3

Control Statements: Part 1

Let's all move one place on.
—Lewis Carroll

The wheel is come full circle.
—William Shakespeare

How many apples fell on Newton's head before he took the hint!
—Robert Frost

All the evolution we know of proceeds from the vague to the definite.
—Charles Sanders Peirce

Objectives

In this chapter you'll learn:

- Basic problem-solving techniques.
- To develop algorithms through the process of top-down, stepwise refinement.
- To use the `if` and `if...else` selection statements to choose among alternative actions.
- To use the `while` repetition statement to execute statements in a program repeatedly.
- Counter-controlled repetition and sentinel-controlled repetition.
- To use the increment, decrement and assignment operators.

Outline

3.1 Introduction

Before writing a program to solve a problem, we must have a thorough understanding of the problem and a carefully planned approach to solving it. When writing a program, we must also understand the types of building blocks that are available and employ proven program construction techniques. In this chapter and in Chapter 4, Control Statements: Part 2, we discuss these issues as we present the theory and principles of structured programming. The concepts presented here are crucial to building effective classes and manipulating objects.

In this chapter, we introduce C++'s `if`, `if...else` and `while` statements, three of the building blocks that allow you to specify the logic required for functions to perform their tasks. We introduce C++'s assignment operators and explore C++'s increment and decrement operators. These additional operators abbreviate and simplify many program statements.

3.2 Algorithms

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

1. the **actions** to execute and
2. the **order** in which the actions execute

is called an **algorithm**. The following example demonstrates that correctly specifying the order in which the actions execute 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 (actions) execute in a computer program is called **program control**. This chapter investigates program control using C++'s **control statements**.

3.3 Pseudocode

Pseudocode (or "fake" code) is an artificial and informal language that helps you develop algorithms without having to worry about the strict details of C++ language syntax. The pseudocode we present here is particularly useful for developing algorithms that will be converted to structured portions of C++ programs. Pseudocode is similar to everyday English; it's convenient and user friendly, although it isn't an actual computer programming language.

Pseudocode does not execute on computers. Rather, it helps you "think out" a program before attempting to write it in a programming language, such as C++. This chapter provides several examples of how to use pseudocode to develop C++ programs.

The style of pseudocode we present consists purely of characters, so you can type pseudocode conveniently, using any editor program. The computer can produce a freshly printed copy of a pseudocode program on demand. A carefully prepared pseudocode program can easily be converted to a corresponding C++ program. In many cases, this simply requires replacing pseudocode statements with C++ equivalents.

Pseudocode normally describes only **executable statements**, which cause specific actions to occur after a programmer converts a program from pseudocode to C++ and the program is run on a computer. We typically do not include variable declarations in our pseudocode. However, some programmers choose to list variables and mention their purposes at the beginning of pseudocode programs.

Let's look at an example of pseudocode that may be written to help a programmer create the addition program of Fig. 2.5. This pseudocode (Fig. 3.1) corresponds to the algorithm that inputs two integers from the user, adds these integers and displays their sum. Although we show the complete pseudocode listing here, we'll show how to *create* pseudocode from a problem statement later in the chapter.

```
1 Prompt the user to enter the first integer
2 Input the first integer
3
4 Prompt the user to enter the second integer
5 Input the second integer
6
7 Add first integer and second integer, store result
8 Display result
```

Fig. 3.1 | Pseudocode for the addition program of Fig. 2.5.

Lines 1–2 correspond to the statements in lines 13–14 of Fig. 2.5. Notice that the pseudocode statements are simply English statements that convey what task is to be performed in C++. Likewise, lines 4–5 correspond to the statements in lines 16–17 of Fig. 2.5 and lines 7–8 correspond to the statements in lines 19 and 21 of Fig. 2.5.

Notice that the pseudocode in Fig. 3.1 corresponds to code only in function `main`. This occurs because pseudocode is normally used for algorithms, not complete programs. In this case, the pseudocode represents the algorithm. The function in which this code is placed is not important to the algorithm itself.

3.4 Control Structures

Normally, statements in a program execute one after the other in the order in which they're written. This is called **sequential execution**. Various C++ statements we'll soon discuss enable you to specify that the next statement to execute 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**, which allows you to specify a transfer of control to one of a wide range of possible destinations in a program (creating what's often called "spaghetti code"). The notion of so-called **structured programming** became almost synonymous with "**goto elimination**."

The research of Böhm and Jacopini[1] demonstrated that programs could be written without any `goto` statements. It became the challenge of the era 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, are easier to debug, test and modify and are more likely to be bug-free in the first place.

Böhm 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 term "control structures" comes from the field of computer science. When we introduce C++'s implementations of control structures, we'll refer to them in the terminology of the C++ standard document as "control statements."

Sequence Structure in C++

The sequence statement is built into C++. Unless directed otherwise, the computer executes C++ statements one after the other in the order in which they're written—that is, in sequence. The Unified Modeling Language (UML) **activity diagram** of Fig. 3.2 illustrates

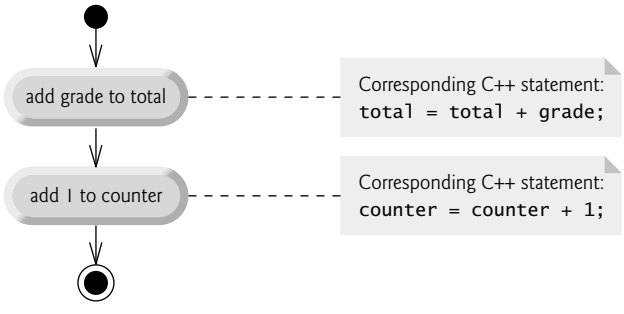

Fig. 3.2 | Sequence-statement activity diagram.

1. Böhm, C., and G. Jacopini, "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules," *Communications of the ACM*, Vol. 9, No. 5, May 1966, pp. 366–371.

a typical sequence statement in which two calculations are performed in order. C++ allows us to have as many actions as we want in a sequence statement. As we'll soon see, anywhere a single action may be placed, we may place several actions in sequence.

In this figure, the two statements add a grade to a `total` variable and add the value 1 to a `counter` variable. Such statements might appear in a program that averages several student grades. To calculate an average, the total of the grades being averaged is divided by the number of grades. A counter variable would be used to keep track of the number of values being averaged. You'll see similar statements in programs we develop in this chapter.

Activity diagrams are part of the UML. An activity diagram models the **workflow** (also called the **activity**) of a portion of a software system. Such workflows may include a portion of an algorithm, such as the sequence statement in Fig. 3.2. Activity diagrams are composed of special-purpose symbols, such as **action state symbols** (a rectangle with its left and right sides replaced with arcs curving outward), **diamonds** and **small circles**; these symbols are connected by **transition arrows**, which represent the flow of the activity.

Activity diagrams help you develop and represent algorithms, but many programmers prefer pseudocode. Activity diagrams clearly show how control statements operate.

Consider the sequence-statement activity diagram of Fig. 3.2. It contains two **action states** that represent actions to perform. Each action state contains an **action expression**—e.g., "add grade to total" or "add 1 to counter"—that specifies a particular action to perform. Other actions might include calculations or input/output operations. The arrows in the activity diagram are called transition arrows. These arrows represent **transitions**, which indicate the order in which the actions represented by the action states occur—the program that implements the activities illustrated by the activity diagram in Fig. 3.2 first adds `grade` to `total`, then adds `1` to `counter`.

The **solid circle** at the top of the diagram represents the activity's **initial state**—the beginning of the workflow before the program performs the modeled activities. The solid circle surrounded by a hollow circle that appears at the bottom of the activity diagram represents the **final state**—the end of the workflow after the program performs its activities.

Figure 3.2 also includes rectangles with the upper-right corners folded over. These are called **notes** in the UML. Notes are explanatory remarks that describe the purpose of symbols in the diagram. Figure 3.2 uses UML notes to show the C++ code associated with each action state in the activity diagram. A **dotted line** connects each note with the element that the note describes. Activity diagrams normally do not show the C++ code that implements the activity. We use notes for this purpose here to illustrate how the diagram relates to C++ code.

Selection Statements in C++

C++ provides three types of selection statements (discussed in this chapter and Chapter 4). The `if` selection statement either performs (selects) an action if a condition (predicate) is true or skips the action if the condition is false. The `if...else` selection statement performs an action if a condition is true or performs a different action if the condition is false. The `switch` selection statement (Chapter 4) performs one of many different actions, depending on the value of an integer expression.

The `if` selection statement is a **single-selection statement** because it selects or ignores a single action (or, as we'll soon see, a single group of actions). The `if...else` statement is called a **double-selection statement** because it selects between two different actions (or

groups of actions). The `switch` selection statement is called a **multiple-selection statement** because it selects among many different actions (or groups of actions).

Repetition Statements in C++

C++ provides three types of repetition statements (also called **looping statements** or **loops**) for performing statements repeatedly while a condition (called the **loop-continuation condition**) remains true. These are the **`while`**, **`do...while`** and **`for`** statements. (Chapter 4 presents the `do...while` and `for` statements.) The `while` and `for` statements perform the action (or group of actions) in their bodies zero or more times—if the loop-continuation condition is initially false, the action (or group of actions) will not execute. The `do...while` statement performs the action (or group of actions) in its body at least once.

Each of the words `if`, `else`, `switch`, `while`, `do` and `for` is a C++ keyword. These words are reserved by the C++ programming language to implement various features, such as C++'s control statements. Figure 3.3 provides a complete list of C++ keywords.

C++ Keywords

Keywords common to the C and C++ programming languages

`auto`	`break`	`case`	`char`	`const`
`continue`	`default`	`do`	`double`	`else`
`enum`	`extern`	`float`	`for`	`goto`
`if`	`int`	`long`	`register`	`return`
`short`	`signed`	`sizeof`	`static`	`struct`
`switch`	`typedef`	`union`	`unsigned`	`void`
`volatile`	`while`			

C++-only keywords

`and`	`and_eq`	`asm`	`bitand`	`bitor`
`bool`	`catch`	`class`	`compl`	`const_cast`
`delete`	`dynamic_cast`	`explicit`	`export`	`false`
`friend`	`inline`	`mutable`	`namespace`	`new`
`not`	`not_eq`	`operator`	`or`	`or_eq`
`private`	`protected`	`public`	`reinterpret_cast`	`static_cast`
`template`	`this`	`throw`	`true`	`try`
`typeid`	`typename`	`using`	`virtual`	`wchar_t`
`xor`	`xor_eq`			

Fig. 3.3 | C++ keywords.

Common Programming Error 3.1

Using a keyword as an identifier is a syntax error.

