must be entered into the computer in **straight-line form**. Thus, expressions such as "a divided by b" must be written as `a / b`, so that all constants, variables and operators appear in a straight line. The algebraic notation

$$\frac{a}{b}$$

is generally not acceptable to compilers, although some special-purpose software packages do support more natural notation for complex mathematical expressions.

Parentheses for Grouping Subexpressions

Parentheses are used in C++ expressions in the same manner as in algebraic expressions. For example, to multiply a times the quantity `b + c` we write `a * ( b + c )`.

C++ operation	C++ arithmetic operator	Algebraic expression	C++ expression
Addition	`+`	$f + 7$	`f + 7`
Subtraction	`-`	$p - c$	`p - c`
Multiplication	`*`	bm or $b \cdot m$	`b * m`
Division	`/`	x / y or $\frac{x}{y}$ or $x \div y$	`x / y`
Modulus	`%`	$r \bmod s$	`r % s`

Fig. 2.9 | Arithmetic operators.

ISBN: 0-558-13856-X

Rules of Operator Precedence

C++ applies the operators in arithmetic expressions in a precise sequence determined by the following **rules of operator precedence**, which are generally the same as those followed in algebra:

1. Operators in expressions contained within pairs of parentheses are evaluated first. Parentheses are said to be at the "highest level of precedence." In cases of **nested**, or **embedded**, **parentheses**, such as

   ```
   ( ( a + b ) + c )
   ```

   the operators in the innermost pair of parentheses are applied first.
2. Multiplication, division and modulus operations are applied next. If an expression contains several multiplication, division and modulus operations, operators are applied from left to right. Multiplication, division and modulus are said to be on the same level of precedence.
3. Addition and subtraction operations are applied last. If an expression contains several addition and subtraction operations, operators are applied from left to right. Addition and subtraction also have the same level of precedence.

The set of rules of operator precedence defines the order in which C++ applies operators. When we say that certain operators are applied from left to right, we are referring to the **associativity** of the operators. For example, in the expression

```
a + b + c
```

the addition operators (+) associate from left to right, so `a + b` is calculated first, then `c` is added to that sum to determine the value of the whole expression. We'll see that some operators associate from right to left. Figure 2.10 summarizes these rules of operator precedence. This table will be expanded as additional C++ operators are introduced. A complete precedence chart is included in Appendix A.

Operator(s)	Operation(s)	Order of evaluation (precedence)
`( )`	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses "on the same level" (i.e., not nested), they are evaluated left to right.
`*`, `/`, `%`	Multiplication, Division, Modulus	Evaluated second. If there are several, they are evaluated left to right.
`+` `-`	Addition Subtraction	Evaluated last. If there are several, they are evaluated left to right.

Fig. 2.10 | Precedence of arithmetic operators.

ISBN: 0-558-13856-X

Sample Algebraic and C++ Expressions

Now consider several expressions in light of the rules of operator precedence. Each example lists an algebraic expression and its C++ equivalent. The following is an example of an arithmetic mean (average) of five terms:

Algebra: $m = \frac{a + b + c + d + e}{5}$

C++:
```
m = ( a + b + c + d + e ) / 5;
```

The parentheses are required because division has higher precedence than addition. The entire quantity `( a + b + c + d + e )` is to be divided by `5`. If the parentheses are erroneously omitted, we obtain `a + b + c + d + e / 5`, which evaluates incorrectly as

$$a + b + c + d + \frac{e}{5}$$

The following is an example of the equation of a straight line:

Algebra: $y = mx + b$

C++:
```
y = m * x + b;
```

No parentheses are required. The multiplication is applied first because multiplication has a higher precedence than addition.

The following example contains modulus (%), multiplication, division, addition, subtraction and assignment operations:

Algebra: $z = pr\%q + w/x - y$

C++:
```
z = p * r % q + w / x - y;
```
6 1 2 4 3 5

The circled numbers under the statement indicate the order in which C++ applies the operators. The multiplication, modulus and division are evaluated first in left-to-right order (i.e., they associate from left to right) because they have higher precedence than addition and subtraction. The addition and subtraction are applied next. These are also applied left to right. Then the assignment operator is applied.

Evaluation of a Second-Degree Polynomial

To develop a better understanding of the rules of operator precedence, consider the evaluation of a second-degree polynomial ($y = ax^2 + bx + c$):

```
y = a * x * x + b * x + c;
```
6 1 2 4 3 5

The circled numbers under the statement indicate the order in which C++ applies the operators. There is no arithmetic operator for exponentiation in C++, so we have represented x^2 as `x * x`. We'll soon discuss the standard library function `pow` ("power") that performs exponentiation. Because of some subtle issues related to the data types required by `pow`, we defer a detailed explanation of `pow` until Chapter 6.

ISBN: 0-558-13856-X

Common Programming Error 2.4

*Some programming languages use operators ** or ^ to represent exponentiation. C++ does not support these exponentiation operators; using them for exponentiation results in errors.*

Suppose variables `a`, `b`, `c` and `x` in the preceding second-degree polynomial are initialized as follows: `a = 2`, `b = 3`, `c = 7` and `x = 5`. Figure 2.11 illustrates the order in which the operators are applied.

As in algebra, it is acceptable to place unnecessary parentheses in an expression to make the expression clearer. These are called **redundant parentheses**. For example, the preceding assignment statement could be parenthesized as follows:

```
y = ( a * x * x ) + ( b * x ) + c;
```

Good Programming Practice 2.14

Using redundant parentheses in complex arithmetic expressions can make the expressions clearer.

2.7 Decision Making: Equality and Relational Operators

This section introduces a simple version of C++'s **`if` statement** that allows a program to take alternative action based on the truth or falsity of some **condition**. If the condition is met, i.e., the condition is true, the statement in the body of the `if` statement is executed. If the condition is not met, i.e., the condition is false, the body statement is not executed. We'll see an example shortly.

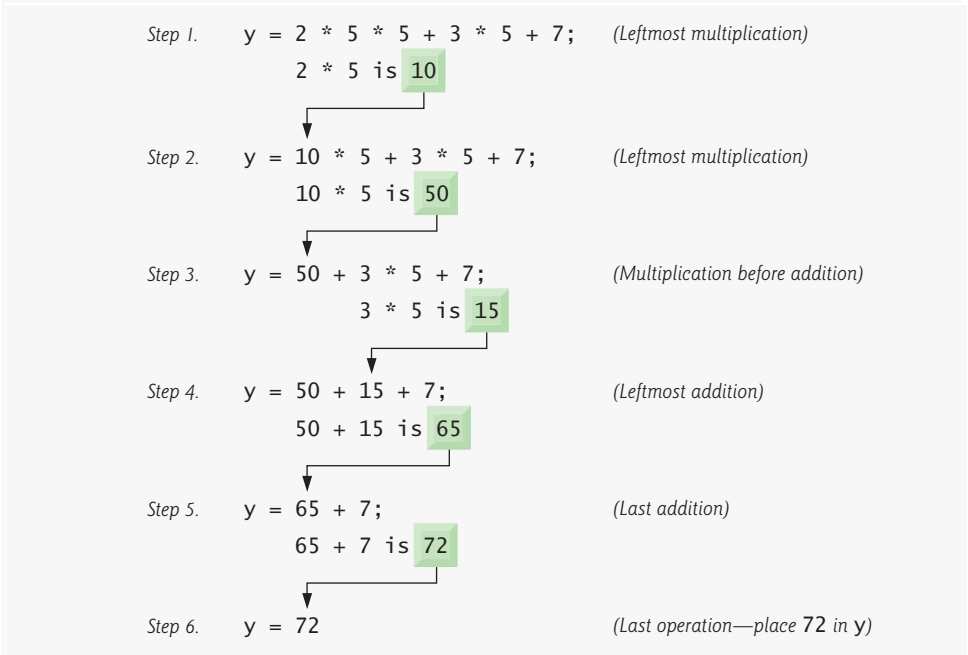

Fig. 2.11 | Order in which a second-degree polynomial is evaluated.

