# Algorithm Development and Control Statements: Part 1

Chapter 4 of C++ How to Program, 10/e

# OBJECTIVES

In this chapter you'll:

- Learn basic problem-solving techniques.
- Develop algorithms through the process of top-down, stepwise refinement.
- Use the `if` and `if...else` selection statements to choose between alternative actions.
- Use the `while` iteration statement to execute statements in a program repeatedly.
- Use counter-controlled iteration and sentinel-controlled iteration.
- Use nested control statements.
- Use the compound assignment operator and the increment and decrement operators.
- Learn about the portability of fundamental data types.

# 4.1 Introduction

- Before writing a program to solve a problem, have a thorough understanding of the problem and a carefully planned approach to solving it.
  - Understand the available building blocks and employ proven program-construction techniques.

# 4.2 Algorithms

- Any solvable computing problem can be solved by the execution a series of actions in a specific order.
- An algorithm is a procedure for solving a problem in terms of
  - the actions to execute and
  - the order in which these actions execute
- 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.

# 4.3 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 structured portions of C++ programs.
- Normally describes only executable statements.
- Declarations (that do not have initializers or do not involve constructor calls) are not executable statements.
- Fig. 4.1 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. 4.1** | Pseudocode for the addition program of Fig. 2.5.

# 4.4 Control Structures

- Normally, statements in a program are executed 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 iteration 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."

# 4.4.1 Sequence Structure

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

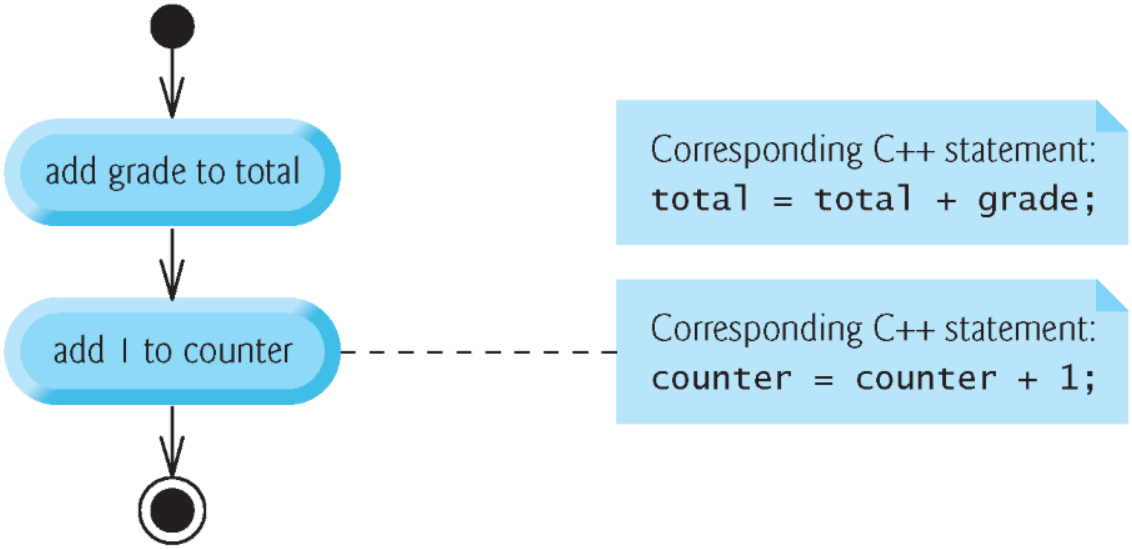


**Fig. 4.2** | Sequence-structure activity diagram.

# 4.4.1 Sequence Structure

- 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. 4.2.
- Activity diagrams are composed of special-purpose symbols, such as action state symbols (rectangles with their 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.

# 4.4.1 Sequence Structure

- 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.

# 4.4.1 Sequence Structure

- 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.

# 4.4.1 Sequence Structure

- 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 4.2 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.

# 4.4.2 Selection Statements

- C++ has three types of selection statements (discussed in this chapter and Chapter 5).
  - The `if` selection statement either performs (selects) an action (or group of actions) if a condition (predicate) is true or skips the action (or group of actions) if the condition is false.
  - The `if...else` selection statement performs an action (or group of actions) if a condition is true or performs a different action (or group of actions) if the condition is false.
  - The `switch` selection statement (Chapter 5) performs one of many different actions (or groups of actions), depending on the value of an integer expression.

# 4.4.2 Selection Statements

- The `if` selection statement is a single-selection statement because it selects or ignores a *single* action (or 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).

# 4.4.3 Iteration Statements

- C++ provides three types of iteration statements (also called looping statements or loops) for performing statements repeatedly while a condition (called the loop-continuation condition) remains true—`while`, `do…while` and `for`.
  - 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*.

# 4.4.3 Iteration Statements

- 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 *cannot* be used as identifiers, such as variable names.
- Figure 4.3 provides a complete list of C++ keywords-
.