Common Programming Error 3.2

Spelling a keyword with any uppercase letters is a syntax error. All of C++'s keywords contain only lowercase letters.

Summary of Control Statements in C++
C++ has only three kinds of control statements: the sequence statement, selection statements (three types—`if`, `if...else` and `switch`) and repetition statements (three types—`while`, `for` and `do...while`). Each program combines as many of these control statements as is appropriate for the algorithm the program implements. We can model each control statement as an activity diagram with an initial state and a final state that represent a control statement's entry point and exit point, respectively. These **single-entry/single-exit control statements** make it easy to build programs—control statements are attached to one another by connecting the exit point of one to the entry point of the next. This is similar to the way a child stacks building blocks, so we call this **control-statement stacking**. We'll learn shortly that there is only one other way to connect control statements—called **control-statement nesting**, in which one control statement is contained *inside* another.

Software Engineering Observation 3.1
Any C++ program we'll ever build can be constructed from only seven different types of control statements (sequence, `if`, `if...else`, `switch`, `while`, `do...while` and `for`) combined in only two ways (control-statement stacking and control-statement nesting).

3.5 `if` Selection Statement

Programs use selection statements 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 whether the condition "student's grade is greater than or equal to 60" is `true` or `false`. If the condition is `true`, "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. The indentation of the second line is optional, but it's recommended because it emphasizes the inherent structure of structured programs.

Good Programming Practice 3.1
Consistently applying reasonable indentation conventions throughout your programs greatly improves program readability. We suggest three blanks per indent. Some people prefer using tabs, but these can vary across editors, causing a program written on one editor to align differently when used with another.

The preceding pseudocode *If* statement can be written in C++ as

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

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

Figure 3.4 illustrates the single-selection `if` statement. It contains what is perhaps the most important symbol in an activity diagram—the diamond or **decision symbol**, which indicates that a decision is to be made. A decision symbol indicates that the workflow will continue along a path determined by the symbol's associated **guard conditions**, which can

be true or false. Each transition arrow emerging from a decision symbol has a guard condition (specified in square brackets above or next to the transition arrow). If a particular guard condition is true, the workflow enters the action state to which that transition arrow points. In Fig. 3.4, if the grade is greater than or equal to 60, the program prints "Passed" to the screen, then transitions to the final state of this activity. If the grade is less than 60, the program immediately transitions to the final state without displaying a message.

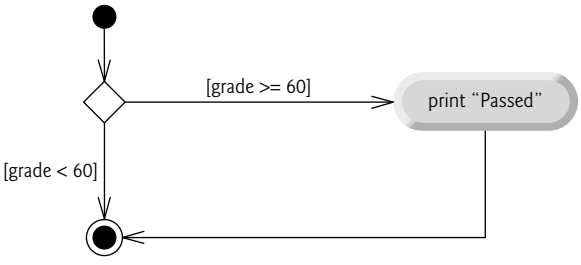

Fig. 3.4 | `if` single-selection statement activity diagram.

We learned in Chapter 2 that decisions can be based on conditions containing relational or equality operators. Actually, in C++, a decision can be based on *any* expression—if the expression evaluates to zero, it's treated as false; if the expression evaluates to nonzero, it's treated as true. C++ provides the data type **`bool`** for variables that can hold only the values **`true`** and **`false`**—each of these is a C++ keyword.

Portability Tip 3.1

For compatibility with earlier versions of C, which used integers for Boolean values, the `bool` *value* `true` *also can be represented by any nonzero value (compilers typically use 1) and the* `bool` *value* `false` *also can be represented as the value zero.*

The `if` statement is a single-entry/single-exit statement. We'll see that the activity diagrams for the remaining control statements also contain initial states, transition arrows, action states that indicate actions to perform, decision symbols (with associated guard conditions) that indicate decisions to be made and final states. This is consistent with the **action/decision model of programming** we've been emphasizing.

Envision seven bins, each containing only empty UML activity diagrams of one of the seven types of control statements. Your task, then, is assembling a program from the activity diagrams of as many of each type of control statement as the algorithm demands, combining the activity diagrams in only two possible ways (stacking or nesting), then filling in the action states and decisions with action expressions and guard conditions in a manner appropriate to form a structured implementation for the algorithm. We'll discuss the variety of ways in which actions and decisions may be written.

3.6 `if...else` Double-Selection Statement

The `if` single-selection statement performs an indicated action only when the condition is `true`; otherwise the action is skipped. The `if...else` double-selection statement allows

you to specify an action to perform when the condition is true and a different action to perform 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, but 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."

The preceding pseudocode *If…Else* statement can be written in C++ as

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

The body of the `else` is also indented.

Good Programming Practice 3.2

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

Good Programming Practice 3.3

Indent both body statements of an `if…else` *statement.*

Good Programming Practice 3.4

If there are several levels of indentation, each level should be indented the same additional amount of space to promote readability and maintainability.

Figure 3.5 illustrates the the `if…else` statement's flow of control. Once again, the symbols in the UML activity diagram (besides the initial state, transition arrows and final state) represent action states and decisions.

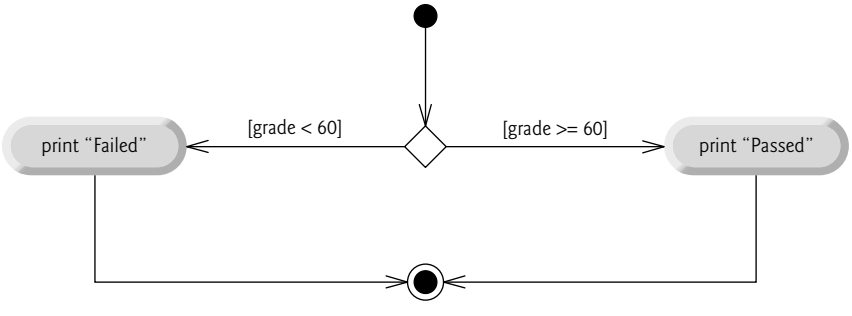

Fig. 3.5 | `if…else` double-selection statement activity diagram.

Conditional Operator (?:)

C++ provides the **conditional operator** (`?:`), which is closely related to the `if...else` statement. 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

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

contains a conditional expression, `grade >= 60 ? "Passed" : "Failed"`, that evaluates to the string `"Passed"` if the condition `grade >= 60` is `true`, but evaluates to `"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'll see, the precedence of the conditional operator is low, so the parentheses in the preceding expression are required.

Error-Prevention Tip 3.1

To avoid precedence problems (and for clarity), place conditional expressions (that appear in larger expressions) in parentheses.

The values in a conditional expression also can be actions to execute. For example, the following conditional expression also prints `"Passed"` or `"Failed"`:

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

The preceding conditional expression 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` statement. Conditional expressions can appear in some program locations where `if...else` statements cannot.

Nested `if...else` Statements

Nested `if...else` statements test for multiple cases by placing `if...else` selection statements inside other `if...else` selection statements. For example, the following pseudocode `if...else` statement prints `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

```
if ( studentGrade >= 90 ) // 90 and above gets "A"
   cout << "A";
else
   if ( studentGrade >= 80 ) // 80-89 gets "B"
      cout << "B";
   else
      if ( studentGrade >= 70 ) // 70-79 gets "C"
         cout << "C";
      else
         if ( studentGrade >= 60 ) // 60-69 gets "D"
            cout << "D";
         else // less than 60 gets "F"
            cout << "F";
```

If `studentGrade` is greater than or equal to 90, the first four conditions are `true`, but only the output statement after the first test executes. Then, the program skips the `else`-part of the "outermost" `if...else` statement. Most write the preceding `if...else` statement as

```
if ( studentGrade >= 90 ) // 90 and above gets "A"
   cout << "A";
else if ( studentGrade >= 80 ) // 80-89 gets "B"
   cout << "B";
else if ( studentGrade >= 70 ) // 70-79 gets "C"
   cout << "C";
else if ( studentGrade >= 60 ) // 60-69 gets "D"
   cout << "D";
else // less than 60 gets "F"
   cout << "F";
```

The two forms are identical except for the spacing and indentation, which the compiler ignores. The latter form is popular because it avoids deep indentation of the code to the right, which can force lines to wrap.

Performance Tip 3.1

A nested `if...else` statement can perform much faster than a series of single-selection `if` statements because of the possibility of early exit after one of the conditions is satisfied.

Performance Tip 3.2

In a nested `if...else` statement, test the conditions that are more likely to be `true` at the beginning of the nested statement. This will enable the nested `if...else` statement to run faster by exiting earlier than they would if infrequently occurring cases were tested first.

Dangling-`else` Problem

The C++ compiler always associates an `else` with the immediately preceding `if` unless told to do otherwise by the placement of braces (`{` and `}`). This behavior can lead to what's referred to as the **dangling-`else` problem**. For example,

```
if ( x > 5 )
   if ( y > 5 )
      cout << "x and y are > 5";
else
   cout << "x is <= 5";
```

appears to indicate that if x is greater than 5, the nested if statement determines whether y is also greater than 5. If so, "x and y are > 5" is output. Otherwise, it appears that if x is not greater than 5, the else part of the if...else outputs "x is <= 5".

Beware! This nested if...else statement does not execute as it appears. The compiler actually interprets the statement as

```
if ( x > 5 )
   if ( y > 5 )
      cout << "x and y are > 5";
   else
      cout << "x is <= 5";
```

in which the body of the first if is a nested if...else. The outer if statement tests whether x is greater than 5. If so, execution continues by testing whether y is also greater than 5. If the second condition is true, the proper string—"x and y are > 5"—is displayed. However, if the second condition is false, the string "x is <= 5" is displayed, even though we know that x is greater than 5.

To force the nested if...else statement to execute as originally intended, we can write it as follows:

```
if ( x > 5 )
{
   if ( y > 5 )
      cout << "x and y are > 5";
}
else
   cout << "x is <= 5";
```

The braces ({}) indicate to the compiler that the second if statement is in the body of the first if and that the else is associated with the first if. Exercises 3.23–3.24 further investigate the dangling-else problem.

Blocks

The if selection statement expects only one statement in its body. Similarly, the if and else parts of an if...else statement each expect only one body statement. To include several statements in the body of an if or in either part of an if...else, enclose the statements in braces ({ and }). A set of statements contained within a pair of braces is called a **compound statement** or a **block**. We use the term "block" from this point forward.

Software Engineering Observation 3.2

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

The following example includes a block in the else part of an if...else statement.

```
if ( studentGrade >= 60 )
   cout << "Passed.\n";
else
{
   cout << "Failed.\n";
   cout << "You must take this course again.\n";
}
```

In this case, if `studentGrade` 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 was less than 60. This is a logic error.

