Chapter 3, Control Statements: Part 1

C++ How to Program, Late Objects Version, 7/e

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.

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.

3.2 Algorithms

- Any solvable computing problem can be solved by the execution of a series of actions in a specific order.
- An algorithm is procedure for solving a problem in terms of
  - the actions to execute and
  - the order in which the actions execute
- Specifying the or-der 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.
- Similar to everyday English
- Convenient and user friendly.
- Helps you "think out" a program before attempting to write it.
- Carefully prepared pseudocode can easily be converted to a corresponding C++ program.
- Normally describes only executable statements.
- Declarations (that do not have initializers or do not involve constructor calls) are not executable statements.
- Fig. 3.1 corresponds to the algorithm that inputs two integers from the user, adds these integers and displays their sum.

```
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.

3.4 Control Structures

- Normally, statements in a program execute one after the other in the order in which they're written.
  - Called sequential execution.
- Various C++ statements enable you to specify that the next statement to execute may be other than the next one in sequence.
  - Called transfer of control.
- All programs could be written in terms of only three control structures
  - the sequence structure
  - the selection structure and
  - the repetition structure
- 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."

3.4 Control Structures (cont.)

- 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 a typical sequence structure in which two calculations are performed in order.
- C++ allows us to have as many actions as we want in a sequence structure.
- Anywhere a single action may be placed, we may place several actions in sequence.

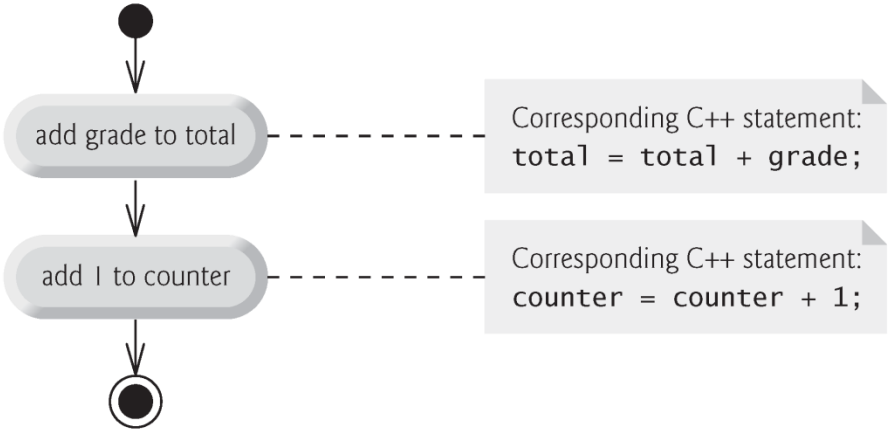

Fig. 3.2 | Sequence-statement activity diagram.

3.4 Control Structures (cont.)

- 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 structure 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 structures operate.
- Action states represent actions to perform.
  - Each contains an action expression that specifies a particular action to perform.

3.4 Control Structures (cont.)

- The arrows in the activity diagram are called transition arrows.
  - Represent transitions, which indicate the order in which the actions represented by the action states occur.
- 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.

3.4 Control Structures (cont.)

- Rectangles with the upper-right corners folded over are called notes in the UML.
  - Explanatory remarks that describe the purpose of symbols in the diagram.
  - Notes can be used in any UML 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.

3.4 Control Structures (cont.)

- 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.

3.4 Control Structures (cont.)

- 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).

3.4 Control Structures (cont.)

- 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.
- The `while` and `for` statements perform the action (or group of actions) in their bodies zero or more times.
- The `do…while` statement performs the action (or group of actions) in its body at least once.

3.4 Control Structures (cont.)

- 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.
- Keywords must not be used as identifiers, such as variable names.
- 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.

3.4 Control Structures (cont.)

- Each program combines control statements as appropriate for the algorithm the program implements.
- Cn 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.
- Single-entry/single-exit control statements
  - Control statements are attached to one another by con-necting the exit point of one to the entry point of the next.
  - Called control-statement stacking.
  - 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.
- The following pseudocode determines whether "student's grade is greater than or equal to 60" is `true` or `false`.

  If student's grade is greater than or equal to 60
  Print "Passed"

  - If `true`, "Passed" is printed and the next pseudocode statement in order is "performed" (remember that pseudocode is not a real programming language).
  - If `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.

3.5 if Selection Statement (cont.)

- The preceding pseudocode *If* statement can be written in C++ as
  - ```
    if ( grade >= 60 )
       cout << "Passed";
    ```
- Figure 3.4 illustrates the single-selection `if` statement.
- The diamond or decision symbol indicates that a decision is to be made.
  - 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 in square brackets above or next to the transition arrow.
  - If a guard condition is true, the workflow enters the action state to which that transition arrow points.
    ```

