Control Statements: Part 2; Logical Operators

Chapter 5 of C++ How to Program, 10/e

OBJECTIVES

In this chapter you'll:

- Learn the essentials of counter-controlled iteration.
- Use the `for` and `do...while` iteration statements to execute statements in a program repeatedly.
- Understand multiple selection using the `switch` selection statement.
- Use the `break` and `continue` program-control statements to alter the flow of control.
- Use the logical operators to form compound conditions in control statements.
- Understand the representational errors associated with using floating-point data types to hold monetary values.
- Understand some of the challenges of processing monetary amounts as we begin building a `DollarAmount` class, which uses integers and integer arithmetic to represent and manipulate monetary amounts.

5.1 Introduction

- This chapter continues our presentation of structured-programming theory and principles.
- Demonstrates C++'s `for`, `do…while` and `switch` statements.
- We explore the essentials of counter-controlled iteration.
- We use compound-interest calculations to begin investigating the issues of processing monetary amounts, and we develop a new `DollarAmount` class that uses very large integers to precisely represent monetary amounts.

5.1 Introduction

- We introduce the `break` and `continue` program-control statements.
- We discuss C++'s logical operators, which enable you to combine simple conditions in control statements.
- We summarize C++'s control statements and the proven problem-solving techniques presented in this chapter and Chapter 4.

5.2 Essentials of Counter-Controlled Iteration

- Counter-controlled iteration requires
  - a control variable (or loop counter)
  - the control variable's initial value
  - the control variable's increment that's applied during each iteration of the loop
  - the loop-continuation condition that determines if looping should continue.

5.2 Essentials of Counter-Controlled Iteration

- The program in Fig. 5.1 prints the numbers from 1 to 10. The declaration in line 8 *names* the control variable (`counter`), declares it to be an `unsigned int`, reserves space for it in memory and sets it to an *initial value* of 1.
- Declarations that require initialization are *executable* statements.
- In C++, it's more precise to call a declaration that also reserves memory a definition.

```
1  // Fig. 5.1: WhileCounter.cpp
2  // Counter-controlled iteration with the while iteration statement.
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     unsigned int counter{1}; // declare and initialize control variable
8
9     while (counter <= 10) { // loop-continuation condition
10       cout << counter << "   ";
11       ++counter; // increment control variable
12    }
13
14    cout << endl;
15 }
```

```
1   2   3   4   5   6   7   8   9   10
```

Fig. 5.1 | Counter-controlled iteration with the `while` iteration statement.

Error-Prevention Tip 5.1

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

5.3 `for` Iteration Statement

- The `for` iteration statement (Fig. 5.2) specifies the counter-controlled iteration details in a single line of code.
- The initialization occurs once when the loop is encountered.
- The condition is tested next and each time the body completes.
- The body executes if the condition is true.
- The increment occurs after the body executes.
- Then, the condition is tested again.

```
1  // Fig. 5.2: ForCounter.cpp
2  // Counter-controlled iteration with the for iteration statement.
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     // for statement header includes initialization,
8     // loop-continuation condition and increment
9     for (unsigned int counter{1}; counter <= 10; ++counter) {
10       cout << counter << "   ";
11    }
12
13    cout << endl;
14 }
```

```
1   2   3   4   5   6   7   8   9   10
```

Fig. 5.2 | Counter-controlled iteration with the `for` iteration statement.

Common Programming Error 5.1

Using an incorrect relational operator or an incorrect final value of a loop counter in the loop-continuation condition of an iteration statement can cause an off-by-one error.

Error-Prevention Tip 5.2

Using the final value and operator `<=` *in a loop's condition helps avoid off-by-one errors. For a loop that outputs 1 to 10, the loop-continuation condition should be* `counter <= 10` *rather than* `counter < 10` *(which causes an off-by-one error) or* `counter < 11` *(which is correct). Many programmers prefer so-called zero-based counting, in which to count 10 times,* `counter` *would be initialized to zero and the loop-continuation test would be* `counter < 10`.

Error-Prevention Tip 5.3

Write loop conditions carefully to prevent loop counters from overflowing.

5.3 `for` Iteration Statement (cont.)

`for` Statement Header Components

- Figure 5.3 takes a closer look at the `for` statement header (line 10) of Fig. 5.2.