ISBN: 0-558-13856-X

Conditions in `if` statements can be formed by using the **equality operators** and **relational operators** summarized in Fig. 2.12. The relational operators all have the same level of precedence and associate left to right. The equality operators both have the same level of precedence, which is lower than that of the relational operators, and associate left to right.

Common Programming Error 2.5

A syntax error will occur if any of the operators ==, !=, >= and <= appears with spaces between its pair of symbols.

Common Programming Error 2.6

Reversing the order of the pair of symbols in any of the operators !=, >= and <= (by writing them as =!, => and =<, respectively) is normally a syntax error. In some cases, writing != as =! will not be a syntax error, but almost certainly will be a ***logic error*** *that has an effect at execution time. You'll understand why when you learn about logical operators in Chapter 5. A* ***fatal logic error*** *causes a program to fail and terminate prematurely. A* ***nonfatal logic error*** *allows a program to continue executing, but usually produces incorrect results.*

Common Programming Error 2.7

Confusing the equality operator == with the assignment operator = results in logic errors. The equality operator should be read "is equal to," and the assignment operator should be read "gets" or "gets the value of" or "is assigned the value of." Some people prefer to read the equality operator as "double equals." As we discuss in Section 5.9, confusing these operators may not necessarily cause an easy-to-recognize syntax error, but may cause extremely subtle logic errors.

The following example uses six `if` statements to compare two numbers input by the user. If the condition in any of these `if` statements is satisfied, the output statement associated with that `if` statement is executed. Figure 2.13 shows the program and the input/output dialogs of three sample executions.

Standard algebraic equality or relational operator	C++ equality or relational operator	Sample C++ condition	Meaning of C++ condition
Relational operators			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y
Equality operators			
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y

Fig. 2.12 | Equality and relational operators.

ISBN: 0-558-13856-X

Lines 6–8

```
using std::cout; // program uses cout
using std::cin; // program uses cin
using std::endl; // program uses endl
```

are **using declarations** that eliminate the need to repeat the `std::` prefix as we did in earlier programs. Once we insert these `using` declarations, we can write `cout` instead of `std::cout`, `cin` instead of `std::cin` and `endl` instead of `std::endl`, respectively, in the remainder of the program. [*Note:* From this point forward in the book, each example contains one or more `using` declarations.]

```
1  // Fig. 2.13: fig02_13.cpp
2  // Comparing integers using if statements, relational operators
3  // and equality operators.
4  #include <iostream> // allows program to perform input and output
5
6  using std::cout; // program uses cout
7  using std::cin; // program uses cin
8  using std::endl; // program uses endl
9
10 // function main begins program execution
11 int main()
12 {
13    int number1; // first integer to compare
14    int number2; // second integer to compare
15
16    cout << "Enter two integers to compare: "; // prompt user for data
17    cin >> number1 >> number2; // read two integers from user
18
19    if ( number1 == number2 )
20       cout << number1 << " == " << number2 << endl;
21
22    if ( number1 != number2 )
23       cout << number1 << " != " << number2 << endl;
24
25    if ( number1 < number2 )
26       cout << number1 << " < " << number2 << endl;
27
28    if ( number1 > number2 )
29       cout << number1 << " > " << number2 << endl;
30
31    if ( number1 <= number2 )
32       cout << number1 << " <= " << number2 << endl;
33
34    if ( number1 >= number2 )
35       cout << number1 << " >= " << number2 << endl;
36
37    return 0; // indicate that program ended successfully
38
39 } // end function main
```

Fig. 2.13 | Comparing integers using `if` statements, relational operators and equality operators. (Part 1 of 2.)

ISBN: 0-558-13856-X

```
Enter two integers to compare: 3 7
3 != 7
3 < 7
3 <= 7
```

```
Enter two integers to compare: 22 12
22 != 12
22 > 12
22 >= 12
```

```
Enter two integers to compare: 7 7
7 == 7
7 <= 7
7 >= 7
```

Fig. 2.13 | Comparing integers using `if` statements, relational operators and equality operators. (Part 2 of 2.)

Good Programming Practice 2.15

Place `using` declarations immediately after the `#include` to which they refer.

Lines 13–14

```
int number1; // first integer to compare
int number2; // second integer to compare
```

declare the variables used in the program. Remember that variables may be declared in one declaration or in separate declarations.

The program uses cascaded stream extraction operations (line 17) to input two integers. Remember that we are allowed to write `cin` (instead of `std::cin`) because of line 7. First a value is read into variable `number1`, then a value is read into variable `number2`.

The `if` statement in lines 19–20

```
if ( number1 == number2 )
   cout << number1 << " == " << number2 << endl;
```

compares the values of variables `number1` and `number2` to test for equality. If the values are equal, the statement in line 20 displays a line of text indicating that the numbers are equal. If the conditions are `true` in one or more of the `if` statements starting in lines 22, 25, 28, 31 and 34, the corresponding body statement displays an appropriate line of text.

Notice that each `if` statement in Fig. 2.13 has a single statement in its body and that each body statement is indented. In Chapter 4 we show how to specify `if` statements with multiple-statement bodies (by enclosing the body statements in a pair of braces, `{ }`, creating what is called a **compound statement** or a **block**).

Good Programming Practice 2.16

Indent the statement(s) in the body of an `if` statement to enhance readability.

ISBN: 0-558-13856-X

Good Programming Practice 2.17

For readability, there should be no more than one statement per line in a program.

Common Programming Error 2.8

Placing a semicolon immediately after the right parenthesis after the condition in an `if` statement is often a logic error (although not a syntax error). The semicolon causes the body of the `if` statement to be empty, so the `if` statement performs no action, regardless of whether or not its condition is true. Worse yet, the original body statement of the `if` statement now becomes a statement in sequence with the `if` statement and always executes, often causing the program to produce incorrect results.

Note the use of white space in Fig. 2.13. Recall that white-space characters, such as tabs, newlines and spaces, are normally ignored by the compiler. So, statements may be split over several lines and may be spaced according to your preferences. It is a syntax error to split identifiers, strings (such as `"hello"`) and constants (such as the number `1000`) over several lines.

Common Programming Error 2.9

It is a syntax error to split an identifier by inserting white-space characters (e.g., writing `main` as `ma in`).

Good Programming Practice 2.18

A lengthy statement may be spread over several lines. If a single statement must be split across lines, choose meaningful breaking points, such as after a comma in a comma-separated list, or after an operator in a lengthy expression. If a statement is split across two or more lines, indent all subsequent lines and left-align the group of indented lines.

Figure 2.14 shows the precedence and associativity of the operators introduced in this chapter. The operators are shown top to bottom in decreasing order of precedence. Notice that all these operators, with the exception of the assignment operator `=`, associate from left to right. Addition is left-associative, so an expression like `x + y + z` is evaluated as if it had been written `(x + y) + z`. The assignment operator `=` associates from right to left, so an expression such as `x = y = 0` is evaluated as if it had been written `x = (y = 0)`, which, as we'll soon see, first assigns `0` to `y`, then assigns the result of that assignment—`0`—to `x`.

Operators				Associativity	Type
`()`				left to right	parentheses
`*`	`/`	`%`		left to right	multiplicative
`+`	`-`			left to right	additive
`<<`	`>>`			left to right	stream insertion/extraction
`<`	`<=`	`>`	`>=`	left to right	relational
`==`	`!=`			left to right	equality
`=`				right to left	assignment

Fig. 2.14 | Precedence and associativity of the operators discussed so far.