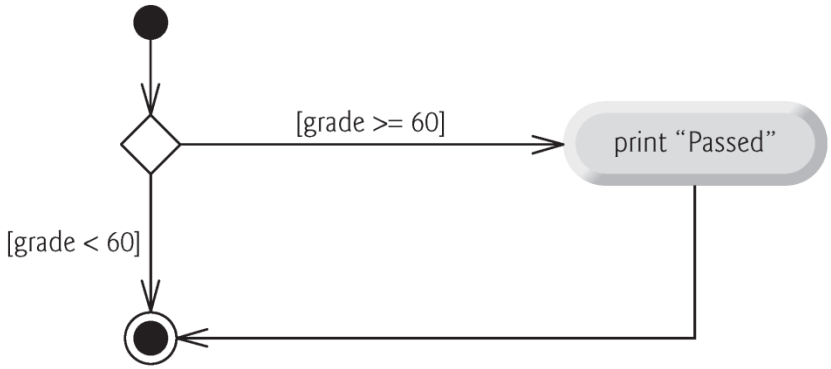

Fig. 3.4 | `if` single-selection statement activity diagram.

3.5 `if` Selection Statement (cont.)

- 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.*

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

- `if...else` double-selection statement
  - specifies an action to perform when the condition is true and a different action to perform when the condition is `false`.
- The following pseudocode prints "Passed" if the student's grade is greater than or equal to 60, or "Failed" if the student's grade is less than 60.

```
If student's grade is greater than or equal to 60
        Print "Passed"
Else
        Print "Failed"
```

- 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";
```

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.

3.6 `if...else` Double-Selection Statement (cont.)

- Figure 3.5 illustrates the the `if...else` statement's flow of control.

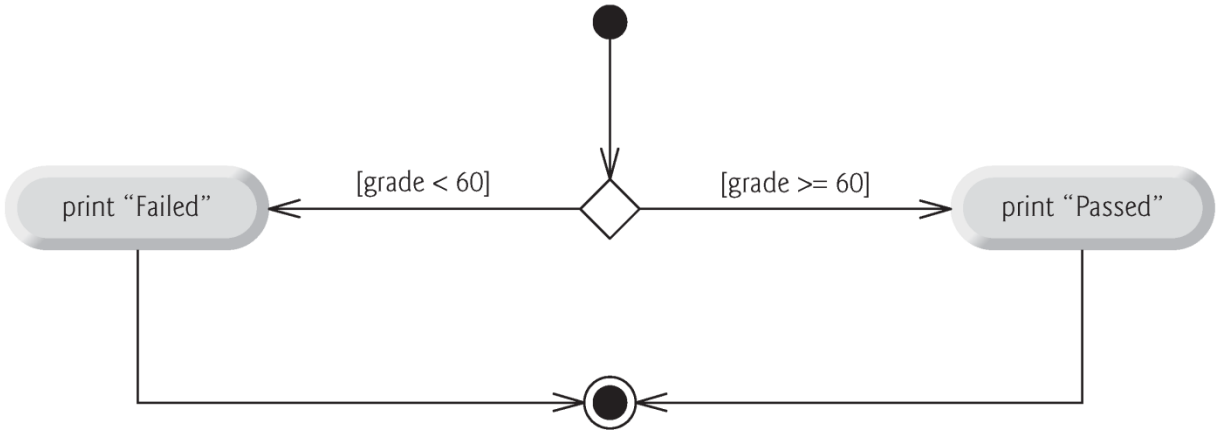

Fig. 3.5 | `if...else` double-selection statement activity diagram.

3.6 `if...else` Double-Selection Statement (cont.)

- Conditional operator (`?:`)
  - Closely related to the `if...else` statement.
- 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`
  - The third operand is the value for the entire conditional expression if the condition is `false`.
- The values in a conditional expression also can be actions to execute.

Error-Prevention Tip 4.1

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

3.6 `if...else` Double-Selection Statement (cont.)

- Nested `if...else` statements test for multiple cases by placing `if...else` selection statements inside other `if...else` selection statements.

