Control Structures

Objectives

• To understand basic problem solving techniques.
• To be able to develop algorithms through the process of top-down, stepwise refinement.
• To be able to use the if, if/else and switch selection structures to choose among alternative actions.
• To be able to use the while, do/while and for repetition structures to execute statements in a program repeatedly.
• To understand counter-controlled repetition and sentinel-controlled repetition.
• To be able to use the increment, decrement, assignment and logical operators.
• To be able to use the break and continue program control statements.

Let’s all move one place on.
Lewis Carroll
The wheel is come full circle.
William Shakespeare, King Lear
Who can control his fate?
William Shakespeare, Othello
The used key is always bright.
Benjamin Franklin
2.1 Introduction

Before writing a program to solve a particular problem, it is essential to have a thorough understanding of the problem and a carefully planned approach to solving the problem. When writing a program, it is equally essential to understand the types of building blocks that are available and to employ proven program-construction principles. In this chapter, we discuss all of these issues in our presentation of the theory and principles of structured programming. The techniques that you will learn here are applicable to most high-level lan-
languages, including C++. When we begin our treatment of object-oriented programming in C++ in Chapter 6, we will see that the control structures we study here in Chapter 2 are helpful in building and manipulating objects.

2.2 Algorithms

Any computing problem can be solved by executing a series of actions in a specific order. A procedure for solving a problem in terms of

1. the actions to be executed and
2. the order in which these actions are to be executed

is called an algorithm. The following example demonstrates that correctly specifying the order in which the actions are to be executed is important.

Consider the “rise-and-shine algorithm” followed by one junior executive for getting out of bed and going to work: (1) Get out of bed, (2) take off pajamas, (3) take a shower, (4) get dressed, (5) eat breakfast, (6) carpool to work.

This routine gets the executive to work well prepared to make critical decisions. Suppose that the same steps are performed in a slightly different order: (1) Get out of bed, (2) take off pajamas, (3) get dressed, (4) take a shower, (5) eat breakfast, (6) carpool to work.

In this case, our junior executive shows up for work soaking wet. Specifying the order in which statements are to be executed in a computer program is called program control. In this chapter, we investigate the program-control capabilities of C++.

2.3 Pseudocode

Pseudocode is an artificial and informal language that helps programmers develop algorithms. The pseudocode we present here is useful for developing algorithms that will be converted to structured C++ programs. Pseudocode is similar to everyday English; it is convenient and user-friendly although it is not an actual computer programming language.

Pseudocode programs are not actually executed on computers. Rather, they help the programmer “think out” a program before attempting to write it in a programming language such as C++. In this chapter, we give several examples of how pseudocode can be used effectively in developing structured C++ programs.

The style of pseudocode we present consists purely of characters, so programmers can conveniently type pseudocode programs using an editor program. The computer can display a fresh copy of a pseudocode program on demand. A carefully prepared pseudocode program can be converted easily to a corresponding C++ program. This is done in many cases simply by replacing pseudocode statements with their C++ equivalents.

Pseudocode consists only of executable statements—those that are executed when the program has been converted from pseudocode to C++ and is run. Declarations are not executable statements. For example, the declaration

```c++
int i;
```

simply tells the compiler the type of variable `i` and instructs the compiler to reserve space in memory for the variable, but this declaration does not cause any action—such as input, output, or a calculation—to occur when the program is executed. Some programmers choose to list variables and briefly mention the purpose of each at the beginning of a pseudocode program.
2.4 Control Structures

Normally, statements in a program are executed one after the other in the order in which they are written. This is called sequential execution. Various C++ statements we will soon discuss enable the programmer to specify that the next statement to be executed may be other than the next one in sequence. This is called transfer of control.

During the 1960s, it became clear that the indiscriminate use of transfers of control was the root of much difficulty experienced by software-development groups. The finger of blame was pointed at the goto statement that allows the programmer to specify a transfer of control to one of a very wide range of possible destinations in a program. The notion of so-called structured programming became almost synonymous with "goto elimination."

The research of Bohm and Jacopini had demonstrated that programs could be written without any goto statements. The challenge of the era became for programmers to shift their styles to "goto-less programming." It was not until the 1970s that programmers started taking structured programming seriously. The results have been impressive as software development groups have reported reduced development times, more frequent on-time delivery of systems and more frequent within-budget completion of software projects. The key to these successes is that structured programs are clearer, easier to debug and modify and more likely to be bug-free in the first place.

Bohm and Jacopini’s work demonstrated that all programs could be written in terms of only three control structures, namely the sequence structure, the selection structure and the repetition structure. The sequence structure is built into C++. Unless directed otherwise, the computer executes C++ statements one after the other in the order in which they are written. The flowchart segment of Fig. 2.1 illustrates a typical sequence structure in which two calculations are performed in order.

A flowchart is a graphical representation of an algorithm or of a portion of an algorithm. Flowcharts are drawn using certain special-purpose symbols, such as rectangles, diamonds, ovals and small circles; these symbols are connected by arrows called flowlines.

Like pseudocode, flowcharts are useful for developing and representing algorithms, although pseudocode is strongly preferred by most programmers. Flowcharts clearly show how control structures operate; that is all we use them for in this text.

Consider the flowchart segment for the sequence structure in Fig. 2.1. We use the rectangle symbol, also called the action symbol, to indicate any type of action, including a calculation or an input/output operation. The flowlines in the figure indicate the order in which the actions are to be performed—first, grade is to be added to total, then 1 is to be added to counter. C++ allows us to have as many actions as we want in a sequence structure. As we will soon see, anywhere a single action may be placed, we can place several actions in sequence.

When drawing a flowchart that represents a complete algorithm, an oval symbol containing the word “Begin” is the first symbol used in the flowchart; an oval symbol containing the word “End” is the last symbol used. When drawing only a portion of an algorithm, as in Fig. 2.1, the oval symbols are omitted in favor of using small circle symbols also called connector symbols.

Perhaps the most important flowcharting symbol is the diamond symbol, also called the decision symbol, that indicates a decision is to be made. We will discuss the diamond symbol in the next section.

C++ provides three types of selection structures. The if selection structure either performs (selects) an action if a condition (predicate) is true or skips the action if the condition is false. The if/else selection structure performs an action if a condition is true and performs a different action if the condition is false. The switch selection structure performs one of many different actions depending on the value of an integer expression.

The if selection structure is a single-selection structure—it selects or ignores a single action. The if/else selection structure is a double-selection structure—it selects between two different actions. The switch selection structure is a multiple-selection structure—it selects the action to perform from many different actions.

C++ provides three types of repetition structures, namely while, do/while and for. Each of the words if, else, switch, while, do and for is a C++ keyword. These words are reserved by the language to implement various features such as C++’s control structures. Keywords must not be used as identifiers, such as for variable names. A complete list of C++ keywords is shown in Fig. 2.2.

Common Programming Error 2.1

Using a keyword as an identifier is a syntax error.

Well, that is all there is. C++ has only seven control structures: sequence, three types of selection and three types of repetition. Each C++ program is formed by combining as many of each type of control structure as is appropriate for the algorithm the program implements. As with the sequence structure of Fig. 2.1, we will see that each control structure is flowcharted with two small circle symbols, one at the entry point to the control structure and one at the exit point. These single-entry/single-exit control structures make it easy to build programs—the control structures are attached to one another by connecting the exit point of one control structure to the entry point of the next. This is similar to the way a child stacks building blocks, so we call this control-structure stacking. We will learn that there is only one other way to connect control structures—called control-structure nesting.

Software Engineering Observation 2.1

Any C++ program we will ever build can be constructed from only seven different types of control structures (sequence, if, if/else, switch, while, do/while and for) combined in only two ways (control-structure stacking and control-structure nesting).
2.5 The if Selection Structure

A selection structure is used to choose among alternative courses of action. For example, suppose the passing grade on an exam is 60. The pseudocode statement

```
If student’s grade is greater than or equal to 60
Print “Passed”
```

determines if the condition “student’s grade is greater than or equal to 60” is true or false. If the condition is true, then “Passed” is printed and the next pseudocode statement in order is “performed” (remember that pseudocode is not a real programming language). If the condition is false, the print statement is ignored and the next pseudocode statement in order is performed. Note that the second line of this selection structure is indented. Such indentation is optional, but it is highly recommended because it emphasizes the inherent structure of structured programs. When you convert your pseudocode into C++ code, the C++ compiler ignores whitespace characters like blanks, tabs and newlines used for indentation and vertical spacing.

Good Programming Practice 2.1

Consistently applying reasonable indentation conventions throughout your programs greatly improves program readability. We suggest a fixed-size tab of about 1/4 inch or three blanks per indent.
The preceding pseudocode *If* statement can be written in C++ as

```cpp
if ( grade >= 60 )
    cout << "Passed";
```

Notice that the C++ code corresponds closely to the pseudocode. This is one of the properties of pseudocode that makes it such a useful program development tool.

**Good Programming Practice 2.2**

Pseudocode is often used to “think out” a program during the program-design process. Then the pseudocode program is converted to C++.

The flowchart of Fig. 2.3 illustrates the single-selection *if* structure. This flowchart contains what is perhaps the most important flowcharting symbol—the *diamond symbol*, also called the *decision symbol*, which indicates that a decision is to be made. The decision symbol contains an expression, such as a condition, that can be either *true* or *false*. The decision symbol has two flowlines emerging from it. One indicates the direction to be taken when the expression in the symbol is *true*; the other indicates the direction to be taken when the expression is *false*. We learned in Chapter 1 that decisions can be based on conditions containing relational or equality operators. Actually, a decision can be based on any expression—if the expression evaluates to zero, it is treated as *false* and if the expression evaluates to nonzero, it is treated as *true*. The C++ standard provides the data type *bool* to represent *true* and *false*. The keywords *true* and *false* are used to represent values of type *bool*.

Note that the *if* structure, too, is a single-entry/single-exit structure. We will soon learn that the flowcharts for the remaining control structures also contain (besides small circle symbols and flowlines) only rectangle symbols to indicate the actions to be performed and diamond symbols to indicate decisions to be made. This is the *action/decision model of programming* we have been emphasizing.

We can envision seven bins, each containing only control structures of one of the seven types. These control structures are empty. Nothing is written in the rectangles or in the diamonds. The programmer’s task, then, is assembling a program from as many of each type of control structure as the algorithm demands, combining those control structures in only two possible ways (stacking or nesting), then filling in the actions and decisions in a manner appropriate for the algorithm. We will discuss the variety of ways in which actions and decisions may be written.
2.6 The if/else Selection Structure

The if selection structure performs an indicated action only when the condition is true; otherwise the action is skipped. The if/else selection structure allows the programmer to specify that a different action is to be performed when the condition is true than when the condition is false. For example, the pseudocode statement

If student’s grade is greater than or equal to 60
    Print “Passed”
else
    Print “Failed”

prints Passed if the student’s grade is greater than or equal to 60 and prints Failed if the student’s grade is less than 60. In either case, after printing occurs, the next pseudocode statement in sequence is “performed.” Note that the body of the else is also indented.

Good Programming Practice 2.3

Indent both body statements of an if/else structure.

Whatever indentation convention you choose should be applied carefully throughout your programs. It is difficult to read programs that do not obey uniform spacing conventions.

Good Programming Practice 2.4

If there are several levels of indentation, each level should be indented the same additional amount of space.

The preceding pseudocode If/else structure can be written in C++ as

```cpp
if ( grade >= 60 )
    cout << "Passed";
else
    cout << "Failed";
```

The flowchart of Fig. 2.4 nicely illustrates the flow of control in the if/else structure. Once again, note that (besides small circles and arrows) the only symbols in the flowchart are rectangles (for actions) and a diamond (for a decision). We continue to emphasize this action/decision model of computing. Imagine again a deep bin containing as many empty double-selection structures as might be needed to build any C++ program. The programmer’s job is to assemble these selection structures (by stacking and nesting) with any other control structures required by the algorithm, and to fill in the empty rectangles and empty diamonds with actions and decisions appropriate to the algorithm being implemented.

C++ provides the conditional operator (?:) that is closely related to the if/else structure. The conditional operator is C++’s only ternary operator—it takes three operands. The operands, together with the conditional operator, form a conditional expression. The first operand is a condition, the second operand is the value for the entire conditional expression if the condition is true and the third operand is the value for the entire conditional expression if the condition is false. For example, the output statement

```cpp
cout << ( grade >= 60 ? "Passed" : "Failed" );
```
Fig. 2.4 Flowcharting the double-selection if/else structure.

contains a conditional expression that evaluates to the string "Passed" if the condition grade >= 60 is true and evaluates to the string "Failed" if the condition is false. Thus, the statement with the conditional operator performs essentially the same as the preceding if/else statement. As we will see, the precedence of the conditional operator is low, so the parentheses in the preceding expression are required.

The values in a conditional expression can also be actions to execute. For example, the conditional expression

```
grade >= 60 ? cout << "Passed" : cout << "Failed";
```

is read, “If grade is greater than or equal to 60, then cout << "Passed"; otherwise cout << "Failed".” This, too, is comparable to the preceding if/else structure. We will see that conditional operators can be used in some situations where if/else statements cannot.

Nested if/else structures test for multiple cases by placing if/else selection structures inside if/else selection structures. For example, the following pseudocode statement will print A for exam grades greater than or equal to 90, B for grades in the range 80 to 89, C for grades in the range 70 to 79, D for grades in the range 60 to 69 and F for all other grades.

```
If student's grade is greater than or equal to 90
    Print "A"
else
    If student's grade is greater than or equal to 80
        Print "B"
    else
        If student's grade is greater than or equal to 70
            Print "C"
        else
            If student's grade is greater than or equal to 60
                Print "D"
            else
                Print "F"
```
This pseudocode can be written in C++ as

```cpp
if ( grade >= 90 )
    cout << "A";
else
    if ( grade >= 80 )
        cout << "B";
    else
        if ( grade >= 70 )
            cout << "C";
        else
            if ( grade >= 60 )
                cout << "D";
            else
                cout << "F";
```

If `grade` is greater than or equal to 90, the first four conditions will be `true`, but only the `cout` statement after the first test will be executed. After that `cout` is executed, the `else`-part of the “outer” `if/else` statement is skipped. Many C++ programmers prefer to write the preceding `if` structure as

```cpp
if ( grade >= 90 )
    cout << "A";
else if ( grade >= 80 )
    cout << "B";
else if ( grade >= 70 )
    cout << "C";
else if ( grade >= 60 )
    cout << "D";
else
    cout << "F";
```

The two forms are equivalent. The latter form is popular because it avoids the deep indentation of the code to the right. Such indentation often leaves little room on a line, forcing lines to be split and decreasing program readability.

**Performance Tip 2.1**

A nested `if/else` structure can be much faster than a series of single selection `if` structures because of the possibility of early exit after one of the conditions is satisfied.

**Performance Tip 2.2**

In a nested `if/else` structure, test the conditions that are more likely to be `true` at the beginning of the nested `if/else` structure. This will enable the nested `if/else` structure to run faster and exit earlier than will testing infrequently occurring cases first.

The `if` selection structure expects only one statement in its body. To include several statements in the body of an `if`, enclose the statements in braces (`{` and `}`). A set of statements contained within a pair of braces is called a `compound statement`.

**Software Engineering Observation 2.2**

A compound statement can be placed anywhere in a program that a single statement can be placed.

The following example includes a compound statement in the `else` part of an `if/else` structure.
if ( grade >= 60 )
  cout << "Passed.\n";
else {
  cout << "Failed.\n";
  cout << "You must take this course again.\n";
}

In this case, if grade is less than 60, the program executes both statements in the body of the else and prints

Failed.
You must take this course again.

Notice the braces surrounding the two statements in the else clause. These braces are important. Without the braces, the statement

cout << "You must take this course again.\n";

would be outside the body of the else-part of the if and would execute regardless of whether the grade is less than 60.

Common Programming Error 2.2
Forgetting one or both of the braces that delimit a compound statement can lead to syntax errors or logic errors in a program.

Good Programming Practice 2.5
Always putting the braces in an if/else structure (or any control structure) helps prevent their accidental omission, especially when adding statements to an if or else clause at a later time.

A syntax error is caught by the compiler. A logic error has its effect at execution time. A fatal logic error causes a program to fail and terminate prematurely. A nonfatal logic error allows a program to continue executing, but might produce incorrect results.

Software Engineering Observation 2.3
Just as a compound statement can be placed anywhere a single statement can be placed, it is also possible to have no statement at all, i.e., the empty statement. The empty statement is represented by placing a semicolon (;) where a statement would normally be.

Common Programming Error 2.3
Placing a semicolon after the condition in an if structure leads to a logic error in single-selection if structures and a syntax error in double-selection if structures (if the if-part contains an actual body statement).

Good Programming Practice 2.6
Some programmers prefer to type the beginning and ending braces of compound statements before typing the individual statements within the braces. This helps avoid omitting one or both of the braces.

In this section, we introduced the notion of a compound statement. A compound statement can contain declarations (as does the body of main, for example). If so, the compound statement is called a block. The declarations in a block are commonly placed first in
the block before any action statements, but declarations can be intermixed with action statements. We will discuss the use of blocks in Chapter 3. The reader should avoid using blocks (other than as the body of `main`, of course) until that time.

### 2.7 The while Repetition Structure

A repetition structure allows the programmer to specify that an action is to be repeated while some condition remains true. The pseudocode statement

```
While there are more items on my shopping list
  Purchase next item and cross it off my list
```

describes the repetition that occurs during a shopping trip. The condition, “there are more items on my shopping list” is either true or false. If it is true, then the action, “Purchase next item and cross it off my list” is performed. This action will be performed repeatedly while the condition remains `true`. The statement(s) contained in the `while` repetition structure constitute the body of the `while`. The `while` structure body can be a single statement or a compound statement. Eventually, the condition will become `false` (when the last item on the shopping list has been purchased and crossed off the list). At this point, the repetition terminates and the first pseudocode statement after the repetition structure is executed.

**Common Programming Error 2.4**

Not providing, in the body of a `while` structure, an action that eventually causes the condition in the `while` to become `false` normally results in an error called an “infinite loop” in which the repetition structure never terminates.

**Common Programming Error 2.5**

Spelling the keyword `while` with an uppercase `W`, as in `While` (remember that C++ is a case-sensitive language), is a syntax error. All of C++’s reserved keywords such as `while`, `if` and `else` contain only lowercase letters.

As an example of an actual `while`, consider a program segment designed to find the first power of 2 larger than 1000. Suppose the integer variable `product` has been initialized to 2. When the following `while` repetition structure finishes executing, `product` will contain the desired answer:

```
int product = 2;
while ( product <= 1000 )
  product = 2 * product;
```

The flowchart of Fig. 2.5 illustrates the flow of control in the `while` structure that corresponds to the preceding `while` structure. Once again, note that (besides small circles and arrows) the flowchart contains only a rectangle symbol and a diamond symbol. Imagine a deep bin of empty `while` structures that can be stacked and nested with other control structures to form a structured implementation of an algorithm’s flow of control. The empty rectangles and diamonds are then filled in with appropriate actions and decisions. The flowchart clearly shows the repetition. The flowline emerging from the rectangle wraps back to the decision that is tested each time through the loop until the decision becomes `false`. Then, the `while` structure exits and control passes to the next statement in the program.
When the `while` structure is entered, the value of `product` is 2. The variable `product` is repeatedly multiplied by 2, taking on the values 4, 8, 16, 32, 64, 128, 256, 512, and 1024 successively. When `product` becomes 1024, the `while` structure condition, `product <= 1000`, becomes `false`. This terminates the repetition—the final value of `product` is 1024. Program execution continues with the next statement after the `while`.

### 2.8 Formulating Algorithms: Case Study 1 (Counter-Controlled Repetition)

To illustrate how algorithms are developed, we solve several variations of a class-averaging problem. Consider the following problem statement:

*A class of ten students took a quiz. The grades (integers in the range 0 to 100) for this quiz are available to you. Determine the class average on the quiz.*

The class average is equal to the sum of the grades divided by the number of students. The algorithm for solving this problem on a computer must input each of the grades, perform the averaging calculation and print the result.

Let us use pseudocode to list the actions to be executed and specify the order in which these actions should be executed. We use *counter-controlled repetition* to input the grades one at a time. This technique uses a variable called a *counter* to control the number of times a set of statements will execute. In this example, repetition terminates when the counter exceeds 10. In this section, we present a pseudocode algorithm (Fig. 2.6) and the corresponding program (Fig. 2.7). In the next section, we show how pseudocode algorithms are developed. Counter-controlled repetition is often called *definite repetition* because the number of repetitions is known before the loop begins executing.

Note the references in the algorithm to a total and a counter. A *total* is a variable used to accumulate the sum of a series of values. A counter is a variable used to count— in this case, to count the number of grades entered. Variables used to store totals should normally be initialized to zero before being used in a program; otherwise, the sum would include the previous value stored in the total’s memory location.