ISBN: 0-558-13856-X

Good Programming Practice 2.19

Refer to the operator precedence and associativity chart when writing expressions containing many operators. Confirm that the operators in the expression are performed in the order you expect. If you are uncertain about the order of evaluation in a complex expression, break the expression into smaller statements or use parentheses to force the order of evaluation, exactly as you would do in an algebraic expression. Be sure to observe that some operators such as assignment (=) associate right to left rather than left to right.

2.8 (Optional) Software Engineering Case Study: Examining the ATM Requirements Specification

Now we begin our optional object-oriented design and implementation case study. The Software Engineering Case Study sections at the ends of this and the next several chapters will ease you into object orientation. We'll develop software for a simple automated teller machine (ATM) system, providing you with a concise, carefully paced, complete design and implementation experience. In Chapters 3–7, 9 and 13, we'll perform the various steps of an object-oriented design (OOD) process using the UML, while relating these steps to the object-oriented concepts discussed in the chapters. Appendix G implements the ATM using the techniques of object-oriented programming (OOP) in C++. We present the complete case study solution. This is not an exercise; rather, it is an end-to-end learning experience that concludes with a detailed walkthrough of the C++ code that implements our design. It will acquaint you with the kinds of substantial problems encountered in industry and their potential solutions.

We begin our design process by presenting a **requirements specification** that specifies the overall purpose of the ATM system and *what* it must do. Throughout the case study, we refer to the requirements specification to determine precisely what functionality the system must include.

Requirements Specification

A local bank intends to install a new automated teller machine (ATM) to allow users (i.e., bank customers) to perform basic financial transactions (Fig. 2.15). Each user can have only one account at the bank. ATM users should be able to view their account balance, withdraw cash (i.e., take money out of an account) and deposit funds (i.e., place money into an account).

The user interface of the automated teller machine contains the following hardware components:

- a screen that displays messages to the user
- a keypad that receives numeric input from the user
- a cash dispenser that dispenses cash to the user and
- a deposit slot that receives deposit envelopes from the user.

The cash dispenser begins each day loaded with 500 $20 bills. [*Note:* Owing to the limited scope of this case study, certain elements of the ATM described here do not accurately mimic those of a real ATM. For example, a real ATM typically contains a device that reads a user's account number from an ATM card, whereas this ATM asks the user to type an account number using the keypad. A real ATM also usually prints a receipt at the end of a session, but all output from this ATM appears on the screen.]

ISBN: 0-558-13856-X

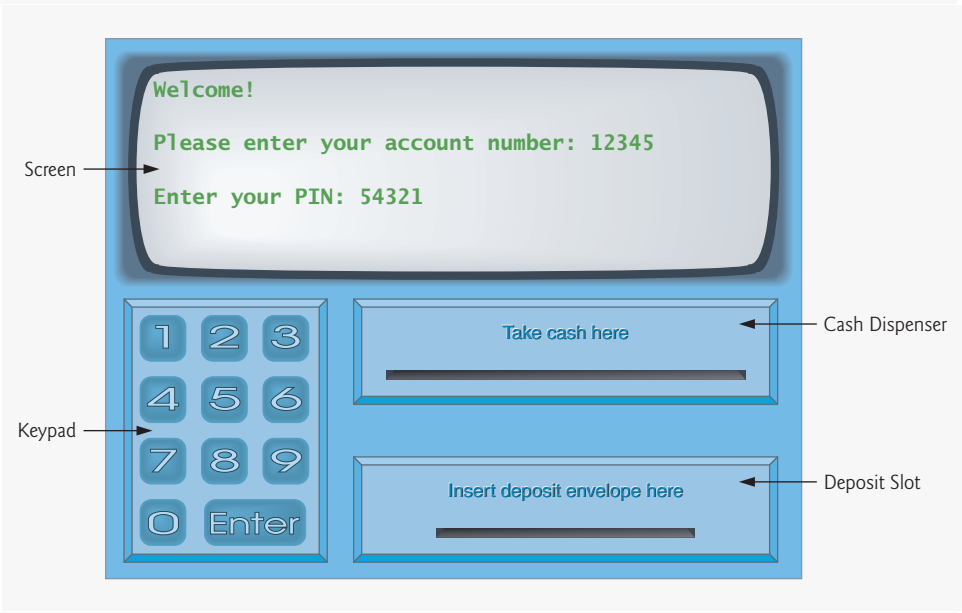

Fig. 2.15 | Automated teller machine user interface.

The bank wants you to develop software to perform the financial transactions initiated by bank customers through the ATM. The bank will integrate the software with the ATM's hardware at a later time. The software should encapsulate the functionality of the hardware devices (e.g., cash dispenser, deposit slot) within software components, but it need not concern itself with how these devices perform their duties. The ATM hardware has not been developed yet, so instead of writing your software to run on the ATM, you should develop a first version of the software to run on a personal computer. This version should use the computer's monitor to simulate the ATM's screen, and the computer's keyboard to simulate the ATM's keypad.

An ATM session consists of authenticating a user (i.e., proving the user's identity) based on an account number and personal identification number (PIN), followed by creating and executing financial transactions. To authenticate a user and perform transactions, the ATM must interact with the bank's account information database. [*Note:* A database is an organized collection of data stored on a computer.] For each bank account, the database stores an account number, a PIN and a balance indicating the amount of money in the account. [*Note:* For simplicity, we assume that the bank plans to build only one ATM, so we do not need to worry about multiple ATMs accessing this database at the same time. Furthermore, we assume that the bank does not make any changes to the information in the database while a user is accessing the ATM. Also, any business system like an ATM faces reasonably complicated security issues that go well beyond the scope of a first- or second-semester computer science course. We make the simplifying assumption, however, that the bank trusts the ATM to access and manipulate the information in the database without significant security measures.]

Upon first approaching the ATM, the user should experience the following sequence of events (shown in Fig. 2.15):

ISBN: 0-558-13856-X

1. The screen displays a welcome message and prompts the user to enter an account number.
2. The user enters a five-digit account number, using the keypad.
3. The screen prompts the user to enter the PIN (personal identification number) associated with the specified account number.
4. The user enters a five-digit PIN, using the keypad.
5. If the user enters a valid account number and the correct PIN for that account, the screen displays the main menu (Fig. 2.16). If the user enters an invalid account number or an incorrect PIN, the screen displays an appropriate message, then the ATM returns to *Step 1* to restart the authentication process.

After the ATM authenticates the user, the main menu (Fig. 2.16) displays a numbered option for each of the three types of transactions: balance inquiry (option 1), withdrawal (option 2) and deposit (option 3). The main menu also displays an option that allows the user to exit the system (option 4). The user then chooses either to perform a transaction (by entering 1, 2 or 3) or to exit the system (by entering 4). If the user enters an invalid option, the screen displays an error message, then redisplays to the main menu.

If the user enters 1 to make a balance inquiry, the screen displays the user's account balance. To do so, the ATM must retrieve the balance from the bank's database.

The following actions occur when the user enters 2 to make a withdrawal:

1. The screen displays a menu (shown in Fig. 2.17) containing standard withdrawal amounts: $20 (option 1), $40 (option 2), $60 (option 3), $100 (option 4) and $200 (option 5). The menu also contains an option to allow the user to cancel the transaction (option 6).

