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.
Chapter 2   Introduction to C++ Programming

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.

2.1 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.

```cpp
// Fig. 2.1: fig02_01.cpp
// Text-printing program.
#include <iostream> // allows program to output data to the screen

// function main begins program execution
int main()
{
    std::cout << "Welcome to C++!\n"; // display message
    return 0; // indicate that program ended successfully
} // end function main

Welcome to C++!
```

Fig. 2.1   |  Text-printing program.
Lines 1 and 2

```cpp
// 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’re 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 `*/`.]

**Good Programming Practice 2.1**

Every program should begin with a comment that describes the purpose of the program.

Line 3

```cpp
#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 1.14). 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.

Line 5

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

is another single-line comment indicating that program execution begins at the next line.
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 named 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 3.4 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 11) must end each function’s body. Line 8

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

instructions 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 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++!
" 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.

The std:: before cout 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 24, 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. 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.

<table>
<thead>
<tr>
<th>Escape sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>Newline. Position the screen cursor to the beginning of the next line.</td>
</tr>
<tr>
<td>\t</td>
<td>Horizontal tab. Move the screen cursor to the next tab stop.</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return. Position the screen cursor to the beginning of the current line; do not advance to the next line.</td>
</tr>
<tr>
<td>\a</td>
<td>Alert. Sound the system bell.</td>
</tr>
<tr>
<td>&quot;</td>
<td>Backslash. Used to print a backslash character.</td>
</tr>
<tr>
<td>'</td>
<td>Single quote. Use to print a single quote character.</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double quote. Used to print a double quote character.</td>
</tr>
</tbody>
</table>

**Fig. 2.2** | Escape sequences.

**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.

**Good Programming Practice 2.3**

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.


Chapter 2  Introduction to C++ Programming

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 yellow 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 because this string did not end with \n, 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.

```
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 } // end function main
```

Welcome to C++!

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

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. 2.4: fig02_04.cpp
2 // Printing multiple lines of text with a single statement.
3 #include <iostream> // allows program to output data to the screen
```

Fig. 2.4  |  Printing multiple lines of text with a single statement. (Part 1 of 2.)

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. In the output window, we highlight the user’s input in bold.

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

int main()
{
    int number1; // first integer to add
    int number2; // second integer to add
    int sum; // sum of number1 and number2

    std::cout << "Enter first integer: "; // prompt user for data
    std::cin >> number1; // read first integer from user into number1

    std::cout << "Enter second integer: "; // prompt user for data
    std::cin >> number2; // read second integer from user into number2

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

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

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.

The comments in lines 1 and 2 state the name of the file and the purpose of the program. The C++ preprocessor directive 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) begins main’s body and the corresponding right brace (line 22) ends it.

Lines 9–11 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 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 2.5
Place a space after each comma (,) to make programs more readable.

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, double and char are called fundamental 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—upercase 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 restrict identifier lengths. Use identifiers of 31 characters or fewer to ensure portability.
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
```

do 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
```

**Good Programming Practice 2.9**

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.
displays Enter first integer: followed by a space. 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

```cpp
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’s often called conversational computing or interactive computing.

Line 16

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

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

```cpp
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

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

adds the values of 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 2.10**

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

Line 21

```cpp
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 end1 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.

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’s 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

```cpp
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

```cpp
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++ 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.

![Fig. 2.6](image)

Memory location showing the name and value of variable number1.