Lines 11 through 14:

```plaintext
int total,       // sum of grades
    gradeCounter, // number of grades entered
    grade,        // one grade
    average;      // average of grades
```

declare variables `total`, `gradeCounter`, `grade` and `average` to be of type `int`. Variable `grade` will store the value the user inputs into the program.
Set total to zero
Set grade counter to one

While grade counter is less than or equal to ten
  Input the next grade
  Add the grade into the total
  Add one to the grade counter

Set the class average to the total divided by ten
Print the class average

Fig. 2.6  Pseudocode algorithm that uses counter-controlled repetition to solve the class average problem.

Fig. 2.7  C++ program and sample execution for the class average problem with counter-controlled repetition (part 1 of 2).
Notice that the preceding declarations appear in the body of function `main`. Remember that variables declared in a function definition’s body are *local variables* and can be used only from the line of their declaration in the function to the closing right brace (}) of the function definition. The declaration of a local variable in a function must appear before the variable is used in that function.

Lines 17 and 18:

```cpp
    total = 0;                           // clear total
    gradeCounter = 1;                    // prepare to loop
```

are assignment statements that initialize `total` to 0 and `gradeCounter` to 1.

Note that variables `total` and `gradeCounter` are initialized before they are used in a calculation. Counter variables are normally initialized to zero or one, depending on their use (we will present examples showing each of these uses). An uninitialized variable contains a "garbage" value (also called an *undefined value*)—the value last stored in the memory location reserved for that variable.

Common Programming Error 2.6

*If a counter or total is not initialized, the results of your program will probably be incorrect. This is an example of a logic error.*

Good Programming Practice 2.7

*Initialize counters and totals.*

Good Programming Practice 2.8

*Declare each variable on a separate line.*

Line 21:

```cpp
    while ( gradeCounter <= 10 ) {       // loop 10 times
```

indicates that the `while` structure should continue as long as `gradeCounter`’s value is less than or equal to 10.
Lines 22 and 23

```cpp
    cout << "Enter grade: " ;  // prompt for input
    cin >> grade;           // input grade
```

 correspond to the pseudocode statement “Input the next grade.” The first statement displays the prompt “Enter grade:” on the screen. The second statement inputs the grade value from the user.

Next, the program updates `total` with the new `grade` entered by the user. Line 24

```cpp
    total = total + grade;  // add grade to total
```

adds `grade` to the previous value of `total` and assigns the result to `total`.

The program is now ready to increment the variable `gradeCounter` to indicate that a grade has been processed, then read the next grade from the user. Line 25

```cpp
    gradeCounter = gradeCounter + 1;  // increment counter
```

adds 1 to `gradeCounter`, so the condition in the `while` structure will eventually become `false` and terminate the loop.

Line 29

```
    average = total / 10;          // integer division
```

assigns the results of the average calculation to variable `average`. Line 30

```cpp
    cout << "Class average is " << average << endl;
```

displays the string "Class average is " followed by the value of variable `average`.

Note that the averaging calculation in the program produced an integer result. Actually, the sum of the grades in this example is 817, which, when divided by 10, should yield 81.7, i.e., a number with a decimal point. We will see how to deal with such numbers (called floating-point numbers) in the next section.

**Common Programming Error 2.7**

In a counter-controlled loop, because the loop counter (when counting up by one each time through the loop) is one higher than its last legitimate value (i.e., 11 in the case of counting from 1 to 10), using the counter value in a calculation after the loop is often an off-by-one-error.

In Fig. 2.7, if line 29 used `gradeCounter` rather than 10 for the calculation, the output for this program would display an incorrect value of 74.

---

2.9 Formulating Algorithms with Top-Down, Stepwise Refinement: Case Study 2 (Sentinel-Controlled Repetition)

Let us generalize the class-average problem. Consider the following problem:

*Develop a class-averaging program that will process an arbitrary number of grades each time the program is run.*

In the first class-average example, the number of grades (10) was known in advance. In this example, no indication is given of how many grades are to be entered. The program must
process an arbitrary number of grades. How can the program determine when to stop the
input of grades? How will it know when to calculate and print the class average?

One way to solve this problem is to use a special value called a sentinel value (also
called a signal value, a dummy value or a flag value) to indicate “end of data entry.” The
user types grades in until all legitimate grades have been entered. The user then types the
sentinel value to indicate that the last grade has been entered. Sentinel-controlled repetition
is often called indefinite repetition because the number of repetitions is not known before
the loop begins executing.

Clearly, the sentinel value must be chosen so that it cannot be confused with an accept-
able input value. Because grades on a quiz are normally nonnegative integers, –1 is an
acceptable sentinel value for this problem. Thus, a run of the class-average program might
process a stream of inputs such as 95, 96, 75, 74, 89 and –1. The program would then com-
pute and print the class average for the grades 95, 96, 75, 74 and 89 (–1 is the sentinel value,
so it should not enter into the averaging calculation).

Common Programming Error 2.8
Choosing a sentinel value that is also a legitimate data value is a logic error.

We approach the class average program with a technique called top-down, stepwise
refinement, a technique that is essential to the development of well-structured programs.
We begin with a pseudocode representation of the top:

Determine the class average for the quiz

The top is a single statement that conveys the overall function of the program. As such, the
top is, in effect, a complete representation of a program. Unfortunately, the top (as in this
case) rarely conveys a sufficient amount of detail from which to write the C++ program. So
we now begin the refinement process. We divide the top into a series of smaller tasks and
list these in the order in which they need to be performed. This results in the following first
refinement.

Initialize variables
Input, sum and count the quiz grades
Calculate and print the class average

Here, only the sequence structure has been used—the steps listed are to be executed in or-
der, one after the other.

Software Engineering Observation 2.4
Each refinement, as well as the top itself, is a complete specification of the algorithm; only
the level of detail varies.

Software Engineering Observation 2.5
Many programs can be divided logically into three phases: an initialization phase that ini-
tializes the program variables; a processing phase that inputs data values and adjusts pro-
gram variables accordingly; and a termination phase that calculates and prints the final
results.

The preceding Software Engineering Observation is often all you need for the first
refinement in the top-down process. To proceed to the next level of refinement, i.e., the
second refinement, we commit to specific variables. We need a running total of the num-
bers, a count of how many numbers have been processed, a variable to receive the value of each grade as it is input and a variable to hold the calculated average. The pseudocode statement

\[\text{Initialize variables}\]

can be refined as follows:

- Initialize total to zero
- Initialize counter to zero

Notice that only the variables total and counter need to be initialized before they are used; the variables average and grade (for the calculated average and the user input, respectively) need not be initialized, because their values will be written over as they are calculated or input.

The pseudocode statement

\[\text{Input, sum and count the quiz grades}\]

requires a repetition structure (i.e., a loop) that successively inputs each grade. Because we do not know in advance how many grades are to be processed, we will use sentinel-controlled repetition. The user will type legitimate grades in one at a time. After the last legitimate grade is typed, the user will type the sentinel value. The program will test for the sentinel value after each grade is input and will terminate the loop when the sentinel value is entered by the user. The second refinement of the preceding pseudocode statement is then

\[\text{Input the first grade (possibly the sentinel)}\]

\[\text{While the user has not as yet entered the sentinel} \]

- Add this grade into the running total
- Add one to the grade counter
- Input the next grade (possibly the sentinel)

Notice that, in pseudocode, we do not use braces around the set of statements that form the body of the while structure. We simply indent the statements under the while to show that they belong to the while. Again, pseudocode is only an informal program-development aid.

The pseudocode statement

\[\text{Calculate and print the class average}\]

can be refined as follows:

\[\text{If the counter is not equal to zero}\]

- Set the average to the total divided by the counter
- Print the average

\[\text{else}\]

- Print “No grades were entered”

Notice that we are being careful here to test for the possibility of division by zero—a fatal logic error that, if undetected, would cause the program to fail (often called “bombing” or “crashing”). The complete second refinement of the pseudocode for the class average problem is shown in Fig. 2.8.

**Common Programming Error 2.9**

An attempt to divide by zero causes a fatal error.
Initialize total to zero
Initialize counter to zero

Input the first grade (possibly the sentinel)
While the user has not as yet entered the sentinel
   Add this grade into the running total
   Add one to the grade counter
   Input the next grade (possibly the sentinel)

If the counter is not equal to zero
   Set the average to the total divided by the counter
   Print the average
else
   Print “No grades were entered”

---

**Fig. 2.8** Pseudocode algorithm that uses sentinel-controlled repetition to solve the class-average problem.

**Good Programming Practice 2.9**

When performing division by an expression whose value could be zero, explicitly test for this case and handle it appropriately in your program (such as by printing an error message) rather than allowing the fatal error to occur.

In Fig. 2.6 and Fig. 2.8, we include some completely blank lines in the pseudocode to make the pseudocode more readable. The blank lines separate these programs into their various phases.

The pseudocode algorithm in Fig. 2.8 solves the more general class-averaging problem. This algorithm was developed after only two levels of refinement. Sometimes more levels are necessary.

**Software Engineering Observation 2.6**

The programmer terminates the top-down, stepwise refinement process when the pseudocode algorithm is specified in sufficient detail for the programmer to be able to convert the pseudocode to C++. Implementing the C++ program is then normally straightforward.

The C++ program and a sample execution are shown in Fig. 2.9. Although only integer grades are entered, the averaging calculation is likely to produce a number with a decimal point, i.e., a real number. The type `int` cannot represent real numbers. The program introduces the data type `double` to handle numbers with decimal points (also called floating-point numbers) and introduces a special operator called a *cast operator* to force the averaging calculation to produce a floating-point numeric result. These features are explained in detail after the program is presented.

In this example, we see that control structures can be stacked on top of one another (in sequence) just as a child stacks building blocks. The `while` structure (lines 30 through 35) is immediately followed by an `if/else` structure (lines 38 through 45) in sequence. Much of the code in this program is identical to the code in Fig. 2.7, so we concentrate in this example on the new features and issues.
Line 20 declares `double` variable `average`. This change allows us to store the class-average calculation’s result as a floating-point number. Line 24 initializes the variable `gradeCounter` to 0, because no grades have been entered yet. Remember that this program uses sentinel-controlled repetition. To keep an accurate record of the number of grades entered, variable `gradeCounter` is incremented only when a valid grade value is entered.

Notice that both input statements (lines 28 and 34)

```cpp
cin >> grade;
```

are preceded by an output statement that prompts the user for input.
Good Programming Practice 2.10

Prompt the user for each keyboard input. The prompt should indicate the form of the input and any special input values (such as the sentinel value the user should enter to terminate a loop).

Good Programming Practice 2.11

In a sentinel-controlled loop, the prompts requesting data entry should explicitly remind the user what the sentinel value is.

Study the difference between the program logic for sentinel-controlled repetition compared with that for the counter-controlled repetition in Fig. 2.7. In counter-controlled repetition, we read a value from the user during each pass of the while structure for the specified number of passes. In sentinel-controlled repetition, we read one value (line 28) before the program reaches the while structure. This value is used to determine if the program’s flow of control should enter the body of the while structure. If the while structure condition is false (i.e., the user has already typed the sentinel), the body of the while structure does not execute (no grades were entered). If, on the other hand, the condition is true, the body begins execution and the value entered by the user is processed (added to the total in this example). After the value is processed, the next value is input from the user before the end of the while structure’s body. As the closing right brace ({}) of the body is reached at line 35, execution continues with the next test of the while structure condition, using the new value just entered by the user to determine if the while structure’s body should execute again. Notice that the next value is always input from the user.

```cpp
// termination phase
if ( gradeCounter != 0 ) {
    average = static_cast< double >( total ) / gradeCounter;
    cout << "Class average is " << setprecision( 2 )
        << setiosflags( ios::fixed | ios::showpoint )
        << average << endl;
} else
    cout << "No grades were entered" << endl;
return 0;   // indicate program ended successfully
```

Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70
Enter grade, -1 to end: 64
Enter grade, -1 to end: 83
Enter grade, -1 to end: 89
Enter grade, -1 to end: -1
Class average is 82.50

Fig. 2.9 C++ program and sample execution for the class-average problem with sentinel-controlled repetition (part 2 of 2).
immediately before the `while` structure condition is evaluated. This allows us to determine if the value just entered by the user is the sentinel value before that value is processed (i.e., added to the `total`). If the value entered is the sentinel value, the `while` structure terminates and the value is not added to the `total`.

Notice the compound statement in the `while` loop in Fig 2.9. Without the braces, the last three statements in the body of the loop would fall outside the loop, causing the computer to interpret this code incorrectly, as follows:

```cpp
while ( grade != -1 )
    total = total + grade;
    gradeCounter = gradeCounter + 1;
    cout << "Enter grade, -1 to end: ";
    cin >> grade;
```

This would cause an infinite loop if the user does not input –1 for the first grade.

Averages do not always evaluate to integer values. Often, an average is a value that contains a fractional part, such as 7.2 or –93.5. These values are referred to as floating-point numbers and are represented in C++ by the data types `float` and `double`. A variable of type `double` can store a value of much greater magnitude or with greater precision than `float`. For this reason, we tend to use type `double` rather than type `float` to represent floating-point values in our programs. Constants (like `1000.0` and `.05`) are treated as type `double` by C++.

The variable `average` is declared to be of type `double` to capture the fractional result of our calculation. However, the result of the calculation `total / counter` is an integer, because `total` and `counter` are both integer variables. Dividing two integers results in integer division, in which any fractional part of the calculation is lost (i.e., truncated). Because the calculation is performed first, the fractional part is lost before the result is assigned to `average`. To produce a floating-point calculation with integer values, we must create temporary values that are floating-point numbers for the calculation. C++ provides the unary cast operator to accomplish this task. The statement

```cpp
average = static_cast< double >( total ) / gradeCounter;
```

includes the cast operator `static_cast< double >( )`, which creates a temporary floating-point copy of its operand in parentheses—`total`. Using a cast operator in this manner is called explicit conversion. The value stored in `total` is still an integer. The calculation now consists of a floating-point value (the temporary `double` version of `total`) divided by the integer `counter`.

The C++ compiler knows how to evaluate only expressions in which the data types of the operands are identical. To ensure that the operands are of the same type, the compiler performs an operation called promotion (also called implicit conversion) on selected operands. For example, in an expression containing the data types `int` and `double`, `int` operands are promoted to `double`. In our example, after `counter` is promoted to `double`, the calculation is performed and the result of the floating-point division is assigned to `average`. Later in this chapter, we discuss all the standard data types and their order of promotion.

Cast operators are available for any data type. The `static_cast` operator is formed by following keyword `static_cast` with angle brackets (`<` and `>`) around a data type.
name. The cast operator is a *unary operator*, i.e., an operator that takes only one operand. In Chapter 1, we studied the binary arithmetic operators. C++ also supports unary versions of the plus (+) and minus (−) operators, so that the programmer can write expressions like −7 or +5. Cast operators have higher precedence than other unary operators such as unary + and unary −. This precedence is higher than that of the *multiplicative operators* *, / and %, and lower than that of parentheses. We indicate the cast operator with the notation

\[ \text{static} \text{cast< type>}() \]

in our precedence charts.

The formatting capabilities in Fig. 2.9 are explained in depth in Chapter 11 and discussed here briefly. The call \textit{setprecision(2)} in the output statement

\begin{verbatim}
cout << "Class average is " << setprecision( 2 )
<< setiosflags( ios::fixed | ios::showpoint )
<< average << endl;
\end{verbatim}

indicates that \textit{double} variable \textit{average} is to be printed with two digits of \textit{precision} to the right of the decimal point (e.g., 92.37). This call is referred to as a *parameterized stream manipulator*. Programs that use these calls must contain the preprocessor directive

\begin{verbatim}
#include <iomanip>
\end{verbatim}

Lines 11 and 12 specify the names from the \textit{iomanip} header file that are used in this program. Note that \textit{endl} is a *nonparameterized stream manipulator* and does not require the \textit{iomanip} header file. If the precision is not specified, floating-point values are normally output with six digits of precision (i.e., the *default precision*), although we will see an exception to this in a moment.

The stream manipulator \textit{setiosflags( ios::fixed | ios::showpoint )} in the preceding statement sets two output formatting options, namely \textit{ios::fixed} and \textit{ios::showpoint}. The vertical bar character (|) separates multiple options in a \textit{setiosflags} call (we will explain the | notation in depth in Chapter 16). [Note: Although commas (,) are often used to separate a list of items, they can not be used with the stream manipulator \textit{setiosflags}; otherwise, only the last option in the list will be set.] The option \textit{ios::fixed} causes a floating-point value to be output in so-called *fixed-point format* (as opposed to *scientific notation*, which we will discuss in Chapter 11). The \textit{ios::showpoint} option forces the decimal point and trailing zeros to print even if the value is a whole number amount such as 88.00. Without the \textit{ios::showpoint} option, such a value prints in C++ as 88 without the trailing zeros and without the decimal point. When the preceding formatting is used in a program, the printed value is rounded to the indicated number of decimal positions, although the value in memory remains unaltered. For example, the values 87.945 and 67.543 are output as 87.95 and 67.54, respectively.

**Common Programming Error 2.10**

Using floating-point numbers in a manner that assumes they are represented precisely can lead to incorrect results. Floating-point numbers are represented only approximately by most computers.

**Good Programming Practice 2.12**

Do not compare floating-point values for equality or inequality. Rather, test that the absolute value of the difference is less than a specified small value.
Despite the fact that floating-point numbers are not always “100% precise,” they have numerous applications. For example, when we speak of a “normal” body temperature of 98.6 we do not need to be precise to a large number of digits. When we view the temperature on a thermometer and read it as 98.6, it may actually be 98.5999473210643. The point here is that calling this number simply 98.6 is fine for most applications.

Another way floating-point numbers develop is through division. When we divide 10 by 3, the result is 3.3333333… with the sequence of 3s repeating infinitely. The computer allocates a fixed amount of space to hold such a value, so clearly the stored floating-point value can only be an approximation.

2.10 Formulating Algorithms with Top-Down, Stepwise Refinement: Case Study 3 (Nested Control Structures)

Let us work another complete problem. We will once again formulate the algorithm by using pseudocode and top-down, stepwise refinement and write a corresponding C++ program. We have seen that control structures can be stacked on top of one another (in sequence) just as a child stacks building blocks. In this case study, we will see the only other structured way control structures can be connected in C++, namely through nesting of one control structure within another.

Consider the following problem statement:

A college offers a course that prepares students for the state licensing exam for real estate brokers. Last year, several of the students who completed this course took the licensing examination. Naturally, the college wants to know how well its students did on the exam. You have been asked to write a program to summarize the results. You have been given a list of these 10 students. Next to each name is written a 1 if the student passed the exam and a 2 if the student failed.

Your program should analyze the results of the exam as follows:

1. Input each test result (i.e., a 1 or a 2). Display the message “Enter result” on the screen each time the program requests another test result.
2. Count the number of test results of each type.
3. Display a summary of the test results indicating the number of students who passed and the number of students who failed.
4. If more than 8 students passed the exam, print the message “Raise tuition.”

After reading the problem statement carefully, we make the following observations:

1. The program must process 10 test results. A counter-controlled loop will be used.
2. Each test result is a number—either a 1 or a 2. Each time the program reads a test result, the program must determine if the number is a 1 or a 2. We test for a 1 in our algorithm. If the number is not a 1, we assume that it is a 2. (An exercise at the end of the chapter considers the consequences of this assumption.)
3. Two counters are used—one to count the number of students who passed the exam and one to count the number of students who failed the exam.
4. After the program has processed all the results, it must decide if more than 8 students passed the exam.

Let us proceed with top-down, stepwise refinement. We begin with a pseudocode representation of the top:
Analyze exam results and decide if tuition should be raised

Once again, it is important to emphasize that the top is a complete representation of the program, but several refinements are likely to be needed before the pseudocode can be naturally evolved into a C++ program. Our first refinement is

*Initialize variables*

*Input the ten quiz grades and count passes and failures*

*Print a summary of the exam results and decide if tuition should be raised*

Here, too, even though we have a complete representation of the entire program, further refinement is necessary. We now commit to specific variables. Counters are needed to record the passes and failures, a counter will be used to control the looping process and a variable is needed to store the user input. The pseudocode statement

*Initialize variables*

*Input the ten quiz grades and count passes and failures*

*Print a summary of the exam results and decide if tuition should be raised*

requires a loop that successively inputs the result of each exam. Here it is known in advance that there are precisely ten exam results, so counter-controlled looping is appropriate. Inside the loop (i.e., nested within the loop), a double-selection structure will determine whether each exam result is a pass or a failure and will increment the appropriate counter accordingly. The refinement of the preceding pseudocode statement is then

*Initialize variables*

*Input the ten quiz grades and count passes and failures*

*Print a summary of the exam results and decide if tuition should be raised*

The complete second refinement appears in Fig. 2.10. Notice that blank lines are also used to set off the while structure for program readability.
Initialize passes to zero
Initialize failures to zero
Initialize student counter to one

While student counter is less than or equal to ten
   Input the next exam result
   If the student passed
      Add one to passes
   else
      Add one to failures
      Add one to student counter

Print the number of passes
Print the number of failures

If more than eight students passed
   Print “Raise tuition”

---

Fig. 2.10 Pseudocode for examination-results problem.