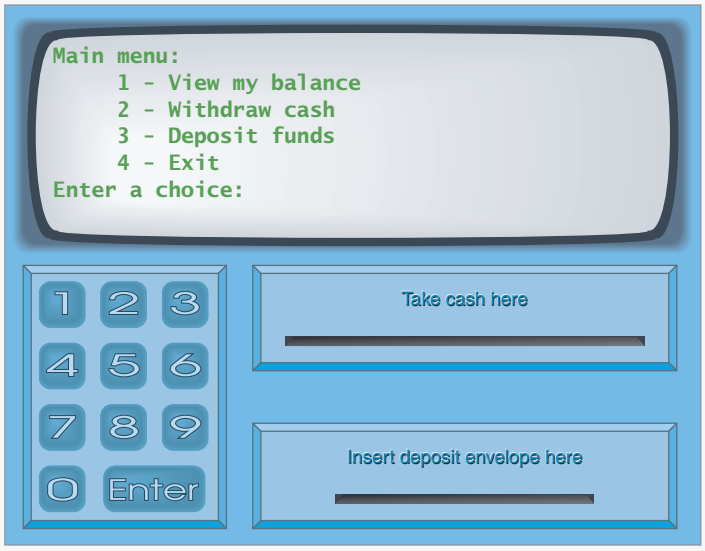

Fig. 2.16 | ATM main menu.

ISBN: 0-558-13856-X

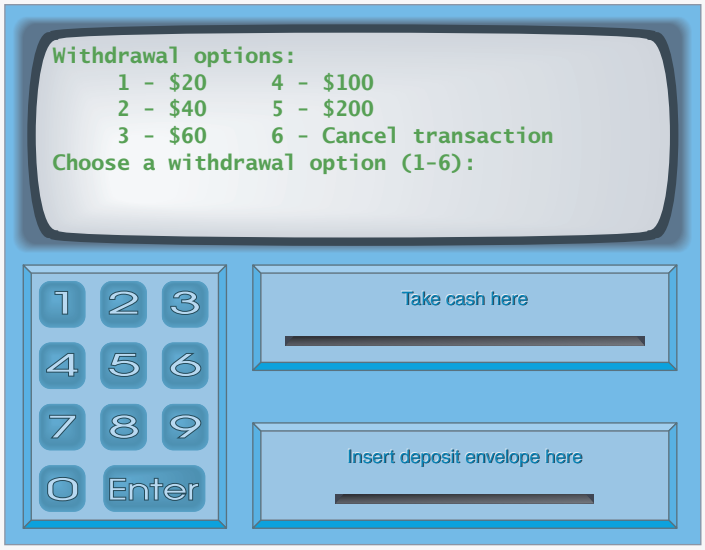

Fig. 2.17 | ATM withdrawal menu.

2. The user enters a menu selection (1–6) using the keypad.
3. If the withdrawal amount chosen is greater than the user's account balance, the screen displays a message stating this and telling the user to choose a smaller amount. The ATM then returns to *Step 1*. If the withdrawal amount chosen is less than or equal to the user's account balance (i.e., an acceptable withdrawal amount), the ATM proceeds to *Step 4*. If the user chooses to cancel the transaction (option 6), the ATM displays the main menu (Fig. 2.16) and waits for user input.
4. If the cash dispenser contains enough cash to satisfy the request, the ATM proceeds to *Step 5*. Otherwise, the screen displays a message indicating the problem and telling the user to choose a smaller withdrawal amount. The ATM then returns to *Step 1*.
5. The ATM debits (i.e., subtracts) the withdrawal amount from the user's account balance in the bank's database.
6. The cash dispenser dispenses the desired amount of money to the user.
7. The screen displays a message reminding the user to take the money.

The following actions occur when the user enters 3 (while the main menu is displayed) to make a deposit:

1. The screen prompts the user to enter a deposit amount or to type 0 (zero) to cancel the transaction.
2. The user enters a deposit amount or 0, using the keypad. [*Note:* The keypad does not contain a decimal point or a dollar sign, so the user cannot type a real dollar amount (e.g., $1.25). Instead, the user must enter a deposit amount as a number of cents (e.g., 125). The ATM then divides this number by 100 to obtain a number representing a dollar amount (e.g., 125 ÷ 100 = 1.25).]

ISBN: 0-558-13856-X

3. If the user specifies a deposit amount, the ATM proceeds to *Step 4*. If the user chooses to cancel the transaction (by entering 0), the ATM displays the main menu (Fig. 2.16) and waits for user input.
4. The screen displays a message telling the user to insert a deposit envelope into the deposit slot.
5. If the deposit slot receives a deposit envelope within two minutes, the ATM credits (i.e., adds) the deposit amount to the user's account balance in the bank's database. [*Note:* This money is not immediately available for withdrawal. The bank first must physically verify the amount of cash in the deposit envelope, and any checks in the envelope must clear (i.e., money must be transferred from the check writer's account to the check recipient's account). When either of these events occurs, the bank appropriately updates the user's balance stored in its database. This occurs independently of the ATM system.] If the deposit slot does not receive a deposit envelope within this time period, the screen displays a message that the system has canceled the transaction due to inactivity. The ATM then displays the main menu and waits for user input.

After the system successfully executes a transaction, the system should redisplay the main menu (Fig. 2.16) so that the user can perform additional transactions. If the user chooses to exit the system (option 4), the screen should display a thank you message, then display the welcome message for the next user.

Analyzing the ATM System

The preceding statement is a simplified example of a requirements specification. Typically, such a document is the result of a detailed process of **requirements gathering** that might include interviews with potential users of the system and specialists in fields related to the system. For example, a systems analyst who is hired to prepare a requirements specification for banking software (e.g., the ATM system described here) might interview financial experts to gain a better understanding of *what* the software must do. The analyst would use the information gained to compile a list of **system requirements** to guide systems designers.

The process of requirements gathering is a key task of the first stage of the software life cycle. The **software life cycle** specifies the stages through which software evolves from the time it is first conceived to the time it is retired from use. These stages typically include: analysis, design, implementation, testing and debugging, deployment, maintenance and retirement. Several software life-cycle models exist, each with its own preferences and specifications for when and how often software engineers should perform each of these stages. **Waterfall models** perform each stage once in succession, whereas **iterative models** may repeat one or more stages several times throughout a product's life cycle.

The analysis stage of the software life cycle focuses on defining the problem to be solved. When designing any system, one must certainly *solve the problem right*, but of equal importance, one must *solve the right problem*. Systems analysts collect the requirements that indicate the specific problem to solve. Our requirements specification describes our ATM system in sufficient detail that you do not need to go through an extensive analysis stage—it has been done for you.

To capture what a proposed system should do, developers often employ a technique known as **use case modeling**. This process identifies the **use cases** of the system, each of which represents a different capability that the system provides to its clients. For example,

ISBN: 0-558-13856-X

ATMs typically have several use cases, such as "View Account Balance," "Withdraw Cash," "Deposit Funds," "Transfer Funds Between Accounts" and "Buy Postage Stamps." The simplified ATM system we build in this case study allows only the first three of these use cases (Fig. 2.18).

Each use case describes a typical scenario in which the user uses the system. You have already read descriptions of the ATM system's use cases in the requirements specification; the lists of steps required to perform each type of transaction (i.e., balance inquiry, withdrawal and deposit) actually described the three use cases of our ATM—"View Account Balance," "Withdraw Cash" and "Deposit Funds."

Use Case Diagrams

We now introduce the first of several UML diagrams in our ATM case study. We create a **use case diagram** to model the interactions between a system's clients (in this case study, bank customers) and the system. The goal is to show the kinds of interactions users have with a system without providing the details—these are provided in other UML diagrams (which we present throughout the case study). Use case diagrams are often accompanied by informal text that describes the use cases in more detail—like the text that appears in the requirements specification. Use case diagrams are produced during the analysis stage of the software life cycle. In larger systems, use case diagrams are simple but indispensable tools that help system designers remain focused on satisfying the users' needs.

Figure 2.18 shows the use case diagram for our ATM system. The stick figure represents an **actor**, which defines the roles that an external entity—such as a person or another system—plays when interacting with the system. For our automated teller machine, the actor is a User who can view an account balance, withdraw cash and deposit funds from the ATM. The User is not an actual person, but instead comprises the roles that a real person—when playing the part of a User—can play while interacting with the ATM. Note that a use case diagram can include multiple actors. For example, the use case diagram for a real bank's ATM system might also include an actor named Administrator who refills the cash dispenser each day.