## C++ Keywords

*Keywords common to the C and C++ programming languages*

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

*C++-only keywords*

| | | | | |
|---|---|---|---|---|
| and | and_eq | bitand | bitor | bool |
| catch | class | compl | const_cast | delete |
| dynamic_cast | explicit | export | false | friend |
| 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. 4.3** | C++ keywords. (Part 1 of 2.)

| C++ Keywords | | | | |
|---|---|---|---|---|
| *C++11 keywords* | | | | |
| `alignas` | `alignof` | `char16_t` | `char32_t` | `constexpr` |
| `decltype` | `noexcept` | `nullptr` | `static_assert` | `thread_local` |

**Fig. 4.3** | C++ keywords. (Part 2 of 2.)

# 4.4.4 Summary of Control Statements

- Each program is formed by combining as many of each of these control statements as appropriate for the algorithm the program implements.
- We can model each control statement as an activity diagram with initial and final states representing that control statement's entry and exit points, respectively.
- Single-entry/single-exit control statements
  - Connecting the exit point of one control statement 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.

# 4.5 `if` Single-Selection Statement

- Programs use selection statements to choose among alternative courses of action.
- Pseudocode to determine whether "student's grade is greater than or equal to 60" is true.

  *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.

# 4.5 `if` Selection Statement (cont.)

- The preceding pseudocode *If* statement can be written in C++ as
  - ```
    if (studentGrade >= 60) {
       cout << "Passed";
    }
    ```
- In C++, a decision can be based on any expression that evaluates to zero or nonzero
  - If the expression evaluates to zero, it's treated as false; if the expression evaluates to nonzero, it's treated as true.
- C++ also provides the data type `bool` for Boolean variables that can hold only the values `true` and `false`.
    ```

## Portability Tip 4.1

*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.

# 4.5 `if` Selection Statement (cont.)

- Figure 4.4 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 arrow.
  - If a guard condition is true, the workflow enters the action state to which that transition arrow points.

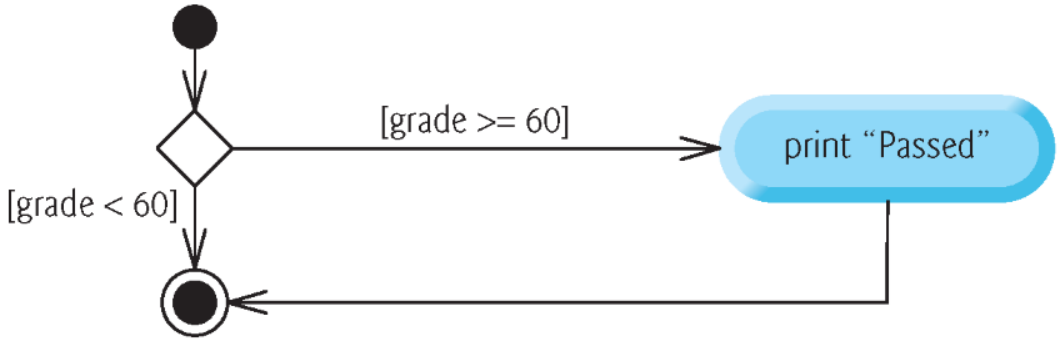


**Fig. 4.4** | `if` single-selection statement UML activity diagram.

# 4.6 `if...else` Double-Selection Statement

- `if...else` double-selection statement
  - specifies an action (or group of actions) to perform when the condition is `true` and a different action to perform when the condition is `false`.
- Pseudocode that prints "Passed" if the student's grade is greater than or equal to 60, or "Failed" otherwise.

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

# 4.6 if...else Double-Selection Statement

- The preceding *If...Else* pseudocode can be written in C++ as

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

## Good Programming Practice 4.1

*Indent both body statements (or groups of statements) of an* `if...else` *statement. Many IDEs do this for you.*

## Good Programming Practice 4.2

*If there are several levels of indentation, each level should be indented the same additional amount of space. We prefer three-space indents.*

# 4.6 `if...else` Double-Selection Statement (cont.)

- Figure 4.5 illustrates the the `if...else` statement's flow of control.

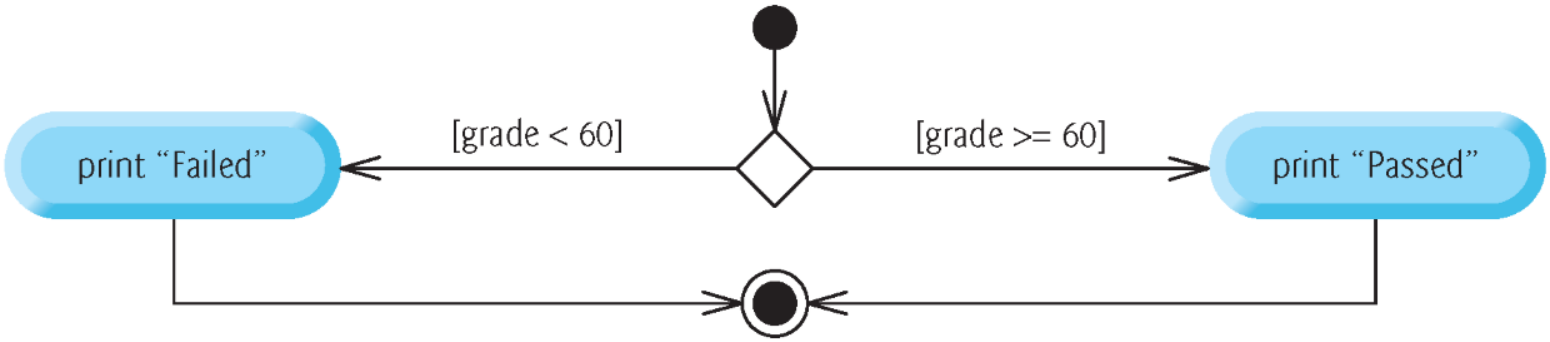


**Fig. 4.5** | `if...else` double-selection statement UML activity diagram.

# 4.6.1 Nested if…else Statements

- 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"
```