This pseudocode is now sufficiently refined for conversion to C++. The C++ program and two sample executions are shown in Fig. 2.11.

```cpp
// Fig. 2.11: fig02_11.cpp
// Analysis of examination results
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int main()
{
    // initialize variables in declarations
    int passes = 0, failures = 0, studentCounter = 1, result;

    // process 10 students; counter-controlled loop
    while (studentCounter <= 10) {
        cout << "Enter result (1=pass,2=fail): ";
        cin >> result;
        if (result == 1) // if/else nested in while
            passes = passes + 1;
    }

    // print number of passes and failures
    cout << "Number of passes: " << passes << endl;
    cout << "Number of failures: " << failures << endl;

    // print tuition decision
    if (passes > 8)
        cout << "Raise tuition" << endl;
}
```

---

Fig. 2.11 C++ program and sample executions for examination-results problem (part 1 of 2).
int passes = 0,           // number of passes
failures = 0,         // number of failures
studentCounter = 1,   // student counter
result;               // one exam result

else
    failures = failures + 1;
    studentCounter = studentCounter + 1;

// termination phase
cout << "Passed " << passes << endl;
cout << "Failed " << failures << endl;

if ( passes > 8 )
    cout << "Raise tuition " << endl;

return 0;   // successful termination

Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Passed 9
Failed 1
Raise tuition

Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Passed 6
Failed 4

Fig. 2.11 C++ program and sample executions for examination-results problem (part 2 of 2).

Lines 12 through 15

    int passes = 0,       // number of passes
        failures = 0,     // number of failures
        studentCounter = 1,// student counter
        result;           // one exam result
declare the variables used in main to process the examination results. Note that we have taken advantage of a feature of C++ that allows variable initialization to be incorporated into declarations (passes is assigned 0, failures is assigned 0 and student-Counter is assigned 1). Looping programs may require initialization at the beginning of each repetition; such initialization would normally occur in assignment statements.

Good Programming Practice 2.13
Initializing variables when they are declared helps the programmer avoid the problems of uninitialized data.

Software Engineering Observation 2.7
Experience has shown that the most difficult part of solving a problem on a computer is developing the algorithm for the solution. Once a correct algorithm has been specified, the process of producing a working C++ program from the algorithm is normally straightforward.

Software Engineering Observation 2.8
Many experienced programmers write programs without ever using program development tools like pseudocode. These programmers feel that their ultimate goal is to solve the problem on a computer and that writing pseudocode merely delays the production of final outputs. Although this may work for simple and familiar problems, it can lead to serious errors and delays on large, complex projects.

2.11 Assignment Operators
C++ provides several assignment operators for abbreviating assignment expressions. For example, the statement

\[ c = c + 3; \]

can be abbreviated with the addition assignment operator += as

\[ c += 3; \]

The += operator adds the value of the expression on the right of the operator to the value of the variable on the left of the operator and stores the result in the variable on the left of the operator. Any statement of the form

\[ variable = variable \ operator \ expression; \]

where \( operator \) is one of the binary operators +, -, *, /, or % (or others we will discuss later in the text), can be written in the form

\[ variable \ operator = expression; \]

Thus the assignment \( c += 3 \) adds 3 to \( c \). Figure 2.12 shows the arithmetic assignment operators, sample expressions using these operators and explanations.

Performance Tip 2.3
Programmers can write programs a bit faster and compilers can compile programs a bit faster when the “abbreviated” assignment operators are used. Some compilers generate code that runs faster when “abbreviated” assignment operators are used.
Performance Tip 2.4

Many of the performance tips we mention in this text result in nominal improvements, so the reader might be tempted to ignore them. Significant performance improvement is often realized when a supposedly nominal improvement is placed in a loop that repeats many times.

2.12 Increment and Decrement Operators

C++ also provides the `++` unary increment operator and the `--` unary decrement operator, which are summarized in Fig. 2.13. If a variable \( c \) is incremented by 1, the increment operator `++` can be used rather than the expressions \( c = c + 1 \) or \( c += 1 \). If an increment or decrement operator is placed before a variable, it is referred to as the preincrement or predecrement operator, respectively. If an increment or decrement operator is placed after a variable, it is referred to as the postincrement or postdecrement operator, respectively. Preincrementing (predecrementing) a variable causes the variable to be incremented (decremented) by 1, then the new value of the variable is used in the expression in which it appears. Postincrementing (postdecrementing) a variable causes the current value of the variable to be used in the expression in which it appears, then the variable value is incremented (decremented) by 1.

Fig. 2.12 Arithmetic assignment operators.

<table>
<thead>
<tr>
<th>Assignment operator</th>
<th>Sample expression</th>
<th>Explanation</th>
<th>Assigns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+=</code></td>
<td>( c += 7 )</td>
<td>( c = c + 7 )</td>
<td>10 to ( c )</td>
</tr>
<tr>
<td><code>-=</code></td>
<td>( d -= 4 )</td>
<td>( d = d - 4 )</td>
<td>1 to ( d )</td>
</tr>
<tr>
<td><code>*=</code></td>
<td>( e *= 5 )</td>
<td>( e = e * 5 )</td>
<td>20 to ( e )</td>
</tr>
<tr>
<td><code>/=</code></td>
<td>( f /= 3 )</td>
<td>( f = f / 3 )</td>
<td>2 to ( f )</td>
</tr>
<tr>
<td><code>%=</code></td>
<td>( g %= 9 )</td>
<td>( g = g % 9 )</td>
<td>3 to ( g )</td>
</tr>
</tbody>
</table>

Assume: \( \text{int } c = 3, d = 5, e = 4, f = 6, g = 12; \)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Called</th>
<th>Sample expression</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>++</code></td>
<td>preincrement</td>
<td><code>++a</code></td>
<td>Increment ( a ) by 1, then use the new value of ( a ) in the expression in which ( a ) resides.</td>
</tr>
<tr>
<td><code>++</code></td>
<td>postincrement</td>
<td><code>a++</code></td>
<td>Use the current value of ( a ) in the expression in which ( a ) resides, then increment ( a ) by 1.</td>
</tr>
<tr>
<td><code>--</code></td>
<td>predecrement</td>
<td><code>--b</code></td>
<td>Decrement ( b ) by 1, then use the new value of ( b ) in the expression in which ( b ) resides.</td>
</tr>
<tr>
<td><code>--</code></td>
<td>postdecrement</td>
<td><code>b--</code></td>
<td>Use the current value of ( b ) in the expression in which ( b ) resides, then decrement ( b ) by 1.</td>
</tr>
</tbody>
</table>

Fig. 2.13 The increment and decrement operators.
The program of Fig. 2.14 demonstrates the difference between the preincrementing version and the postincrementing version of the \texttt{++} operator. Postincrementing the variable \texttt{c} causes it to be incremented after it is used in the output statement. Preincrementing the variable \texttt{c} causes it to be incremented before it is used in the output statement.

The program displays the value of \texttt{c} before and after the \texttt{++} operator is used. The decrement operator (\texttt{--}) works similarly.

\textbf{Good Programming Practice 2.14}

\textit{Unary operators should be placed next to their operands with no intervening spaces.}

The three assignment statements in Fig 2.11

\begin{verbatim}
passes = passes + 1;
failures = failures + 1;
studentCounter = studentCounter + 1;
\end{verbatim}

\begin{lstlisting}[language=C++]
// Fig. 2.14: fig02_14.cpp
// Preincrementing and postincrementing
#include <iostream>

using std::cout;
using std::endl;

int main()
{
    int c;
    c = 5;
    cout << c << endl; // print 5
    cout << c++ << endl; // print 5 then postincrement
    cout << c << endl << endl; // print 6
    c = 5;
    cout << c << endl; // print 5
    cout << ++c << endl; // preincrement then print 6
    cout << c << endl; // print 6
    return 0; // successful termination
}
\end{lstlisting}
can be written more concisely with assignment operators as

```c
passes += 1;
failures += 1;
studentCounter += 1;
```

with preincrement operators as

```c
++passes;
++failures;
++studentCounter;
```

or with postincrement operators as

```c
passes++;
failures++;
studentCounter++;
```

Note that, when incrementing or decrementing a variable in a statement by itself, the preincrement and postincrement forms have the same effect, and the predecrement and postdecrement forms have the same effect. It is only when a variable appears in the context of a larger expression that preincrementing the variable and postincrementing the variable have different effects (and similarly for predecrementing and postdecrementing). Also, preincrement and predecrement operate slightly faster than postincrement and postdecrement.

For now, only a simple variable name may be used as the operand of an increment or decrement operator. (We will see that these operators can be used on so-called lvalues.)

Common Programming Error 2.11

Attempting to use the increment or decrement operator on an expression other than a simple variable name, e.g., writing `++(x + 1)`, is a syntax error.

Figure 2.15 shows the precedence and associativity of the operators introduced up to this point. The operators are shown top-to-bottom in decreasing order of precedence. The second column describes the associativity of the operators at each level of precedence. Notice that the conditional operator (`?:`), the unary operators increment (`++`), decrement (`--`), plus (`+`), minus (`-`) and casts, and the assignment operators `=`, `+=`, `-=` `*=` `/=` and `%=` associate from right to left. All other operators in the operator precedence chart of Fig. 2.15 associate from left to right. The third column names the various groups of operators.

2.13 Essentials of Counter-Controlled Repetition

Counter-controlled repetition requires the following:

1. the name of a control variable (or loop counter);
2. the initial value of the control variable;
3. the condition that tests for the final value of the control variable (i.e., whether looping should continue);
4. the increment (or decrement) by which the control variable is modified each time through the loop.
Consider the simple program shown in Fig. 2.16, which prints the numbers from 1 to 10. The declaration at line 10

```cpp
int counter = 1;
```

names the control variable (`counter`), declares it to be an integer, reserves space for it in memory and sets it to an `initial value` of 1. Declarations that require initialization are, in effect, executable statements. In C++, it is more precise to call a declaration that also reserves memory—as the preceding declaration does—a `definition`.

```cpp
// Fig. 2.16: fig02_16.cpp
// Counter-controlled repetition
#include <iostream>
using std::cout;
using std::endl;
int main()
{
    int counter = 1;       // initialization
    while ( counter <= 10 ) { // repetition condition
        cout << counter << endl;
        ++counter;            // increment
    }
    return 0;
}
```

Fig. 2.16 Counter-controlled repetition.

<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>unary (postfix)</td>
</tr>
<tr>
<td>++ -- + -</td>
<td>right to left</td>
<td>unary (prefix)</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>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>conditional</td>
</tr>
<tr>
<td>= += -= *= /= %+=</td>
<td>right to left</td>
<td>assignment</td>
</tr>
<tr>
<td>,</td>
<td>left to right</td>
<td>comma</td>
</tr>
</tbody>
</table>
The declaration and initialization of `counter` could also have been accomplished with the statements:

```cpp
int counter;
counter = 1;
```

We use both methods of initializing variables.

The statement

```cpp
++counter;
```

increments the loop counter by 1 each time the loop is performed. The loop-continuation condition in the `while` structure tests if the value of the control variable is less than or equal to `10` (the last value for which the condition is `true`). Note that the body of this `while` is performed even when the control variable is `10`. The loop terminates when the control variable is greater than `10` (i.e., `counter` becomes `11`).

The program in Fig. 2.16 can be made more concise by initializing `counter` to `0` and by replacing the `while` structure with

```cpp
while ( ++counter <= 10 )
    cout << counter << endl;
```

This code saves a statement because the incrementing is done directly in the `while` condition before the condition is tested. Also, this code eliminates the braces around the body of the `while`, because the `while` now contains only one statement. Coding in such a condensed fashion takes some practice and can lead to programs that are more difficult to debug, modify and maintain.

**Common Programming Error 2.12**

Because floating-point values are approximate, controlling counting loops with floating-point variables can result in imprecise counter values and inaccurate tests for termination.

**Good Programming Practice 2.15**

Control counting loops with integer values.

**Good Programming Practice 2.16**

Indent the statements in the body of each control structure.
Good Programming Practice 2.17
Put a blank line before and after each control structure to make it stand out in the program.

Good Programming Practice 2.18
Too many levels of nesting can make a program difficult to understand. As a general rule, try to avoid using more than three levels of indentation.

Good Programming Practice 2.19
Vertical spacing above and below control structures, and indentation of the bodies of control structures within the control-structure headers, give programs a two-dimensional appearance that greatly improves readability.

2.14 The for Repetition Structure
The for repetition structure handles all the details of counter-controlled repetition. To illustrate the power of for, let us rewrite the program of Fig. 2.16. The result is shown in Fig. 2.17. The program operates as follows.

When the for structure begins executing, the control variable counter is declared and initialized to 1. Then, the loop-continuation condition counter ≤ 10 is checked. Because the initial value of counter is 1, the condition is satisfied, so the body statement prints the value of counter, namely 1. The control variable counter is then incremented in the expression counter++ and the loop begins again with the loop-continuation test. Because the control variable is now equal to 2, the final value is not exceeded, so the program performs the body statement again. This process continues until the control variable counter is incremented to 11—this causes the loop-continuation test to fail and repetition terminates. The program continues by performing the first statement after the for structure (in this case, the return statement at the end of the program).

```
// Fig. 2.17: fig02_17.cpp
// Counter-controlled repetition with the for structure
#include <iostream>

using std::cout;
using std::endl;

int main()
{
    // Initialization, repetition condition, and incrementing
    // are all included in the for structure header.
    for ( int counter = 1; counter <= 10; counter++ )
        cout << counter << endl;
    return 0;
}
```

Fig. 2.17 Counter-controlled repetition with the for structure.
Figure 2.18 takes a closer look at the for structure of Fig. 2.17. Notice that the for structure “does it all”—it specifies each of the items needed for counter-controlled repetition with a control variable. If there is more than one statement in the body of the for, braces are required to enclose the body of the loop.

Notice that Fig. 2.17 uses the loop-continuation condition $\text{counter} \leq 10$. If the programmer incorrectly wrote $\text{counter} < 10$, then the loop would be executed only 9 times. This is a common logic error called an off-by-one error.

**Common Programming Error 2.13**

Using an incorrect relational operator or using an incorrect final value of a loop counter in the condition of a while or for structure can cause off-by-one errors.

**Good Programming Practice 2.20**

Using the final value in the condition of a while or for structure and using the $\leq$ relational operator will help avoid off-by-one errors. For a loop used to print the values 1 to 10, for example, the loop-continuation condition should be $\text{counter} \leq 10$ rather than $\text{counter} < 10$ (which is an off-by-one error) or $\text{counter} < 11$ (which is nevertheless correct). Many programmers nevertheless prefer so-called zero-based counting, in which, to count 10 times through the loop, counter would be initialized to zero and the loop-continuation test would be $\text{counter} < 10$.

The general format of the for structure is

\[
\text{for ( initialization; loopContinuationTest; increment )}
\]

where the initialization expression initializes the loop’s control variable, loopContinuationTest is the loop-continuation condition (containing the final value of the control variable for which the condition is true) and increment increments the control variable. In most cases, the for structure can be represented by an equivalent while structure, as follows:

\[
\text{initialization;}
\]

\[
\text{while ( loopContinuationTest )}
\]

\[
\text{statement increment;}
\]

There is an exception to this rule, which we will discuss in Section 2.18.
If the initialization expression in the for structure header defines the control variable (i.e., the control variable’s type is specified before the variable name), the control variable can be used only in the body of the for structure, i.e., the value of the control variable will be unknown outside the for structure. This restricted use of the control variable name is known as the variable’s scope. The scope of a variable specifies where it can be used in a program. Scope is discussed in detail in Chapter 3, “Functions.”

**Common Programming Error 2.14**

When the control variable of a for structure is initially defined in the initialization section of the for structure header, using the control variable after the body of the structure is a syntax error.

**Portability Tip 2.1**

In the C++ standard, the scope of the control variable declared in the initialization section of a for structure is different from the scope in older C++ compilers. C++ code created with old C++ compilers can break when compiled on compilers that are compatible with the C++ standard. These are two defensive programming strategies that can be used to prevent this problem: Either define control variables with different names in every for structure, or, if you prefer to use the same name for the control variable in several for structures, define the control variable outside and before the first for loop.

Sometimes, the initialization and increment expressions are comma-separated lists of expressions. The commas, as used here, are comma operators that guarantee that lists of expressions evaluate from left to right. The comma operator has the lowest precedence of all C++ operators. The value and type of a comma-separated list of expressions is the value and type of the rightmost expression in the list. The comma operator is most often used in for structures. Its primary application is to enable the programmer to use multiple initialization expressions and/or multiple increment expressions. For example, there may be several control variables in a single for structure that must be initialized and incremented.

**Good Programming Practice 2.21**

Place only expressions involving the control variables in the initialization and increment sections of a for structure. Manipulations of other variables should appear either before the loop (if they execute only once, like initialization statements) or in the loop body (if they execute once per repetition, like incrementing or decrementing statements).

The three expressions in the for structure are optional. If the loopContinuationTest is omitted, C++ assumes that the loop-continuation condition is true, thus creating an infinite loop. One might omit the initialization expression if the control variable is initialized elsewhere in the program. One might omit the increment expression if the increment is calculated by statements in the body of the for or if no increment is needed. The increment expression in the for structure acts like a stand-alone statement at the end of the body of the for. Therefore, the expressions

```c++
counter = counter + 1
counter += 1
++counter
counter++
```

are all equivalent in the incrementing portion of the for structure. Many programmers prefer the form `counter++` because the incrementing occurs after the loop body is executed.
The postincrementing form therefore seems more natural. Because the variable being incremented here does not appear in an expression, both preincrementing and postincrementing have the same effect. The two semicolons in the for structure are required.

Common Programming Error 2.15
Using commas instead of the two required semicolons in a for header is a syntax error.

Common Programming Error 2.16
Placing a semicolon immediately to the right of the right parenthesis of a for header makes the body of that for structure an empty statement. This is normally a logic error.

Software Engineering Observation 2.9
Placing a semicolon immediately after a for header is sometimes used to create a so-called delay loop. Such a for loop with an empty body still loops the indicated number of times doing nothing other than the counting. For example, you might use a delay loop to slow down a program that is producing outputs on the screen too quickly for you to read them.

The initialization, loop-continuation condition and increment portions of a for structure can contain arithmetic expressions. For example, assume that \( x = 2 \) and \( y = 10 \). If \( x \) and \( y \) are not modified in the loop body, the statement

\[
\text{for ( int j = x; j <= 4 * x * y; j += y / x )}
\]

is equivalent to the statement

\[
\text{for ( int j = 2; j <= 80; j += 5 )}
\]

The “increment” of a for structure can be negative (in which case it is really a decrement and the loop actually counts downwards). If the loop-continuation condition is initially false, the body of the for structure is not performed. Instead, execution proceeds with the statement following the for.

The control variable is frequently printed or used in calculations in the body of a for structure, but it does not have to be. It is common to use the control variable for controlling repetition while never mentioning it in the body of the for structure.

Good Programming Practice 2.22
Although the value of the control variable can be changed in the body of a for loop, avoid doing so, because this practice can lead to subtle logic errors.

The for structure is flowcharted much like the while structure. Figure 2.19 shows the flowchart of the for statement

\[
\text{for ( int counter = 1; counter <= 10; counter++ )}
\]

\[
\text{cout << counter << endl;}
\]

The flowchart makes it clear that the initialization occurs once and that incrementing occurs each time after the body statement executes. Note that (besides small circles and arrows) the flowchart contains only rectangle symbols and a diamond symbol. Imagine, again, that the programmer has a bin of empty for structures—as many as needed to stack and nest with other control structures to form a structured implementation of an algorithm. The rectangles and diamonds are filled with actions and decisions appropriate to the algorithm.
Fig. 2.19  Flowcharting a typical for repetition structure.