Common Programming Error 3.3

Forgetting one or both of the braces that delimit a block can lead to syntax errors or logic errors in a program.

Good Programming Practice 3.5

Always putting the braces in an `if...else` statement (or any control statement) helps prevent their accidental omission, especially when adding statements to an `if` or `else` clause at a later time. To avoid omitting one or both of the braces, some programmers prefer to type the beginning and ending braces of blocks even before typing the individual statements within the braces.

Just as a block can be placed anywhere a single statement can be placed, it's also possible to have no statement at all—called a **null statement** (or an **empty statement**). The null statement is represented by placing a semicolon (`;`) where a statement would normally be.

Common Programming Error 3.4

Placing a semicolon after the condition in an `if` statement leads to a logic error in single-selection `if` statements and a syntax error in double-selection `if...else` statements (when the `if` part contains an actual body statement).

3.7 `while` Repetition Statement

A **repetition statement** (also called a **looping statement** or a **loop**) allows you to specify that a program should repeat an action 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's 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 contained in the *While* repetition statement constitutes the body of the *While*, which can be a single statement or a block. 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 statement executes.

As an example of C++'s `while` repetition statement, consider a program segment designed to find the first power of 3 larger than 100. Suppose the integer variable `product` has been initialized to `3`. When the following `while` repetition statement finishes executing, `product` contains the result:

```
int product = 3;

while ( product <= 100 )
   product = 3 * product;
```

When the `while` statement begins execution, `product`'s value is 3. Each repetition multiplies `product` by 3, so `product` takes on the values 9, 27, 81 and 243 successively. When `product` becomes 243, the `while` statement condition—`product <= 100`—becomes `false`. This terminates the repetition, so the final value of `product` is 243. At this point, program execution continues with the next statement after the `while` statement.

Common Programming Error 3.5

Not providing, in the body of a `while` *statement, an action that eventually causes the condition in the* `while` *to become false normally results in a logic error called an* infinite loop, *in which the repetition statement never terminates. This can make a program appear to "hang" or "freeze" if the loop body does not contain statements that interact with the user.*

The UML activity diagram of Fig. 3.6 illustrates the flow of control that corresponds to the preceding `while` statement. Once again, the symbols in the diagram (besides the initial state, transition arrows, a final state and three notes) represent an action state and a decision. This diagram also introduces the UML's **merge symbol**, which joins two flows of activity into one flow of activity. The UML represents both the merge symbol and the decision symbol as diamonds. In this diagram, the merge symbol joins the transitions from the initial state and from the action state, so they both flow into the decision that determines whether the loop should begin (or continue) executing. The decision and merge symbols can be distinguished by the number of "incoming" and "outgoing" transition arrows. A decision symbol has one transition arrow pointing to the diamond and two or more transition arrows pointing out from the diamond to indicate possible transitions

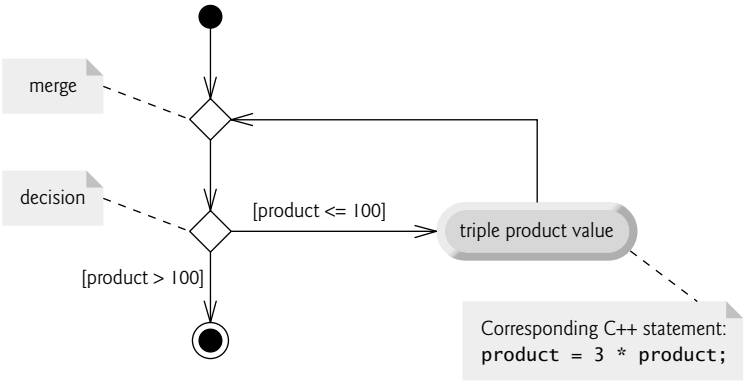

Fig. 3.6 | `while` repetition statement UML activity diagram.

from that point. In addition, each transition arrow pointing out of a decision symbol has a guard condition next to it. A merge symbol has two or more transition arrows pointing to the diamond and only one transition arrow pointing from the diamond, to indicate multiple activity flows merging to continue the activity. Unlike the decision symbol, the merge symbol does not have a counterpart in C++ code.

The diagram of Fig. 3.6 clearly shows the repetition of the `while` statement discussed earlier in this section. The transition arrow emerging from the action state points to the merge, which transitions back to the decision that's tested each time through the loop until the guard condition `product > 100` becomes true. Then the `while` statement exits (reaches its final state) and control passes to the next statement in sequence in the program.

Imagine a deep bin of empty UML `while` repetition statement activity diagrams—as many as you might need to stack and nest with the activity diagrams of other control statements to form a structured implementation of an algorithm. You fill in the action states and decision symbols with action expressions and guard conditions appropriate to the algorithm.

Performance Tip 3.3

Many of the Performance Tips *we mention in this text result in only small improvements, so you might be tempted to ignore them. However, a small performance improvement for code that executes many times in a loop can result in substantial overall performance improvement.*

3.8 Formulating Algorithms: Counter-Controlled Repetition

To illustrate how programmers develop algorithms, this section and Section 3.9 solve two variations of a class average 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. Calculate and display the total of all student grades and 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, calculate the average and print the result.

Pseudocode Algorithm with Counter-Controlled Repetition

Let's use pseudocode to list the actions to execute and specify the order in which these actions should occur. 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 group of statements will execute (also known as the number of **iterations** of the loop).

Counter-controlled repetition is often called **definite repetition** because the number of repetitions is known before the loop begins executing. In this example, repetition terminates when the counter exceeds 10. This section presents a fully developed pseudocode algorithm (Fig. 3.7) and the C++ program (Fig. 3.8) that implements the algorithm. In Section 3.9 we demonstrate how to use pseudocode to develop such an algorithm from scratch.

Software Engineering Observation 3.3

Experience has shown that the most difficult part of solving a problem on a computer is developing the algorithm for the solution. The process of producing a working C++ program from the algorithm is typically straightforward.

```
1  Set total to zero
2  Set grade counter to one
3
4  While grade counter is less than or equal to ten
5      Prompt the user to enter the next grade
6      Input the next grade
7      Add the grade into the total
8      Add one to the grade counter
9
10 Set the class average to the total divided by ten
11 Print the total of the grades for all students in the class
12 Print the class average
```

Fig. 3.7 | Pseudocode for solving the class average problem with counter-controlled repetition.

Note the references in the pseudocode algorithm of Fig. 3.7 to a total and a counter. A **total** is a variable used to accumulate the sum of several values. A **counter** is a variable used to count—in this case, the grade counter indicates which of the 10 grades is about to be entered by the user. Variables used to store totals are normally initialized to zero before being used in a program; otherwise, the sum would include the previous value stored in the total's memory location.

Implementing Counter-Controlled Repetition

Lines 14–31 of Fig. 3.8 implement the class average algorithm described by the pseudocode in Fig. 3.7.

```
1  // Fig. 3.8: fig03_08.cpp
2  // Class average program with counter-controlled repetition.
3  #include <iostream>
4  using namespace std;
5
6  int main ()
7  {
8     int total; // sum of grades entered by user
9     int gradeCounter; // number of the grade to be entered next
10    int grade; // grade value entered by user
11    int average; // average of grades
12
13    // initialization phase
14    total = 0; // initialize total
15    gradeCounter = 1; // initialize loop counter
16
```

Fig. 3.8 | Class average problem using counter-controlled repetition. (Part 1 of 2.)

```
17      // processing phase
18      while ( gradeCounter <= 10 ) // loop 10 times
19      {
20         cout << "Enter grade: "; // prompt for input
21         cin >> grade; // input next grade
22         total = total + grade; // add grade to total
23         gradeCounter = gradeCounter + 1; // increment counter by 1
24      } // end while
25
26      // termination phase
27      average = total / 10; // integer division yields integer result
28
29      // display total and average of grades
30      cout << "\nTotal of all 10 grades is " << total << endl;
31      cout << "Class average is " << average << endl;
32   } // end main
```

```
Enter grade: 67
Enter grade: 78
Enter grade: 89
Enter grade: 67
Enter grade: 87
Enter grade: 98
Enter grade: 93
Enter grade: 85
Enter grade: 82
Enter grade: 100

Total of all 10 grades is 846
Class average is 84
```

Fig. 3.8 | Class average problem using counter-controlled repetition. (Part 2 of 2.)

Lines 8–11 (Fig. 3.8) declare local variables `total`, `gradeCounter`, `grade` and `average` to be of type `int`. Variable `grade` stores the user input. Notice that the preceding declarations appear in the body of function `main`.

Good Programming Practice 3.6

Separate declarations from other statements in functions with a blank line for readability.

Lines 14–15 initialize `total` to 0 and `gradeCounter` to 1 before they're used in calculations. Counter variables are normally initialized to zero or one, depending on their use. An uninitialized variable contains a "**garbage**" **value** (also called an **undefined value**)—the value last stored in the memory location reserved for that variable. The variables `grade` and `average` (for the user input and calculated average, respectively) need not be initialized before they're used—their values will be assigned as they're input or calculated later in the function.

Common Programming Error 3.6

Not initializing counters and totals can lead to logic errors.

Error-Prevention Tip 3.2

Initialize each counter and total, either in its declaration or in an assignment statement. Totals are normally initialized to 0. Counters are normally initialized to 0 or 1, depending on how they're used.

Good Programming Practice 3.7

Declare each variable on a separate line with its own comment for readability.

Line 18 indicates that the `while` statement should continue looping (also called **iterating**) as long as `gradeCounter`'s value is less than or equal to `10`. While this condition remains true, the `while` statement repeatedly executes the statements between the braces that delimit its body (lines 55–60).

Line 20 displays the prompt `"Enter grade: "`. This line corresponds to the pseudocode statement *"Prompt the user to enter the next grade."* Line 21 reads the grade entered by the user and assigns it to variable `grade`. This line corresponds to the pseudocode statement *"Input the next grade."* Recall that variable `grade` was not initialized earlier in the program, because the program obtains the value for `grade` from the user during each iteration of the loop. Line 22 adds the new grade entered by the user to the `total` and assigns the result to `total`, which replaces its previous value.

Line 23 adds `1` to `gradeCounter` to indicate that the program has processed a grade and is ready to input the next grade from the user. Incrementing `gradeCounter` eventually causes `gradeCounter` to exceed `10`. At that point the `while` loop terminates because its condition (line 18) becomes false.

When the loop terminates, line 27 performs the averaging calculation and assigns its result to the variable `average`. Line 30 displays the text `"Total of all 10 grades is "` followed by variable `total`'s value. Line 31 then displays the text `"Class average is "` followed by variable `average`'s value.