We identify the actor in our system by examining the requirements specification, which states, "ATM users should be able to view their account balance, withdraw cash and deposit funds." Therefore, the actor in each of the three use cases is the User who interacts with the ATM. An external entity—a real person—plays the part of the User to perform financial transactions. Figure 2.18 shows one actor, whose name, User, appears below the actor in the diagram. The UML models each use case as an oval connected to an actor with a solid line.

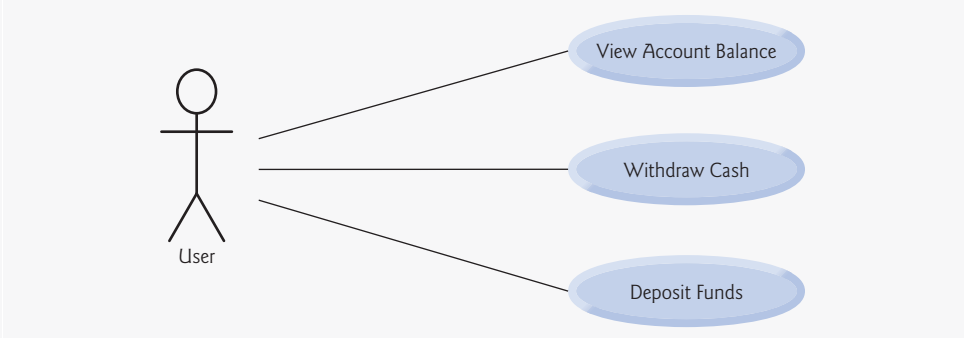

Fig. 2.18 | Use case diagram for the ATM system from the User's perspective.

ISBN: 0-558-13856-X

Software engineers (more precisely, systems analysts) must analyze the requirements specification or a set of use cases and design the system before programmers implement it. During the analysis stage, systems analysts focus on understanding the requirements specification to produce a high-level specification that describes *what* the system is supposed to do. The output of the design stage—a **design specification**—should specify clearly *how* the system should be constructed to satisfy these requirements. In the next several Software Engineering Case Study sections, we perform the steps of a simple object-oriented design (OOD) process on the ATM system to produce a design specification containing a collection of UML diagrams and supporting text. Recall that the UML is designed for use with any OOD process. Many such processes exist, the best known of which is the Rational Unified Process™ (RUP) developed by Rational Software Corporation (now a division of IBM). RUP is a rich process intended for designing "industrial strength" applications. For this case study, we present our own simplified design process.

Designing the ATM System

We now begin the design stage of our ATM system. A **system** is a set of components that interact to solve a problem. For example, to perform the ATM system's designated tasks, our ATM system has a user interface (Fig. 2.15), contains software that executes financial transactions and interacts with a database of bank account information. **System structure** describes the system's objects and their interrelationships. **System behavior** describes how the system changes as its objects interact with one another. Every system has both structure and behavior—designers must specify both. There are several distinct types of system structures and behaviors. For example, the interactions among objects in the system differ from those between the user and the system, yet both constitute a portion of the system behavior.

The UML 2 specifies 13 diagram types for documenting the models of systems. Each models a distinct characteristic of a system's structure or behavior—six diagrams relate to system structure; the remaining seven relate to system behavior. We list here only the six types of diagrams used in our case study—one of these (class diagrams) models system structure—the remaining five model system behavior. We overview the remaining seven UML diagram types in Appendix H, UML 2: Additional Diagram Types.

1. **Use case diagrams**, such as the one in Fig. 2.18, model the interactions between a system and its external entities (actors) in terms of use cases (system capabilities, such as "View Account Balance," "Withdraw Cash" and "Deposit Funds").
2. **Class diagrams**, which you'll study in Section 3.11, model the classes, or "building blocks," used in a system. Each noun or "thing" described in the requirements specification is a candidate to be a class in the system (e.g., "account," "keypad"). Class diagrams help us specify the structural relationships between parts of the system. For example, the ATM system class diagram will specify that the ATM is physically composed of a screen, a keypad, a cash dispenser and a deposit slot.
3. **State machine diagrams**, which you'll study in Section 3.11, model the ways in which an object changes state. An object's **state** is indicated by the values of all the object's attributes at a given time. When an object changes state, that object may behave differently in the system. For example, after validating a user's PIN, the ATM transitions from the "user not authenticated" state to the "user authenticated" state, at which point the ATM allows the user to perform financial transactions (e.g., view account balance, withdraw cash, deposit funds).

ISBN: 0-558-13856-X

4. **Activity diagrams**, which you'll also study in Section 5.11, model an object's **activity**—the object's workflow (sequence of events) during program execution. An activity diagram models the actions the object performs and specifies the order in which it performs these actions. For example, an activity diagram shows that the ATM must obtain the balance of the user's account (from the bank's account information database) before the screen can display the balance to the user.
5. **Communication diagrams** (called **collaboration diagrams** in earlier versions of the UML) model the interactions among objects in a system, with an emphasis on *what* interactions occur. You'll learn in Section 7.12 that these diagrams show which objects must interact to perform an ATM transaction. For example, the ATM must communicate with the bank's account information database to retrieve an account balance.
6. **Sequence diagrams** also model the interactions among the objects in a system, but unlike communication diagrams, they emphasize *when* interactions occur. You'll learn in Section 7.12 that these diagrams help show the order in which interactions occur in executing a financial transaction. For example, the screen prompts the user to enter a withdrawal amount before cash is dispensed.

In Section 3.11, we continue designing our ATM system by identifying the classes from the requirements specification. We accomplish this by extracting key nouns and noun phrases from the requirements specification. Using these classes, we develop our first draft of the class diagram that models the structure of our ATM system.

Internet and Web Resources

The following URLs provide information on object-oriented design with the UML.

`www-306.ibm.com/software/rational/uml/`
Lists frequently asked questions (FAQs) about the UML, provided by IBM Rational.

`www.douglass.co.uk/documents/softdocwiz.com.UML.htm`
Hosts the Unified Modeling Language Dictionary. Lists and defines all terms used in the UML.

`www-306.ibm.com/software/rational/offerings/design.html`
Provides information about IBM Rational software available for designing systems. Provides downloads of 30-day trial versions of several products, such as IBM Rational Rose® XDE Developer.

`www.embarcadero.com/products/describe/index.html`
Provides a free 14-day license to download a trial version of Describe™—a UML modeling tool from Embarcadero Technologies®.

`www.borland.com/us/products/together/index.html`
Provides a free 30-day license to download a trial version of Borland® Together® ControlCenter™—a software-development tool that supports the UML.

`www.ilogix.com/sublevel.aspx?id=53`
Provides a free 30-day license to download a trial version of I-Logix Rhapsody®—a UML 2 based model-driven development environment.

`argouml.tigris.org`
Contains information and downloads for ArgoUML, a free open-source UML tool written in Java.

`www.objectsbydesign.com/books/booklist.html`
Lists books on the UML and object-oriented design.

`www.objectsbydesign.com/tools/umltools_byCompany.html`
Lists software tools that use the UML, such as IBM Rational Rose, Embarcadero Describe, Sparx Systems Enterprise Architect, I-Logix Rhapsody and Gentleware Poseidon for UML.

ISBN: 0-558-13856-X

`www.ootips.org/ood-principles.html`
Provides answers to the question, "What Makes a Good Object-Oriented Design?"

`parlezuml.com/tutorials/umlforjava.htm`
Provides a UML tutorial for Java developers that presents UML diagrams side by side with the Java code that implements them.

`www.cetus-links.org/oo_uml.html`
Introduces the UML and provides links to numerous UML resources.

`www.agilemodeling.com/essays/umlDiagrams.htm`
Provides in-depth descriptions and tutorials on each of the 13 UML 2 diagram types.