2.15  Examples Using the for Structure

The following examples show methods of varying the control variable in a for structure. In each case, we write the appropriate for header. Note the change in the relational operator for loops that decrement the control variable.

a) Vary the control variable from 1 to 100 in increments of 1.

\[ \text{for ( int } i = 1; i \leq 100; i++ ) \]

b) Vary the control variable from 100 to 1 in increments of -1 (decrements of 1).

\[ \text{for ( int } i = 100; i \geq 1; i-- ) \]

Common Programming Error 2.17

Not using the proper relational operator in the loop-continuation condition of a loop that counts downwards (such as incorrectly using \( i \leq 1 \) in a loop counting down to 1) is usually a logic error that will yield incorrect results when the program runs.

c) Vary the control variable from 7 to 77 in steps of 7.

\[ \text{for ( int } i = 7; i \leq 77; i += 7 ) \]

d) Vary the control variable from 20 to 2 in steps of -2.

\[ \text{for ( int } i = 20; i \geq 2; i -= 2 ) \]

e) Vary the control variable over the following sequence of values: 2, 5, 8, 11, 14, 17, 20.

\[ \text{for ( int } j = 2; j \leq 20; j += 3 ) \]

f) Vary the control variable over the following sequence of values: 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.

\[ \text{for ( int } j = 99; j \geq 0; j -= 11 ) \]

The next two examples provide simple applications of the for structure. The program of Fig. 2.20 uses the for structure to sum all the even integers from 2 to 100.
Note that the body of the \texttt{for} structure in Fig. 2.20 could actually be merged into the rightmost portion of the \texttt{for} header, by using the comma operator as follows:

\begin{verbatim}
for ( int number = 2; number <= 100; number += 2 )
  sum += number, number += 2
\end{verbatim}

The assignment \texttt{sum = 0} could also be merged into the initialization section of the \texttt{for}.

\textbf{Good Programming Practice 2.23}

Although statements preceding a \texttt{for} and statements in the body of a \texttt{for} can often be merged into the \texttt{for} header, avoid doing so, because it can make the program more difficult to read.

\textbf{Good Programming Practice 2.24}

Limit the size of control structure headers to a single line, if possible.

The next example computes compound interest using the \texttt{for} structure. Consider the following problem statement:

A person invests $1000.00 in a savings account yielding 5 percent interest. Assuming that all interest is left on deposit in the account, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula for determining these amounts:

\[ a = p (1 + r)^n \]
where
\[ p \] is the original amount invested (i.e., the principal),
\[ r \] is the annual interest rate,
\[ n \] is the number of years and
\[ a \] is the amount on deposit at the end of the \( n \)th year.

This problem involves a loop that performs the indicated calculation for each of the 10 years the money remains on deposit. The solution is shown in Fig. 2.21.

The for structure executes the body of the loop 10 times, varying a control variable from 1 to 10 in increments of 1. C++ does not include an exponentiation operator, so we use the standard library function pow for this purpose. The function pow\((x, y)\) calculates the value of \(x\) raised to the \(y\)th power. In this example, the algebraic expression \((1 + r)^n\) is written as pow\((1 + \text{rate}, \text{year})\) where variable \text{rate} represents \(r\) and variable \text{year} represents \(n\). Function pow takes two arguments of type double and returns a double value.

This program would not compile without the inclusion of <cmath>. Function pow requires two double arguments. Note that year is an integer. The <cmath> file includes information that tells the compiler to convert the value of year to a temporary double representation before calling the function. This information is contained in pow’s function prototype. Function prototypes are explained in Chapter 3. We provide a summary of the pow function and other math library functions in Chapter 3.

### Common Programming Error 2.18

Forgetting to include the <cmath> file in a program that uses math library functions is a syntax error.

Notice that we declared the variables amount, principal and rate to be of type double. We have done this for simplicity because we are dealing with fractional parts of dollars and we need a type that allows decimal points in its values. Unfortunately, this can cause trouble. Here is a simple explanation of what can go wrong when using float or double to represent dollar amounts (assuming setprecision(2) is used to specify two digits of precision when printing): Two dollar amounts stored in the machine could be 14.234 (which prints as 14.23) and 18.673 (which prints as 18.67). When these amounts are added, they produce the internal sum 32.907 which prints as 32.91. Thus your printout could appear as

\[
\begin{align*}
14.23 \\
+ 18.67 \\
\hline
32.91
\end{align*}
\]

but a person adding the individual numbers as printed would expect the sum 32.90! You have been warned!

### Good Programming Practice 2.25

Do not use variables of type float or double to perform monetary calculations. The imprecision of floating-point numbers can cause errors that will result in incorrect monetary values. In the exercises, we explore the use of integers to perform monetary calculations. Note: C++ class libraries from third-party vendors are available for properly performing monetary calculations.
The output statement

```cpp
cout << setiosflags( ios::fixed | ios::showpoint )
    << setprecision( 2 );
```
before the for loop and the output statement

```cpp
    cout << setw( 4 ) << year << setw( 21 ) << amount << endl;
```

in the for loop combine to print the values of the variables `year` and `amount` with the formatting specified by the parameterized stream manipulators `setw`, `setiosflags` and `setprecision`. The call `setw(4)` specifies that the next value output is printed in a field width of 4, i.e., the value is printed with at least 4 character positions. If the value to be output is less than 4 character positions wide, the value is right justified in the field by default. If the value to be output is more than 4 character positions wide, the field width is extended to accommodate the entire value. The call `setiosflags(ios::left)` can be used to specify that values should be output left justified.

The other formatting in the preceding output statements indicates that variable `amount` is printed as a fixed-point value with a decimal point (specified with the stream manipulator `setiosflags(ios::fixed | ios::showpoint)`) right-justified in a field of 21 character positions (specified with `setw(21)`) and two digits of precision to the right of the decimal point (specified with `setprecision(2)`). We will discuss the powerful input/output formatting capabilities of C++ in detail in Chapter 11. We placed the `setiosflags` and `setprecision` stream manipulators in a `cout` before the for loop because these settings remain in effect until they are changed. Thus, they do not need to be applied during each iteration of the loop.

Note that the calculation `1.0 + rate`, which appears as an argument to the `pow` function, is contained in the body of the for statement. In fact, this calculation produces the same result each time through the loop, so repeating the calculation is wasteful.

**Performance Tip 2.5**

Avoid placing expressions whose values do not change inside loops—but, even if you do, many of today’s sophisticated optimizing compilers will automatically place such expressions outside loops in the generated machine language code.

**Performance Tip 2.6**

Many compilers contain optimization features that improve the code you write, but it is still better to write good code from the start.

For fun, be sure to try our Peter Minuit problem in the chapter exercises. This problem demonstrates the wonders of compound interest.

### 2.16 The switch Multiple-Selection Structure

We have discussed the if single-selection structure and the if/else double-selection structure. Occasionally, an algorithm will contain a series of decisions in which a variable or expression is tested separately for each of the constant integral values it can assume and different actions are taken. C++ provides the switch multiple-selection structure to handle such decision making.

The switch structure consists of a series of case labels and an optional default case. The program in Fig. 2.22 uses switch to count the number of each different letter grade that students earned on an exam.
// Fig. 2.22: fig02_22.cpp
// Counting letter grades
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int main()
{
    int grade,       // one grade
        aCount = 0,  // number of A's
        bCount = 0,  // number of B's
        cCount = 0,  // number of C's
        dCount = 0,  // number of D's
        fCount = 0;  // number of F's

    cout << "Enter the letter grades." << endl
         << "Enter the EOF character to end input." << endl;

    while ( ( grade = cin.get() ) != EOF ) {
        switch ( grade ) {  // switch nested in while
            case 'A':  // grade was uppercase A
            case 'a':  // or lowercase a
                ++aCount;
                break;  // necessary to exit switch
            case 'B':  // grade was uppercase B
            case 'b':  // or lowercase b
                ++bCount;
                break;
            case 'C':  // grade was uppercase C
            case 'c':  // or lowercase c
                ++cCount;
                break;
            case 'D':  // grade was uppercase D
            case 'd':  // or lowercase d
                ++dCount;
                break;
            case 'F':  // grade was uppercase F
            case 'f':  // or lowercase f
                ++fCount;
                break;
            case '\n': // ignore newlines,
                case '\t': // tabs,
                case ' ': // and spaces in input
                break;
        }
    }
    return 0;
}
In the program, the user enters letter grades for a class. Inside the `while` header,

```c++
while ( ( grade = cin.get() ) != EOF )
```

the parenthesized assignment `( grade = cin.get() )` is executed first. The `cin.get()` function reads one character from the keyboard and stores that character in integer variable `grade`. The dot notation used in `cin.get()` will be explained in Chapter 6, “Classes.” Characters normally are stored in variables of type `char`; however, an important feature of C++ is that characters can be stored in any integer data type because they
are represented as 1-byte integers in the computer. Thus, we can treat a character as either an integer or a character depending on its use. For example, the statement

```cpp
    cout << "The character (" << 'a' << ") has the value "
        << static_cast<int>( 'a' ) << endl;
```

prints the character `a` and its integer value as follows:

The character (a) has the value 97

The integer 97 is the character’s numerical representation in the computer. Many computers today use the ASCII (American Standard Code for Information Interchange) character set, in which 97 represents the lowercase letter ‘a’. A list of the ASCII characters and their decimal values is presented in the appendices.

Assignment statements as a whole have the value that is assigned to the variable on the left side of the `=`. Thus, the value of the assignment `grade = cin.get()` is the same as the value returned by `cin.get()` and assigned to the variable `grade`.

The fact that assignment statements have values can be useful for initializing several variables to the same value. For example,

```cpp
    a = b = c = 0;
```

first evaluates the assignment `c = 0` (because the `=` operator associates from right to left). The variable `b` is then assigned the value of the assignment `c = 0` (which is 0). Then, the variable `a` is assigned the value of the assignment `b = (c = 0)` (which is also 0). In the program, the value of the assignment `grade = cin.get()` is compared with the value of `EOF` (a symbol whose acronym stands for “end-of-file”). We use `EOF` (which normally has the value –1) as the sentinel value. However, you do not type the value –1 nor do you type the letters `EOF` as the sentinel value. Rather, you type a system-dependent keystroke combination to mean “end-of-file,” i.e., “I have no more data to enter.” `EOF` is a symbolic integer constant defined in the `<iostream>` header file. If the value assigned to `grade` is equal to `EOF`, the program terminates. We have chosen to represent characters in this program as `ints` because `EOF` has an integer value (again, normally –1).

### Portability Tip 2.2

The keystroke combinations for entering end-of-file are system dependent.

### Portability Tip 2.3

Testing for the symbolic constant `EOF` rather than –1 makes programs more portable. The ANSI standard states that `EOF` is a negative integral value (but not necessarily –1). Thus, `EOF` could have different values on different systems.

On UNIX systems and many others, end-of-file is entered by typing the sequence

```
<ctrl-d>
```

on a line by itself. This notation means to simultaneously press both the `ctrl` key and the `d` key. On other systems such as Digital Equipment Corporation’s VAX VMS or Microsoft Corporation’s MS-DOS, end-of-file can be entered by typing

```
<ctrl-c>
```

Note: In some cases, you may have to press `Enter` after the preceding key sequence.
The user enters grades at the keyboard. When the Enter (or Return) key is pressed, the characters are read by the `cin.get()` function, one character at a time. If the character entered is not end-of-file, the switch structure is entered. The keyword `switch` is followed by the variable name `grade` in parentheses. This is called the controlling expression. The value of this expression is compared with each of the case labels. Assume the user has entered the letter `C` as a grade. `C` is automatically compared to each case in the switch. If a match occurs (case `'C'`), the statements for that case are executed. For the letter `C`, `cCount` is incremented by 1 and the switch structure is exited immediately with the break statement. Note that, unlike other control structures, it is not necessary to enclose a multistatement case in braces.

The break statement causes program control to proceed with the first statement after the switch structure. The break statement is used because the cases in a switch statement would otherwise run together. If break is not used anywhere in a switch structure, then, each time a match occurs in the structure, the statements for all the remaining cases will be executed. (This feature is sometimes useful when performing the same actions for several cases, as in the program of Fig. 2.22.) If no match occurs, the default case is executed and an error message is printed.

Each case can have one or more actions. The switch structure is different from all other structures in that braces are not required around multiple actions in a case of a switch. The general switch multiple-selection structure (using a break in each case) is flowcharted in Fig. 2.23.

![Flowchart of switch structure with breaks](image)
The flowchart makes it clear that each break statement at the end of a case causes control to immediately exit the switch structure. Again, note that (besides small circles and arrows) the flowchart contains only rectangle symbols and diamond symbols. Imagine, again, that the programmer has access to a deep bin of empty switch structures—as many as the programmer might need to stack and nest with other control structures to form a structured implementation of an algorithm’s flow of control. And again, the rectangles and diamonds are then filled with actions and decisions appropriate to the algorithm. Nested control structures are common, but it is rare to find nested switch structures in a program.

Common Programming Error 2.19
Forgetting a break statement when one is needed in a switch structure is a logic error.

Common Programming Error 2.20
Omitting the space between the word case and the integral value being tested in a switch structure can cause a logic error. For example, writing case 3: instead of writing case 3:
simply creates an unused label. (We will say more about this in Chapter 18.) The problem is that the switch structure will not perform the appropriate actions when the switch’s controlling expression has a value of 3.

Good Programming Practice 2.26
Provide a default case in switch statements. Cases not explicitly tested in a switch statement without a default case are ignored. Including a default case focuses the programmer on the need to process exceptional conditions. There are situations in which no default processing is needed. Although the case clauses and the default case clause in a switch structure can occur in any order, it is considered a good programming practice to place the default clause last.

Good Programming Practice 2.27
In a switch structure when the default clause is listed last, the break statement is not required. Some programmers include this break for clarity and symmetry with other cases.

In the switch structure of Fig. 2.22, lines 50 through 53

```c
    case '\n':
    case ' ':
        break;
```

cause the program to skip newline, tab and blank characters. Reading characters one at a time can cause some problems. To have the program read the characters, they must be sent to the computer by pressing the Enter key on the keyboard. This places a newline character in the input after the character we wish to process. Often, this newline character must be specially processed to make the program work correctly. By including the preceding cases in our switch structure, we prevent the error message in the default case from being printed each time a newline, tab or space is encountered in the input.

Common Programming Error 2.21
Not processing newline and other whitespace characters in the input when reading characters one at a time can cause logic errors.
Note that several case labels listed together (such as case 'D': case 'd': in Fig. 2.22) simply means that the same set of actions is to occur for each of the cases.

When using the switch structure, remember that it can only be used for testing a constant integral expression, i.e., any combination of character constants and integer constants that evaluates to a constant integer value. A character constant is represented as the specific character in single quotes such as 'A'. An integer constant is simply an integer value.

When we get to the part of the book on object-oriented programming, we will present a more elegant way to implement switch logic. We will use a technique called polymorphism to create programs that are often clearer, more concise, easier to maintain and easier to extend than programs using switch logic.

Portable languages like C++ must have flexible data type sizes. Different applications might need integers of different sizes. C++ provides several data types to represent integers. The range of integer values for each type depends on the particular computer’s hardware. In addition to the types int and char, C++ provides the types short (an abbreviation of short int) and long (an abbreviation of long int). The minimum range of values for short integers is -32,768 to 32,767. For the vast majority of integer calculations, long integers are sufficient. The minimum range of values for long integers is -2,147,483,648 to 2,147,483,647. On most computers, ints are equivalent either to short or to long. The range of values for an int is at least the same as the range for short integers and no larger than the range for long integers. The data type char can be used to represent any of the characters in the computer’s character set. The data type char can also be used to represent small integers.

**Portability Tip 2.4**
Because ints vary in size between systems, use long integers if you expect to process integers outside the range -32,768 to 32,767 and you would like to be able to run the program on several different computer systems.

**Performance Tip 2.7**
In performance-oriented situations where memory is at a premium or execution speed is crucial, it might be desirable to use smaller integer sizes.

**Performance Tip 2.8**
Using smaller integer sizes can result in a slower program if the machine’s instructions for manipulating them are not as efficient as for the natural-size integers (e.g., sign extension must be done on them).

**Common Programming Error 2.22**
Providing identical case labels in a switch structure is a syntax error.

### 2.17 The do/while Repetition Structure

The do/while repetition structure is similar to the while structure. In the while structure, the loop-continuation condition is tested at the beginning of the loop before the body of the loop is performed. The do/while structure tests the loop-continuation condition after the loop body is performed; therefore, the loop body will be executed at least once. When a do/while terminates, execution continues with the statement after the while
clause. Note that it is not necessary to use braces in the `do/while` structure if there is only one statement in the body; however, the braces are usually included to avoid confusion between the `while` and `do/while` structures. For example,

```cpp
while ( condition )
```

is normally regarded as the header to a `while` structure. A `do/while` with no braces around the single statement body appears as

```cpp
do
statement
while ( condition );
```

which can be confusing. The last line—`while ( condition );`—might be misinterpreted by the reader as a `while` structure containing an empty statement. Thus, the `do/while` with one statement is often written as follows to avoid confusion:

```cpp
do {
    statement
} while ( condition );
```

**Good Programming Practice 2.28**

Some programmers always include braces in a `do/while` structure, even if the braces are not necessary. This helps eliminate ambiguity between the `while` structure and the `do/while` structure containing one statement.

**Common Programming Error 2.23**

Infinite loops are caused when the loop-continuation condition in a `while`, `for` or `do/while` structure never becomes `false`. To prevent this, make sure the value of the condition does change somewhere in the header or body of the loop so the condition can eventually become `false`.

The program in Fig. 2.24 uses a `do/while` repetition structure to print the numbers from 1 to 10. Note that the control variable `counter` is preincremented in the loop-continuation test. Note also the use of the braces to enclose the single-statement body of the `do/while` structure.

```cpp
// Fig. 2.24: fig02_24.cpp
// Using the do/while repetition structure
#include <iostream>
using std::cout;
using std::endl;

int main()
{
    int counter = 1;
    do {
        cout << counter << " ";
    } while ( ++counter <= 10 );
}
```

**Fig. 2.24** Using the `do/while` structure (part 1 of 2).
The `do/while` structure is flowcharted in Fig. 2.25. This flowchart makes it clear that the loop-continuation condition is not executed until after the action is performed at least once. Again, note that (besides small circles and arrows) the flowchart contains only a rectangle symbol and a diamond symbol. Imagine, again, that the programmer has access to a deep bin of empty `do/while` structures—as many as the programmer might need to stack and nest with other control structures to form a structured implementation of an algorithm’s flow of control. And again, the rectangles and diamonds are then filled with actions and decisions appropriate to the algorithm.

### 2.18 The break and continue Statements

The `break` and `continue` statements alter the flow of control. The `break` statement, when executed in a `while`, `for`, `do/while` or `switch` structure, causes immediate exit from that structure. Program execution continues with the first statement after the structure. Common uses of the `break` statement are to escape early from a loop or to skip the remainder of a `switch` structure (as in Fig. 2.22). Figure 2.26 demonstrates the `break` statement in a `for` repetition structure. When the `if` structure detects that `x` has become 5, `break` is executed. This terminates the `for` statement and the program continues with the `cout` after the `for`. The loop executes fully only four times.

Note that the control variable `x` in this program is defined outside the `for` structure header. This is because we intend to use the control variable both in the body of the loop and after the loop completes its execution.

```cpp
cout << endl;
return 0;
}
```

---

**Fig. 2.25** Flowcharting the `do/while` repetition structure.
The `continue` statement, when executed in a `while`, `for` or `do/while` structure, skips the remaining statements in the body of that structure and proceeds with the next iteration of the loop. In `while` and `do/while` structures, the loop-continuation test is evaluated immediately after the `continue` statement is executed. In the `for` structure, the increment expression is executed, then the loop-continuation test is evaluated. Earlier, we stated that the `while` structure could be used in most cases to represent the `for` structure. The one exception occurs when the increment expression in the `while` structure follows the `continue` statement. In this case, the increment is not executed before the repetition-continuation condition is tested and the `while` does not execute in the same manner as the `for`. Figure 2.27 uses the `continue` statement in a `for` structure to skip the output statement in the structure and begin the next iteration of the loop.

```cpp
#include <iostream>
using std::cout;
using std::endl;

int main()
{
    // x declared here so it can be used after the loop
    int x;
    for ( x = 1; x <= 10; x++ )
    {
        if ( x == 5 )
            break;    // break loop only if x is 5
        cout << x << " ";
    }
    cout << "\nBroke out of loop at x of " << x << endl;
    return 0;
}
```

Fig. 2.26 Using the `break` statement in a `for` structure.

```
// Fig. 2.26: fig02_26.cpp
// Using the break statement in a for structure
#include <iostream>
using std::cout;
using std::endl;

int main()
{
    // x declared here so it can be used after the loop
    int x;
    for ( x = 1; x <= 10; x++ )
    {
        if ( x == 5 )
            break;    // break loop only if x is 5
        cout << x << " ";
    }
    cout << "\nBroke out of loop at x of " << x << endl;
    return 0;
}
```

Broke out of loop at x of 5
Some programmers feel that `break` and `continue` violate structured programming. Because the effects of these statements can be achieved by structured programming techniques we will soon learn, these programmers do not use `break` and `continue`.

**Performance Tip 2.9**
The `break` and `continue` statements, when used properly, perform faster than the corresponding structured techniques we will soon learn.

**Software Engineering Observation 2.10**
There is a tension between achieving quality software engineering and achieving the best-performing software. Often, one of these goals is achieved at the expense of the other.