# 4.6.1 Nested if…else Statements

- This pseudocode can be written in C++ as

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

- If variable `studentGrade` is greater than or equal to 90, the first four conditions in the nested `if…else` will be `true`, but only the statement in the `if` part of the first `if…else` will execute. After that, the `else` part of the "outermost" `if…else` statement is skipped.

# 4.6.1 Nested if…else Statements

- Most programmers write the preceding statement as

```
if (studentGrade >= 90) {
   cout << "A";
}
else if (studentGrade >= 80) {
   cout << "B";
}
else if (studentGrade >= 70) {
   cout << "C";
}
else if (studentGrade >= 60) {
   cout << "D";
}
else {
   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 4.1

*In a nested* `if...else` *statement, ensure that you test for all possible cases.*

# 4.6.2 Dangling-else Problem

- Throughout the text, we always enclose control statement bodies in braces ({ and }).
- This avoids a logic error called the "dangling-`else`" problem.
- We investigate this problem in Exercises 4.23–4.25.

**4.23** *(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;
}
```

# 4.6.3 Blocks

- The `if` statement normally expects only one statement in its body.
- To include several statements in an `if` (or the body of an `else` for an `if…else` statement), enclose the statements in braces.
- Good practice to always use the braces.
- Statements contained in a pair of braces form a block.
- A block can be placed anywhere in a function that a single statement can be placed.

# 4.6.3 Blocks

- Syntax errors (such as when one brace in a block is left out of the program) are caught by the compiler.
- A logic error (such as an incorrect calculation) has its effect at execution time.
- A fatal logic error causes a program to fail and terminate prematurely.
- A nonfatal logic error allows a program to continue executing but causes it to produce incorrect results.

# 4.6.3 Blocks

- 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 state-ment is represented by placing a semicolon (`;`) where a statement would normally be.

## Common Programming Error 4.1

*Placing a semicolon after the parenthesized condition in an* `if` *or* `if...else` *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 a body statement).*

# 4.6.4 Conditional Operator (?:)

- 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" also can be actions to execute.

# 4.7 Student Class: Nested if…else Statements

- The example of Figs. 4.6–4.7 demonstrates a nested `if…else` statement that determines a student's letter grade based on the student's average in a course.
- Class Student (Fig. 4.6) stores a student's name and average and provides member functions for manipulating these values.
  - Data member `name` of type `string` (line 65) to store a `Student`'s name.
  - Data member `average` of type `int` (line 66) to store a `Student`'s average in a course.
  - A constructor (lines 8–13) that initializes the `name` and `average`.
  - Member functions `setName` and `getName` (lines 16–23) to set and get `name`.
  - Member functions `setAverage` and `getAverage` (lines 26–39) to set and get `average`.
  - Member function `getLetterGrade` (lines 42–63), which uses nested `if…else` statements to determine the `Student`'s letter grade based on the `Student`'s `average`.

```
1  // Fig. 4.6: Student.h
2  // Student class that stores a student name and average.
3  #include <string>
4
5  class Student {
6  public:
7     // constructor initializes data members
8     Student(std::string studentName, int studentAverage)
9        : name(studentName) {
10
11       // sets average data member if studentAverage is valid
12       setAverage(studentAverage);
13    }
14
15    // sets the Student's name
16    void setName(std::string studentName) {
17       name = studentName;
18    }
19
20    // retrieves the Student's name
21    std::string getName() const {
22       return name;
23    }
24
```

**Fig. 4.6** | `Student` class that stores a student name and average. (Part 1 of 4.)

```
25      // sets the Student's average
26      void setAverage(int studentAverage) {
27         // validate that studentAverage is > 0 and <= 100; otherwise,
28         // keep data member average's current value
29         if (studentAverage > 0) {
30            if (studentAverage <= 100) {
31               average = studentAverage; // assign to data member
32            }
33         }
34      }
35
36      // retrieves the Student's average
37      int getAverage() const {
38         return average;
39      }
40
```