Notes on Integer Division and Truncation

The averaging calculation performed in Fig. 3.8 produces an integer result. The sample execution indicates that the sum of the grade values is 846, which, when divided by 10, should yield 84.6—a number with a decimal point. However, the result of the calculation `total / 10` (line 27 of Fig. 3.8) is the integer 84, because `total` and 10 are both integers. Dividing two integers results in integer division—any fractional part of the calculation is lost (i.e., **truncated**). We'll see how to obtain a result that includes a decimal point from the averaging calculation in the next section.

Common Programming Error 3.7

Assuming that integer division rounds (rather than truncates) can lead to incorrect results. For example, 7 ÷ 4, which yields 1.75 in conventional arithmetic, truncates to 1 in integer arithmetic, rather than rounding to 2.

In Fig. 3.8, if line 27 used `gradeCounter` rather than 10, the output for this program would display an incorrect value, 76. This would occur because in the final iteration of the `while` statement, `gradeCounter` was incremented to the value 11 in line 23.

Common Programming Error 3.8

Using a loop's counter-control variable in a calculation after the loop often causes a common logic error called an ***off-by-one error****. In a counter-controlled loop that counts up by one each time through the loop, the loop terminates when the counter's value is one higher than its last legitimate value (i.e., 11 in the case of counting from 1 to 10).*

3.9 Formulating Algorithms: Sentinel-Controlled Repetition

Let's generalize the class average problem. Consider the following problem:

> *Develop a class average program that processes grades for an arbitrary number of students each time it's run.*

In the previous example, the problem statement specified the number of students, so the number of grades (10) was known in advance. In this example, no indication is given of how many grades the user will enter during the program's execution. 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?

To solve this problem, we can 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." After typing the legitimate grades, the user 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.

The sentinel value must be chosen so that it's not confused with an acceptable input value. Grades are normally nonnegative integers, so –1 is an acceptable sentinel value. Thus, a run of the program might process inputs such as 95, 96, 75, 74, 89 and –1. The program would then compute and print the class average for the grades 95, 96, 75, 74 and 89. Since –1 is the sentinel value, it should not enter into the averaging calculation.

Common Programming Error 3.9

Choosing a sentinel value that's also a legitimate data value is a logic error.

Developing the Pseudocode Algorithm with Top-Down, Stepwise Refinement: The Top and First Refinement

We approach the program with a technique called **top-down, stepwise refinement**, which is essential in developing well-structured programs. We begin with a pseudocode representation of the **top**—a single statement that conveys the overall function of the program:

Determine the class average for the quiz for an arbitrary number of students

The top is, in effect, a *complete* representation of a program. Unfortunately, the top (as in this case) rarely conveys sufficient detail from which to write a 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 total of all student grades and the class average

This refinement uses only the sequence statement—these steps execute in order.

Software Engineering Observation 3.4

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

Software Engineering Observation 3.5

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 (such as counters and totals) accordingly; and a termination phase that calculates and outputs the final results.

Proceeding to the Second Refinement

The preceding *Software Engineering Observation* is often all you need for the first refinement in the top-down process. In the **second refinement**, we commit to specific variables. In this example, we need a running total of the numbers, a count of how many numbers have been processed, a variable to receive the value of each grade as it's input by the user and a variable to hold the calculated average. The pseudocode statement

```
Initialize variables
```

can be refined as follows:

```
Initialize total to zero
Initialize counter to zero
```

Only the variables *total* and *counter* need to be initialized before they're used. The variables *average* and *grade* (for the calculated average and the user input, respectively) need not be initialized, because their values will be replaced as they're calculated or input.

The pseudocode statement

```
Input, sum and count the quiz grades
```

requires a repetition statement (i.e., a loop) that successively inputs each grade. We don't know in advance how many grades are to be processed, so we'll use **sentinel-controlled repetition**. The user enters legitimate grades one at a time. After entering the last legitimate grade, the user enters the sentinel value. The program tests for the sentinel value after each grade is input and terminates the loop when the user enters the sentinel value. The second refinement of the preceding pseudocode statement is then

```
Prompt the user to enter the first grade
Input the first grade (possibly the sentinel)

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

In pseudocode, we do not use braces around the statements that form the body of the *While* statement. 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

Calculate and print the total of all student grades and the class average

can be refined as follows:

```
If the counter is not equal to zero
    Set the average to the total divided by the counter
    Print the total of the grades for all students in the class
    Print the class average
else
    Print "No grades were entered"
```

We're careful here to test for the possibility of division by zero—normally a **fatal logic error** that, if undetected, would cause the program to fail (often called "**crashing**"). The complete second refinement of the pseudocode for the class average problem is shown in Fig. 3.9.

Common Programming Error 3.10

An attempt to divide by zero normally causes a fatal runtime error.

Error-Prevention Tip 3.3

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

```
1  Initialize total to zero
2  Initialize counter to zero
3
4  Prompt the user to enter the first grade
5  Input the first grade (possibly the sentinel)
6
7  While the user has not yet entered the sentinel
8      Add this grade into the running total
9      Add one to the grade counter
10     Prompt the user to enter the next grade
11     Input the next grade (possibly the sentinel)
12
13 If the counter is not equal to zero
14     Set the average to the total divided by the counter
15     Print the total of the grades for all students in the class
16     Print the class average
17 else
18     Print "No grades were entered"
```

Fig. 3.9 | Class average problem pseudocode algorithm with sentinel-controlled repetition.

In Fig. 3.7 and Fig. 3.9, we include some blank lines and indentation in the pseudocode to make it more readable. The blank lines separate the pseudocode algorithms into their various phases, and the indentation emphasizes the control statement bodies.

The pseudocode algorithm in Fig. 3.9 solves the more general class average problem. This algorithm was developed after only two levels of refinement. Sometimes more levels are necessary.

Software Engineering Observation 3.6

Terminate the top-down, stepwise refinement process when the pseudocode algorithm is specified in sufficient detail for you to be able to convert the pseudocode to C++. Typically, implementing the C++ program is then straightforward.

Software Engineering Observation 3.7

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 method might work for simple and familiar problems, it can lead to serious difficulties in large, complex projects.

Implementing Sentinel-Controlled Repetition in a C++ Program

Fig. 3.10 shows the C++ implementation the pseudocode algorithm of Fig. 3.9. Although each grade entered is an integer, the averaging calculation is likely to produce a number with a decimal point—in other words, a real number or **floating-point number** (e.g., 7.33, 0.0975 or 1000.12345). The type `int` cannot represent such a number, so this class must use another type to do so. C++ provides several data types for storing floating-point numbers in memory, including **`float`** and **`double`**. The primary difference between these types is that, compared to `float` variables, `double` variables can typically store numbers with larger magnitude and finer detail (i.e., more digits to the right of the decimal point—also known as the number's **precision**). This program 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 as we discuss the program.

```
1  // Fig. 3.10: fig03_10.cpp
2  // Class average program with sentinel-controlled repetition.
3  #include <iostream>
4  #include <iomanip> // parameterized stream manipulators
5  using namespace std;
6
7  // determine class average based on 10 grades entered by user
8  int main()
9  {
10    int total; // sum of grades entered by user
11    int gradeCounter; // number of grades entered
12    int grade; // grade value
13    double average; // number with decimal point for average
14
```

Fig. 3.10 | Class average problem using sentinel-controlled repetition: `GradeBook` source code file. (Part 1 of 2.)

```
15      // initialization phase
16      total = 0; // initialize total
17      gradeCounter = 0; // initialize loop counter
18
19      // processing phase
20      // prompt for input and read grade from user
21      cout << "Enter grade or -1 to quit: ";
22      cin >> grade; // input grade or sentinel value
23
24      // loop until sentinel value read from user
25      while ( grade != -1 ) // while grade is not -1
26      {
27         total = total + grade; // add grade to total
28         gradeCounter = gradeCounter + 1; // increment counter
29
30         // prompt for input and read next grade from user
31         cout << "Enter grade or -1 to quit: ";
32         cin >> grade; // input grade or sentinel value
33      } // end while
34
35      // termination phase
36      if ( gradeCounter != 0 ) // if user entered at least one grade...
37      {
38         // calculate average of all grades entered
39         average = static_cast< double >( total ) / gradeCounter;
40
41         // display total and average (with two digits of precision)
42         cout << "\nTotal of all " << gradeCounter << " grades entered is "
43            << total << endl;
44         cout << "Class average is " << setprecision( 2 ) << fixed << average
45            << endl;
46      } // end if
47      else // no grades were entered, so output appropriate message
48         cout << "No grades were entered" << endl;
49   } // end main
```

```
Enter grade or -1 to quit: 97
Enter grade or -1 to quit: 88
Enter grade or -1 to quit: 72
Enter grade or -1 to quit: -1

Total of all 3 grades entered is 257
Class average is 85.67
```

Fig. 3.10 | Class average problem using sentinel-controlled repetition: `GradeBook` source code file. (Part 2 of 2.)

In this example, we see that control statements can be stacked on top of one another (in sequence) just as a child stacks building blocks. The `while` statement (lines 25–33 of Fig. 3.10) is immediately followed by an `if...else` statement (lines 36–48) in sequence. Much of the code in this program is identical to the code in Fig. 3.8, so we concentrate on the new features and issues.

Line 13 declares the `double` variable `average`. Recall that we used an `int` variable in the preceding example to store the class average. Using type `double` in the current example

allows us to store the class average calculation's result as a floating-point number. Line 17 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, the program increments variable `gradeCounter` only when the user enters a valid grade value (i.e., not the sentinel value) and the program completes the processing of the grade. Finally, notice that both input statements (lines 22 and 32) are preceded by an output statement that prompts the user for input.

Good Programming Practice 3.8

Prompt the user for each keyboard input. The prompt should indicate the form of the input and any special input values. For example, in a sentinel-controlled loop, the prompts requesting data entry should explicitly remind the user what the sentinel value is.

Program Logic for Sentinel-Controlled Repetition vs. Counter-Controlled Repetition

Compare the program logic for sentinel-controlled repetition in this application with that for counter-controlled repetition in Fig. 3.8. In counter-controlled repetition, each iteration of the `while` statement (lines 18–24 of Fig. 3.8) reads a value from the user, for the specified number of iterations. In sentinel-controlled repetition, the program reads the first value (lines 21–22 of Fig. 3.10) before reaching the `while`. This value determines whether the program's flow of control should enter the body of the `while`. If the condition of the `while` is false, the user entered the sentinel value, so the body of the `while` does not execute (i.e., no grades were entered). If, on the other hand, the condition is true, the body begins execution, and the loop adds the `grade` value to the `total` (line 27) and increments `gradeCounter` (line 28). Then lines 31–32 in the loop's body prompt for and input the next value from the user. Next, program control reaches the closing right brace (`}`) of the body in line 33, so execution continues with the test of the `while`'s condition (line 25). The condition uses the most recent `grade` input by the user to determine whether the loop's body should execute again. The value of variable `grade` is always input from the user immediately before the program tests the `while` condition. This allows the program to determine whether the value just input is the sentinel value *before* the program processes that value (i.e., adds it to the `total` and increments `gradeCounter`). If the sentinel value is input, the loop terminates, and the program does not add –1 to the `total`.

After the loop terminates, the `if...else` statement in lines 36–48 executes. The condition in line 36 determines whether any grades were entered. If none were, the `else` part (lines 47–48) of the `if...else` statement executes and displays the message `"No grades were entered"`.

Notice the block in the `while` loop in Fig. 3.10. 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:

```
// loop until sentinel value read from user
while ( grade != -1 )
   total = total + grade; // add grade to total
gradeCounter = gradeCounter + 1; // increment counter

// prompt for input and read next grade from user
cout << "Enter grade or -1 to quit: ";
cin >> grade;
```

This would cause an infinite loop in the program if the user did not input -1 for the first grade (in line 22).

Common Programming Error 3.11

Omitting the braces that delimit a block can lead to logic errors, such as infinite loops. To prevent this problem, some programmers enclose the body of every control statement in braces, even if the body contains only a single statement.

Floating-Point Number Precision and Memory Requirements

Variables of type `float` represent **single-precision floating-point numbers** and have seven significant digits on most 32-bit systems. Variables of type `double` represent **double-precision floating-point numbers**. These require twice as much memory as `float` variables and provide 15 significant digits on most 32-bit systems—approximately double the precision of `float` variables. For most programs, variables of type `float` should suffice, but you can use `double` to "play it safe." In some programs, even variables of type `double` will be inadequate—such programs are beyond the scope of this book. Most programmers represent floating-point numbers with type `double`. In fact, C++ treats all floating-point numbers you type in a program's source code (such as 7.33 and 0.0975) as `double` values by default. Such values in the source code are known as **floating-point constants**. See Appendix C, Fundamental Types, for the ranges of values for `float`s and `double`s.

Floating-point numbers often arise as a result of division. In conventional arithmetic, when we divide 10 by 3, the result is 3.3333333…, with the sequence of 3s repeating infinitely. The computer allocates only a fixed amount of space to hold such a value, so clearly the stored floating-point value can be only an approximation.

Common Programming Error 3.12

Using floating-point numbers in a manner that assumes they're represented exactly (e.g., using them in comparisons for equality) can lead to incorrect results. Floating-point numbers are represented only approximately.

Although floating-point numbers are not always 100 percent 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 read the temperature on a thermometer as 98.6, it may actually be 98.5999473210643. Calling this number simply 98.6 is fine for most applications involving body temperatures. Due to the imprecise nature of floating-point numbers, type `double` is preferred over type `float`, because `double` variables can represent floating-point numbers more accurately. For this reason, we use type `double` throughout the book.

Converting Between Fundamental Types Explicitly and Implicitly

The variable `average` is declared to be of type `double` (line 13 of Fig. 3.10) to capture the fractional result of our calculation. However, `total` and `gradeCounter` are both integer variables. Recall that dividing two integers results in integer division, in which any fractional part of the calculation is lost (i.e., **truncated**). In the following statement:

```
average = total / gradeCounter;
```

the division occurs first—the result's fractional part is lost before it's assigned to `average`. To perform a floating-point calculation with integers, we must create temporary floating-

point values. C++ provides the **unary cast operator** to accomplish this task. Line 74 uses the cast operator `static_cast<double>(total)` to create 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 `gradeCounter`. The compiler knows how to evaluate only expressions in which the operand types of 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 values of data types `int` and `double`, C++ **promotes** `int` operands to `double` values. In our example, we're treating `total` as a `double` (by using the unary cast operator), so the compiler promotes `gradeCounter` to `double`, allowing the calculation to be performed—the result of the floating-point division is assigned to `average`. In Chapter 5, Functions and an Introduction to Recursion, we discuss all the fundamental data types and their order of promotion.

Common Programming Error 3.13

The cast operator can be used to convert between fundamental numeric types, such as `int` *and* `double`*, and between related class types (as we discuss in Chapter 13, Object-Oriented Programming: Polymorphism). Casting to the wrong type may cause errors.*

Cast operators are available for use with every data type and with class types as well. 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**—an operator that takes only one operand. In Chapter 2, we studied the binary arithmetic operators. C++ also supports unary versions of the plus (+) and minus (-) operators, so that you can write such expressions as `-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 `static_cast<`*type*`>()` in our precedence charts.

Formatting for Floating-Point Numbers

The formatting capabilities in Fig. 3.10 are discussed here briefly and explained in depth in Chapter 15, Stream Input/Output. The call to **`setprecision`** in line 44 (with an argument of `2`) indicates that `double` variable `average` should be printed with two digits of **precision** to the right of the decimal point (e.g., 92.37). This call is referred to as a **parameterized stream manipulator** (because of the `2` in parentheses). Programs that use these calls must contain the preprocessor directive (line 5)

```
#include <iomanip>
```

The manipulator `endl` is a **nonparameterized stream manipulator** (because it isn't followed by a value or expression in parentheses) and does not require the `<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** on most 32-bit systems today), although we'll see an exception to this in a moment.

The stream manipulator **`fixed`** (line 79) indicates that floating-point values should be output in so-called **fixed-point format**, as opposed to **scientific notation**. Scientific notation is a way of displaying a number as a floating-point number between the values of 1.0 and 10.0, multiplied by a power of 10. For instance, the value 3,100.0 would be displayed

in scientific notation as 3.1×10^3. Scientific notation is useful when displaying values that are very large or very small. Formatting using scientific notation is discussed further in Chapter 15. Fixed-point formatting, on the other hand, is used to force a floating-point number to display a specific number of digits. Specifying fixed-point formatting also forces the decimal point and trailing zeros to print, even if the value is a whole number amount, such as 88.00. Without the fixed-point formatting option, such a value prints in C++ as 88 without the trailing zeros and without the decimal point. When the stream manipulators `fixed` and `setprecision` are used in a program, the printed value is **rounded** to the number of decimal positions indicated by the value passed to `setprecision` (e.g., the value `2` in line 79), although the value in memory remains unaltered. For example, the values 87.946 and 67.543 are output as 87.95 and 67.54, respectively. It's also possible to force a decimal point to appear by using stream manipulator **`showpoint`**. If `showpoint` is specified without `fixed`, then trailing zeros will not print. Like `endl`, stream manipulators `fixed` and `showpoint` do not use parameters, nor do they require the `<iomanip>` header file. Both can be found in header `<iostream>`.

Lines 44 and 45 of Fig. 3.10 output the class average. In this example, we display the class average rounded to the nearest hundredth and output it with exactly two digits to the right of the decimal point. The parameterized stream manipulator (line 44) indicates that variable `average`'s value should be displayed with two digits of precision to the right of the decimal point—indicated by `setprecision( 2 )`. The three grades entered during the sample execution of the program in Fig. 3.10 total 257, which yields the average 85.666666…. The parameterized stream manipulator `setprecision` causes the value to be rounded to the specified number of digits. In this program, the average is rounded to the hundredths position and displayed as `85.67`.

3.10 Formulating Algorithms: Nested Control Statements

For the next example, we once again formulate an algorithm by using pseudocode and top-down, stepwise refinement, and write a corresponding C++ program. We've seen that control statements can be stacked on top of one another (in sequence) just as a child stacks building blocks. In this case study, we examine the only other structured way control statements can be connected, namely, by **nesting** one control statement 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, ten of the students who completed this course took the exam. The college wants to know how well its students did on the exam. You've been asked to write a program to summarize the results. You've been given a list of these 10 students. Next to each name is written a 1 if the student passed the exam or 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 prompting message "Enter result" 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 who failed.*
> 4. *If more than eight students passed the exam, print the message "Bonus to instructor!"*

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

1. The program must process test results for 10 students. A counter-controlled loop can be used because the number of test results is known in advance.
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 whether 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's a 2. (Exercise 3.20 considers the consequences of this assumption.)
3. Two counters are used to keep track of the exam results—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 whether more than eight students passed the exam.

Let's proceed with top-down, stepwise refinement. We begin with a pseudocode representation of the top:

```
Analyze exam results and decide whether tuition should be raised
```

Once again, it's important to emphasize that the top is a *complete* representation of the program, but several refinements are likely to be needed before the pseudocode evolves naturally into a C++ program.

Our first refinement is

```
Initialize variables
Input the 10 exam results, 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 last variable is not initialized, because its value is read from the user during each iteration of the loop.

The pseudocode statement

```
Initialize variables
```

can be refined as follows:

```
Initialize passes to zero
Initialize failures to zero
Initialize student counter to one
```

Notice that only the counters are initialized at the start of the algorithm.

The pseudocode statement

```
Input the 10 exam results, and count passes and failures
```

requires a loop that successively inputs the result of each exam. Here it's known in advance that there are precisely 10 exam results, so counter-controlled looping is appropriate. Inside the loop (i.e., **nested** within the loop), an `if...else` statement will determine whether

each exam result is a pass or a failure and will increment the appropriate counter. The refinement of the preceding pseudocode statement is then

```
While student counter is less than or equal to 10
    Prompt the user to enter the next exam result
    Input the next exam result

    If the student passed
        Add one to passes
    Else
        Add one to failures

    Add one to student counter
```

We use blank lines to isolate the *If…Else* control statement, which improves readability.

The pseudocode statement

```
Print a summary of the exam results and decide whether tuition should be raised
```

can be refined as follows:

```
Print the number of passes
Print the number of failures

If more than eight students passed
    Print "Bonus to instructor!"
```

The complete second refinement appears in Fig. 3.11. Blank lines set off the *While* statement for readability. This pseudocode is now sufficiently refined for conversion to C++.

```
1  Initialize passes to zero
2  Initialize failures to zero
3  Initialize student counter to one
4
5  While student counter is less than or equal to 10
6      Prompt the user to enter the next exam result
7      Input the next exam result
8
9      If the student passed
10         Add one to passes
11     Else
12         Add one to failures
13
14     Add one to student counter
15
16 Print the number of passes
17 Print the number of failures
18
19 If more than eight students passed
20     Print "Bonus to instructor!"
```

Fig. 3.11 | Pseudocode for examination-results problem.

Conversion to C++

The program that implements the pseudocode algorithm and two sample executions are shown in Fig. 3.12.