Recommended Readings

The following books provide information on object-oriented design with the UML.

Booch, G. *Object-Oriented Analysis and Design with Applications*. 3rd ed. Boston: Addison-Wesley, 2004.

Eriksson, H., et al. *UML 2 Toolkit*. New York: John Wiley, 2003.

Fowler, M. *UML Distilled*. 3rd ed. Boston: Addison-Wesley Professional, 2004.

Kruchten, P. *The Rational Unified Process: An Introduction*. Boston: Addison-Wesley, 2004.

Larman, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. 2nd ed. Upper Saddle River, NJ: Prentice Hall, 2002.

Roques, P. *UML in Practice: The Art of Modeling Software Systems Demonstrated Through Worked Examples and Solutions*. New York: John Wiley, 2004.

Rosenberg, D., and K. Scott. *Applying Use Case Driven Object Modeling with UML: An Annotated e-Commerce Example*. Reading, MA: Addison-Wesley, 2001.

Rumbaugh, J., I. Jacobson and G. Booch. *The Complete UML Training Course*. Upper Saddle River, NJ: Prentice Hall, 2000.

Rumbaugh, J., I. Jacobson and G. Booch. *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley, 1999.

Rumbaugh, J., I. Jacobson and G. Booch. *The Unified Software Development Process*. Reading, MA: Addison-Wesley, 1999.

Schneider, G. and J. Winters. *Applying Use Cases: A Practical Guide.* 2nd ed. Boston: Addison-Wesley Professional, 2002.

Software Engineering Case Study Self-Review Exercises

2.1 Suppose we enabled a user of our ATM system to transfer money between two bank accounts. Modify the use case diagram of Fig. 2.18 to reflect this change.

2.2 _____ model the interactions among objects in a system with an emphasis on *when* these interactions occur.

a) Class diagrams
b) Sequence diagrams
c) Communication diagrams
d) Activity diagrams

2.3 Which of the following choices lists stages of a typical software life cycle in sequential order?

a) design, analysis, implementation, testing
b) design, analysis, testing, implementation
c) analysis, design, testing, implementation
d) analysis, design, implementation, testing

ISBN: 0-558-13856-X

Answers to Software Engineering Case Study Self-Review Exercises

2.1 Figure 2.19 shows a use case diagram for a modified version of our ATM system that also allows users to transfer money between accounts.

2.2 b.

2.3 d.

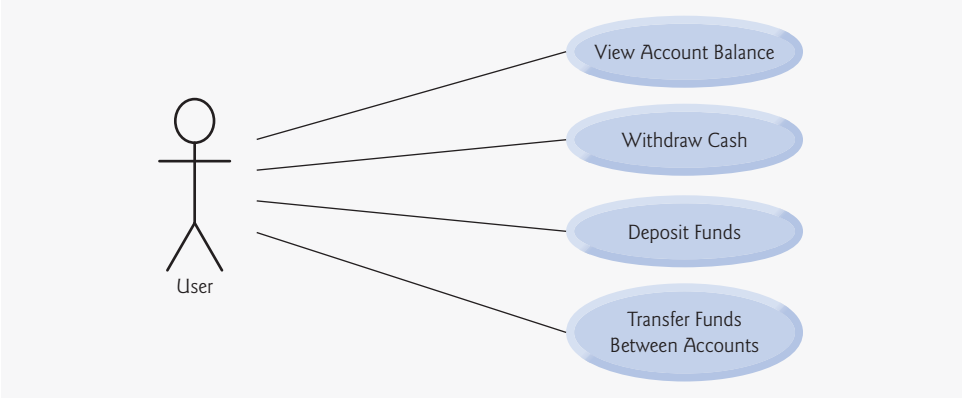

Fig. 2.19 | Use case diagram for a modified version of our ATM system that also allows users to transfer money between accounts.

2.9 Wrap-Up

You learned many important basic features of C++ in this chapter, including displaying data on the screen, inputting data from the keyboard and declaring variables of fundamental types. In particular, you learned to use the output stream object `cout` and the input stream object `cin` to build simple interactive programs. We explained how variables are stored in and retrieved from memory. You also learned how to use arithmetic operators to perform calculations. We discussed the order in which C++ applies operators (i.e., the rules of operator precedence), as well as the associativity of the operators. You also learned how C++'s `if` statement allows a program to make decisions. Finally, we introduced the equality and relational operators, which you use to form conditions in `if` statements.

The non-object-oriented applications presented here introduced you to basic programming concepts. As you'll see in Chapter 3, C++ applications typically contain just a few lines of code in function `main`—these statements normally create the objects that perform the work of the application, then the objects "take over from there." In Chapter 3, you'll learn how to implement your own classes and use objects of those classes in applications.

Summary

Section 2.2 First Program in C++: Printing a Line of Text

- Single-line comments begin with `//`. You insert comments to document your programs and improve their readability.
- Comments do not cause the computer to perform any action when the program is run—they are ignored by the C++ compiler and do not cause any machine-language object code to be generated.

ISBN: 0-558-13856-X

- A preprocessor directive begins with # and is a message to the C++ preprocessor. Preprocessor directives are processed by the preprocessor before the program is compiled and don't end with a semicolon as C++ statements do.
- The line `#include <iostream>` tells the C++ preprocessor to include the contents of the input/output stream header file in the program. This file contains information necessary to compile programs that use `std::cin` and `std::cout` and operators << and >>.
- White space (i.e., blank lines, space characters and tab characters) makes programs easier to read. White-space characters outside of literals are ignored by the compiler.
- C++ programs begin executing at `main`, even if `main` does not appear first in the program.
- The keyword `int` to the left of `main` indicates that `main` "returns" an integer value.
- A left brace, `{`, must begin the body of every function. A corresponding right brace, `}`, must end each function's body.
- A string in double quotes is sometimes referred to as a character string, message or string literal. White-space characters in strings are not ignored by the compiler.
- Every statement must end with a semicolon (also known as the statement terminator).
- Output and input in C++ are accomplished with streams of characters.
- The output stream object `std::cout`—normally connected to the screen—is used to output data. Multiple data items can be output by concatenating stream insertion (<<) operators.
- The input stream object `std::cin`—normally connected to the keyboard—is used to input data. Multiple data items can be input by concatenating stream extraction (>>) operators.
- The `std::cout` and `std::cin` stream objects facilitate interaction between the user and the computer. Because this interaction resembles a dialog, it is often called conversational computing or interactive computing.
- The notation `std::cout` specifies that we are using a name, in this case `cout`, that belongs to "namespace" `std`.
- When a backslash (i.e., an escape character) is encountered in a string of characters, the next character is combined with the backslash to form an escape sequence.
- The escape sequence `\n` means newline. It causes the cursor (i.e., the current screen-position indicator) to move to the beginning of the next line on the screen.
- A message that directs the user to take a specific action is known as a prompt.
- C++ keyword `return` is one of several means to exit a function.

Section 2.4 Another C++ Program: Adding Integers

- All variables in a C++ program must be declared before they can be used.
- A variable name in C++ is any valid identifier that is not a keyword. An identifier is a series of characters consisting of letters, digits and underscores (_). Identifiers cannot start with a digit. C++ identifiers can be any length; however, some systems and/or C++ implementations may impose some restrictions on the length of identifiers.
- C++ is case sensitive.
- Most calculations are performed in assignment statements.
- A variable is a location in the computer's memory where a value can be stored for use by a program.
- Variables of type `int` hold integer values, i.e., whole numbers such as 7, –11, 0, 31914.

Section 2.5 Memory Concepts

- Every variable stored in the computer's memory has a name, a value, a type and a size.