**Fig. 4.6** | `Student` class that stores a student name and average. (Part 2 of 4.)

```
41      // determines and returns the Student's letter grade
42      std::string getLetterGrade() const {
43         // initialized to empty string by class string's constructor
44         std::string letterGrade;
45
46         if (average >= 90) {
47            letterGrade = "A";
48         }
49         else if (average >= 80) {
50            letterGrade = "B";
51         }
52         else if (average >= 70) {
53            letterGrade = "C";
54         }
55         else if (average >= 60) {
56            letterGrade = "D";
57         }
58         else {
59            letterGrade = "F";
60         }
61
62         return letterGrade;
63      }
```

**Fig. 4.6** | `Student` class that stores a student name and average. (Part 3 of 4.)

```
64   private:
65      std::string name;
66      int average{0}; // initialize average to 0
67   } // end class Student
```

**Fig. 4.6** | `Student` class that stores a student name and average. (Part 4 of 4.)

```cpp
// Fig. 4.7: StudentTest.cpp
// Create and test Student objects.
#include <iostream>
#include "Student.h"
using namespace std;

int main() {
   Student account1{"Jane Green", 93};
   Student account2{"John Blue", 72};

   cout << account1.getName() << "'s letter grade equivalent of "
      << account1.getAverage() << " is: "
      << account1.getLetterGrade() << "\n";
   cout << account2.getName() << "'s letter grade equivalent of "
      << account2.getAverage() << " is: "
      << account2.getLetterGrade() << endl;
}
```

```
Jane Green's letter grade equivalent of 93 is: A
John Blue's letter grade equivalent of 72 is: C
```

**Fig. 4.7** | Create and test `Student` objects.

# 4.8 `while` Iteration Statement

- An iteration 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* iteration statement constitutes the body of the *While*
  - Eventually, the condition will become false, the iteration will terminate, and the first pseudocode statement after the iteration statement will execute.

# 4.8 while Iteration Statement (cont.)

- Consider a program segment that finds the first power of 3 larger than 100. When the following `while` iteration statement finishes executing, `product` contains the result:
  - ```
    int product = 3;

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

## Common Programming Error 4.2

*Not providing in the body of a* `while` *statement an action that eventually causes the condition in the* `while` *to become false results in a logic error called an* ***infinite loop*** *(the loop never terminates).*

# 4.8 `while` Iteration Statement (cont.)

- The UML activity diagram of Fig. 4.8 illustrates the flow of control that corresponds to the preceding `while` statement.
- 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.

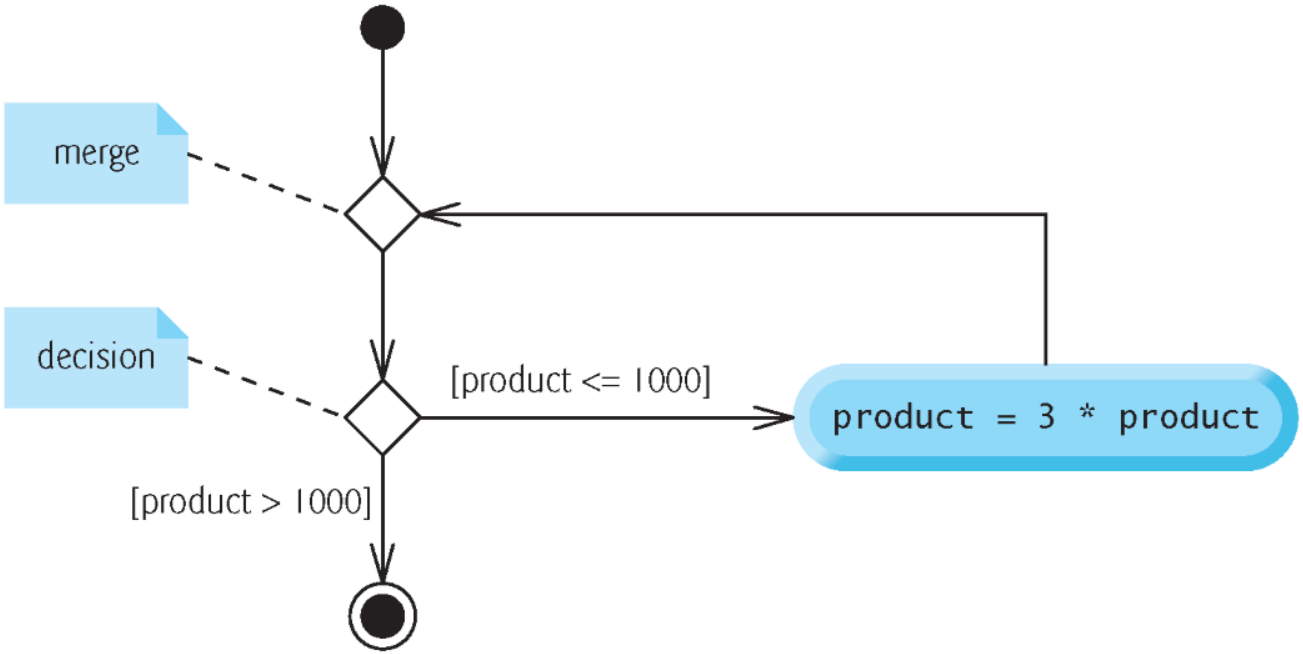


**Fig. 4.8** | `while` iteration statement UML activity diagram.

# 4.8 `while` Iteration 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.

# 4.9 Formulating Algorithms: Counter-Controlled Iteration

- Consider the following problem statement:
  - A class of ten students took a quiz. The grades (integers in the range 0-100) for this quiz are available to you. Determine the class average on the quiz.
- The class average is equal to the sum of the grades divided by the number of students.
- The algorithm for solving this problem on a computer must input each grade, keep track of the total of all grades entered, perform the averaging calculation and print the result.

# 4.9.1 Pseudocode Algorithm with Counter Controlled Iteration

- We use counter-controlled iteration to input the grades one at a time.
  - This technique uses a variable called a counter (or control variable) to control the number of times a group of statements will execute.
  - Often called definite iteration because the number of iterations is known *before* the loop begins executing.