Lines 9–12 declare the variables used to process the examination results. We've taken advantage of a feature of C++ that allows variable initialization to be incorporated into declarations (`passes` is initialized to `0`, `failures` is initialized to `0` and `studentCounter` is initialized to `1`). Looping programs may require initialization at the beginning of each repetition; such reinitialization normally would be performed by assignment statements rather than in declarations or by moving the declarations inside the loop bodies.

The `while` statement (lines 15–29) loops 10 times. Each iteration inputs and processes one exam result. The `if...else` statement (lines 22–25) for processing each result is nested in the `while` statement. If the `result` is `1`, the `if...else` statement increments

```
1  // Fig. 3.12: fig03_12.cpp
2  // Examination-results problem: Nested control statements.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     // initializing variables in declarations
9     int passes = 0; // number of passes
10    int failures = 0; // number of failures
11    int studentCounter = 1; // student counter
12    int result; // one exam result (1 = pass, 2 = fail)
13
14    // process 10 students using counter-controlled loop
15    while ( studentCounter <= 10 )
16    {
17       // prompt user for input and obtain value from user
18       cout << "Enter result (1 = pass, 2 = fail): ";
19       cin >> result; // input result
20
21       // if...else nested in while
22       if ( result == 1 )          // if result is 1,
23          passes = passes + 1;     // increment passes;
24       else                        // else result is not 1, so
25          failures = failures + 1; // increment failures
26
27       // increment studentCounter so loop eventually terminates
28       studentCounter = studentCounter + 1;
29    } // end while
30
31    // termination phase; display number of passes and failures
32    cout << "Passed " << passes << "\nFailed " << failures << endl;
33
34    // determine whether more than eight students passed
35    if ( passes > 8 )
36       cout << "Bonus to instructor!" << endl;
37 } // end main
```

Fig. 3.12 | Examination-results problem: Nested control statements. (Part 1 of 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
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
Enter result (1 = pass, 2 = fail): 1
Passed 9
Failed 1
Bonus to instructor!
```

```
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): 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): 2
Passed 6
Failed 4
```

Fig. 3.12 | Examination-results problem: Nested control statements. (Part 2 of 2.)

`passes`; otherwise, it assumes the `result` is `2` and increments `failures`. Line 28 increments `studentCounter` before the loop condition is tested again in line 15. After 10 values have been input, the loop terminates and line 32 displays the number of `passes` and the number of `failures`. The `if` statement in lines 35–36 determines whether more than eight students passed the exam and, if so, outputs the message `"Bonus to instructor!"`.

Figure 3.12 shows the input and output from two sample executions of the program. At the end of the first sample execution, the condition in line 35 is true—more than eight students passed the exam, so the program outputs a message indicating that the instructor should receive a bonus.

3.11 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;
```

in which the same *variable* appears on both sides of the assignment operator and *operator* is one of the binary operators +, -, *, /, or % (or others we'll discuss later in the text), can be written in the form

```
variable operator= expression;
```

Thus the assignment `c += 3` adds `3` to `c`. Figure 3.13 shows the arithmetic assignment operators, sample expressions using these operators and explanations.

Assignment operator	Sample expression	Explanation	Assigns
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	`10` to `c`
`-=`	`d -= 4`	`d = d - 4`	`1` to `d`
`*=`	`e *= 5`	`e = e * 5`	`20` to `e`
`/=`	`f /= 3`	`f = f / 3`	`2` to `f`
`%=`	`g %= 9`	`g = g % 9`	`3` to `g`

Fig. 3.13 | Arithmetic assignment operators.

3.12 Increment and Decrement Operators

In addition to the arithmetic assignment operators, C++ also provides two unary operators for adding 1 to or subtracting 1 from the value of a numeric variable. These are the unary **increment operator**, ++, and the unary **decrement operator**, --, which are summarized in Fig. 3.14. A program can increment by 1 the value of a variable called `c` using the increment operator, ++, rather than the expression `c = c + 1` or `c += 1`. An increment or decrement operator that's prefixed to (placed before) a variable is referred to as the **prefix increment** or **prefix decrement operator**, respectively. An increment or decrement operator that's postfixed to (placed after) a variable is referred to as the **postfix increment** or **postfix decrement operator**, respectively.

Operator	Called	Sample expression	Explanation
`++`	preincrement	`++a`	Increment `a` by 1, then use the new value of `a` in the expression in which `a` resides.
`++`	postincrement	`a++`	Use the current value of `a` in the expression in which `a` resides, then increment `a` by 1.
`--`	predecrement	`--b`	Decrement `b` by 1, then use the new value of `b` in the expression in which `b` resides.
`--`	postdecrement	`b--`	Use the current value of `b` in the expression in which `b` resides, then decrement `b` by 1.

Fig. 3.14 | Increment and decrement operators.

Using the prefix increment (or decrement) operator to add (or subtract) 1 from a variable is known as **preincrementing** (or **predecrementing**) the variable. Preincrementing (or predecrementing) 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. Using the postfix increment (or decrement) operator to add (or subtract) 1 from a variable is known as **postincrementing** (or **postdecrementing**) the variable. Postincrementing (or postdecrementing) causes the current value of the variable to be used in the expression in which it appears, then the variable's value is incremented (decremented) by 1.

Good Programming Practice 3.9

Unlike binary operators, the unary increment and decrement operators should be placed next to their operands, with no intervening spaces.

Figure 3.15 demonstrates the difference between the prefix increment and postfix increment versions of the ++ increment operator. The decrement operator (--) works similarly.

```
1  // Fig. 3.15: fig03_15.cpp
2  // Preincrementing and postincrementing.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int c;
9
10    // demonstrate postincrement
11    c = 5; // assign 5 to c
12    cout << c << endl; // print 5
13    cout << c++ << endl; // print 5 then postincrement
14    cout << c << endl; // print 6
15
16    cout << endl; // skip a line
17
18    // demonstrate preincrement
19    c = 5; // assign 5 to c
20    cout << c << endl; // print 5
21    cout << ++c << endl; // preincrement then print 6
22    cout << c << endl; // print 6
23 } // end main
```

```
5
5
6

5
6
6
```

Fig. 3.15 | Preincrementing and postincrementing.

Line 11 initializes c to 5, and line 12 outputs c's initial value. Line 13 outputs the value of the expression c++. This postincrements the variable c, so c's original value (5) is output, then c's value is incremented. Thus, line 13 outputs c's initial value (5) again. Line 14 outputs c's new value (6) to prove that the variable's value was incremented in line 13.

Line 19 resets c's value to 5, and line 20 outputs that value. Line 21 outputs the value of the expression ++c. This expression preincrements c, so its value is incremented, then the new value (6) is output. Line 22 outputs c's value again to show that the value of c is still 6 after line 21 executes.

The arithmetic assignment operators and the increment and decrement operators can be used to simplify program statements. The three assignment statements in Fig. 3.12:

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

can be written more concisely with assignment operators as

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

with prefix increment operators as

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

or with postfix increment operators as

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

When you increment (++) or decrement (--) 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's 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).

Common Programming Error 3.14

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

Figure 3.16 shows the precedence and associativity of the operators introduced to this point. The operators are shown top-to-bottom in decreasing order of precedence. The second column indicates the associativity of the operators at each level of precedence. Notice that the conditional operator (?:), the unary operators preincrement (++), predecrement (--), plus (+) and minus (-), and the assignment operators =, +=, -=, *=, /= and %= associate from right to left. All other operators in the operator precedence chart of Fig. 3.16 associate from left to right. The third column names the various groups of operators.

Operators						Associativity	Type
`::`						left to right	scope resolution
`()`						left to right	parentheses
`++`	`--`	`static_cast<`*type*`>()`				left to right	unary (postfix)
`++`	`--`	`+`	`-`			right to left	unary (prefix)
`*`	`/`	`%`				left to right	multiplicative
`+`	`-`					left to right	additive
`<<`	`>>`					left to right	insertion/extraction
`<`	`<=`	`>`	`>=`			left to right	relational
`==`	`!=`					left to right	equality
`?:`						right to left	conditional
`=`	`+=`	`-=`	`*=`	`/=`	`%=`	right to left	assignment

Fig. 3.16 | Operator precedence for the operators encountered so far in the text.

3.13 Wrap-Up

This chapter presented basic problem-solving techniques. We demonstrated how to construct an algorithm (i.e., an approach to solving a problem) in pseudocode, then how to refine the algorithm through pseudocode development, resulting in C++ code that can be executed as part of a function. You learned how to use top-down, stepwise refinement to plan out the actions that a function must perform and the order in which it must perform them.

You learned that only three types of control statements—sequence, selection and repetition—are needed to develop any algorithm. We demonstrated two of C++'s selection statements—the `if` single-selection statement and the `if...else` double-selection statement. The `if` statement is used to execute a set of statements based on a condition—if the condition is true, the statements execute; if it isn't, the statements are skipped. The `if...else` double-selection statement is used to execute one set of statements if a condition is true, and another set of statements if the condition is false. We then discussed the `while` repetition statement, where a set of statements are executed repeatedly as long as a condition is true. We used control-statement stacking to total and compute the average of a set of student grades with counter- and sentinel-controlled repetition, and we used control-statement nesting to analyze and make decisions based on a set of exam results. We introduced assignment operators, which can be used for abbreviating statements. We presented the increment and decrement operators, which can be used to add or subtract the value 1 from a variable. In Chapter 4, Control Statements: Part 2, we continue our discussion of control statements, introducing the `for`, `do...while` and `switch` statements.

Summary

Section 3.2 Algorithms

- An algorithm is a procedure for solving a problem in terms of the actions to execute and the order in which to execute them.
- Specifying the order in which statements execute in a program is called program control.

Section 3.3 Pseudocode

- Pseudocode helps you think out a program before writing it in a programming language.
- Activity diagrams are part of the UML—an industry standard for modeling software systems.

Section 3.4 Control Structures

- An activity diagram models the workflow (also called the activity) of a software system.
- Activity diagrams are composed of symbols, such as action state symbols, diamonds and small circles, that are connected by transition arrows representing the flow of the activity.
- Like pseudocode, activity diagrams help you develop and represent algorithms.
- An action state is represented as a rectangle with its left and right sides replaced with arcs curving outward. The action expression appears inside the action state.
- The arrows in an activity diagram represent transitions, which indicate the order in which the actions represented by action states occur.
- The solid circle in an activity diagram represents the initial state—the beginning of the workflow before the program performs the modeled actions.
- The solid circle surrounded by a hollow circle that appears at the bottom of the activity diagram represents the final state—the end of the workflow after the program performs its actions.
- Rectangles with the upper-right corners folded over are called notes in the UML. A dotted line connects each note with the element that the note describes.
- A decision symbol in an activity diagram indicates that a decision is to be made. The workflow follows a path determined by the associated guard conditions. Each transition arrow emerging from a decision symbol has a guard condition. If a guard condition is true, the workflow enters the action state to which the transition arrow points.
- A merge symbol has two or more transition arrows pointing to the diamond and only one transition arrow pointing from it, to indicate multiple activity flows merging to continue the activity.
- Top-down, stepwise refinement is a process for refining pseudocode by maintaining a complete representation of the program during each refinement.
- There are three types of control statements—sequence, selection and repetition.
- The sequence statement is built in—by default, statements execute in the order they appear.
- A selection statement chooses among alternative courses of action.