When 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

```cpp
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. These locations are not necessarily adjacent in memory.

![Fig. 2.7](image-url) Memory locations after storing values for number1 and number2.

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. 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. 2.8](image-url) Memory locations after calculating and storing the sum of number1 and number2.

### 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. Any fractional part in integer division is discarded (i.e., truncated)—no rounding occurs.

2.6 Arithmetic

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).

**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 \( a \times (b + c) \).

**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

\[
( a \times ( b + c ) )
\]

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, oper-
ators 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.

<table>
<thead>
<tr>
<th>Operator(s)</th>
<th>Operation(s)</th>
<th>Order of evaluation (precedence)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>Parentheses</td>
<td>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’re evaluated left to right.</td>
</tr>
<tr>
<td>* , , / , %</td>
<td>Multiplication, Division, Modulus</td>
<td>Evaluated second. If there are several, they’re evaluated left to right.</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
<td>Evaluated last. If there are several, they’re evaluated left to right.</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 2.10** | Precedence of arithmetic operators.

*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: \( m = \frac{a + b + c + d + e}{5} \)

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:

\[
\begin{aligned}
\text{Algebra:} & \quad z = pr \% q + w/x - y \\
\text{C++:} & \quad z = p * r \% q + w / x - y;
\end{aligned}
\]

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 because its precedence is lower than that of any of the arithmetic operators.

**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 \(y = ax^2 + bx + c\):

\[
\begin{aligned}
\text{y = a} & \ast x \ast x + b \ast x + c;
\end{aligned}
\]

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’ve represented \(x^2\) as \(x \ast x\). 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.

**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’s 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:

\[
y = (a \ast x \ast x) + (b \ast x) + c;
\]

**Good Programming Practice 2.11**

Using redundant parentheses in complex arithmetic expressions can make the expressions clearer.
Chapter 2  Introduction to C++ Programming

We now introduce a simple version of C++’s if statement that allows a program to take alternative action based on whether a condition is true or false. If the condition is true, the statement in the body of the if statement is executed. If the condition is false, the body statement is not executed. We’ll see an example shortly.

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.

2.7 Decision Making: Equality and Relational Operators

We now introduce a simple version of C++’s if statement that allows a program to take alternative action based on whether a condition is true or false. If the condition is true, the statement in the body of the if statement is executed. If the condition is false, the body statement is not executed. We’ll see an example shortly.

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.

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

### 2.7 Decision Making: Equality and Relational Operators

We now introduce a simple version of C++’s if statement that allows a program to take alternative action based on whether a condition is true or false. If the condition is true, the statement in the body of the if statement is executed. If the condition is false, the body statement is not executed. We’ll see an example shortly.

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.
2.7 Decision Making: Equality and Relational Operators

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.

<table>
<thead>
<tr>
<th>Standard algebraic equality or relational operator</th>
<th>C++ equality or relational operator</th>
<th>Sample C++ condition</th>
<th>Meaning of C++ condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relational operators</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;</td>
<td><code>x &gt; y</code></td>
<td><code>x</code> is greater than <code>y</code></td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;</td>
<td><code>x &lt; y</code></td>
<td><code>x</code> is less than <code>y</code></td>
</tr>
<tr>
<td>≥</td>
<td>≥</td>
<td><code>x &gt;= y</code></td>
<td><code>x</code> is greater than or equal to <code>y</code></td>
</tr>
<tr>
<td>≤</td>
<td>≤</td>
<td><code>x &lt;= y</code></td>
<td><code>x</code> is less than or equal to <code>y</code></td>
</tr>
<tr>
<td>Equality operators</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>==</td>
<td><code>x == y</code></td>
<td><code>x</code> is equal to <code>y</code></td>
</tr>
<tr>
<td>≠</td>
<td>!=</td>
<td><code>x != y</code></td>
<td><code>x</code> is not equal to <code>y</code></td>
</tr>
</tbody>
</table>

**Fig. 2.12** Equality and relational operators.

**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.

```cpp
// Fig. 2.13: fig02_13.cpp
// Comparing integers using if statements, relational operators and equality operators.
#include <iostream> // allows program to perform input and output
using std::cout; // program uses cout
using std::cin; // program uses cin
using std::endl; // program uses endl

// function main begins program execution
int main()
{
    int number1; // first integer to compare
    int number2; // second integer to compare
    cout << "Enter two integers to compare: "; // prompt user for data
    cin >> number1 >> number2; // read two integers from user
```

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

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.
In place of lines 6–8, many programmers prefer to use the declaration

```cpp
using namespace std;
```

which enables a program to use all the names in any standard C++ header file (such as `<iostream>`) that a program might include. From this point forward in the book, we’ll use the preceding declaration in our programs.

Lines 13–14