```
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 class average
```

**Fig. 4.9** | Pseudocode algorithm that uses counter-controlled iteration to solve the class-average problem.

# 4.9.1 Pseudocode Algorithm with Counter Controlled Iteration

- 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 that are 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.

## Software Engineering Observation 4.1

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

# 4.9.2 Implementing Counter-Controlled Iteration

- Fig. 4.10 implements the class-average problem using counter-controlled iteration.

```
1  // Fig. 4.10: ClassAverage.cpp
2  // Solving the class-average problem using counter-controlled iteration.
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     // initialization phase
8     int total{0}; // initialize sum of grades entered by the user
9     unsigned int gradeCounter{1}; // initialize grade # to be entered next
10
11    // processing phase uses counter-controlled iteration
12    while (gradeCounter <= 10) { // loop 10 times
13       cout << "Enter grade: "; // prompt
14       int grade;
15       cin >> grade; // input next grade
16       total = total + grade; // add grade to total
17       gradeCounter = gradeCounter + 1; // increment counter by 1
18    }
19
20    // termination phase
21    int average{total / 10}; // int division yields int result
22
```

**Fig. 4.10** | Solving the class-average problem using counter-controlled iteration. (Part 1 of 2.)

```
23      // display total and average of grades
24      cout << "\nTotal of all 10 grades is " << total;
25      cout << "\nClass average is " << average << endl;
26  }
```

```
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. 4.10** | Solving the class-average problem using counter-controlled iteration. (Part 2 of 2.)

# 4.9.2 Implementing Counter-Controlled Iteration

- Variable `gradeCounter` is of type `unsigned int`, because it can assume only the values from 1 through 11 (11 terminates the loop), which are all positive values.
  - In general, counters that should store only nonnegative values should be declared with `unsigned` types.
- Variables of unsigned integer types can represent values from 0 to approximately twice the positive range of the corresponding signed integer types.

# 4.9.2 Implementing Counter-Controlled Iteration

- A variable declared in a function body is a local variable and can be used only from the line of its declaration to the closing right brace of the block in which the variable is declared.
- A local variable's declaration must appear before the variable is used; otherwise, a compilation error occurs.
- Variable grade—declared in the body of the while loop—can be used only in that block.

# 4.9.2 Implementing Counter-Controlled Iteration

- You'll normally initialize counter variables to zero or one, depending on how they are used in an algorithm.

## Error-Prevention Tip 4.2

*Initialize each total and counter, 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 (we'll show examples of when to use 0 and when to use 1).*

# 4.9.3 Notes on Integer Division and Truncation

- Dividing two integers results in integer division—any fractional part of the calculation is truncated (discarded).

## Common Programming Error 4.3

*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.*

# 4.9.3 Arithmetic Overflow

- In Fig. 4.10, line 16

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

  added each grade entered by the user to the total.
- Even this simple statement has a *potential* problem—adding the integers could result in a value that's *too large* to store in an `int` variable.
- This is known as arithmetic overflow and causes *undefined behavior*, which can lead to security problems or unintended results
  - http://en.wikipedia.org/wiki/Integer_overflow#Security_ramifications

# 4.9.3 Arithmetic Overflow

- Figure 2.5's addition program had the same issue in line 19, which calculated the sum of two `int` values entered by the user:

```
sum = number1 + number2;
```

- The maximum and minimum values that can be stored in an `int` variable are represented by the constants `INT_MAX` and `INT_MIN`, respectively, which are defined in the header `<climits>`.

# 4.9.3 Arithmetic Overflow

- There are similar constants for the other integral types and for floating-point types.
- You can see your platform's values for these constants by opening the headers `<climits>` and `<cfloat>` in a text editor.
- It's considered a good practice to ensure that *before* you perform arithmetic calculations like the ones in line 16 of Fig. 4.10 and line 19 of Fig. 2.5, they will *not* overflow.
- The code for doing this is shown on the CERT website `www.securecoding.cert.org`—just search for guideline "INT32-CPP." (Uses operators defined in next chapter.)