Section 3.5 `if` Selection Statement

- The `if` single-selection statement either performs (selects) an action if a condition is true, or skips the action if the condition is false.

Section 3.6 `if...else` Double-Selection Statement

- The `if...else` double-selection statement performs (selects) an action if a condition is true and performs a different action if the condition is false.
- To include several statements in an `if`'s body (or the body of an `else` for an `if...else` statement), enclose the statements in braces (`{` and `}`). A set of statements contained in braces is called a block. A block can be placed anywhere in a program that a single statement can be placed.
- A null statement, indicating that no action is to be taken, is indicated by a semicolon (`;`).

Section 3.7 `while` Repetition Statement

- A repetition statement repeats an action while some condition remains true.
- A value that contains a fractional part is referred to as a floating-point number and is represented approximately by data types such as `float` and `double`.

Section 3.8 Formulating Algorithms: Counter-Controlled Repetition

- Counter-controlled repetition is used when the number of repetitions is known before a loop begins executing, i.e., when there is definite repetition.
- The unary cast operator `static_cast<double>` can be used to create a temporary floating-point copy of its operand.
- Unary operators take only one operand; binary operators take two.
- The parameterized stream manipulator `setprecision` indicates the number of digits of precision that should be displayed to the right of the decimal point.
- The stream manipulator `fixed` indicates that floating-point values should be output in so-called fixed-point format, as opposed to scientific notation.

Section 3.9 Formulating Algorithms: Sentinel-Controlled Repetition

- Sentinel-controlled repetition is used when the number of repetitions is not known before a loop begins executing, i.e., when there is indefinite repetition.

Section 3.10 Formulating Algorithms: Nested Control Statements

- A nested control statement appears in the body of another control statement.

Section 3.11 Assignment Operators

- The arithmetic operators `+=`, `-=`, `*=`, `/=` and `%=` abbreviate assignment expressions.

Section 3.12 Increment and Decrement Operators

- The increment operator, `++`, and the decrement operator, `--`, increment or decrement a variable by 1, respectively. If the operator is prefixed to the variable, the variable is incremented or decremented by 1 first, then its new value is used in the expression in which it appears. If the operator is postfixed to the variable, the variable is first used in the expression in which it appears, then the variable's value is incremented or decremented by 1.

Terminology

action/decision model of programming 73
action 67
action expression 70
action state symbol 70
activity 70
activity diagram 69
addition assignment operator (+=) 96
algorithm 67
assignment operators 96
block 77
`bool` fundamental type 73
cast operator 87
compound statement 77
conditional expression 75
conditional operator (?:) 75
control-statement nesting 72
control-statement stacking 72
control statements 67
control structures 69
counter 80
counter-controlled repetition 80
"crashing" 86
dangling-`else` problem 76
decision symbol 72
decrement operator (--) 97
default precision 91
definite repetition 80
diamond 70
divide by zero 86
`do...while` repetition statement 71
dotted line 70
`double` fundamental type 87
double-precision floating-point number 90
double-selection statement 70
dummy value 84
empty statement 78
executable statement 68
explicit conversion 91
`false` 73
fatal logic error 86

final state symbol 70
first refinement 84
`fixed` 91
fixed-point format 91
`fixed` stream manipulator 91
flag value 84
`float` fundamental type 87
floating-point constants 90
floating-point number 87
`for` repetition statement 71
"garbage" value 82
`goto` elimination 69
`goto` statement 69
guard condition 72
implicit conversion 91
increment operator (++) 97
indefinite repetition 84
initial state 70
iteration of a loop 80
loop-continuation condition 71
looping statement 71
loops 71
merge symbol 79
multiple-selection statement 71
multiplicative operator 91
nested control statement 92
nested `if...else` statement 75
nonparameterized stream manipulator 91
note in the UML 70
null statement 78
off-by-one error 84
order 67
parameterized stream manipulator 91
postdecrement 98
postfix decrement operator 97
postfix increment operator 97
postincrement 98
precision 87
predecrement 98
prefix decrement operator 97
prefix increment operator 97
preincrement 98
procedure 67
program control 67
promote 91
promotion 91
pseudocode 68
repetition statement 78
repetition structure 69
rounding a floating-point number 92
rounding 83
scientific notation 91
second refinement 85
selection structure 69
sentinel value 84
sentinel-controlled repetition 85
sequence structure 69
sequential execution 69
`setprecision` stream manipulator 91
`showpoint` stream manipulator 92
signal value 84
single-entry/single-exit control statement 72
single-precision floating-point number 90
single-selection statement 70
solid circle symbol 70
`static_cast` operator 91
structured programming 67
ternary operator 75
top-down, stepwise refinement 84
total 81
transfer of control 69
transition 70
transition arrow symbol 70
`true` 73
truncate 83
unary cast operator 91
unary operator 91
undefined value 82
`while` repetition statement 71
workflow 70

Self-Review Exercises

3.1 Answer each of the following questions.

a) All programs can be written in terms of three types of control statements: _____, _____ and _____.

b) The _____selection statement is used to execute one action when a condition is `true` or a different action when that condition is `false`.

c) Repeating a set of instructions a specific number of times is called _____ repetition.

d) When it isn't known in advance how many times a set of statements will be repeated, a(n)_____value can be used to terminate the repetition.

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

3.3 Write C++ statements to accomplish each of the following:
a) In one statement, assign the sum of the current value of `x` and `y` to `z` and postincrement the value of `x`.
b) Determine whether the value of the variable `count` is greater than 10. If it is, print `"Count is greater than 10."`
c) Predecrement 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.

3.4 Write C++ statements to accomplish each of the following tasks.
a) Declare variables `sum` and `x` to be of type `int`.
b) Set variable `x` to `1`.
c) Set 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`.

3.5 Combine the statements that you wrote in Exercise 3.4 into a program that calculates and prints the sum of the integers from 1 to 10. Use the `while` statement to loop through the calculation and increment statements. The loop should terminate when the value of `x` becomes 11.

3.6 State 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;`

3.7 Write single C++ statements or portions of statements that do the following:
a) Input integer variable `x` with `cin` and `>>`.
b) Input integer variable `y` with `cin` and `>>`.
c) Set integer variable `i` to `1`.
d) Set integer variable `power` to `1`.
e) Multiply variable `power` by `x` and assign the result to `power`.
f) Preincrement variable `i` by `1`.
g) Determine whether `i` is less than or equal to `y`.
h) Output integer variable `power` with `cout` and `<<`.

3.8 Write a C++ program that uses the statements in Exercise 3.7 to calculate `x` raised to the `y` power. The program should have a `while` repetition statement.

3.9 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;
```

3.10 What's wrong with the following `while` repetition statement?

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

Answers to Self-Review Exercises

3.1 a) Sequence, selection and repetition. b) `if...else`. c) Counter-controlled or definite. d) Sentinel, signal, flag or dummy.

3.2
```
x = x + 1;
x += 1;
++x;
x++;
```

3.3 a)
```
z = x++ + y;
```
b)
```
if ( count > 10 )
   cout << "Count is greater than 10" << endl;
```
c)
```
total -= --x;
```
d)
```
q %= divisor;
```
or
```
q = q % divisor;
```

3.4 a)
```
int sum;
int x;
```
b)
```
x = 1;
```
c)
```
sum = 0;
```
d)
```
sum += x;
```
or
```
sum = sum + x;
```
e)
```
cout << "The sum is: " << sum << endl;
```

3.5 See the following code:

```
1  // Exercise 3.5 Solution: ex03_05.cpp
2  // Calculate the sum of the integers from 1 to 10.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int sum; // stores sum of integers 1 to 10
9     int x; // counter
10
11    x = 1; // count from 1
12    sum = 0; // initialize sum
13
14    while ( x <= 10 ) // loop 10 times
15    {
16       sum += x; // add x to sum
17       ++x; // increment x
18    } // end while
19
20    cout << "The sum is: " << sum << endl;
21 } // end main
```

```
The sum is: 55
```

3.6 a)
```
product = 25, x = 6;
```
b)
```
quotient = 0, x = 6;
```

```
// Exercise 3.6 Solution: ex03_06.cpp
// Calculate the value of product and quotient.
#include <iostream>
using namespace std;

int main()
{
   int x = 5;
   int product = 5;
   int quotient = 5;

   // part a
   product *= x++; // part a statement
   cout << "Value of product after calculation: " << product << endl;
   cout << "Value of x after calculation: " << x << endl << endl;

   // part b
   x = 5; // reset value of x
   quotient /= ++x; // part b statement
   cout << "Value of quotient after calculation: " << quotient << endl;
   cout << "Value of x after calculation: " << x << endl << endl;
} // end main
```

```
Value of product after calculation: 25
Value of x after calculation: 6

Value of quotient after calculation: 0
Value of x after calculation: 6
```

3.7
a) `cin >> x;`
b) `cin >> y;`
c) `i = 1;`
d) `power = 1;`
e) `power *= x;`
   or
   `power = power * x;`
f) `++i;`
g) `if ( i <= y )`
h) `cout << power << endl;`

3.8 See the following code:

```
// Exercise 3.8 Solution: ex03_08.cpp
// Raise x to the y power.
#include <iostream>
using namespace std;

int main()
{
   int x; // base
   int y; // exponent
   int i; // counts from 1 to y
   int power; // used to calculate x raised to power y

   i = 1; // initialize i to begin counting from 1
   power = 1; // initialize power
```

```
15
16      cout << "Enter base as an integer: ";  // prompt for base
17      cin >> x; // input base
18
19      cout << "Enter exponent as an integer: "; // prompt for exponent
20      cin >> y; // input exponent
21
22      // count from 1 to y and multiply power by x each time
23      while ( i <= y )
24      {
25         power *= x;
26         ++i;
27      } // end while
28
29      cout << power << endl; // display result
30   } // end main
```

```
Enter base as an integer: 2
Enter exponent as an integer: 3
8
```

3.9 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`.

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

Exercises

3.11 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;
   }
```

3.12 What does the following program print?

```
// Exercise 3.12: ex03_12.cpp
// What does this program print?
#include <iostream>
using namespace std;

int main()
{
   int y; // declare y
   int x = 1; // initialize x
   int total = 0; // initialize total

   while ( x <= 10 ) // loop 10 times
   {
      y = x * x; // perform calculation
      cout << y << endl; // output result
      total += y; // add y to total
      x++; // increment counter x
   } // end while

   cout << "Total is " << total << endl; // display result
} // end main
```

For Exercises 3.13–3.16, 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.