```cpp
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

```cpp
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.

Each `if` statement in Fig. 2.13 has a single statement in its body and 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 2.12**

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

**Good Programming Practice 2.13**

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’s a syntax error to split identifiers, strings (such as "hello") and constants (such as the number 1000) over several lines.
Chapter 2 Introduction to C++ Programming

Common Programming Error 2.9
It’s a syntax error to split an identifier by inserting white-space characters (e.g., writing main as ma in).

Good Programming Practice 2.14
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. 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 \(x\).

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>left to right</td>
<td>parentheses</td>
</tr>
<tr>
<td>* / %</td>
<td>left to right</td>
<td>multiplicative</td>
</tr>
<tr>
<td>+ -</td>
<td>left to right</td>
<td>additive</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>left to right</td>
<td>stream insertion/extraction</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>left to right</td>
<td>relational</td>
</tr>
<tr>
<td>== !=</td>
<td>left to right</td>
<td>equality</td>
</tr>
<tr>
<td>=</td>
<td>right to left</td>
<td>assignment</td>
</tr>
</tbody>
</table>

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

Good Programming Practice 2.15
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’d 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 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're ignored by the compiler and do not cause any machine-language object code to be generated.
- A preprocessor directive begins with # and is a message to the C++ preprocessor. Preprocessor directives are processed before the program is compiled and don’t end with a semicolon.
- 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 the stream insertion (<<) and stream extraction (>>) operators.
- 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’s often called conversational computing or interactive computing.
- The notation std::cout specifies that we are using cout from “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 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.

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

```cpp
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

```cpp
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. The declaration

```cpp
using namespace std;
```

enables the program to use all the names in any included standard library header file.
Terminology

// comment 41
<< operator 42
arithmetic operator 50
assignment operator (=) 48
associativity of operators 52
asterisk (*) 50
binary operator 48
block 57
body of a function 42
cascading stream insertion operations 49
case sensitive 46
chaining stream insertion operations 49
character string 42
comma-separated list 46
comment (//) 41
compilation error 43
compile-time error 43
compiler error 43
compound statement 57
concatenating stream insertion operations 49
condition 54
conversational computing 48
cursor 43
declaration 46
destructive write 49
equality operators 54
    == “is equal to” 55
        != “is not equal to” 55
embedded parentheses 51
escape character (\) 42
escape sequence 42
exit a function 43
fatal logic error 54
function 42
fundamental type 46
identifier 46
if statement 54
input stream object (cin) 45
input/output stream header file <iostream> 41
int data type 46
integer 46
integer division 50
interactive computing 48
keyword 42
left brace ( {) 42
location in memory 49
logic error 54
main function 42
modulus operator (%) 50
multiplication operator (*) 50
nested parentheses 51
newline character (\n) 42
nondestructive read 50
nonfatal logic error 54
operand 42
operator overloading 49
percent sign (%) 50
perform an action 42
preprocessor directive 41
prompt 48
redundant parentheses 53
relational operators 54
    < “is less than” 55
        <= “is less than or equal to” 55
> “is greater than” 55
    >= “is greater than or equal to” 55
return statement 43
right brace (}) 42
rules of operator precedence 51
self-documenting program 47
semicolon (;) statement terminator 42
single-line comment 41
standard input stream object (cin) 48
standard output stream object (cout) 42
statement 42
std::cin 45
straight-line form 51
stream 42
stream extraction operator (>>) 45
stream insertion operator (<<) 42
stream manipulator 49
string 42
string literal 42
syntax 43
syntax error 43
using declaration 56
truncated 50
value 48
variable 46
white space 41

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’re used.
d) All variables must be given a type when they’re 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.

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
";
b) if ( c => 7 )
   cout << "c is equal to or greater than 7";
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’re 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";
   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.)

```cpp
// Calculate the product of three integers
#include <iostream> // allows program to perform input and output
using namespace std; // program uses names from the std namespace

// function main begins program execution
int main()
{
  int x; // first integer to multiply
  int y; // second integer to multiply
  int z; // third integer to multiply
  int result; // the product of the three integers