### 2.19 Logical Operators

So far we have studied only simple conditions, such as `counter <= 10`, `total > 1000` and `number != sentinelValue`. We have expressed these conditions in terms of the relational operators `>`, `<`, `>=` and `<=`, and the equality operators `==` and `!=`. Each decision tested precisely one condition. To test multiple conditions while making a decision, we performed these tests in separate statements or in nested `if` or `if/else` structures.

C++ provides logical operators that are used to form more complex conditions by combining simple conditions. The logical operators are `&&` (logical AND), `||` (logical OR) and `!` (logical NOT, also called logical negation). We consider examples of each of these.

Suppose we wish to ensure that two conditions are both `true` before we choose a certain path of execution. In this case we can use the logical `&&` operator as follows:

```c++
if ( gender == 1 && age >= 65 )
    ++seniorFemales;
```

---

Fig. 2.27 Using the `continue` statement in a `for` structure (part 2 of 2).
This if statement contains two simple conditions. The condition `gender == 1` might be evaluated, for example, to determine if a person is a female. The condition `age >= 65` is evaluated to determine if a person is a senior citizen. The simple condition to the left of the `&&` operator is evaluated first because the precedence of `==` is higher than the precedence of `&&`. If necessary, the simple condition to the right of the `&&` operator is evaluated next because the precedence of `>=` is higher than the precedence of `&&` (as we will discuss shortly, the right side of a logical AND expression is evaluated only if the left side is true). The if statement then considers the combined condition

```
gender == 1 && age >= 65
```

This condition is true if and only if both of the simple conditions are true. Finally, if this combined condition is indeed true, then the count of `seniorFemales` is incremented by 1. If either or both of the simple conditions are false, then the program skips the incrementing and proceeds to the statement following the if. The preceding combined condition can be made more readable by adding redundant parentheses

```
( gender == 1 ) && ( age >= 65 )
```

**Common Programming Error 2.24**

Although `3 < x < 7` is a mathematically correct condition, it does not evaluate correctly in C++. Use `(3 < x && x < 7)` to get the proper evaluation in C++.

The table of Fig. 2.28 summarizes the `&&` operator. The table shows all four possible combinations of false and true values for expression1 and expression2. Such tables are often called truth tables. C++ evaluates to false or true all expressions that include relational operators, equality operators and/or logical operators.

**Portability Tip 2.5**

For compatibility with earlier versions of the C++ standard, the bool value true can also be represented by any nonzero value and the bool value false can also be represented as the value 0.

Now let us consider the `||` (logical OR) operator. Suppose we wish to ensure at some point in a program that either or both of two conditions are true before we choose a certain path of execution. In this case we use the `||` operator as in the following program segment:

```
if ( semesterAverage >= 90 || finalExam >= 90 )
    cout << "Student grade is A" << endl;
```

<table>
<thead>
<tr>
<th>expression1</th>
<th>expression2</th>
<th>expression1 &amp;&amp; expression2</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

**Fig. 2.28** Truth table for the && (logical AND) operator.
This preceding condition also contains two simple conditions. The simple condition $\text{semesterAverage} \geq 90$ is evaluated to determine if the student deserves an “A” in the course because of a solid performance throughout the semester. The simple condition $\text{finalExam} \geq 90$ is evaluated to determine if the student deserves an “A” in the course because of an outstanding performance on the final exam. The if statement then considers the combined condition

$$\text{semesterAverage} \geq 90 \ || \ \text{finalExam} \geq 90$$

and awards the student an “A” if either or both of the simple conditions are true. Note that the message “Student grade is A” is not printed only when both of the simple conditions are false. Figure 2.29 is a truth table for the logical OR operator (| |).

The && operator has a higher precedence than the || operator. Both operators associate from left to right. An expression containing && or || operators is evaluated only until truth or falsehood is known. Thus, evaluation of the expression

$$\text{gender} == 1 \ &\& \ \text{age} \geq 65$$

will stop immediately if gender is not equal to 1 (i.e., the entire expression is false) and continue if gender is equal to 1 (i.e., the entire expression could still be true if the condition age $\geq 65$ is true).

**Common Programming Error 2.25**

In expressions using operator &&, it is possible that a condition—we will call this the dependent condition—might require another condition to be true for it to be meaningful to evaluate the dependent condition. In this case, the dependent condition should be placed after the other condition, or an error might occur.

**Performance Tip 2.10**

In expressions using operator &&, if the separate conditions are independent of one another make the condition that is most likely to be false the leftmost condition. In expressions using operator ||, make the condition that is most likely to be true the leftmost condition. This can reduce a program’s execution time.

C++ provides the ! (logical negation) operator to enable a programmer to “reverse” the meaning of a condition. Unlike the && and || operators, which combine two conditions (binary operators), the logical negation operator has only a single condition as an operand (unary operator). The logical negation operator is placed before a condition when we are interested in choosing a path of execution if the original condition (without the logical negation operator) is false, such as in the following program segment:

```c++
if ( !( grade == sentinelValue ) )
    cout << "The next grade is " << grade << endl;
```

The parentheses around the condition grade == sentinelValue are needed because the logical negation operator has a higher precedence than the equality operator. Figure 2.30 is a truth table for the logical negation operator.

In most cases, the programmer can avoid using logical negation by expressing the condition differently with an appropriate relational or equality operator. For example, the preceding statement can also be written as follows:
if ( grade != sentinelValue )
    cout << "The next grade is " << grade << endl;

This flexibility can often help a programmer express a condition in a more “natural” or convenient manner.

Figure 2.31 shows the precedence and associativity of the C++ operators introduced to this point. The operators are shown from top to bottom, in decreasing order of precedence.

<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>++ -- static_cast&lt;type&gt;()</td>
<td>left to right</td>
<td>unary (postfix)</td>
</tr>
<tr>
<td>+ -- static_cast&lt;type&gt;()</td>
<td>right to left</td>
<td>unary (prefix)</td>
</tr>
<tr>
<td>* / %</td>
<td>left to right</td>
<td>multiplicative</td>
</tr>
<tr>
<td>+ -- static_cast&lt;type&gt;()</td>
<td>right to left</td>
<td>additive</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>left to right</td>
<td>insertion/extraction</td>
</tr>
<tr>
<td>&lt;= &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>&amp;&amp;</td>
<td>left to right</td>
<td>logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>right to left</td>
<td>conditional</td>
</tr>
</tbody>
</table>

Fig. 2.31  Operator precedence and associativity (part 1 of 2).
2.20 Confusing Equality (==) and Assignment (=) Operators

There is one type of error that C++ programmers, no matter how experienced, tend to make so frequently that we felt it was worth a separate section. That error is accidentally swapping the operators == (equality) and = (assignment). What makes these swaps so damaging is the fact that they do not ordinarily cause syntax errors. Rather, statements with these errors ordinarily compile correctly and the programs run to completion, probably generating incorrect results through run-time logic errors.

There are two aspects of C++ that cause these problems. One is that any expression that produces a value can be used in the decision portion of any control structure. If the value is 0, it is treated as false, and if the value is nonzero, it is treated as true. The second is that C++ assignments produce a value, namely the value that is assigned to the variable on the left side of the assignment operator. For example, suppose we intend to write

```cpp
if ( payCode == 4 )
    cout << "You get a bonus!" << endl;
```

but we accidentally write

```cpp
if ( payCode = 4 )
    cout << "You get a bonus!" << endl;
```

The first if statement properly awards a bonus to the person whose payCode is equal to 4. The second if statement—the one with the error—evaluates the assignment expression in the if condition to the constant 4. Because any nonzero value is interpreted as true, the condition in this if statement is always true and the person always receives a bonus regardless of what the actual paycode is! Even worse, the paycode has been modified when it was only supposed to be examined!

**Common Programming Error 2.26**

Using operator == for assignment and using operator = for equality are logic errors.

**Testing and Debugging Tip 2.1**

Programmers normally write conditions such as x == 7 with the variable name on the left and the constant on the right. By reversing these so that the constant is on the left and the variable name is on the right as in 7 == x, the programmer who accidentally replaces the == operator with = will be protected by the compiler. The compiler will treat this as a syntax error because only a variable name can be placed on the left-hand side of an assignment statement. At least this will prevent the potential devastation of a run-time logic error.
Variable names are said to be *lvalues* (for “left values”) because they can be used on the left side of an assignment operator. Constants are said to be *rvalues* (for “right values”) because they can be used on only the right side of an assignment operator. Note that *lvalues* can also be used as *rvalues*, but not vice versa.

The other side of the coin can be equally unpleasant. Suppose the programmer wants to assign a value to a variable with a simple statement like

\[ x = 1; \]

but instead writes

\[ x == 1; \]

Here, too, this is not a syntax error. Rather the compiler simply evaluates the conditional expression. If \( x \) is equal to 1, the condition is *true* and the expression returns the value *true*. If \( x \) is not equal to 1, the condition is *false* and the expression returns the value *false*. Regardless of what value is returned, there is no assignment operator, so the value is simply lost and the value of \( x \) remains unaltered, probably causing an execution-time logic error. Unfortunately, we do not have a handy trick available to help you with this problem!

**Testing and Debugging Tip 2.2**

Use your text editor to search for all occurrences of `=` in your program and check that you have the correct operator in each place.

## 2.21 Structured-Programming Summary

Just as architects design buildings by employing the collective wisdom of their profession, so should programmers design programs. Our field is younger than architecture is and our collective wisdom is considerably sparser. We have learned that structured programming produces programs that are easier than unstructured programs to understand and hence are easier to test, debug, modify, and even prove correct in a mathematical sense.

Figure 2.32 summarizes C++’s control structures. Small circles are used in the figure to indicate the single entry point and the single exit point of each structure. Connecting individual flowchart symbols arbitrarily can lead to unstructured programs. Therefore, the programming profession has chosen to combine flowchart symbols to form a limited set of control structures and to build structured programs by properly combining control structures in two simple ways.

For simplicity, only single-entry/single-exit control structures are used—there is only one way to enter and only one way to exit each control structure. Connecting control structures in sequence to form structured programs is simple—the exit point of one control structure is connected to the entry point of the next control structure, i.e., the control structures are simply placed one after another in a program; we have called this “control structure stacking.” The rules for forming properly structured programs also allow for control structures to be nested.

Figure 2.33 shows the rules for forming properly structured programs. The rules assume that the rectangle flowchart symbol may be used to indicate any action, including input/output. The rules also assume that we begin with the simplest flowchart (Fig. 2.34).
Fig. 2.32  C++’s single-entry/single-exit sequence, selection and repetition structures.

Rules for Forming Structured Programs

1) Begin with the “simplest flowchart” (Fig. 2.34).
2) Any rectangle (action) can be replaced by two rectangles (actions) in sequence.

Fig. 2.33  Rules for forming structured programs (part 1 of 2).
Fig. 2.34  The simplest flowchart.

Applying the rules of Fig. 2.33 always results in a structured flowchart with a neat, building-block appearance. For example, repeatedly applying rule 2 to the simplest flowchart results in a structured flowchart containing many rectangles in sequence (Fig. 2.35). Notice that rule 2 generates a stack of control structures, so let us call rule 2 the stacking rule.

Rule 3 is called the nesting rule. Repeatedly applying rule 3 to the simplest flowchart results in a flowchart with neatly nested control structures. For example, in Fig. 2.36, the rectangle in the simplest flowchart is first replaced with a double-selection (if/else) structure. Then rule 3 is applied again to both of the rectangles in the double-selection structure, replacing each of these rectangles with double-selection structures. The dashed boxes around each of the double-selection structures represent the rectangle that was replaced in the original simplest flowchart.
Rule 4 generates larger, more involved and more deeply nested structures. The flowcharts that emerge from applying the rules in Fig. 2.33 constitute the set of all possible structured flowcharts and hence the set of all possible structured programs.

The beauty of the structured approach is that we use only seven simple single-entry/single-exit pieces and we assemble them in only two simple ways. Figure 2.37 shows the kinds of stacked building blocks that emerge from applying rule 2 and the kinds of nested building blocks that emerge from applying rule 3. The figure also shows the kind of overlapped building blocks that cannot appear in structured flowcharts (because of the elimination of the goto statement).

If the rules in Fig. 2.33 are followed, an unstructured flowchart (such as that in Fig. 2.38) cannot be created. If you are uncertain if a particular flowchart is structured, apply the rules of Fig. 2.33 in reverse to try to reduce the flowchart to the simplest flowchart. If the flowchart is reducible to the simplest flowchart, the original flowchart is structured; otherwise, it is not.

Fig. 2.36 Applying rule 3 of Fig. 2.33 to the simplest flowchart.
Structured programming promotes simplicity. Bohm and Jacopini have given us the result that only three forms of control are needed:

- **Sequence**
- **Selection**
- **Repetition**

Sequence is trivial. Selection is implemented in one of three ways:

- **if** structure (single selection)
- **if/else** structure (double selection)
- **switch** structure (multiple selection)

In fact, it is straightforward to prove that the simple **if** structure is sufficient to provide any form of selection—everything that can be done with the **if/else** structure and the **switch** structure can be implemented by combining **if** structures (although perhaps not as clearly and efficiently).

Repetition is implemented in one of three ways:

- **while** structure
- **do/while** structure
- **for** structure
It is straightforward to prove that the `while` structure is sufficient to provide any form of repetition. Everything that can be done with the `do/while` structure and the `for` structure can be done with the `while` structure (although perhaps not as smoothly).

Combining these results illustrates that any form of control ever needed in a C++ program can be expressed in terms of the following:

- sequence
- `if` structure (selection)
- `while` structure (repetition)

and that these control structures can be combined in only two ways—stacking and nesting. Indeed, structured programming promotes simplicity.

In this chapter, we discussed how to compose programs from control structures containing actions and decisions. In Chapter 3, we will introduce another program-structuring unit called the `function`. We will learn to compose large programs by combining functions that, in turn, are composed of control structures. We will also discuss how functions promote software reusability. In Chapter 6, we will introduce C++’s other program-structuring unit called the `class`. We will then create objects from classes and proceed with our treatment of object-oriented programming. Now, we continue our introduction to objects by introducing a problem that the reader will attack with the techniques of object-oriented design.

### 2.22 (Optional Case Study) Thinking About Objects: Identifying the Classes in a Problem

Now we begin our optional, object-oriented design/implementation case study. These “Thinking About Objects” sections at the ends of this and the next several chapters will ease you into object orientation by examining an elevator simulation case study. This case study will provide you with a substantial, carefully paced, complete design and implementation experience. In Chapters 2 through 5 we will perform the various steps of an object-oriented design (OOD) using the UML. In Chapters 6, 7 and 9, we will implement the elevator simulator using the techniques of object-oriented programming (OOP) in C++. We present this case study in a fully solved format. This is not an exercise; rather it is an end-to-end learning experience that concludes with a detailed walkthrough of the C++ code. We have provided this case study so you can become accustomed to the kinds of substantial problems that are attacked in industry. We hope you enjoy this experience.

**Problem Statement**

A company intends to build a two-floor office building and equip it with an elevator. The company wants you to develop an object-oriented software simulator in C++ that models the operation of the elevator to determine whether or not it will meet their needs.

Your simulator should include a clock that begins with its time, in seconds, set to zero. The clock ticks (increments the time by one) every second; it does not keep track of hours and minutes. Your simulator should also include a scheduler that begins the day by randomly scheduling two times: the time when a person will step onto floor 1 and press the button on the floor to summon the elevator, and the time when a person will step onto floor
and press the button on the floor to summon the elevator. Each of these times is a random integer in the range of 5 to 20, inclusive (i.e., 5, 6, 7, ..., 20). [Note: You will learn how to schedule random times in Chapter 3.] When the clock time equals the earlier of these two times, the scheduler creates a person, who then walks onto the appropriate floor and presses the floor button. [Note: It is possible that these two randomly scheduled times will be identical, in which case people will step onto both floors and press both floor buttons at the same time.] The floor button illuminates, indicating that it has been pressed. [Note: The illumination of the floor button occurs automatically when the button is pressed and needs no programming; the light built into the button turns off automatically when the button is reset.] The elevator starts the day waiting with its door closed on floor 1. To conserve energy, the elevator moves only when necessary. The elevator alternates directions between moving up and moving down.

For simplicity, the elevator and each of the floors have a capacity of one person. The scheduler first verifies that a floor is unoccupied before creating a person to walk onto that floor. If the floor is occupied, the scheduler delays creating the person by one second (thus giving the elevator an opportunity to pick up the person and clear the floor). After a person walks onto a floor, the scheduler creates the next random time (between 5 and 20 seconds into the future) for a person to walk onto that floor and press the floor button.

When the elevator arrives at a floor, it resets the elevator button and sounds the elevator bell (which is inside the elevator). The elevator then signals its arrival to the floor. The floor, in response, resets the floor button and turns on the floor's elevator arrival light. The elevator then opens its door. [Note: The door on the floor opens automatically with the elevator door and needs no programming.] The elevator’s passenger, if there is one, exits the elevator, and a person, if there is one waiting on that floor, enters the elevator. Although each floor has a capacity of one person, assume there is enough room on each floor for a person to wait on that floor while the elevator’s passenger, if there is one, exits.

A person entering the elevator presses the elevator button, which illuminates (automatically, without programming) when pressed and turns off when the elevator arrives on the floor and resets the elevator button. [Note: Because there are only two floors, only one elevator button is necessary; this button simply tells the elevator to move to the other floor.] Next, the elevator closes its door and begins moving to the other floor. When the elevator arrives at a floor, if a person does not enter the elevator and the floor button on the other floor has not been pressed, the elevator closes its door and remains on that floor until a button on a floor is pressed.

For simplicity, assume that all the activities that happen once the elevator reaches a floor, and until the elevator closes its door, take zero time. [Note: Although these activities take zero time, they still occur sequentially, e.g., the elevator door must open before the passenger exits the elevator.] The elevator takes five seconds to move from either floor to the other. Once per second, the simulator provides the time to the scheduler and to the elevator. The scheduler and elevator use the time to determine what actions each needs to take at that particular time, e.g., the scheduler may determine that it is time to create a person; and the elevator, if moving, may determine that it is time to arrive at its destination floor.

The simulator should display messages on the screen describing the activities that occur in the system. These include a person pressing a floor button, the elevator arriving on a floor, the clock ticking, a person entering the elevator, etc. The output should resemble the following:
Enter run time: 30
(scheduler schedules next person for floor 1 at time 5)
(scheduler schedules next person for floor 2 at time 17)

*** ELEVATOR SIMULATION BEGINS ***

TIME: 1
elevator at rest on floor 1

TIME: 2
elevator at rest on floor 1

TIME: 3
elevator at rest on floor 1

TIME: 4
elevator at rest on floor 1

TIME: 5
scheduler creates person 1
person 1 steps onto floor 1
person 1 presses floor button on floor 1
floor 1 button summons elevator
(scheduler schedules next person for floor 1 at time 20)
elevator resets its button
elevator rings its bell
floor 1 resets its button
floor 1 turns on its light
elevator opens its door on floor 1
person 1 enters elevator from floor 1
person 1 presses elevator button
elevator button tells elevator to prepare to leave
floor 1 turns off its light
elevator closes its door on floor 1
elevator begins moving up to floor 2 (arrives at time 10)

TIME: 6
elevator moving up

TIME: 7
elevator moving up

TIME: 8
elevator moving up

TIME: 9
elevator moving up

TIME: 10
elevator arrives on floor 2
elevator resets its button
elevator rings its bell
floor 2 resets its button
floor 2 turns on its light
Control Structures Chapter 2

elevator opens its door on floor 2
person 1 exits elevator on floor 2
floor 2 turns off its light
elevator closes its door on floor 2
elevator at rest on floor 2

TIME: 11
elevator at rest on floor 2

TIME: 12
elevator at rest on floor 2

TIME: 13
elevator at rest on floor 2

TIME: 14
elevator at rest on floor 2

TIME: 15
elevator at rest on floor 2

TIME: 16
elevator at rest on floor 2

TIME: 17
scheduler creates person 2
person 2 steps onto floor 2
person 2 presses floor button on floor 2
floor 2 button summons elevator
(scheduler schedules next person for floor 2 at time 34)
elevator resets its button
elevator rings its bell
floor 2 resets its button
floor 2 turns on its light
elevator opens its door on floor 2
person 2 enters elevator from floor 2
person 2 presses elevator button
elevator button tells elevator to prepare to leave
floor 2 turns off its light
elevator closes its door on floor 2
elevator begins moving down to floor 1 (arrives at time 22)

TIME: 18
elevator moving down

TIME: 19
elevator moving down

TIME: 20
scheduler creates person 3
person 3 steps onto floor 1
person 3 presses floor button on floor 1
floor 1 button summons elevator
(scheduler schedules next person for floor 1 at time 26)
Chapter 2

Control Structures

elevator moving down

TIME: 21
elevator moving down

TIME: 22
elevator arrives on floor 1
elevator resets its button
elevator rings its bell
floor 1 resets its button
floor 1 turns on its light
elevator opens its door on floor 1
person 2 exits elevator on floor 1
person 3 enters elevator from floor 1
person 3 presses elevator button
elevator button tells elevator to prepare to leave
floor 1 turns off its light
elevator closes its door on floor 1
elevator begins moving up to floor 2 (arrives at time 27)

TIME: 23
elevator moving up

TIME: 24
elevator moving up

TIME: 25
elevator moving up

TIME: 26
scheduler creates person 4
person 4 steps onto floor 1
person 4 presses floor button on floor 1
floor 1 button summons elevator
(scheduler schedules next person for floor 1 at time 35)
elevator moving up

TIME: 27
elevator arrives on floor 2
elevator resets its button
elevator rings its bell
floor 2 resets its button
floor 2 turns on its light
elevator opens its door on floor 2
person 3 exits elevator on floor 2
floor 2 turns off its light
elevator closes its door on floor 2
elevator begins moving down to floor 1 (arrives at time 32)

TIME: 28
elevator moving down

TIME: 29
elevator moving down
Our goal (over these “Thinking About Objects” sections in Chapters 2 through 7 and Chapter 9) is to implement a working software simulator that models the operation of the elevator for the number of seconds entered by simulator user.