```
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"
```

3.6 if...else Double-Selection Statement (cont.)

- 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.

3.6 if...else Double-Selection Statement (cont.)

- 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.
    ```

Error-Prevention Tip 3.1

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

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.*

3.6 if...else Double-Selection Statement (cont.)

- 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.
  - ```
    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`.
    ```

3.6 if...else Double-Selection Statement (cont.)

- The compiler actually interprets the statement as

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

- To force the nested `if...else` statement to execute as intended, use:

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

- Braces ({}) indicate that the second `if` statement is in the body of the first `if` and that the `else` is associated with the first `if`.

3.6 `if...else` Double-Selection Statement (cont.)

- 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 in-clude 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.

Software Engineering Observation 3.2

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

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.*

3.6 `if...else` Double-Selection Statement (cont.)

- 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.

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

- "There are more items on my shopping list" is true or false.
  - If true, "Purchase next item and cross it off my list" is performed.
    - 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, the repetition will terminate, and the first pseudocode statement after the repetition statement will execute.

3.7 while Repetition Statement (cont.)

- 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;
    ```
    ```

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.*

3.7 `while` Repetition Statement (cont.)

- The UML activity diagram of Fig. 3.6 illustrates the flow of control that corresponds to the preceding `while` statement.
- This diagram 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.
- 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.

3.7 while Repetition Statement (cont.)

- 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 from that point.
  - Each transition arrow 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.

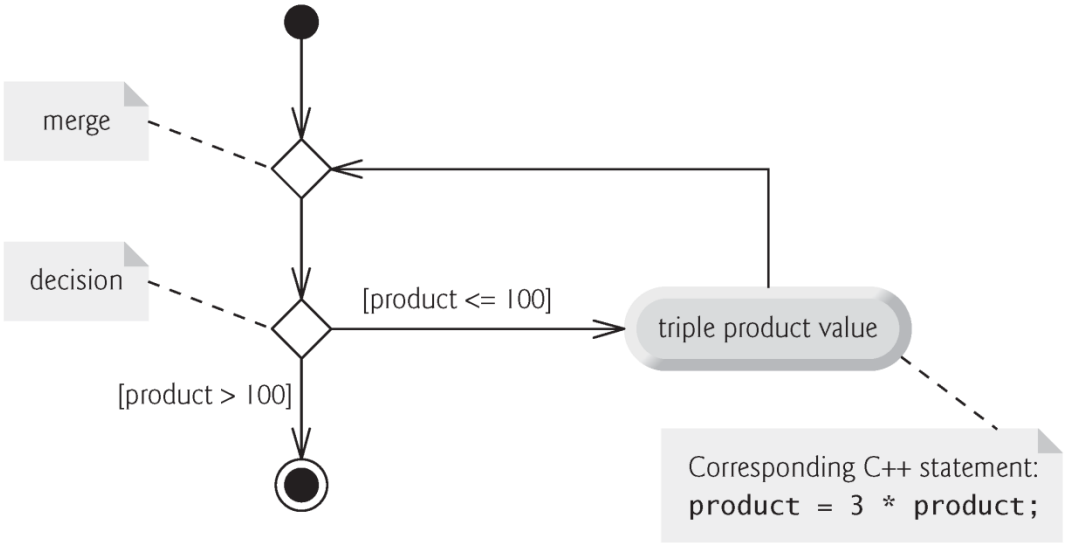

Fig. 3.6 | `while` repetition statement UML activity diagram.

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

- 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.