  cout << "Enter three integers: "; // prompt user for data
  cin >> x >> y >> z; // read three integers from user
  result = x * y * z; // multiply the three integers; store result
  cout << "The product is " << result << endl; // print result; end line
}
```
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 executes whether or not the condition in the `if` statement is true.] The semicolon after the right parenthesis is a null (or empty) statement that does nothing. We'll learn more about the null statement in Chapter 4.

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`

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) \( \texttt{cout} << "x + y = " << x + y; \)

i) \( \texttt{cout} << "\n"; \)

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

a) \( \texttt{cin} >> \texttt{b} >> \texttt{c} >> \texttt{d} >> \texttt{e} >> \texttt{f}; \)

b) \( \texttt{p} = \texttt{i} + \texttt{j} + \texttt{k} + 7; \)

c) \( \texttt{cout} << "\text{variables whose values are replaced}"; \)

d) \( \texttt{cout} << "\text{a} = 5"; \)

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

a) \( y = \texttt{a} * \texttt{x} * \texttt{x} * \texttt{x} + 7; \)

b) \( y = \texttt{a} * \texttt{x} * \texttt{x} * ( \texttt{x} + 7 ); \)

c) \( y = ( \texttt{a} * \texttt{x} ) * \texttt{x} * ( \texttt{x} + 7 ); \)

d) \( y = (\texttt{a} * \texttt{x}) * \texttt{x} * \texttt{x} + 7; \)

e) \( y = \texttt{a} * ( \texttt{x} * \texttt{x} * \texttt{x} ) + 7; \)

f) \( y = \texttt{a} * ( \texttt{x} * \texttt{x} * \texttt{x} ) * \texttt{x} + 7; \)

**2.15** *(Order of Evaluation)* 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** *(Arithmetic)* 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** *(Printing)* 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** *(Comparing Integers)* 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** *(Arithmetic, Smallest and Largest)* 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** *(Diameter, Circumference and Area of a Circle)* 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 \( \pi \). Do all calculations in output statements. *[Note: In this chapter, we’ve discussed only integer constants and variables. In Chapter 4 we discuss floating-point numbers, i.e., values that can have decimal points.]*

**2.21** *(Displaying Shapes with Asterisks)* Write a program that prints a box, an oval, an arrow and a diamond as follows:
2.22 What does the following code print?
```cpp
cout << "**
***
****
*****" << endl;
```

2.23 (Largest and Smallest Integers) 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 (Odd or Even) Write a program that reads an integer and determines and prints whether it’s 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.]

2.25 (Multiples) 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 (Checkerboard Pattern) Display the following checkerboard pattern with eight output statements, then display the same pattern using as few statements as possible.

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

2.27 (Integer Equivalent of a Character) 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

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

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

```cpp
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 (Digits of an Integer) Write a program that inputs a five-digit integer, separates the integer into its digits and prints them separated 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 \((Table)\) Using the techniques of this chapter, write a program that calculates the squares and cubes of the integers from 0 to 10. Use tabs to print the following neatly formatted table of values:

<table>
<thead>
<tr>
<th>integer</th>
<th>square</th>
<th>cube</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>64</td>
</tr>
<tr>
<td>5</td>
<td>25</td>
<td>125</td>
</tr>
<tr>
<td>6</td>
<td>36</td>
<td>216</td>
</tr>
<tr>
<td>7</td>
<td>49</td>
<td>343</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
<td>512</td>
</tr>
<tr>
<td>9</td>
<td>81</td>
<td>729</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>1000</td>
</tr>
</tbody>
</table>

Making a Difference

2.30 \((Body Mass Index Calculator)\) We introduced the body mass index (BMI) calculator in Exercise 1.12. The formulas for calculating BMI are

\[
BMI = \frac{\text{weightInPounds} \times 703}{\text{heightInInches} \times \text{heightInInches}}
\]

or

\[
BMI = \frac{\text{weightInKilograms}}{\text{heightInMeters} \times \text{heightInMeters}}
\]

Create a BMI calculator application that reads the user’s weight in pounds and height in inches (or, if you prefer, the user’s weight in kilograms and height in meters), then calculates and displays the user’s body mass index. Also, the application should display the following information from the Department of Health and Human Services/National Institutes of Health so the user can evaluate his/her BMI:

<table>
<thead>
<tr>
<th>BMI VALUES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Underweight: less than 18.5</td>
</tr>
<tr>
<td>Normal: between 18.5 and 24.9</td>
</tr>
<tr>
<td>Overweight: between 25 and 29.9</td>
</tr>
<tr>
<td>Obese: 30 or greater</td>
</tr>
</tbody>
</table>

[Note: In this chapter, you learned to use the \texttt{int} type to represent whole numbers. The BMI calculations when done with \texttt{int} values will both produce whole-number results. In Chapter 4 you’ll learn to use the \texttt{double} type to represent numbers with decimal points. When the BMI calculations are performed with \texttt{doubles}, they’ll both produce numbers with decimal points—the numbers are called “floating-point” numbers.]  

2.31 \((Car-Pool Savings Calculator)\) Research several car-pooling websites. Create an application that calculates your daily driving cost, so that you can estimate how much money could be saved by car pooling, which also has other advantages such as reducing carbon emissions and reducing traffic congestion. The application should input the following information and display the user’s cost per day of driving to work:

a) Total miles driven per day.  
b) Cost per gallon of gasoline.  
c) Average miles per gallon.  
d) Parking fees per day.  
e) Tolls per day.