Analyzing and Designing the System
In this and the next several “Thinking About Objects” sections, we perform the steps of an object-oriented design process for the elevator system. The UML is designed for use with any OOAD process—many such processes exist. One popular method is the Rational Unified Process™ developed by Rational Software Corporation. For this case study, we present our own simplified design process for your first OOD/UML experience.

Before we begin, we must examine the nature of simulations. A simulation consists of two portions. One contains all the elements that belong to the world we want to simulate. These elements include the elevator, the floors, the buttons, the lights, etc. Let us call this the world portion. The other portion contains all the elements needed to simulate this world. These elements include the clock and the scheduler. We call this the controller portion. We will keep these two portions in mind as we design our system.

Use Case Diagrams
When developers begin a project, they rarely start with a detailed problem statement, such as the one we have provided at the beginning of this section (Section 2.22). This document and others are usually the result of the object-oriented analysis (OOA) phase. In this phase you interview the people who want you to build the system and the people who will eventually use the system. You use the information gained in these interviews to compile a list of system requirements. These requirements guide you and your fellow developers as you design the system. In our case study, the problem statement contains the system requirements for the elevator system. The output of the analysis phase is intended to specify clearly what the system is supposed to do. The output of the design phase is intended to clearly specify how the system should be constructed to do what is needed.

The UML provides the use case diagram to facilitate the process of requirements gathering. The use case diagram models the interactions between the system’s external clients and the use cases of the system. Each use case represents a different capability that the system provides the client. For example, an automated teller machine has several use cases, including “Deposit,” “Withdraw” and “Transfer Funds.”

Figure 2.39 shows the use case diagram for the elevator system. The stick figure represents an actor. Actors are any external entities such as people, robots, other systems, etc., that use the system. The only actors in our system are the people who want to ride the elevator. We therefore model one actor called “Person.” The actor’s “name” appears underneath the stick figure.
The *system box* (i.e., the enclosing rectangle in the figure) contains the use cases for the system. Notice that the box is labeled “Elevator System.” This title shows that *this use case model focuses on the behaviors of the system we want to simulate* (i.e., elevator transporting people), *as opposed to the behaviors of the simulation* (i.e., creating people and scheduling arrivals).

The UML models each use case as an oval. In our simple system, actors use the elevator for only one purpose: to move to another floor. The system provides only one capability to its users; therefore, “Move to other floor” is the only use case in our elevator system.

As you build your system, you rely on the use case diagram to ensure that all the clients’ needs are met. Our case study contains only one use case. In larger systems, use case diagrams are indispensable tools that help systems designers remain focused on satisfying the users’ needs. The goal of the use case diagram is to show the kinds of interactions users have with a system without providing the details of those interactions.

**Identifying the Classes in a System**

The next step of our OOD process is to *identify the classes* in our problem. We will eventually describe these classes in a formal way and implement them in C++ (we begin implementing the elevator simulator in C++ in Chapter 6). First we review the problem statement and locate all the *nouns*; with high likelihood, these represent most of the classes (or instances of classes) necessary to implement the elevator simulator. Figure 2.40 is a list of these nouns.

### List of nouns in the problem statement

- company
- building
- elevator
- simulator
- clock

*Fig. 2.40* List of nouns in problem statement (part 1 of 2).
We choose only the nouns that perform important duties in our system. For this reason we omit the following:

- company
- simulator
- time
- energy
- capacity

We do not need to model “company” as a class, because the company is not part of the simulation; the company simply wants us to model the elevator. The “simulator” is our entire C++ program, not an individual class. The “time” is a property of the clock, not an entity itself. We do not model “energy” in our simulation (although electric, gas or oil companies might certainly be interested in doing so in their simulation programs) and, finally, “capacity” is a property of the elevator and of the floor—not a separate entity itself.

We determine the classes for our system by filtering the remaining nouns into categories. Each remaining noun from Fig. 2.40 refers to one or more of the following categories:

- building
- elevator
- clock
- scheduler
- person (person waiting on a floor, elevator’s passenger)
• floor (floor 1, floor 2)
• floor button
• elevator button
• bell
• light
• door

These categories are likely to be the classes we will need to implement for our system. Notice that we create one category for the buttons on the floors and one category for the button on the elevator. The two types of buttons perform different duties in our simulation—the buttons on the floors summon the elevator, and the button in the elevator tells the elevator to begin moving to the other floor.

We can now model the classes in our system based on the categories we derived. By convention, we will capitalize class names. If the name of a class contains more than one word, we run the words together and capitalize each word (e.g., **MultipleWordName**). Using this convention, we create classes **Elevator**, **Clock**, **Scheduler**, **Person**, **Floor**, **Door**, **Building**, **FloorButton**, **ElevatorButton**, **Bell** and **Light**. We construct our system using all of these classes as building blocks. Before we begin building the system, however, we must gain a better understanding of how the classes relate to one another.

**Class Diagrams**

The UML enables us to model the classes in the elevator system and their relationships via the **class diagram**. Figure 2.41 shows how to represent a class using the UML. Here, we model class **Elevator**. In a class diagram, each class is modeled as a rectangle. This rectangle can then be divided into three parts. The top part contains the name of the class.

The middle part contains the class’s *attributes*. We discuss attributes in the “Thinking About Objects” section at the end of Chapter 3. The bottom contains the class’s *operations*. We discuss operations in the “Thinking About Objects” section at the end of Chapter 4.

Classes relate to one another via *associations*. Figure 2.42 shows how our classes **Building**, **Elevator** and **Floor** relate to one another. Notice that the rectangles in this diagram are not subdivided into three sections. The UML allows the truncation of class symbols in this manner in order to create more readable diagrams.
In this class diagram, a solid line that connects classes represents an association. An association is a relationship between classes. The numbers near the lines express multiplicity values. Multiplicity values indicate how many objects of a class participate in the association. From the diagram, we see that two objects of class `Floor` participate in the association with one object of class `Building`. Therefore, class `Building` has a one-to-two relationship with class `Floor`; we can also say that class `Floor` has a two-to-one relationship with class `Building`. From the diagram, you can see that class `Building` has a one-to-one relationship with class `Elevator` and vice versa. Using the UML, we can model many types of multiplicity. Figure 2.43 shows the multiplicity types and how to represent them.

An association can be named. For example, the word “Services” above the line connecting classes `Floor` and `Elevator` indicates the name of that association—the arrow shows the direction of the association. This part of the diagram reads: “one object of class `Elevator` services two objects of class `Floor`.”

The solid diamond attached to the association lines of class `Building` indicates that class `Building` has a composition relationship with classes `Floor` and `Elevator`. Composition implies a whole/part relationship. The class that has the composition symbol (the solid diamond) on its end of the association line is the whole (in this case, `Building`), and the class on the other end of the association line is the part (i.e., `Floor` and `Elevator`).

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>None.</td>
</tr>
<tr>
<td>1</td>
<td>One.</td>
</tr>
<tr>
<td>m</td>
<td>An integer value.</td>
</tr>
</tbody>
</table>

According to the UML 1.3 specifications, classes in a composition relationship observe the following three properties: 1) only one class in the relationship may represent the whole (i.e., the diamond can only be placed on one end of the association line); 2) composition implies coincident lifetimes of the parts with the whole, and the whole is responsible for the creation and destruction of its parts; 3) a part may only belong to one whole at a time, although the part may be removed and attached to another whole, which then assumes responsibility for the part.
Figure 2.44 shows the full class diagram for the elevator system. All the classes we created are modeled, as well as the associations between these classes. [Note: In Chapter 9, we expand our class diagram by using the object-oriented concept of inheritance.]

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..1</td>
<td>Zero or one.</td>
</tr>
<tr>
<td>m..n</td>
<td>At least m, but not more than n.</td>
</tr>
<tr>
<td>*</td>
<td>Any non-negative integer.</td>
</tr>
<tr>
<td>0..*</td>
<td>Zero or more</td>
</tr>
<tr>
<td>1..*</td>
<td>One or more</td>
</tr>
</tbody>
</table>

Figure 2.43  Multiplicity table (part 2 of 2).

Fig. 2.44  Full class diagram for elevator simulation.
Class **Building** is represented near the top of the diagram and is composed of four classes, including **Clock** and **Scheduler**. These two classes make up the controller portion of the simulation. Class **Building** is also composed of class **Elevator** and class **Floor** (notice the one-to-two relationship between class **Building** and class **Floor**).

Classes **Floor** and **Elevator** are modeled near the bottom of the diagram. Class **Floor** is composed of one object each of classes **Light** and **FloorButton**. Class **Elevator** is composed of one object each of classes **ElevatorButton**, class **Door** and class **Bell**.

The classes involved in an association can also have **roles**. Roles help clarify the relationship between two classes. For example, class **Person** plays the “waiting passenger” role in its association with class **Floor** (because the person is waiting for the elevator.) Class **Person** plays the passenger role in its association with class **Elevator**. In a class diagram, the name of a class’s role is placed on either side of the association line, near the class’s rectangle. Each class in an association can play a different role.

The association between class **Floor** and class **Person** indicates that an object of class **Floor** can relate to zero or one objects of class **Person**. Class **Elevator** also relates to zero or one objects of class **Person**. The dashed line that bridges these two association lines indicates a **constraint** on the relationship between classes **Person**, **Floor** and **Elevator**. The constraint says that an object of class **Person** can participate in a relationship with an object of class **Floor** or with an object of class **Elevator**, but not both objects at the same time. The notation for this relationship is the word “xor” (which stands for “exclusive or”) placed inside braces. The association between class **Scheduler** and class **Person** states that one object of class **Scheduler** creates zero or more objects of class **Person**.

**Object Diagrams**

The UML also defines *object diagrams*, which are similar to class diagrams, except that they model objects and **links**—links are relationships between objects. Like class diagrams, object diagrams model the structure of the system. Object diagrams present a snapshot of the structure while the system is running—this provides information about which objects are participating in the system at a specific point in time.

Figure 2.45 models a snapshot of the system when no one is in the building (i.e., no objects of class **Person** exist in the system at this point in time). Object names are usually written in the form: `ObjectName:ClassName`. The first word in an object name is not capitalized, but subsequent words are. All object names in an object diagram are underlined. We omit the object name for some of the objects in the diagram (e.g., objects of class **FloorButton**). In large systems, many names of objects will be used in the model. This can cause cluttered, hard-to-read diagrams. If the name of a particular object is unknown or if it is not necessary to include the name (i.e., we only care about the type of the object), we can leave the object name out. In this instance, we simply display the colon and the class name.

---

3. The composite relationship between class **Building** and classes **Clock** and **Scheduler** represents a design decision on our part. We consider class **Building** to be part of both the “world” and the “controller” portions of our simulation. In our design we give the building the responsibility of running the simulation.

4. Constraints in UML diagrams can be written with what is known as the **Object Constraint Language (OCL)**. The OCL was created so that modelers could express constraints on a system in an clearly defined way. To learn more, visit [www-4.ibm.com/software/ad/standards/ocl.html](http://www-4.ibm.com/software/ad/standards/ocl.html).
Now we have identified the classes for this system (although we may discover others in later phases of the design process). We have also examined the system’s use case. In the “Thinking About Objects” section at the end of Chapter 3, we use this knowledge to examine how the system changes over time. As we expand our knowledge, we will also discover new information that will enable us to describe our classes in greater depth.

Notes

1. You will learn how to implement randomness in the next chapter (Chapter 3), where we study random number generation. Random number generation helps you simulate random processes like coin tossing and dice rolling. It will also help you simulate people arriving at random to use the elevator.

2. Because the real world is so object-oriented, it will be quite natural for you to pursue this project, even though you have not yet formally studied object orientation.

Questions

1. How might you decide whether the elevator is able to handle the anticipated traffic volume?

2. Why might it be more complicated to implement a three-story (or taller) building?

3. It is common for large buildings to have many elevators. We will see in Chapter 6 that once we have created one elevator object, it is easy to create as many as we want. What problems or opportunities do you foresee in having several elevators, each of which may pick up and discharge passengers at every floor in a large building?

4. For simplicity, we have given our elevator and each floor a capacity of one passenger. What problems or opportunities do you foresee in being able to increase these capacities?
SUMMARY

• A procedure for solving a problem in terms of the actions to be executed and the order in which these actions should be executed is called an algorithm.

• Specifying the order in which statements are to be executed in a computer program is called program control.

• Pseudocode helps the programmer “think out” a program before attempting to write it in a programming language such as C++.

• Declarations are messages to the compiler telling it the names and attributes of variables and telling it to reserve space for variables.

• A selection structure is used to choose among alternative courses of action.

• The if selection structure executes an indicated action only when the condition is true.

• The if/else selection structure specifies separate actions to be executed when the condition is true and when the condition is false.

• Whenever more than one statement is to be executed where normally only a single statement is expected, these statements must be enclosed in braces forming a compound statement. A compound statement can be placed anywhere a single statement can be placed.

• An empty statement indicating that no action is to be taken is indicated by placing a semicolon (;)

• A repetition structure specifies that an action is to be repeated while some condition remains true.

• The format for the while repetition structure is

    ```c++
    while ( condition )
    statement
    ```

• A value that contains a fractional part is referred to as a floating-point number and is represented by the data types float or double.

• The unary cast operator static_cast<double>( ) creates a temporary floating-point copy of its operand.

• C++ provides the arithmetic assignment operators +=, -=, *=, /= and %= that help abbreviate certain common types of expressions.

• C++ provides the increment (++) and decrement (--) operators to increment or decrement a variable by 1. If the operator is prefixed to the variable, the variable is incremented or decremented by 1 first, then used in its expression. If the operator is postfixed to the variable, the variable is used in its expression, then incremented or decremented by 1.

• A loop is a group of instructions the computer executes repeatedly until some terminating condition is satisfied. Two forms of repetition are counter-controlled repetition and sentinel-controlled repetition.

• A loop counter is used to count repetitions for a group of instructions. It is incremented (or decremented), usually by 1, each time the group of instructions is performed.

• Sentinel values are generally used to control repetition when the precise number of repetitions is not known in advance and the loop includes statements that obtain data each time the loop is performed. A sentinel value is entered after all valid data items have been supplied to the program. Sentinels should be different from valid data items.

• The for repetition structure handles all the details of counter-controlled repetition. The general format of the for structure is

    ```c++
    for ( initialization; loopContinuationTest; increment )
    statement
    ```
where *initialization* initializes the loop’s control variable, *loopContinuationTest* is the loop-continuation condition and *increment* increments the control variable.

- The **do/while** repetition structure tests the loop-continuation condition at the end of the loop, so the body of the loop will be executed at least once. The format for the **do/while** structure is

```c
  do
    statement
  while ( condition );
```

- The **break** statement, when executed in one of the repetition structures (**for**, **while** and **do/while**), causes immediate exit from the structure.

- The **continue** statement, when executed in one of the repetition structures (**for**, **while** and **do/while**), skips any remaining statements in the body of the structure and proceeds with the next iteration of the loop.

- The **switch** statement handles a series of decisions in which a particular variable or expression is tested for values it can assume and different actions are taken. In most programs, it is necessary to include a **break** statement after the statements for each **case**. Several **cases** can execute the same statements by listing the **case** labels together before the statements. The **switch** structure can only test constant integral expressions. It is not necessary to enclose a multistatement **case** in braces.

- On UNIX systems and many others, end-of-file is entered by typing the sequence

```c
  <ctrl-d>
```

on a line by itself. On VMS and DOS, end-of-file is entered by typing

```c
  <ctrl-z>
```

possibly followed by pressing the **Enter** key.

- Logical operators can be used to form complex conditions by combining conditions. The logical operators are **&&**, **||** and **!**, meaning logical AND, logical OR and logical NOT (negation), respectively.

- Any nonzero value implicitly converts to **true**; 0 (zero) implicitly converts to **false**.

### TERMINOLOGY

- 1 operator
- **&** operator
- **||** operator
- **++** operator
- **--** operator
- ?: operator
- action/decision model
- algorithm
- arithmetic assignment operators: `+=, -=, *=, /=` and `%=`
- ASCII character set
- block
- body of a loop
- **bool**
- **break**
- **case** label
- cast operator
- **char**
- **cin.get()** function
- compound statement
- conditional operator (?:)
- **continue**
- control structure
- counter-controlled repetition
- decrement operator (--)**
- **default** case in **switch**
- definite repetition
- definition
- delay loop
- **do/while** repetition structure
double  
double-selection structure  
empty statement (;)  
EOF  
false  
fatal error  
field width  
fixed-point format  
float  
for repetition structure  
garbage value  
if selection structure  
if/else selection structure  
increment operator (++)  
indefinite repetition  
infinite loop  
initialization  
integer division  
ios::fixed  
ios::left  
ios::showpoint  
keyword  
logic error  
logical AND (&&)  
logical negation (!)  
logical OR (||)  
long  
loop counter  
loop-continuation condition  
looping  
lvalue ("left value")  
multiple-selection structure  
nested control structures  
nonfatal error  

off-by-one error  
parameterized stream manipulator  
postdecrement operator  
postincrement operator  
pow function  
predicate operator  
predecrement operator  
preincrement operator  
pseudocode  
repetition  
replication structures  
rvalue ("right value")  
selection  
sentinel value  
sequential execution  
setiosflags stream manipulator  
setprecision stream manipulator  
setw stream manipulator  
short  
single-entry/single-exit control structures  
single-selection structure  
stacked control structures  
static_cast< type >()  
structured programming  
switch selection structure  
syntax error  
ternary operator  
top-down, stepwise refinement  
transfer of control  
true  
unary operator  
undefined value  
while repetition structure  
whitespace characters  
|| operator  

"Thinking About Objects" Terminology  
actor  
association  
association name  
class diagram  
composition  
constraint  
controller portion of a simulation  
identify the classes in a system  
link  
multiplicity  
Object Constraint Language (OCL)  
object diagram  
object-oriented analysis (OOA)  
object-oriented analysis and design (OOAD)  
optimized system requirements
common programming errors

2.1 Using a keyword as an identifier is a syntax error.
2.2 Forgetting one or both of the braces that delimit a compound statement can lead to syntax errors or logic errors in a program.
2.3 Placing a semicolon after the condition in an if structure leads to a logic error in single-selection if structures and a syntax error in double-selection if structures (if the if-part contains an actual body statement).
2.4 Not providing, in the body of a while structure, an action that eventually causes the condition in the while to become false normally results in an error called an “infinite loop” in which the repetition structure never terminates.
2.5 Spelling the keyword while with an uppercase W, as in While (remember that C++ is a case-sensitive language), is a syntax error. All of C++’s reserved keywords such as while, if and else contain only lowercase letters.
2.6 If a counter or total is not initialized, the results of your program will probably be incorrect. This is an example of a logic error.
2.7 In a counter-controlled loop, because the loop counter (when counting up by one each time through the loop) is one higher than its last legitimate value (i.e., 11 in the case of counting from 1 to 10), using the counter value in a calculation after the loop is often an off-by-one-error.
2.8 Choosing a sentinel value that is also a legitimate data value is a logic error.
2.9 An attempt to divide by zero causes a fatal error.
2.10 Using floating-point numbers in a manner that assumes they are represented precisely can lead to incorrect results. Floating-point numbers are represented only approximately by most computers.
2.11 Attempting to use the increment or decrement operator on an expression other than a simple variable name, e.g., writing ++(x + 1), is a syntax error.
2.12 Because floating-point values are approximate, controlling counting loops with floating-point variables can result in imprecise counter values and inaccurate tests for termination.
2.13 Using an incorrect relational operator or using an incorrect final value of a loop counter in the condition of a while or for structure can cause off-by-one errors.
2.14 When the control variable of a for structure is initially defined in the initialization section of the for structure header, using the control variable after the body of the structure is a syntax error.
2.15 Using commas instead of the two required semicolons in a for header is a syntax error.
2.16 Placing a semicolon immediately to the right of the right parenthesis of a for header makes the body of that for structure an empty statement. This is normally a logic error.
2.17 Not using the proper relational operator in the loop-continuation condition of a loop that counts downwards (such as incorrectly using i <= 1 in a loop counting down to 1) is usually a logic error that will yield incorrect results when the program runs.
2.18 Forgetting to include the <cmath> file in a program that uses math library functions is a syntax error.
2.19 Forgetting a break statement when one is needed in a switch structure is a logic error.
Omitting the space between the word `case` and the integral value being tested in a `switch` structure can cause a logic error. For example, writing `case3:` instead of writing `case 3:` simply creates an unused label. (We will say more about this in Chapter 18.) The problem is that the `switch` structure will not perform the appropriate actions when the `switch`'s controlling expression has a value of 3.