3.13 *(Gas Mileage)* 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 uses a `while` statement to input the miles driven and gallons used for each tankful. The program should calculate and display the miles per gallon obtained for each tankful and print the combined miles per gallon obtained for all tankfuls up to this point.

```
Enter miles driven (-1 to quit): 287
Enter gallons used: 13
MPG this tankful: 22.076923
Total MPG: 22.076923

Enter miles driven (-1 to quit): 200
Enter gallons used: 10
MPG this tankful: 20.000000
Total MPG: 21.173913

Enter the miles driven (-1 to quit): 120
Enter gallons used: 5
MPG this tankful: 24.000000
Total MPG: 21.678571

Enter the miles used (-1 to quit): -1
```

3.14 *(Credit Limit Calculator)* Develop a C++ program that will determine whether 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 use a `while` statement to input each of these facts, calculate the new balance (= beginning balance + charges – credits) and determine whether 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."

```
Enter account number (or -1 to quit): 100
Enter beginning balance: 5394.78
Enter total charges: 1000.00
Enter total credits: 500.00
Enter credit limit: 5500.00
New balance is 5894.78
Account:      100
Credit limit: 5500.00
Balance:      5894.78
Credit Limit Exceeded.

Enter Account Number (or -1 to quit): 200
Enter beginning balance: 1000.00
Enter total charges: 123.45
Enter total credits: 321.00
Enter credit limit: 1500.00
New balance is 802.45

Enter Account Number (or -1 to quit): -1
```

3.15 *(Sales Commission Calculator)* A large company pays its salespeople on a commission basis. The salespeople each receive $200 per week plus 9% of their gross sales for that week. For example, a salesperson who sells $5000 worth of chemicals in a week receives $200 plus 9% of $5000, or a total of $650. Develop a C++ program that uses a `while` statement to input each salesperson's gross sales for last week and calculates and displays that salesperson's earnings. Process one salesperson's figures at a time.

```
Enter sales in dollars (-1 to end): 5000.00
Salary is: $650.00

Enter sales in dollars (-1 to end): 6000.00
Salary is: $740.00

Enter sales in dollars (-1 to end): 7000.00
Salary is: $830.00

Enter sales in dollars (-1 to end): -1
```

3.16 *(Salary Calculator)* Develop a C++ program that uses a `while` statement to 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.

```
Enter hours worked (-1 to end): 39
Enter hourly rate of the employee ($00.00): 10.00
Salary is $390.00

Enter hours worked (-1 to end): 40
Enter hourly rate of the employee ($00.00): 10.00
Salary is $400.00

Enter hours worked (-1 to end): 41
Enter hourly rate of the employee ($00.00): 10.00
Salary is $415.00

Enter hours worked (-1 to end): -1
```

3.17 ***(Find the Largest)*** 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 inputs the number of units sold by each salesperson. The salesperson who sells the most units wins the contest. Write a C++ program that uses a `while` statement to determine and print the largest number of 10 numbers input by the user. 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.

3.18 ***(Tabular Output)*** Write a C++ program that uses a `while` statement and the tab escape sequence `\t` to print the following table of values:

```
N       10*N    100*N   1000*N

1       10      100     1000
2       20      200     2000
3       30      300     3000
4       40      400     4000
5       50      500     5000
```

3.19 ***(Find the Two Largest Numbers)*** Using an approach similar to that in Exercise 3.17, find the *two* largest values among the 10 numbers. [*Note:* You must input each number only once.]

3.20 ***(Validating User Input)*** The examination-results program of Fig. 3.12 assumes that any value input by the user that's not a 1 must be a 2. Modify the application 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.

3.21 What does the following program print?

```
1  // Exercise 3.21: ex03_21.cpp
2  // What does this program print?
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int count = 1; // initialize count
9
10    while ( count <= 10 ) // loop 10 times
11    {
12       // output line of text
13       cout << ( count % 2 ? "****" : "++++++++" ) << endl;
14       ++count; // increment count
15    } // end while
16 } // end main
```

3.22 What does the following program print?

```
// Exercise 3.22: ex03_22.cpp
// What does this program print?
#include <iostream>
using namespace std;

int main()
{
   int row = 10; // initialize row
   int column; // declare column

   while ( row >= 1 ) // loop until row < 1
   {
      column = 1; // set column to 1 as iteration begins

      while ( column <= 10 ) // loop 10 times
      {
         cout << ( row % 2 ? "<" : ">" ); // output
         ++column; // increment column
      } // end inner while

      --row; // decrement row
      cout << endl; // begin new output line
   } // end outer while
} // end main
```

3.23 *(Dangling-`else` Problem)* State the output for each of the following when `x` is `9` and `y` is `11` and when `x` is `11` and `y` is `9`. The compiler ignores the indentation in a C++ program. The C++ compiler always associates an `else` with the previous `if` unless told to do otherwise by the placement of braces `{}`. On first glance, you may not be sure which `if` and `else` match, so this is referred to as the "dangling-`else`" problem. We eliminated the indentation from the following code to make the problem more challenging. [*Hint:* Apply indentation conventions you've learned.]

a)
```
if ( x < 10 )
if ( y > 10 )
cout << "*****" << endl;
else
cout << "#####" << endl;
cout << "$$$$$" << endl;
```

b)
```
if ( x < 10 )
{
if ( y > 10 )
cout << "*****" << endl;
}
else
{
cout << "#####" << endl;
cout << "$$$$$" << endl;
}
```

3.24 *(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 eliminated the indentation from the following code to make the problem more challenging. [*Note:* It's possible that no modification is necessary.]

```
if ( y == 8 )
if ( x == 5 )
cout << "@@@@@" << endl;
else
cout << "#####" << endl;
cout << "$$$$$" << endl;
cout << "&&&&&" << endl;
```

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

```
@@@@@
$$$$$
&&&&&
```

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

```
@@@@@
```

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

```
@@@@@
&&&&&
```

d) Assuming `x = 5` and `y = 7`, the following output is produced. [*Note:* The last three output statements after the `else` are all part of a block.]

```
#####
$$$$$
&&&&&
```

3.25 ***(Square of Asterisks)*** Write a program that reads in the size of the side of a square 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

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

3.26 ***(Palindromes)*** A palindrome is a number or a text phrase that reads the same backward as forward. 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's a palindrome. [*Hint:* Use the division and modulus operators to separate the number into its individual digits.]

3.27 ***(Printing the Decimmal Equivalent of a Binary Number)*** Input an integer containing only 0s and 1s (i.e., a "binary" integer) and print its decimal equivalent. Use the modulus and division operators to pick off the "binary" number's digits one at a time from right to left. Much as in the decimal number system, where the rightmost digit has a positional value of 1, the next digit left has a positional value of 10, then 100, then 1000, and so on, 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, and so on. Thus the decimal number 234 can be interpreted as 2 * 100 + 3 * 10 + 4 * 1. The decimal equivalent of binary 1101 is 1 * 1 + 0 * 2 + 1 * 4 + 1 * 8 or 1 + 0 + 4 + 8, or 13. [*Note:* To learn more about binary numbers, refer to Appendix D.]

3.28 *(Checkerboard Pattern of Asterisks)* Write a program that displays the checkerboard pattern shown below. Your program must use only three output statements, one of each of the following forms:

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

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

3.29 *(Multiples of 2 with an Infinite Loop)* Write a program that prints the powers of the integer 2, namely 2, 4, 8, 16, 32, 64, etc. Your `while` loop should not terminate (i.e., you should create an infinite loop). To do this, simply use the keyword `true` as the expression for the `while` statement. What happens when you run this program?

3.30 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 π.

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

```
cout << ++( x + y );
```

3.32 *(Sides of a Triangle)* Write a program that reads three nonzero `double` values and determines and prints whether they could represent the sides of a triangle.

3.33 *(Sides of a Right Triangle)* Write a program that reads three nonzero integers and determines and prints whether they could be the sides of a right triangle.

3.34 *(Factorial)* The factorial of a nonnegative integer n is written $n!$ (pronounced "n factorial") and is defined as follows:

$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$ (for values of n greater than 1)

and

$n! = 1$ (for $n = 0$ or $n = 1$).

For example, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, which is 120. Use `while` statements in each of the following:

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:

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

Prompt the user for the desired accuracy of e (i.e., the number of terms in the summation).

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

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

Prompt the user for the desired accuracy of e (i.e., the number of terms in the summation).

Making a Difference

3.35 *(Enforcing Privacy with Cryptography)* The explosive growth of Internet communications and data storage on Internet-connected computers has greatly increased privacy concerns. The field

of cryptography is concerned with coding data to make it difficult (and hopefully—with the most advanced schemes—impossible) for unauthorized users to read. In this exercise you'll investigate a simple scheme for encrypting and decrypting data. A company that wants to send data over the Internet has asked you to write a program that will encrypt it so that it may be transmitted more securely. All the data is transmitted as four-digit integers. Your application should read a four-digit integer entered by the user and encrypt it as follows: Replace each digit with the result of adding 7 to the digit and getting the remainder after dividing the new value by 10. Then swap the first digit with the third, and swap the second digit with the fourth. Then print the encrypted integer. Write a separate application that inputs an encrypted four-digit integer and decrypts it (by reversing the encryption scheme) to form the original number. [*Optional reading project:* Research "public key cryptography" in general and the PGP (Pretty Good Privacy) specific public key scheme. You may also want to investigate the RSA scheme, which is widely used in industrial-strength applications.]

3.36 ***(World Population Growth)*** World population has grown considerably over the centuries. Continued growth could eventually challenge the limits of breathable air, drinkable water, arable cropland and other limited resources. There is evidence that growth has been slowing in recent years and that world population could peak some time this century, then start to decline.

For this exercise, research world population growth issues online. *Be sure to investigate various viewpoints.* Get estimates for the current world population and its growth rate (the percentage by which it's likely to increase this year). Write a program that calculates world population growth each year for the next 75 years, *using the simplifying assumption that the current growth rate will stay constant*. Print the results in a table. The first column should display the year from year 1 to year 75. The second column should display the anticipated world population at the end of that year. The third column should display the numerical increase in the world population that would occur that year. Using your results, determine the year in which the population would be double what it is today, if this year's growth rate were to persist.