# 4.9.4 Input Validation

- To ensure that inputs are valid, industrial-strength programs must test for all possible erroneous cases.
- A  program that inputs grades should validate the grades by using range checking to ensure that they're values from 0 to 100.
- You can then ask the user to reenter any value that's out of range.
- If a program requires inputs from a specific set of values (e.g., nonsequential product codes), you can ensure that each input matches a value in the set.

# 4.10 Formulating Algorithms: Sentinel-Controlled Iteration

- 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.
- Must process an arbitrary number of grades.
  - How can the program determine when to stop the input?
- Sentinel value (also called a signal value, a dummy value or a flag value) can be used for "end of data entry."
- Sentinel-controlled iteration is often called indefinite iteration
  - the number of iterations is not known in advance.
- Sentinel value must not be an acceptable input value.

# 4.10.1 Top-Down, Stepwise Refinement: The Top and First Refinement

- We approach the class average program with a technique called top-down, stepwise refinement
- Begin with a pseudocode representation of the top—a single statement that conveys program's overall function
  - 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.

# 4.10.1 Top-Down, Stepwise Refinement: The Top and First Refinement

- We divide the top into a series of smaller tasks and list these in the order in which they need to be performed.
- This results in the following first refinement.

  *Initialize variables*

  *Input, sum and count the quiz grades*

  *Calculate and print the class average*

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

## Software Engineering Observation 4.2

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

# 4.10.2 Proceeding to the Second Refinement

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

  *Input, sum and count the quiz grades*

requires an iteration statement (i.e., a loop) that successively inputs each grade.

# 4.10.2 Proceeding to the Second Refinement

- We don't know in advance how many grades are to be processed, so we'll use sentinel-controlled iteration.
- 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.

# 4.10.2 Proceeding to the Second Refinement

- 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)*

# 4.10.2 Proceeding to the Second Refinement

- 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 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").

## Error-Prevention Tip 4.3

*According to the C++ standard, the result of division by zero in floating-point arithmetic is undefined. When performing division (/) or remainder (%) calculations in which the right operand could be zero, test for this and handle it (e.g., display an error message) rather than allowing the calculation to proceed.*

```
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 average
16 else
17     Print "No grades were entered"
```

**Fig. 4.11** | Class-averaging pseudocode algorithm with sentinel-controlled iteration.

## Software Engineering Observation 4.3

*Terminate the top-down, stepwise refinement process when you've specified the pseudocode algorithm in sufficient detail for you to convert the pseudocode to C++.*

## Software Engineering Observation 4.4

*Some programmers do not use program development tools like pseudocode. They feel that their ultimate goal is to solve the problem on a computer and that writing pseudocode merely delays the production of final outputs. Although this may work for simple and familiar problems, it can lead to serious errors and delays in large, complex projects.*

# 4.10.3 Implementing Sentinel-Controlled Iteration

- 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. 4.12: ClassAverage.cpp
2  // Solving the class-average problem using sentinel-controlled iteration.
3  #include <iostream>
4  #include <iomanip> // parameterized stream manipulators
5  using namespace std;
6
7  int main() {
8     // initialization phase
9     int total{0}; // initialize sum of grades
10    unsigned int gradeCounter{0}; // initialize # of grades entered so far
11
```

**Fig. 4.12** | Solving the class-average problem using sentinel-controlled iteration. (Part 1 of 4.)

```
12      // processing phase
13      // prompt for input and read grade from user
14      cout << "Enter grade or -1 to quit: ";
15      int grade;
16      cin >> grade;
17
18      // loop until sentinel value read from user
19      while (grade != -1)  {
20         total = total + grade; // add grade to total
21         gradeCounter = gradeCounter + 1; // increment counter
22
23         // prompt for input and read next grade from user
24         cout << "Enter grade or -1 to quit: ";
25         cin >> grade;
26      }
27
```

**Fig. 4.12** | Solving the class-average problem using sentinel-controlled iteration. (Part 2 of 4.)

```
28      // termination phase
29      // if user entered at least one grade...
30      if (gradeCounter != 0) {
31         // use number with decimal point to calculate average of grades
32         double average{static_cast<double>(total) / gradeCounter};
33
34         // display total and average (with two digits of precision)
35         cout << "\nTotal of the " << gradeCounter
36            << " grades entered is " << total;
37         cout << setprecision(2) << fixed;
38         cout << "\nClass average is " << average << endl;
39      }
40      else { // no grades were entered, so output appropriate message
41         cout << "No grades were entered" << endl;
42      }
43   }
```

**Fig. 4.12** | Solving the class-average problem using sentinel-controlled iteration. (Part 3 of 4.)

```
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 the 3 grades entered is 257
Class average is 85.67
```

**Fig. 4.12** | Solving the class-average problem using sentinel-controlled iteration. (Part 4 of 4.)

# 4.10.3 Implementing Sentinel-Controlled Iteration

- Notice the block in the `while` loop.
- Without the braces, the last three statements in the loop would fall outside its body, 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;
```

- Would cause an infinite loop in the program if the user did not input **–1** for the first grade.