ISBN: 0-558-13856-X

- Whenever a new value is placed in a memory location, the process is destructive; i.e., the new value replaces the previous value in that location. The previous value is lost.
- When a value is read from memory, the process is nondestructive; i.e., a copy of the value is read, leaving the original value undisturbed in the memory location.
- The `std::endl` stream manipulator outputs a newline, then "flushes the output buffer."

Section 2.6 Arithmetic

- C++ evaluates arithmetic expressions in a precise sequence determined by the rules of operator precedence and associativity.
- Parentheses may be used to group expressions.
- Integer division (i.e., both the numerator and the denominator are integers) yields an integer quotient. Any fractional part in integer division is truncated—no rounding occurs.
- The modulus operator, %, yields the remainder after integer division. The modulus operator can be used only with integer operands.

Section 2.7 Decision Making: Equality and Relational Operators

- The `if` statement allows a program to take alternative action based on whether a condition is met. The format for an `if` statement is

```
if ( condition )
   statement;
```

  If the condition is true, the statement in the body of the `if` is executed. If the condition is not met, i.e., the condition is false, the body statement is skipped.
- Conditions in `if` statements are commonly formed by using equality operators and relational operators. The result of using these operators is always the value true or false.
- The declaration

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

  is a `using` declaration that informs the compiler where to find `cout` (namespace `std`) and eliminates the need to repeat the `std::` prefix. Once we include this `using` declaration, we can, for example, write `cout` instead of `std::cout` in the remainder of a program.

Terminology

/* … */ comment (C-style comment)
// comment
arithmetic operator
assignment operator (=)
associativity of operators
binary operator
block
body of a function
cascading stream insertion operations
case sensitive
chaining stream insertion operations
character string
`cin` object
comma-separated list
comment (//)
compilation error
compiler error
compile-time error
compound statement
concatenating stream insertion operations
condition
`cout` object
cursor
data type
decision
declaration
destructive write
equality operators
  == "is equal to"
  != "is not equal to"
escape character (\)
escape sequence

ISBN: 0-558-13856-X

exit a function
fatal error
function
identifier
`if` statement
input/output stream header file `<iostream>`
`int` data type
integer (`int`)
integer division
left-to-right associativity
literal
logic error
`main` function
memory
memory location
message
modulus operator (`%`)
multiplication operator (`*`)
nested parentheses
newline character (`\n`)
nondestructive read
nonfatal logic error
operand
operator
operator associativity
parentheses `()`
perform an action
precedence
preprocessor directive
prompt
redundant parentheses
relational operators
  `<` "is less than"
  `<=` "is less than or equal to"
  `>` "is greater than"
  `>=` "is greater than or equal to"
`return` statement
rules of operator precedence
self-documenting program
semicolon (`;`) statement terminator
standard input stream object (`cin`)
standard output stream object (`cout`)
statement
statement terminator (`;`)
stream
stream extraction operator (`>>`)
stream insertion operator (`<<`)
stream manipulator
string
string literal
syntax error
`using` declaration
variable
white space

Self-Review Exercises

2.1 Fill in the blanks in each of the following.

a) Every C++ program begins execution at the function _____.
b) A _____ begins the body of every function and a _____ ends the body.
c) Every C++ statement ends with a(n) _____.
d) The escape sequence `\n` represents the _____ character, which causes the cursor to position to the beginning of the next line on the screen.
e) The _____ statement is used to make decisions.

2.2 State whether each of the following is *true* or *false*. If *false*, explain why. Assume the statement `using std::cout;` is used.

a) Comments cause the computer to print the text after the `//` on the screen when the program is executed.
b) The escape sequence `\n`, when output with `cout` and the stream insertion operator, causes the cursor to position to the beginning of the next line on the screen.
c) All variables must be declared before they are used.
d) All variables must be given a type when they are declared.
e) C++ considers the variables `number` and `NuMbEr` to be identical.
f) Declarations can appear almost anywhere in the body of a C++ function.
g) The modulus operator (`%`) can be used only with integer operands.
h) The arithmetic operators `*`, `/`, `%`, `+` and `–` all have the same level of precedence.
i) A C++ program that prints three lines of output must contain three statements using `cout` and the stream insertion operator.

ISBN: 0-558-13856-X

2.3 Write a single C++ statement to accomplish each of the following (assume that `using` declarations have not been used):

a) Declare the variables `c`, `thisIsAVariable`, `q76354` and `number` to be of type `int`.
b) Prompt the user to enter an integer. End your prompting message with a colon (`:`) followed by a space and leave the cursor positioned after the space.
c) Read an integer from the user at the keyboard and store it in integer variable `age`.
d) If the variable `number` is not equal to `7`, print `"The variable number is not equal to 7"`.
e) Print the message `"This is a C++ program"` on one line.
f) Print the message `"This is a C++ program"` on two lines. End the first line with `C++`.
g) Print the message `"This is a C++ program"` with each word on a separate line.
h) Print the message `"This is a C++ program"`. Separate each word from the next by a tab.

2.4 Write a statement (or comment) to accomplish each of the following (assume that `using` declarations have been used for `cin`, `cout` and `endl`):

a) State that a program calculates the product of three integers.
b) Declare the variables `x`, `y`, `z` and `result` to be of type `int` (in separate statements).
c) Prompt the user to enter three integers.
d) Read three integers from the keyboard and store them in the variables `x`, `y` and `z`.
e) Compute the product of the three integers contained in variables `x`, `y` and `z`, and assign the result to the variable `result`.
f) Print `"The product is "` followed by the value of the variable `result`.
g) Return a value from `main` indicating that the program terminated successfully.

2.5 Using the statements you wrote in Exercise 2.4, write a complete program that calculates and displays the product of three integers. Add comments to the code where appropriate. [*Note:* You'll need to write the necessary `using` declarations.]

2.6 Identify and correct the errors in each of the following statements (assume that the statement `using std::cout;` is used):

a)
```
if ( c < 7 );
   cout << "c is less than 7\n";
```
b)
```
if ( c => 7 )
   cout << "c is equal to or greater than 7\n";
```

Answers to Self-Review Exercises

2.1 a) `main`. b) left brace ({), right brace (}). c) semicolon. d) newline. e) `if`.

2.2 a) False. Comments do not cause any action to be performed when the program is executed. They are used to document programs and improve their readability.
b) True.
c) True.
d) True.
e) False. C++ is case sensitive, so these variables are unique.
f) True.
g) True.
h) False. The operators `*`, `/` and `%` have the same precedence, and the operators `+` and `-` have a lower precedence.
i) False. One statement with `cout` and multiple `\n` escape sequences can print several lines.

2.3 a)
```
int c, thisIsAVariable, q76354, number;
```
b)
```
std::cout << "Enter an integer: ";
```
c)
```
std::cin >> age;
```
d)
```
if ( number != 7 )
   std::cout << "The variable number is not equal to 7\n";
```

ISBN: 0-558-13856-X

e) `std::cout << "This is a C++ program\n";`
f) `std::cout << "This is a C++\nprogram\n";`
g) `std::cout << "This\nis\na\nC++\nprogram\n";`
h) `std::cout << "This\tis\ta\tC++\tprogram\n";`

2.4 a) `// Calculate the product of three integers`
b)
```
int x;
int y;
int z;
int result;
```
c) `cout << "Enter three integers: ";`
d) `cin >> x >> y >> z;`
e) `result = x * y * z;`
f) `cout << "The product is " << result << endl;`
g) `return 0;`

2.5 (See program below.)

```
1  // Calculate the product of three integers
2  #include <iostream> // allows program to perform input and output
3
4  using std::cout; // program uses cout
5  using std::cin; // program uses cin
6  using std::endl; // program uses endl
7
8  // function main begins program execution
9  int main()
10 {
11    int x; // first integer to multiply
12    int y; // second integer to multiply
13    int z; // third integer to multiply
14    int result; // the product of the three integers
15
16    cout << "Enter three integers: "; // prompt user for data
17    cin >> x >> y >> z; // read three integers from user
18    result = x * y * z; // multiply the three integers; store result
19    cout << "The product is " << result << endl; // print result; end line
20
21    return 0; // indicate program executed successfully
22 } // end function main
```

2.6 a) *Error:* Semicolon after the right parenthesis of the condition in the `if` statement.
Correction: Remove the semicolon after the right parenthesis. [*Note:* The result of this error is that the output statement will be executed whether or not the condition in the `if` statement is true.] The semicolon after the right parenthesis is a null (or empty) statement—a statement that does nothing. We'll learn more about the null statement in the next chapter.
b) *Error:* The relational operator =>.
Correction: Change => to >=, and you may want to change "equal to or greater than" to "greater than or equal to" as well.