3.8 Formulating Algorithms: Counter-Controlled Repetition (cont.)

- 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).
  - Often called definite repetition because the number of repetitions is known before the loop begins exe-cut-ing.

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.

3.8 Formulating Algorithms: Counter-Controlled Repetition (cont.)

- 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 in-clude the previous value stored in the total's memory location.

```
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
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
```

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

```
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.)

3.8 Formulating Algorithms: Counter-Controlled Repetition (cont.)

- The averaging calculation is performed in member function `determineClass-Average` using local variables—we do not preserve any information about student grades in the class's data members.
  - In Chapter 7, Arrays and Vectors, we modify class `GradeBook` to maintain the grades in memory using a data member that refers to a data structure known as an array.

Good Programming Practice 3.6

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

3.8 Formulating Algorithms: Counter-Controlled Repetition (cont.)

- 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.

3.8 Formulating Algorithms: Counter-Controlled Repetition (cont.)

- The averaging calculation performed in response to the function call in line 12 of Fig. 3.10 produces an integer result.
- Dividing two integers results in integer division—any fractional part of the calculation is lost (i.e., truncated).

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.

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.
  - Develop a class average program that processes grades for an arbitrary number of students each time it's run.
- The program must process an arbitrary number of grades.
  - How can the program determine when to stop the input of grades?
- 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."
- Sentinel-controlled repetition is often called indefinite repetition
  - the number of repetitions is not known in advance.
- The sentinel value must not be an acceptable input value.

Common Programming Error 3.9

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

3.9 Formulating Algorithms: Sentinel-Controlled Repetition (cont.)

- We approach the class average program with a technique called top-down, stepwise refinement, a technique that is essential to the development of well-structured programs.
- We begin with a pseudocode representation of the top—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.
  - Rarely conveys sufficient detail from which to write a pro-gram.

3.9 Formulating Algorithms: Sentinel-Controlled Repetition (cont.)

- 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 structure—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.

3.9 Formulating Algorithms: Sentinel-Controlled Repetition (cont.)

- In the second refinement, we commit to specific variables.
- 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.

3.9 Formulating Algorithms: Sentinel-Controlled Repetition (cont.)

- 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.

3.9 Formulating Algorithms: Sentinel-Controlled Repetition (cont.)

- 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)

3.9 Formulating Algorithms: Sentinel-Controlled Repetition (cont.)

- 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"

- 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").

Common Programming Error 3.10

An attempt to divide by zero normally causes a fatal run-time 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.

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.

3.9 Formulating Algorithms: Sentinel-Controlled Repetition (cont.)

- An averaging calculation is likely to produce a number with a decimal point—a real number or floating-point number (e.g., 7.33, 0.0975 or 1000.12345).
- Type `int` cannot represent such a number.
- C++ provides several data types for storing floating-point numbers in memory, including `float` and `double`.
- Compared to `float` variables, `double` variables can typically store numbers with larger magnitude and finer detail
  - more digits to the right of the decimal point—also known as the number's precision.
- Cast operator can be used to force the averaging calculation to produce a floating-point numeric result.

```
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
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
```

Fig. 3.10 | Class average problem using sentinel-controlled repetition: `GradeBook` source code file. (Part 1 of 3.)

```
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
```

Fig. 3.10 | Class average problem using sentinel-controlled repetition: `GradeBook` source code file. (Part 2 of 3.)

```
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 3 of 3.)

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.

3.9 Formulating Algorithms: Sentinel-Controlled Repetition (cont.)

- Notice the block in the `while` loop in Fig. 3.13.
- 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 57).

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.

3.9 Formulating Algorithms: Sentinel-Controlled Repetition (cont.)

- 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
- C++ treats all floating-point numbers in a program's source code as `double` values by default.
  - Known as floating-point constants.
- Floating-point numbers often arise as a result of division.

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.

3.9 Formulating Algorithms: Sentinel-Controlled Repetition (cont.)

- The variable `average` is declared to be of type `double` to capture the fractional result of our calculation.
- `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 the division occurs *first*—the result's fractional part is lost before it's assigned to `average`:
  - `average = total / gradeCounter;`

3.9 Formulating Algorithms: Sentinel-Controlled Repetition (cont.)

- To perform a floating-point calculation with integers, create temporary floating-point values.
- Unary cast operator accomplishes this task.
- The cast operation `static_cast<double>(total)` creates a *temporary* floating-point copy of its operand in parentheses.
  - Known as explicit conversion.
  - The value stored in `total` is still an integer.