# 4.10.4 Converting Between Fundamental Types Explicitly and Implicitly

- 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`:
  - `double average{total / gradeCounter};`

# 4.10.4 Converting Between Fundamental Types Explicitly and Implicitly

- To perform a floating-point calculation with integers, create *temporary* floating-point values.
- `static_cast` operator accomplishes this task.
- The cast operator `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.

# 4.10.4 Converting Between Fundamental Types Explicitly and Implicitly

- 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.

## Good Programming Practice 4.3

*In a sentinel-controlled loop, prompts should remind the user of the sentinel.*

# 4.10.5 Formatting Floating-Point Numbers

- The call to `setprecision` (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.

# 4.10.5 Formatting Floating-Point Numbers

- 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.

# 4.10.5 Formatting Floating-Point Numbers

- 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), although the value in memory remains 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>`.

# 4.10.6 Unsigned Integers and User Input

- Fig. 4.10 declared the variable `gradeCounter` as an unsigned `int` because it can assume only the values from 1 through 11 (11 terminates the loop), which are all non-negative values.
- Could have also declared as `unsigned int` the variables `grade`, `total` and `average`. Grades are normally values from 0 to 100, so the `total` and `average` should each be greater than or equal to 0.
- We declared those variables as `int`s because we can't control what the user actually enters—the user could enter *negative* values.
- Worse yet, the user could enter a value that's not even a number.

# 4.10.6 Unsigned Integers and User Input

- Sometimes sentinel-controlled loops use *intentionally* invalid values to terminate a loop.
- We terminate the loop when the user enters the sentinel `-1` (an invalid grade), so it would be improper to declare variable `grade` as an unsigned int.
- As you'll see, the end-of-file (`EOF`) indicator is also normally implemented internally in the compiler as a negative number.

# 4.11 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.
- In this case study, we examine the only other structured way control statements can be connected—namely, by nesting one control statement within another.

# 4.11.1 Problem Statement

- 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" on the screen each time the program requests another test result.
    - 2.Count the number of test results of each type.
    - 3.Display a summary of the test results indicating the number of students who passed and the number who failed.
    - 4.If more than eight students passed the exam, print "Bonus to instructor!"

# 4.11.1 Problem Statement

- 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 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. For simplicity, we test only for a 1 in our algorithm. If the number is not a 1, we assume that it's a 2.
  - 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.

# 4.11.2 Top-Down Stepwise Refinement: Pseudocode Representation of the Top

- Pseudocode representation of the top:

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

# 4.11.3 Top-Down Stepwise Refinement: First Refinement

- First refinement

  *Initialize variables*

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

  *Print a summary of the exam results and decide whether a bonus should be paid*

- Further refinement is necessary.
- 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.

# 4.11.4 Top-Down Stepwise Refinement: Second Refinement

- The pseudocode statement

  *Initialize variables*

  can be refined as follows:

  *Initialize passes to zero*

  *Initialize failures to zero*

  *Initialize student counter to one*

# 4.11.4 Top-Down Stepwise Refinement: Second Refinement

- 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
```

# 4.11.4 Top-Down Stepwise Refinement: Second Refinement

- The pseudocode statement

  *Print a summary of the exam results and decide whether bonus should be paid*

  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. 4.13.

```
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. 4.13** | Pseudocode for examination-results problem.

# 4.11.6 Program That Implements the Pseudocode Algoritm

- The program that implements the pseudocode algorithm and two sample executions are shown in Fig. 4.14.

## Error-Prevention Tip 4.4

*Initializing local variables when they're declared helps you avoid any compilation warnings that might arise from attempts to use uninitialized variables and helps you avoid logic errors from using uninitialized variables.*

```
1  // Fig. 4.14: Analysis.cpp
2  // Analysis of examination results using nested control statements.
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     // initializing variables in declarations
8     unsigned int passes{0};
9     unsigned int failures{0};
10    unsigned int studentCounter{1};
11
```

**Fig. 4.14** | Analysis of examination results using nested control statements. (Part 1 of 4.)

```
12      // process 10 students using counter-controlled loop
13      while (studentCounter <= 10) {
14         // prompt user for input and obtain value from user
15         cout << "Enter result (1 = pass, 2 = fail): ";
16         int result;
17         cin >> result;
18
19         // if...else is nested in the while statement
20         if (result == 1) {
21            passes = passes + 1;
22         }
23         else {
24            failures = failures + 1;
25         }
26
27         // increment studentCounter so loop eventually terminates
28         studentCounter = studentCounter + 1;
29      }
30
```

**Fig. 4.14** | Analysis of examination results using nested control statements. (Part 2 of 4.)

```
31      // termination phase; prepare and display results
32      cout << "Passed: " << passes << "\nFailed: " << failures << endl;
33
34      // determine whether more than 8 students passed
35      if (passes > 8) {
36         cout << "Bonus to instructor!" << endl;
37      }
38   }
```

```
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
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. 4.14** | Analysis of examination results using 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): 1
Enter result (1 = pass, 2 = fail): 2
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
Passed: 6
Failed: 4
```

**Fig. 4.14** | Analysis of examination results using nested control statements. (Part 4 of 4.)

# 4.11.6 Program That Implements the Pseudocode Algoritm

- The `if...else` statement for processing each result is nested in the `while` statement.
- The `if` statement after the loop determines whether more than eight students passed the exam and, if so, outputs the message `"Bonus to instructor!"`.