Exercises

2.7 Discuss the meaning of each of the following objects:
a) `std::cin`
b) `std::cout`

ISBN: 0-558-13856-X

2.8 Fill in the blanks in each of the following:
a) _____ are used to document a program and improve its readability.
b) The object used to print information on the screen is _____.
c) A C++ statement that makes a decision is _____.
d) Most calculations are normally performed by _____ statements.
e) The _____ object inputs values from the keyboard.

2.9 Write a single C++ statement or line that accomplishes each of the following:
a) Print the message `"Enter two numbers"`.
b) Assign the product of variables `b` and `c` to variable `a`.
c) State that a program performs a payroll calculation (i.e., use text that helps to document a program).
d) Input three integer values from the keyboard into integer variables `a`, `b` and `c`.

2.10 State which of the following are *true* and which are *false*. If *false*, explain your answers.
a) C++ operators are evaluated from left to right.
b) The following are all valid variable names: `_under_bar_`, `m928134`, `t5`, `j7`, `her_sales`, `his_account_total`, `a`, `b`, `c`, `z`, `z2`.
c) The statement `cout << "a = 5;";` is a typical example of an assignment statement.
d) A valid C++ arithmetic expression with no parentheses is evaluated from left to right.
e) The following are all invalid variable names: `3g`, `87`, `67h2`, `h22`, `2h`.

2.11 Fill in the blanks in each of the following:
a) What arithmetic operations are on the same level of precedence as multiplication? _____.
b) When parentheses are nested, which set of parentheses is evaluated first in an arithmetic expression? _____.
c) A location in the computer's memory that may contain different values at various times throughout the execution of a program is called a _____.

2.12 What, if anything, prints when each of the following C++ statements is performed? If nothing prints, then answer "nothing." Assume `x = 2` and `y = 3`.

a)
```
cout << x;
```
b)
```
cout << x + x;
```
c)
```
cout << "x=";
```
d)
```
cout << "x = " << x;
```
e)
```
cout << x + y << " = " << y + x;
```
f)
```
z = x + y;
```
g)
```
cin >> x >> y;
```
h)
```
// cout << "x + y = " << x + y;
```
i)
```
cout << "\n";
```

2.13 Which of the following C++ statements contain variables whose values are replaced?

a)
```
cin >> b >> c >> d >> e >> f;
```
b)
```
p = i + j + k + 7;
```
c)
```
cout << "variables whose values are replaced";
```
d)
```
cout << "a = 5";
```

2.14 Given the algebraic equation $y = ax^3 + 7$, which of the following, if any, are correct C++ statements for this equation?

a)
```
y = a * x * x * x + 7;
```
b)
```
y = a * x * x * ( x + 7 );
```
c)
```
y = ( a * x ) * x * ( x + 7 );
```
d)
```
y = (a * x) * x * x + 7;
```

ISBN: 0-558-13856-X

e) `y = a * ( x * x * x ) + 7;`
f) `y = a * x * ( x * x + 7 );`

2.15 State the order of evaluation of the operators in each of the following C++ statements and show the value of x after each statement is performed.

a) `x = 7 + 3 * 6 / 2 - 1;`
b) `x = 2 % 2 + 2 * 2 - 2 / 2;`
c) `x = ( 3 * 9 * ( 3 + ( 9 * 3 / ( 3 ) ) ) );`

2.16 Write a program that asks the user to enter two numbers, obtains the two numbers from the user and prints the sum, product, difference, and quotient of the two numbers.

2.17 Write a program that prints the numbers 1 to 4 on the same line with each pair of adjacent numbers separated by one space. Do this several ways:

a) Using one statement with one stream insertion operator.
b) Using one statement with four stream insertion operators.
c) Using four statements.

2.18 Write a program that asks the user to enter two integers, obtains the numbers from the user, then prints the larger number followed by the words "`is larger.`" If the numbers are equal, print the message "`These numbers are equal.`"

2.19 Write a program that inputs three integers from the keyboard and prints the sum, average, product, smallest and largest of these numbers. The screen dialog should appear as follows:

```
Input three different integers: 13 27 14
Sum is 54
Average is 18
Product is 4914
Smallest is 13
Largest is 27
```

2.20 Write a program that reads in the radius of a circle as an integer and prints the circle's diameter, circumference and area. Use the constant value 3.14159 for π. Do all calculations in output statements. [*Note:* In this chapter, we have discussed only integer constants and variables. In Chapter 4 we discuss floating-point numbers, i.e., values that can have decimal points.]

2.21 Write a program that prints a box, an oval, an arrow and a diamond as follows:

```
*********          ***             *               *
*       *        *     *          ***             * *
*       *       *       *        *****           *   *
*       *       *       *          *            *     *
*       *       *       *          *           *       *
*       *       *       *          *            *     *
*       *       *       *          *             *   *
*       *        *     *           *              * *
*********          ***             *               *
```

2.22 What does the following code print?

```
cout << "*\n**\n***\n****\n*****" << endl;
```

2.23 Write a program that reads in five integers and determines and prints the largest and the smallest integers in the group. Use only the programming techniques you learned in this chapter.

2.24 Write a program that reads an integer and determines and prints whether it is odd or even. [*Hint:* Use the modulus operator. An even number is a multiple of two. Any multiple of two leaves a remainder of zero when divided by 2.]

ISBN: 0-558-13856-X

2.25 Write a program that reads in two integers and determines and prints if the first is a multiple of the second. [*Hint:* Use the modulus operator.]

2.26 Display the following checkerboard pattern with eight output statements, then display the same pattern using as few statements as possible.

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

2.27 Here is a peek ahead. In this chapter you learned about integers and the type `int`. C++ can also represent uppercase letters, lowercase letters and a considerable variety of special symbols. C++ uses small integers internally to represent each different character. The set of characters a computer uses and the corresponding integer representations for those characters are called that computer's **character set**. You can print a character by enclosing that character in single quotes, as with

```
cout << 'A'; // print an uppercase A
```

You can print the integer equivalent of a character using `static_cast` as follows:

```
cout << static_cast< int >( 'A' ); // print 'A' as an integer
```

This is called a **cast** operation (we formally introduce casts in Chapter 4). When the preceding statement executes, it prints the value 65 (on systems that use the **ASCII character set**). Write a program that prints the integer equivalent of a character typed at the keyboard. Store the input in a variable of type `char`. Test your program several times using uppercase letters, lowercase letters, digits and special characters (like $).

2.28 Write a program that inputs a five-digit integer, separates the integer into its individual digits and prints the digits separated from one another by three spaces each. [*Hint:* Use the integer division and modulus operators.] For example, if the user types in 42339, the program should print:

```
4   2   3   3   9
```

2.29 Using only the techniques you learned in this chapter, write a program that calculates the squares and cubes of the integers from 0 to 10 and uses tabs to print the following neatly formatted table of values:

```
integer square  cube
0       0       0
1       1       1
2       4       8
3       9       27
4       16      64
5       25      125
6       36      216
7       49      343
8       64      512
9       81      729
10      100     1000
```

ISBN: 0-558-13856-X