3.9 Formulating Algorithms: Sentinel-Controlled Repetition (cont.)

- The calculation now consists of a floating-point value divided by the integer `gradeCounter`.
  - The compiler knows how to evaluate only expressions in which the operand types of are identical.
  - Compiler performs promotion (also called implicit conversion) on selected operands.
  - In an expression containing values of data types `int` and `double`, C++ promotes `int` operands to `double` values.
- Cast operators are available for use with every data type and with class types as well.

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.*

3.9 Formulating Algorithms: Sentinel-Controlled Repetition (cont.)

- The call to `setprecision` in line 79 (with an argument of `2`) indicates that `double` values should be printed with two digits of precision to the right of the decimal point (e.g., 92.37).
  - Parameterized stream manipulator (argument in parentheses).
  - Programs that use these must include the header `<iomanip>`.
- `endl` is a nonparameterized stream manipulator 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.

3.9 Formulating Algorithms: Sentinel-Controlled Repetition (cont.)

- Stream manipulator `fixed` indicates that floating-point values should be output in fixed-point format, as opposed to scientific notation.
- Fixed-point formatting 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 decimal point.

3.9 Formulating Algorithms: Sentinel-Controlled Repetition (cont.)

- 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 re-mains unaltered.
- 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.
  - Both can be found in header `<iostream>`.

3.10 Formulating Algorithms: Nested Control Statements

- Consider the following problem statement:
  - A college offers a course that prepares students for the state licensing exam for real es-tate 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!"

3.10 Formulating Algorithms: Nested Control Statements (cont.)

- After reading the problem statement carefully, we make the following observations:
  - Must process test results for 10 students. A counter-controlled loop can be used because the number of test results is known in advance.
  - 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.)
  - Two counters keep track of the exam results—one to count the number of students who passed and one to count the number of students who failed.
  - After the program has processed all the results, it must decide whether more than eight students passed the exam.

3.10 Formulating Algorithms: Nested Control Statements (cont.)

- 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

- Further refinement is necessary.

3.10 Formulating Algorithms: Nested Control Statements (cont.)

- Counters are needed to record the passes and failures, a counter will be used to control the looping process and a variable is needed to store the user input.
- The pseudocode statement

  Initialize variables

  can be refined as follows:

  Initialize passes to zero

  Initialize failures to zero

  Initialize student counter to one

3.10 Formulating Algorithms: Nested Control Statements (cont.)

- The following pseudocode statement requires a loop that successively inputs the result of each exam

  Input the 10 exam results, and count passes and failures

- 10 exam results, so counter-controlled looping is appropriate.
- Nested inside 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
```

3.10 Formulating Algorithms: Nested Control Statements (cont.)

- 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. 3.15.

```
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.

```
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
```

Fig. 3.12 | Examination-results problem: Nested control statements. (Part 1 of 4.)

```
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 2 of 4.)

```
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!
```

Fig. 3.12 | Examination-results problem: Nested control statements. (Part 3 of 4.)

```
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 4 of 4.)

3.10 Formulating Algorithms: Nested Control Statements (cont.)

- C++ allows variable initialization to be incorporated into declarations.
- The `if…else` statement (lines 22–25) for processing each result is nested in the `while` statement.
- 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!"`.

3.11 Assignment Operators

- C++ provides several assignment operators for abbreviating assignment expressions.
- 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 3.17 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

- 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. 3.18.

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.

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.

```
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
```

Fig. 3.15 | Preincrementing and postincrementing. (Part 1 of 2.)

```
5
5
6

5
6
6
```

Fig. 3.15 | Preincrementing and postincrementing. (Part 2 of 2.)

3.12 Increment and Decrement Operators (cont.)

- 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 post-decrementing).
- Figure 3.20 shows the precedence and associativity of the operators introduced to this point.

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.*

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.