Not processing newline and other whitespace characters in the input when reading characters one at a time can cause logic errors.

Providing identical `case` labels in a `switch` structure is a syntax error.

Infinite loops are caused when the loop-continuation condition in a `while`, `for` or `do/while` structure never becomes `false`. To prevent this, make sure the value of the condition does change somewhere in the header or body of the loop so the condition can eventually become `false`.

Although $3 < x < 7$ is a mathematically correct condition, it does not evaluate correctly in C++. Use `(3 < x && x < 7)` to get the proper evaluation in C++.

In expressions using operator `&&`, it is possible that a condition—we will call this the dependent condition—might require another condition to be `true` for it to be meaningful to evaluate the dependent condition. In this case, the dependent condition should be placed after the other condition, or an error might occur.

Using operator `==` for assignment and using operator `=` for equality are logic errors.

**GOOD PROGRAMMING PRACTICES**

1. Consistently applying reasonable indentation conventions throughout your programs greatly improves program readability. We suggest a fixed-size tab of about 1/4 inch or three blanks per indent.

2. Pseudocode is often used to “think out” a program during the program-design process. Then the pseudocode program is converted to C++.

3. Indent both body statements of an `if/else` structure.

4. If there are several levels of indentation, each level should be indented the same additional amount of space.

5. Always putting the braces in an `if/else` structure (or any control structure) helps prevent their accidental omission, especially when adding statements to an `if` or `else` clause at a later time.

6. Some programmers prefer to type the beginning and ending braces of compound statements before typing the individual statements within the braces. This helps avoid omitting one or both of the braces.

7. Initialize counters and totals.

8. Declare each variable on a separate line.

9. When performing division by an expression whose value could be zero, explicitly test for this case and handle it appropriately in your program (such as by printing an error message) rather than allowing the fatal error to occur.

10. Prompt the user for each keyboard input. The prompt should indicate the form of the input and any special input values (like the sentinel value the user should enter to terminate a loop).

11. In a sentinel-controlled loop, the prompts requesting data entry should explicitly remind the user what the sentinel value is.

12. Do not compare floating-point values for equality or inequality. Rather, test that the absolute value of the difference is less than a specified small value.
2.13 Initializing variables when they are declared helps the programmer avoid the problems of uninitialized data.

2.14 Unary operators should be placed next to their operands with no intervening spaces.

2.15 Control counting loops with integer values.

2.16 Indent the statements in the body of each control structure.

2.17 Put a blank line before and after each control structure to make it stand out in the program.

2.18 Too many levels of nesting can make a program difficult to understand. As a general rule, try to avoid using more than three levels of indentation.

2.19 Vertical spacing above and below control structures, and indentation of the bodies of control structures within the control-structure headers give programs a two-dimensional appearance that greatly improves readability.

2.20 Using the final value in the condition of a while or for structure and using the $\leq$ relational operator will help avoid off-by-one errors. For a loop used to print the values 1 to 10, for example, the loop-continuation condition should be $\text{counter} \leq 10$ rather than $\text{counter} < 10$ (which is an off-by-one error) or $\text{counter} < 11$ (which is nevertheless correct). Many programmers nevertheless prefer so-called zero-based counting, in which, to count 10 times through the loop, $\text{counter}$ would be initialized to zero and the loop-continuation test would be $\text{counter} < 10$.

2.21 Place only expressions involving the control variables in the initialization and increment sections of a for structure. Manipulations of other variables should appear either before the loop (if they execute only once, like initialization statements) or in the loop body (if they execute once per repetition, like incrementing or decrementing statements).

2.22 Although the value of the control variable can be changed in the body of a for loop, avoid doing so because this practice can lead to subtle logic errors.

2.23 Although statements preceding a for and statements in the body of a for can often be merged into the for header, avoid doing so because it can make the program more difficult to read.

2.24 Limit the size of control structure headers to a single line, if possible.

2.25 Do not use variables of type float or double to perform monetary calculations. The imprecision of floating-point numbers can cause errors that will result in incorrect monetary values. In the exercises, we explore the use of integers to perform monetary calculations. Note: C++ class libraries from third-party vendors are available for properly performing monetary calculations.

2.26 Provide a default case in switch statements. Cases not explicitly tested in a switch statement without a default case are ignored. Including a default case focuses the programmer on the need to process exceptional conditions. There are situations in which no default processing is needed. Although the case clauses and the default case clause in a switch structure can occur in any order, it is considered a good programming practice to place the default clause last.

2.27 In a switch structure when the default clause is listed last, the break statement is not required. Some programmers include this break for clarity and symmetry with other cases.

2.28 Some programmers always include braces in a do while structure, even if the braces are not necessary. This helps eliminate ambiguity between the while structure and the do while structure containing one statement.

2.29 Some programmers feel that break and continue violate structured programming. Because the effects of these statements can be achieved by structured programming techniques we will soon learn, these programmers do not use break and continue.
PERFORMANCE TIPS

2.1 A nested if/else structure can be much faster than a series of single-selection if structures because of the possibility of early exit after one of the conditions is satisfied.

2.2 In a nested if/else structure, test the conditions that are more likely to be true at the beginning of the nested if/else structure. This will enable the nested if/else structure to run faster and exit earlier than will testing infrequently occurring cases first.

2.3 Programmers can write programs a bit faster and compilers can compile programs a bit faster when the “abbreviated” assignment operators are used. Some compilers generate code that runs faster when “abbreviated” assignment operators are used.

2.4 Many of the performance tips we mention in this text result in nominal improvements, so the reader might be tempted to ignore them. Significant performance improvement is often realized when a supposedly nominal improvement is placed in a loop that repeats many times.

2.5 Avoid placing expressions whose values do not change inside loops—but, even if you do, many of today’s sophisticated optimizing compilers will automatically place such expressions outside loops in the generated machine language code.

2.6 Many compilers contain optimization features that improve the code you write, but it is still better to write good code from the start.

2.7 In performance-oriented situations where memory is at a premium or execution speed is crucial, it might be desirable to use smaller integer sizes.

2.8 Using smaller integer sizes can result in a slower program if the machine’s instructions for manipulating them are not as efficient as for the natural-size integers (e.g., sign extension must be done on them).

2.9 The break and continue statements, when used properly, perform faster than the corresponding structured techniques we will soon learn.

2.10 In expressions using operator &&, if the separate conditions are independent of one another make the condition that is most likely to be false the leftmost condition. In expressions using operator ||, make the condition that is most likely to be true the leftmost condition. This can reduce a program’s execution time.

PORTABILITY TIPS

2.1 In the C++ standard, the scope of the control variable declared in the initialization section of a for structure is different from the scope in older C++ compilers. C++ code created with old C++ compilers can break when compiled on compilers that are compatible with the C++ standard. These are two defensive programming strategies that can be used to prevent this problem: Either define control variables with different names in every for structure, or, if you prefer to use the same name for the control variable in several for structures, define the control variable outside and before the first for loop.

2.2 The keystroke combinations for entering end-of-file are system dependent.

2.3 Testing for the symbolic constant EOF rather than –1 makes programs more portable. The ANSI standard states that EOF is a negative integral value (but not necessarily –1). Thus, EOF could have different values on different systems.

2.4 Because ints vary in size between systems, use long integers if you expect to process integers outside the range –32,768 to 32,767 and you would like to be able to run the program on several different computer systems.

2.5 For compatibility with earlier versions of the C++ standard, the bool value true can also be represented by any nonzero value and the bool value false can also be represented as the value 0.
SOFTWARE ENGINEERING OBSERVATIONS

2.1 Any C++ program we will ever build can be constructed from only seven different types of control structures (sequence, if, if/else, switch, while, do/while and for) combined in only two ways (control-structure stacking and control-structure nesting).

2.2 A compound statement can be placed anywhere in a program that a single statement can be placed.

2.3 Just as a compound statement can be placed anywhere a single statement can be placed, it is also possible to have no statement at all, i.e., the empty statement. The empty statement is represented by placing a semicolon (;) where a statement would normally be.

2.4 Each refinement, as well as the top itself, is a complete specification of the algorithm; only the level of detail varies.

2.5 Many programs can be divided logically into three phases: an initialization phase that initializes the program variables; a processing phase that inputs data values and adjusts program variables accordingly; and a termination phase that calculates and prints the final results.

2.6 The programmer terminates the top-down, stepwise refinement process when the pseudocode algorithm is specified in sufficient detail for the programmer to be able to convert the pseudocode to C++. Implementing the C++ program is then normally straightforward.

2.7 Experience has shown that the most difficult part of solving a problem on a computer is developing the algorithm for the solution. Once a correct algorithm has been specified, the process of producing a working C++ program from the algorithm is normally straightforward.

2.8 Many experienced programmers write programs without ever using program development tools like pseudocode. These programmers feel that their ultimate goal is to solve the problem on a computer and that writing pseudocode merely delays the production of final outputs. Although this may work for simple and familiar problems, it can lead to serious errors and delays on large, complex projects.

2.9 Placing a semicolon immediately after a for header is sometimes used to create a so-called delay loop. Such a for loop with an empty body still loops the indicated number of times doing nothing other than the counting. You might use a delay loop, for example, to slow down a program that is producing outputs on the screen too quickly for you to read them.

2.10 There is a tension between achieving quality software engineering and achieving the best-performing software. Often, one of these goals is achieved at the expense of the other.

TESTING AND DEBUGGING TIPS

2.1 Programmers normally write conditions such as \textit{x == 7} with the variable name on the left and the constant on the right. By reversing these so that the constant is on the left and the variable name is on the right as in \textit{7 == x}, the programmer who accidentally replaces the == operator with \textit{=} will be protected by the compiler. The compiler will treat this as a syntax error because only a variable name can be placed on the left-hand side of an assignment statement. At least this will prevent the potential devastation of a run-time logic error.

2.2 Use your text editor to search for all occurrences of = in your program and check that you have the correct operator in each place.
SELF-REVIEW EXERCISES

Exercises 2.1 through 2.10 correspond to Sections 2.1 through 2.12.
Exercises 2.11 through 2.13 correspond to Sections 2.13 through 2.21.

2.1 Answer each of the following questions.
   a) All programs can be written in terms of three types of control structures: _______, and _______.
   b) The _______ selection structure is used to execute one action when a condition is true and another action when that condition is false.
   c) Repeating a set of instructions a specific number of times is called _______ repetition.
   d) When it is not known in advance how many times a set of statements will be repeated, a _______ value can be used to terminate the repetition.

2.2 Write four different C++ statements that each add 1 to integer variable x.

2.3 Write C++ statements to accomplish each of the following:
   a) Assign the sum of x and y to z and increment the value of x by 1 after the calculation.
   b) Test if the value of the variable count is greater than 10. If it is, print “Count is greater than 10.”
   c) Decrement the variable x by 1 then subtract it from the variable total.
   d) Calculate the remainder after q is divided by divisor and assign the result to q. Write this statement two different ways.

2.4 Write a C++ statement to accomplish each of the following tasks.
   a) Declare variables sum and x to be of type int.
   b) Initialize variable x to 1.
   c) Initialize variable sum to 0.
   d) Add variable x to variable sum and assign the result to variable sum.
   e) Print “The sum is: ” followed by the value of variable sum.

2.5 Combine the statements that you wrote in Exercise 2.4 into a program that calculates and prints the sum of the integers from 1 to 10. Use the while repetition control structure. The loop should terminate when the value of x becomes 11.

2.6 Determine the values of each variable after the calculation is performed. Assume that, when each statement begins executing, all variables have the integer value 5.
   a) product *= x++;
   b) quotient /= ++x;

2.7 Write single C++ statements that do the following:
   a) Input integer variable x with cin and >.
   b) Input integer variable y with cin and >.
   c) Initialize integer variable i to 1.
   d) Initialize integer variable power to 1.
   e) Multiply variable power by x and assign the result to power.
   f) Increment variable y by 1.
   g) Test y to see if it is less than or equal to x.
   h) Output integer variable power with cout and <.

2.8 Write a C++ program that uses the statements in Exercise 2.7 to calculate x raised to the y power. The program should have a while repetition control structure.

2.9 Identify and correct the errors in each of the following:
   a) while ( c <= 5 ) {
      product *= c;
      ++c;
b) `cin << value;`

c) `if ( gender == 1 )`
    `cout << "Woman" << endl;`
    `else;`
    `cout << "Man" << endl;`

2.10 What is wrong with the following `while` repetition structure?

```
while ( z >= 0 )
    sum += z;
```

2.11 State whether the following are true or false. If the answer is false, explain why.

a) The `default` case is required in the `switch` selection structure.

b) The `break` statement is required in the `default` case of a `switch` selection structure to exit the structure properly.

c) The expression `(x > y && a < b)` is `true` if either the expression `x > y` is `true` or the expression `a < b` is `true`.

d) An expression containing the `||` operator is `true` if either or both of its operands are `true`.

2.12 Write a C++ statement or a set of C++ statements to accomplish each of the following:

a) Sum the odd integers between 1 and 99 using a `for` structure. Assume the integer variables `sum` and `count` have been declared.

b) Print the value `333.546372` in a field width of 15 characters with precisions of 1, 2 and 3. Print each number on the same line. Left-justify each number in its field. What three values print?

c) Calculate the value of `2.5` raised to the power `3` using the `pow` function. Print the result with a precision of 2 in a field width of 10 positions. What prints?

d) Print the integers from 1 to 20 using a `while` loop and the counter variable `x`. Assume that the variable `x` has been declared, but not initialized. Print only 5 integers per line. Hint: Use the calculation `x % 5`. When the value of this is 0, print a newline character, otherwise print a tab character.

e) Repeat Exercise 2.12 (d) using a `for` structure.

2.13 Find the error(s) in each of the following code segments and explain how to correct it (them).

a) `x = 1;`
   `while ( x <= 10 );`
   `x++;`

b) `for ( y = .1; y != 1.0; y += .1 )`
   `cout << y << endl;`

c) `switch ( n ) {`
   `case 1:`
   `cout << "The number is 1" << endl;`
   `case 2:`
   `cout << "The number is 2" << endl;`
   `break;`
   `default:`
   `cout << "The number is not 1 or 2" << endl;`
   `break;`

}`
d) The following code should print the values 1 to 10.
   \[
   n = 1;
   \text{while (} n < 10 \text{) }
   \]
   \[
   \text{cout} \ll n++ \ll \text{endl;}
   \]

ANSWERS TO SELF-REVIEW EXERCISES

2.1 a) Sequence, selection and repetition.  b) \textbf{if/else}.  c) Counter-controlled or definite.  
d) Sentinel, signal, flag or dummy.

2.2 \[
\begin{align*}
x &= x + 1; \\
x &= x += 1; \\
++x; \\
x++; \\
\end{align*}
\]

2.3 a) \[
z = x++ + y;
\]
   b) \[
\text{if (} \text{count} > 10 \text{) }
\]
   \[
   \text{cout} \ll \text{"Count is greater than 10"} \ll \text{endl;}
   \]
   c) \[
   \text{total} -= --x;
   \]
   d) \[
   q %= \text{divisor};
   \]
   \[
   q = q \% \text{divisor};
   \]

2.4 a) \[
\text{int sum, x;}
\]
   b) \[
x = 1;
\]
   c) \[
\text{sum} = 0;
\]
   d) \[
\text{sum} += x; \text{ or sum} = \text{sum} + x;
\]
   e) \[
\text{cout} \ll \text{"The sum is: "} \ll \text{sum} \ll \text{endl;}
\]

2.5 See below.

```cpp
1  // Calculate the sum of the integers from 1 to 10
2  #include <iostream>
3  using std::cout;
4  using std::endl;
5  int main()
6  {
7      int sum, x;
8      x = 1;
9      sum = 0;
10     while ( x <= 10 ) {
11         sum += x;
12         ++x;
13     }
14     cout << \"The sum is: \" << sum << \endl;
15     return 0;
16  }
```

2.6 a) \[
\text{product} = 25, x = 6;
\]
   b) \[
\text{quotient} = 0, x = 6;
\]

2.7 a) \[
\text{cin} \ll x;
\]
   b) \[
\text{cin} \ll y;
\]
   c) \[
i = 1;
\]
d) `power = 1;`

c) `power *= x;` or `power = power * x;`

e) `i++;`

f) `if ( i <= y )`

   g) `cout << power << endl;`

2.8 See below.

```cpp
// raise x to the y power
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

int main()
{
    int x, y, i, power;
    i = 1;
    power = 1;
    cout << "Enter base as an integer: ";
    cin >> x;
    cout << "Enter exponent as an integer: ";
    cin >> y;
    while ( i <= y ) {
        power *= x;
        ++i;
    }
    cout << power << endl;
    return 0;
}
```

2.9 a) Error: Missing the closing right brace of the `while` body.
   Correction: Add closing right brace after the statement `;++c;`.

b) Error: Used stream insertion instead of stream extraction.
   Correction: Change `<<` to `>>`.

c) Error: Semicolon after `else` results in a logic error. The second output statement will always be executed.
   Correction: Remove the semicolon after `else`.

2.10 The value of the variable `z` is never changed in the `while` structure. Therefore, if the loop-continuation condition (`z >= 0`) is `true`, an infinite loop is created. To prevent the infinite loop, `z` must be decremented so that it eventually becomes less than 0.

2.11 a) False. The `default` case is optional. If no default action is needed, then there is no need for a `default` case.

b) False. The `break` statement is used to exit the `switch` structure. The `break` statement is not required when the `default` case is the last case.
c) False. Both of the relational expressions must be **true** in order for the entire expression to be **true** when using the **&&** operator.

d) True.

### 2.12

a)  
   ```
   sum = 0;
   for ( count = 1; count <= 99; count += 2 )
       sum += count;
   ```

b)  
   ```
   cout << setiosflags(ios::fixed | ios::showpoint | ios::left)
       << setprecision( 1 ) << setw( 15 ) << 333.546372
       << setprecision( 2 ) << setw( 15 ) << 333.546372
       << setprecision( 3 ) << setw( 15 ) << 333.546372
       << endl;
   ```

   Output is:
   
   333.5          333.55         333.546

   c)  
   ```
   cout << setiosflags( ios::fixed | ios::showpoint )
       << setprecision( 2 ) << setw( 10 ) << pow( 2.5, 3 )
       << endl;
   ```

   Output is:
   
   15.63

d)  
   ```
   x = 1;
   while ( x <= 20 ) {
       cout << x;
       if ( x % 5 == 0 )
           cout << endl;
       else
           cout << '	';
       x++;
   }
   ```

e)  
   ```
   for ( x = 1; x <= 20; x++ ) {
       cout << x;
       if ( x % 5 == 0 )
           cout << endl;
       else
           cout << '	';
   }
   ```

   or

   ```
   for ( x = 1; x <= 20; x++ )
       if ( x % 5 == 0 )
           cout << x << endl;
       else
           cout << x << '\t';
   ```

### 2.13

a) Error: The semicolon after the **while** header causes an infinite loop.
   Correction: Replace the semicolon by a {, or remove both the ; and the }.

b) Error: Using a floating-point number to control a **for** repetition structure.
   Correction: Use an integer and perform the proper calculation in order to get the values you desire.

   ```
   for ( y = 1; y != 10; y++ )
       cout << ( static_cast< double >( y ) / 10 ) << endl;
   ```
c) Error: Missing `break` statement in the statements for the first `case`. 
Correction: Add a `break` statement at the end of the statements for the first `case`. Note that this is not necessarily an error if the programmer wants the statement of `case 1:` to execute every time the `case 1:` statement executes.

d) Error: Improper relational operator used in the while repetition-continuation condition. 
Correction: Use `<=` rather than `<`, or change 10 to 11.