# 4.11.7 Preventing Narrowing Conversions with List Initialization

- Consider from Fig. 4.14

```
unsigned int studentCounter{1};
```

- Can also write as

```
unsigned int studentCounter = {1};
```

- or

```
unsigned int studentCounter = 1;
```

# 4.11.7 Preventing Narrowing Conversions with List Initialization

- For fundamental-type variables, list-initialization syntax also *prevents* so-called narrowing conversions that could result in *data loss*.
- For example, previously you could write

```
int x = 12.7;
```

- which attempts to assign the `double` value `12.7` to the `int` variable `x`.
- A `double` value is converted to an `int`, by truncating the floating-point part (.7), which results in a loss of information—a *narrowing conversion*.

# 4.11.7 Preventing Narrowing Conversions with List Initialization

- The actual value assigned to x is 12.
- Many compilers generate a warning for this statement, but still allow it to compile.
- However, using list initialization, as in

```
int x = {12.7};
```

- or

```
int x{ 12.7 };
```

- yields a *compilation error.*

# 4.12 Compund Assignment Operators

- The compound assignment operators abbreviate 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 4.15 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. 4.15** | Arithmetic compound assignment operators.

# 4.13 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. 4.16.

| Operator | Operator name | Sample expression | Explanation |
|---|---|---|---|
| ++ | prefix increment | ++a | Increment a by 1, then use the new value of a in the expression in which a resides. |
| ++ | postfix increment | a++ | Use the current value of a in the expression in which a resides, then increment a by 1. |
| -- | prefix decrement | --b | Decrement b by 1, then use the new value of b in the expression in which b resides. |
| -- | postfix decrement | b-- | Use the current value of b in the expression in which b resides, then decrement b by 1. |

**Fig. 4.16** | Increment and decrement operators.

## Good Programming Practice 4.4

*Unlike binary operators, the unary increment and decrement operators as a matter of style should be placed next to their operands, with no intervening spaces.*

```
1  // Fig. 4.17: Increment.cpp
2  // Prefix increment and postfix increment operators.
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     // demonstrate postfix increment operator
8     unsigned int c{5}; // initializes c with the value 5
9     cout << "c before postincrement: " << c << endl; // prints 5
10    cout << "    postincrementing c: " << c++ << endl; // prints 5
11    cout << " c after postincrement: " << c << endl; // prints 6
12
13    cout << endl; // skip a line
14
15    // demonstrate prefix increment operator
16    c = 5; // assigns 5 to c
17    cout << " c before preincrement: " << c << endl; // prints 5
18    cout << "     preincrementing c: " << ++c << endl; // prints 6
19    cout << "  c after preincrement: " << c << endl; // prints 6
20 }
```

**Fig. 4.17** | Prefix increment and postfix increment operators. (Part 1 of 2.)

```
c before postincrement: 5
    postincrementing c: 5
 c after postincrement: 6

 c before preincrement: 5
     preincrementing c: 6
  c after preincrement: 6
```

**Fig. 4.17** | Prefix increment and postfix increment operators. (Part 2 of 2.)

# 4.13 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.
- In the context of a larger expression preincrementing a variable and postincrementing a variable have different effects (and similarly for predecrementing and post-decrementing).

## Common Programming Error 4.4

*Attempting to use the increment or decrement operator on an expression other than one to which a value can be assigned is a syntax error. For example, writing* `++(x + 1)` *is a syntax error, because* `(x + 1)` *is not a variable.*

# 4.13 Increment and Decrement Operators (cont.)

- Figure 4.18 shows the precedence and associativity of the operators introduced to this point.

## Good Programming Practice 4.5

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

| Operators | Associativity | Type |
|---|---|---|
| `::` `()` | left to right *[See Fig. 2.10's caution regarding grouping parentheses.]* | primary |
| `++` `--` `static_cast<type>()` | left to right | 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. 4.18** | Operator precedence for the operators encountered so far in the text.

# 4.14 Fundamental Types Are Not Portable

- The table in Appendix C lists C++'s fundamental types.
- C++ requires all variables to have a type.
- In C and C++, programmers frequently have to write separate versions of programs to support different computer platforms, because the fundamental types are not guaranteed to be identical from computer to computer.
  - An int on one machine might be represented by 16 bits (2 bytes) of memory, on a second machine by 32 bits (4 bytes), and on another machine by 64 bits (8 bytes).

**Portability Tip 4.2**

*C++'s fundamental types are not portable across all computer platforms that support C++.*