**EXERCISES**

*Exercises 2.14 through 2.38 correspond to Sections 2.1 through 2.12.*
*Exercises 2.39 through 2.63 correspond to Sections 2.13 through 2.21.*

**2.14** Identify and correct the error(s) in each of the following:

a) `if ( age >= 65 );`
   `cout << "Age is greater than or equal to 65" << endl;`
   `else`
   `cout << "Age is less than 65 << endl;`  

b) `if ( age >= 65 )`
   `cout << "Age is greater than or equal to 65" << endl;`
   `else;`
   `cout << "Age is less than 65 << endl;`  

c) `int x = 1, total;`
   `while ( x <= 10 ) {`
   `   total += x;`
   `   ++x;`
   `}`

d) `While ( x <= 100 )`
   `total += x;`
   `++x;`

e) `while ( y > 0 ) {`
   `   cout << y << endl;`
   `   ++y;`
   `}`

**2.15** What does the following program print?

```cpp
#include <iostream>
using std::cout;
using std::endl;

int main()
{
   int y, x = 1, total = 0;

   while ( x <= 10 ) {
      y = x * x;
      cout << y << endl;
      total += y;
      ++x;
   }
```
For Exercises 2.16 to 2.19, perform each of these steps:
  a) Read the problem statement.
  b) Formulate the algorithm using pseudocode and top-down, stepwise refinement.
  c) Write a C++ program.
  d) Test, debug and execute the C++ program.

2.16 Drivers are concerned with the mileage obtained by their automobiles. One driver has kept track of several tankfuls of gasoline by recording miles driven and gallons used for each tankful. Develop a C++ program that will input the miles driven and gallons used for each tankful. The program should calculate and display the miles per gallon obtained for each tankful. After processing all input information, the program should calculate and print the combined miles per gallon obtained for all tankfuls.

```
16 cout << "Total is " << total << endl;
18 return 0;
19 }
```

```
Enter the gallons used (-1 to end): 12.8
Enter the miles driven: 287
The miles / gallon for this tank was 22.421875

Enter the gallons used (-1 to end): 10.3
Enter the miles driven: 200
The miles / gallon for this tank was 19.417475

Enter the gallons used (-1 to end): 5
Enter the miles driven: 120
The miles / gallon for this tank was 24.000000

Enter the gallons used (-1 to end): -1
The overall average miles/gallon was 21.601423
```

2.17 Develop a C++ program that will determine if a department store customer has exceeded the credit limit on a charge account. For each customer, the following facts are available:
  a) Account number (an integer)
  b) Balance at the beginning of the month
  c) Total of all items charged by this customer this month
  d) Total of all credits applied to this customer’s account this month
  e) Allowed credit limit

The program should input each of these facts, calculate the new balance (= beginning balance + charges – credits) and determine if the new balance exceeds the customer's credit limit. For those customers whose credit limit is exceeded, the program should display the customer's account number, credit limit, new balance and the message “Credit limit exceeded.”
2.18 One large chemical company pays its salespeople on a commission basis. The salespeople receive $200 per week plus 9 percent of their gross sales for that week. For example, a salesperson who sells $5000 worth of chemicals in a week receives $200 plus 9 percent of $5000, or a total of $650. Develop a C++ program that will input each salesperson’s gross sales for last week and calculate and display that salesperson’s earnings. Process one salesperson’s figures at a time.

2.19 Develop a C++ program that will determine the gross pay for each of several employees. The company pays “straight-time” for the first 40 hours worked by each employee and pays “time-and-a-half” for all hours worked in excess of 40 hours. You are given a list of the employees of the company, the number of hours each employee worked last week and the hourly rate of each employee. Your program should input this information for each employee and should determine and display the employee’s gross pay.
2.20 The process of finding the largest number (i.e., the maximum of a group of numbers) is used frequently in computer applications. For example, a program that determines the winner of a sales contest would input the number of units sold by each salesperson. The salesperson who sells the most units wins the contest. Write a pseudocode program, then a C++ program that inputs a series of 10 numbers, and determines and prints the largest of the numbers. Hint: Your program should use three variables, as follows:

- **counter**: A counter to count to 10 (i.e., to keep track of how many numbers have been input and to determine when all 10 numbers have been processed).
- **number**: The current number input to the program.
- **largest**: The largest number found so far.

2.21 Write a C++ program that utilizes looping and the tab escape sequence `\t` to print the following table of values:

<table>
<thead>
<tr>
<th>N</th>
<th>10*N</th>
<th>100*N</th>
<th>1000*N</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>100</td>
<td>1000</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>200</td>
<td>2000</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>300</td>
<td>3000</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>400</td>
<td>4000</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>500</td>
<td>5000</td>
</tr>
</tbody>
</table>

2.22 Using an approach similar to Exercise 2.20, find the two largest values among the 10 numbers. Note: You must input each number only once.

2.23 Modify the program in Fig. 2.11 to validate its inputs. On any input, if the value entered is other than 1 or 2, keep looping until the user enters a correct value.

2.24 What does the following program print?

```c++
#include <iostream>
using std::cout;
using std::endl;
```
Chapter 2
Control Structures

2.25 What does the following program print?

```cpp
int main()
{
    int count = 1;
    while ( count <= 10 ) {
        cout << (count % 2 ? "****" : "+++++++")
             << endl;
        ++count;
    }
    return 0;
}
```

2.26 (Dangling Else Problem) Determine the output for each of the following when `x` is 9 and `y` is 11 and when `x` is 11 and `y` is 9. Note that the compiler ignores the indentation in a C++ program. Also, the C++ compiler always associates an `else` with the previous `if` unless told to do otherwise by the placement of braces `{}`. Because, on first glance, the programmer may not be sure which `if` an `else` matches, this is referred to as the “dangling else” problem. We have eliminated the indentation from the following code to make the problem more challenging. (Hint: Apply indentation conventions you have learned.)

a) `if ( x < 10 )`
   `if ( y > 10 )`
   `cout << "*****" << endl;`
   `else`
   `cout << "#######" << endl;`
   `cout << "$$$$$$" << endl;`
b) \[
\begin{align*}
\text{if (} & \ x < 10 \ \text{)} \{ \\
& \text{if (} \ y > 10 \ \text{)} \\
& \quad \text{cout} \ \ll \ "*****" \ \ll \ \text{endl;} \\
& \} \\
& \quad \text{else} \{} \\
& \quad \quad \text{cout} \ \ll \ "#####" \ \ll \ \text{endl;} \\
& \quad \quad \text{cout} \ \ll \ "$$$$$$" \ \ll \ \text{endl;} \\
& \} \\
\end{align*}
\]

2.27 (Another Dangling Else Problem) Modify the following code to produce the output shown. Use proper indentation techniques. You must not make any changes other than inserting braces. The compiler ignores indentation in a C++ program. We have eliminated the indentation from the following code to make the problem more challenging. Note: It is possible that no modification is necessary.

\[
\begin{align*}
\text{if (} & \ y == 8 \ \text{)} \\
& \text{if (} \ x == 5 \ \text{)} \\
& \quad \text{cout} \ \ll \ "@@@@@" \ \ll \ \text{endl;} \\
& \text{else} \\
& \quad \text{cout} \ \ll \ "#####" \ \ll \ \text{endl;} \\
& \quad \text{cout} \ \ll \ "$$$$$$" \ \ll \ \text{endl;} \\
& \quad \text{cout} \ \ll \ "&&&&&" \ \ll \ \text{endl;} \\
\end{align*}
\]

a) Assuming \( x = 5 \) and \( y = 8 \), the following output is produced.

\[
\begin{array}{cccc}
@&&&@ \\
@&&@@ \\
@&&@@ \\
\end{array}
\]

b) Assuming \( x = 5 \) and \( y = 8 \), the following output is produced.

\[
\begin{array}{cccc}
@&&&@ \\
@&&&@ \\
\end{array}
\]

c) Assuming \( x = 5 \) and \( y = 8 \), the following output is produced.

\[
\begin{array}{cccc}
@&&&@ \\
&&&& \\
\end{array}
\]

d) Assuming \( x = 5 \) and \( y = 7 \), the following output is produced. Note: The last three output statements after the \textit{else} are all part of a compound statement.

\[
\begin{array}{cccc}
@&&&@ \\
@&&@@ \\
&&&& \\
\end{array}
\]
2.28 Write a program that reads in the size of the side of a square and then prints a hollow square of that size out of asterisks and blanks. Your program should work for squares of all side sizes between 1 and 20. For example, if your program reads a size of 5, it should print

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

2.29 A palindrome is a number or a text phrase that reads the same backwards as forwards. For example, each of the following five-digit integers is a palindrome: 12321, 55555, 45554 and 11611. Write a program that reads in a five-digit integer and determines whether it is a palindrome. (Hint: Use the division and modulus operators to separate the number into its individual digits.)

2.30 Input an integer containing only 0s and 1s (i.e., a “binary” integer) and print its decimal equivalent. (Hint: Use the modulus and division operators to pick off the “binary” number’s digits one at a time from right to left. Just as in the decimal number system where the rightmost digit has a positional value of 1 and the next digit left has a positional value of 10, then 100, then 1000, etc., in the binary number system, the rightmost digit has a positional value of 1, the next digit left has a positional value of 2, then 4, then 8, etc. Thus the decimal number 234 can be interpreted as $4 \times 1 + 3 \times 10 + 2 \times 100$. The decimal equivalent of binary 1101 is $1 \times 1 + 0 \times 2 + 1 \times 4 + 1 \times 8$ or $1 + 0 + 4 + 8$, or 13.)

2.31 Write a program that displays the following checkerboard pattern

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

Your program must use only three output statements, one of each of the following forms:

```
cout << "* ";
cout << " ";
cout << endl;
```

2.32 Write a program that keeps printing the multiples of the integer 2, namely 2, 4, 8, 16, 32, 64, etc. Your loop should not terminate (i.e., you should create an infinite loop). What happens when you run this program?

2.33 Write a program that reads the radius of a circle (as a `double` value) and computes and prints the diameter, the circumference and the area. Use the value 3.14159 for $\pi$.

2.34 What's wrong with the following statement? Provide the correct statement to accomplish what the programmer was probably trying to do.

```
cout << ++( x + y );
```
2.35 Write a program that reads three nonzero double values and determines and prints if they could represent the sides of a triangle.

2.36 Write a program that reads three nonzero integers and determines and prints if they could be the sides of a right triangle.

2.37 A company wants to transmit data over the telephone, but they are concerned that their phones may be tapped. All of their data are transmitted as four-digit integers. They have asked you to write a program that encrypts their data so that it can be transmitted more securely. Your program should read a four-digit integer and encrypt it as follows: Replace each digit by \((the\ sum\ of\ that\ digit\ plus\ 7)\) modulus 10. Then, swap the first digit with the third, swap the second digit with the fourth and print the encrypted integer. Write a separate program that inputs an encrypted four-digit integer and decrypts it to form the original number.

2.38 The factorial of a nonnegative integer \(n\) is written \(n!\) (pronounced “\(n\) factorial”) and is defined as follows:

\[
!n = n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1 \quad (\text{for values of } n \text{ greater than or equal to } 1)
\]

and

\[
!n = 1 \quad (\text{for } n = 0).
\]

For example, \(5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1\), which is 120.

a) Write a program that reads a nonnegative integer and computes and prints its factorial.

b) Write a program that estimates the value of the mathematical constant \(e\) by using the formula:

\[
e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots
\]

c) Write a program that computes the value of \(e^x\) by using the formula

\[
e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots
\]

2.39 Find the error(s) in each of the following:

a) For ( \(x = 100\), \(x >= 1\), \(x++\) )

\[
cout \ll x \ll endl;
\]

b) The following code should print whether integer value is odd or even:

\[
switch ( value % 2 ) {
\]

\[
\begin{align*}
case 0: & \\
case 1: & \\
\end{align*}
\]

\[
cout \ll "Even integer\" \ll endl;
\]

\[
cout \ll "Odd integer\" \ll endl;
\]

}\n\]

\[
c}\n\]

\[
c} While ( counter < 100 );
\]
2.40 Write a program that sums a sequence of integers. Assume that the first integer read specifies the number of values remaining to be entered. Your program should read only one value per input statement. A typical input sequence might be

```
5 100 200 300 400 500
```

where the 5 indicates that the subsequent 5 values are to be summed.

2.41 Write a program that calculates and prints the average of several integers. Assume the last value read is the sentinel 9999. A typical input sequence might be

```
10 8 11 7 9 9999
```

indicating that the average of all the values preceding 9999 is to be calculated.

2.42 What does the following program do?

2.43 Write a program that finds the smallest of several integers. Assume that the first value read specifies the number of values remaining and that the first number is not one of the integers to compare.

2.44 Write a program that calculates and prints the product of the odd integers from 1 to 15.

2.45 The factorial function is used frequently in probability problems. The factorial of a positive integer \( n \) (written \( n! \) and pronounced “\( n \) factorial”) is equal to the product of the positive integers from 1 to \( n \). Write a program that evaluates the factorials of the integers from 1 to 5. Print the results in tabular format. What difficulty might prevent you from calculating the factorial of 20?

2.46 Modify the compound interest program of Section 2.15 to repeat its steps for interest rates of 5 percent, 6 percent, 7 percent, 8 percent, 9 percent and 10 percent. Use a `for` loop to vary the interest rate.

```cpp
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

int main()
{
    int x, y;

    cout << "Enter two integers in the range 1-20: ";
    cin >> x >> y;

    for ( int i = 1; i <= y; i++ ) {
        for ( int j = 1; j <= x; j++ )
            cout << '@';
        cout << endl;
    }

    return 0;
}
```
2.47  Write a program that prints the following patterns separately one below the other. Use for loops to generate the patterns. All asterisks (*) should be printed by a single statement of the form cout << '*'; (this causes the asterisks to print side by side). Hint: The last two patterns require that each line begin with an appropriate number of blanks. Extra credit: Combine your code from the four separate problems into a single program that prints all four patterns side by side by making clever use of nested for loops.

(A) *  (B) **********  (C) ********** *  (D) *
    **  *********  **
    ***  ********
    ****  *******
    *****  ******
    ******  *****
    *******  ****
    ********  **
    ********* *
    ***********

2.48 One interesting application of computers is drawing graphs and bar charts (sometimes called “histograms”). Write a program that reads five numbers (each between 1 and 30). For each number read, your program should print a line containing that number of adjacent asterisks. For example, if your program reads the number seven, it should print ********.

2.49 A mail order house sells five different products whose retail prices are product 1 — $2.98, product 2 — $4.50, product 3 — $9.98, product 4 — $4.49 and product 5 — $6.87. Write a program that reads a series of pairs of numbers as follows:
   a) Product number
   b) Quantity sold for one day

Your program should use a switch statement to help determine the retail price for each product. Your program should calculate and display the total retail value of all products sold last week.

2.50 Modify the program of Fig. 2.22 so that it calculates the grade-point average for the class. A grade of ‘A’ is worth 4 points, ‘B’ is worth 3 points, etc.

2.51 Modify the program in Fig. 2.21 so it uses only integers to calculate the compound interest. (Hint: Treat all monetary amounts as integral numbers of pennies. Then “break” the result into its dollar portion and cents portion by using the division and modulus operations. Insert a period.)

2.52 Assume i = 1, j = 2, k = 3 and m = 2. What does each of the following statements print? Are the parentheses necessary in each case?
   a) cout << ( i == 1 ) << endl;
   b) cout << ( j == 3 ) << endl;
   c) cout << ( i >= 1 && j < 4 ) << endl;
   d) cout << ( m <= 99 && k < m ) << endl;
   e) cout << ( j >= i || k == m ) << endl;
   f) cout << ( k + m < j || 3 - j >= k ) << endl;
   g) cout << ( !m ) << endl;
   h) cout << ( !( j - m ) ) << endl;
   i) cout << ( !( k > m ) ) << endl;

2.53 Write a program that prints a table of the binary, octal and hexadecimal equivalents of the decimal numbers in the range 1 through 256. If you are not familiar with these number systems, read Appendix C first.
2.54 Calculate the value of $\pi$ from the infinite series

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \cdots$$

Print a table that shows the value of $\pi$ approximated by 1 term of this series, by two terms, by three terms, etc. How many terms of this series do you have to use before you first get 3.14? 3.141? 3.1415? 3.14159?

2.55 (Pythagorean Triples) A right triangle can have sides that are all integers. The set of three integer values for the sides of a right triangle is called a Pythagorean triple. These three sides must satisfy the relationship that the sum of the squares of two of the sides is equal to the square of the hypotenuse. Find all Pythagorean triples for side1, side2 and hypotenuse all no larger than 500. Use a triple-nested for-loop that tries all possibilities. This is an example of “brute force” computing. You will learn in more advanced computer science courses that there are many interesting problems for which there is no known algorithmic approach other than using sheer brute force.

2.56 A company pays its employees as managers (who receive a fixed weekly salary), hourly workers (who receive a fixed hourly wage for up to the first 40 hours they work and “time-and-a-half,” i.e., 1.5 times their hourly wage, for overtime hours worked), commission workers (who receive $250 plus 5.7% of their gross weekly sales), or pieceworkers (who receive a fixed amount of money per item for each of the items they produce—each pieceworker in this company works on only one type of item). Write a program to compute the weekly pay for each employee. You do not know the number of employees in advance. Each type of employee has its own pay code: Managers have paycode 1, hourly workers have code 2, commission workers have code 3 and pieceworkers have code 4. Use a switch to compute each employee’s pay based on that employee’s paycode. Within the switch, prompt the user (i.e., the payroll clerk) to enter the appropriate facts your program needs to calculate each employee’s pay based on that employee’s paycode.

2.57 (De Morgan’s Laws) In this chapter, we discussed the logical operators &&, || and !. De Morgan’s Laws can sometimes make it more convenient for us to express a logical expression. These laws state that the expression $!(\text{condition1} \&\& \text{condition2})$ is logically equivalent to the expression $(\text{!condition1} || \text{!condition2})$. Also, the expression $!(\text{condition1} || \text{condition2})$ is logically equivalent to the expression $(\text{!condition1} \&\& \text{!condition2})$. Use De Morgan’s Laws to write equivalent expressions for each of the following, then write a program to show that both the original expression and the new expression in each case are equivalent:

a) $!(\ x < 5 \ ) \&\& \ !(\ y >= 7 \ )$

b) $!(\ a == b \ ) || \ !(\ g != 5 \ )$

c) $!(\ (\ x <= 8 \ ) \&\& \ (\ y > 4 \ ) \ )$

d) $!(\ (\ i > 4 \ ) || (\ j <= 6 \ ) \ )$

2.58 Write a program that prints the following diamond shape. You may use output statements that print either a single asterisk (*) or a single blank. Maximize your use of repetition (with nested for structures) and minimize the number of output statements.
2.59 Modify the program you wrote in Exercise 2.58 to read an odd number in the range 1 to 19 to specify the number of rows in the diamond. Your program should then display a diamond of the appropriate size.

2.60 A criticism of the `break` statement and the `continue` statement is that each is unstructured. Actually `break` statements and `continue` statements can always be replaced by structured statements, although doing so can be awkward. Describe in general how you would remove any `break` statement from a loop in a program and replace that statement with some structured equivalent. (Hint: The `break` statement leaves a loop from within the body of the loop. The other way to leave is by failing the loop-continuation test. Consider using in the loop-continuation test a second test that indicates “early exit because of a ‘break’ condition.”) Use the technique you developed here to remove the break statement from the program of Fig. 2.26.

2.61 What does the following program segment do?

```cpp
for ( i = 1; i <= 5; i++ ) {
    for ( j = 1; j <= 3; j++ ) {
        for ( k = 1; k <= 4; k++ )
            cout << '*';
        cout << endl;
    }
    cout << endl;
}
```

2.62 Describe in general how you would remove any `continue` statement from a loop in a program and replace that statement with some structured equivalent. Use the technique you developed here to remove the `continue` statement from the program of Fig. 2.27.

2.63 (“The Twelve Days of Christmas” Song) Write a program that uses repetition and `switch` structures to print the song “The Twelve Days of Christmas.” One `switch` structure should be used to print the day (i.e., “First,” “Second,” etc.). A separate `switch` structure should be used to print the remainder of each verse.

**Exercise 2.64 corresponds to Section 2.22, “Thinking About Objects.”**

2.64 Describe in 200 words or less what an automobile is and does. List the nouns and verbs separately. In the text, we stated that each noun might correspond to an object that will need to be built to implement a system, in this case a car. Pick five of the objects you listed, and, for each, list several attributes and several behaviors. Describe briefly how these objects interact with one another and other objects in your description. You have just performed several of the key steps in a typical object-oriented design.

2.65 (Peter Minuit Problem) Legend has it that in 1626 Peter Minuit purchased Manhattan for $24.00 in barter. Did he make a good investment? To answer this question, modify the compound interest program of Fig. 2.21 to begin with a principal of $24.00 and to calculate the amount of interest on deposit if that money had been kept on deposit until this year (374 years through 2000). Run the program with interest rates of 5%, 6%, 7%, 8%, 9% and 10% to observe the wonders of compound interest.