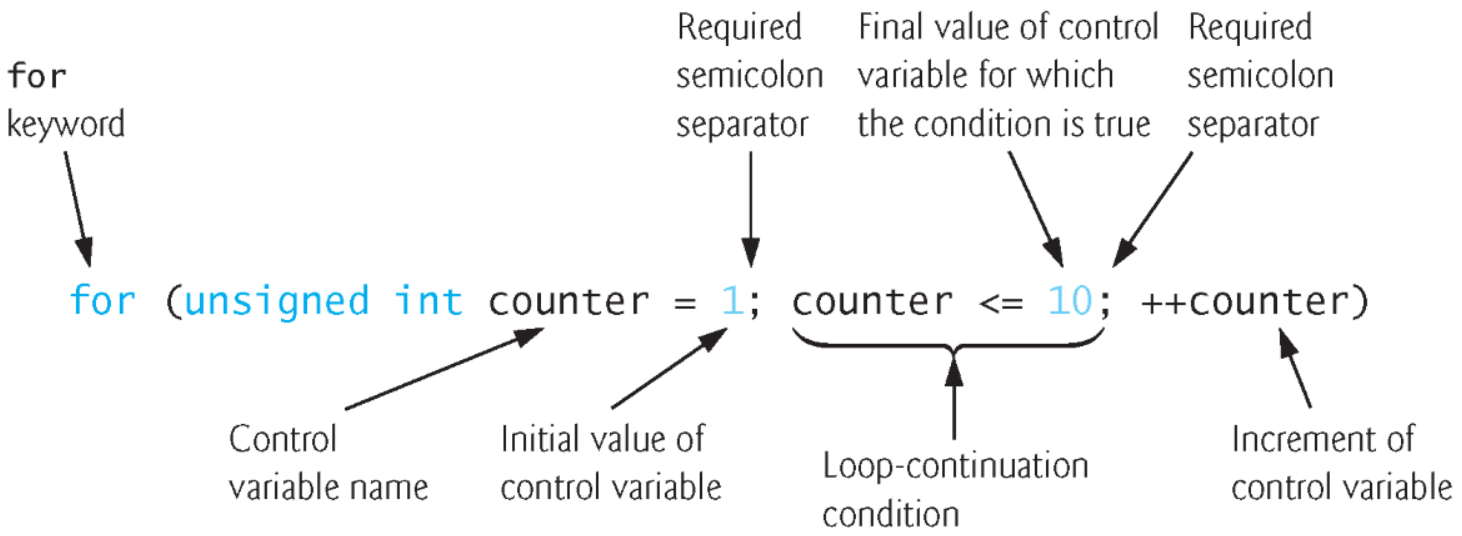

Fig. 5.3 | `for` statement header components.

5.3 `for` Iteration Statement (cont.)

- The general form of the `for` statement is

```
for (initialization; loopContinuationCondition; increment) {
    statement
}
```

- where the *initialization* expression initializes the loop's control variable, *loopContinuationCondition* determines whether the loop should continue executing and *increment* increments the control variable.
- In most cases, the `for` statement can be represented by an equivalent `while` statement, as follows:

```
initialization;

while (loopContinuationCondition)
{
    statement
    increment;
}
```

5.3 `for` Iteration Statement (cont.)

- If the *initialization* expression declares the control variable, the control variable can be used *only* in the body of the `for` statement—the control variable will be unknown *outside* the `for` statement.
- This restricted use of the control variable name is known as the variable's scope.
- The scope of a variable specifies *where* it can be used in a program.

Common Programming Error 5.2

When a `for` *statement's control variable is declared in the initialization section of the* `for`*'s header, using the control variable after the* `for`*'s body is a compilation error.*

5.3 `for` Iteration Statement (cont.)

- The three expressions in the `for` statement header are optional (but the two semicolon separators are *required*).
- If the *loopContinuationCondition* is omitted, C++ assumes that the condition is true, thus creating an *infinite loop.*
- One might omit the *initialization* expression if the control variable is initialized earlier in the program.
- One might omit the *increment* expression if the increment is calculated by statements in the body of the `for` or if no increment is needed.

5.3 `for` Iteration Statement (cont.)

- The increment expression in the `for` statement acts like a standalone statement at the end of the `for` statement's body.
- The expressions
  - ```
    counter = counter + 1
    counter += 1
    ++counter
    counter++
    ```
- are all equivalent in the *increment* expression (when no other code appears there).
    ```

Common Programming Error 5.3

Placing a semicolon immediately to the right of the right parenthesis of a `for` *header makes that* `for`*'s body an empty statement. This is normally a logic error.*

Error-Prevention Tip 5.4

Infinite loops occur when the loop-continuation condition in an iteration statement never becomes false. To prevent this situation in a counter-controlled loop, ensure that the control variable is modified during each iteration of the loop so that the loop-continuation condition will eventually become false. In a sentinel-controlled loop, ensure that the sentinel value is able to be input.

5.3 `for` Iteration Statement (cont.)

- The initialization, loop-continuation condition and increment expressions of a `for` statement can contain arithmetic expressions.
- The "increment" of a `for` statement can be negative, in which case it's really a *decrement* and the loop actually counts *downward*.
- If the loop-continuation condition is *initially false*, the body of the `for` statement is not performed.

Error-Prevention Tip 5.5

Although the value of the control variable can be changed in the body of a `for` *loop, avoid doing so, because this practice can lead to subtle errors. If a program must modify the control variable's value in the loop's body, use* `while` *rather than* `for`.

5.3 `for` Iteration Statement (cont.)

- The `for` iteration statement's UML activity diagram is similar to that of the `while` statement (Fig. 4.6).
- Figure 5.4 shows the activity diagram of the `for` statement in Fig. 5.2.
- The diagram makes it clear that initialization occurs once *before* the loop-continuation test is evaluated the first time, and that incrementing occurs *each time* through the loop *after* the body statement executes.

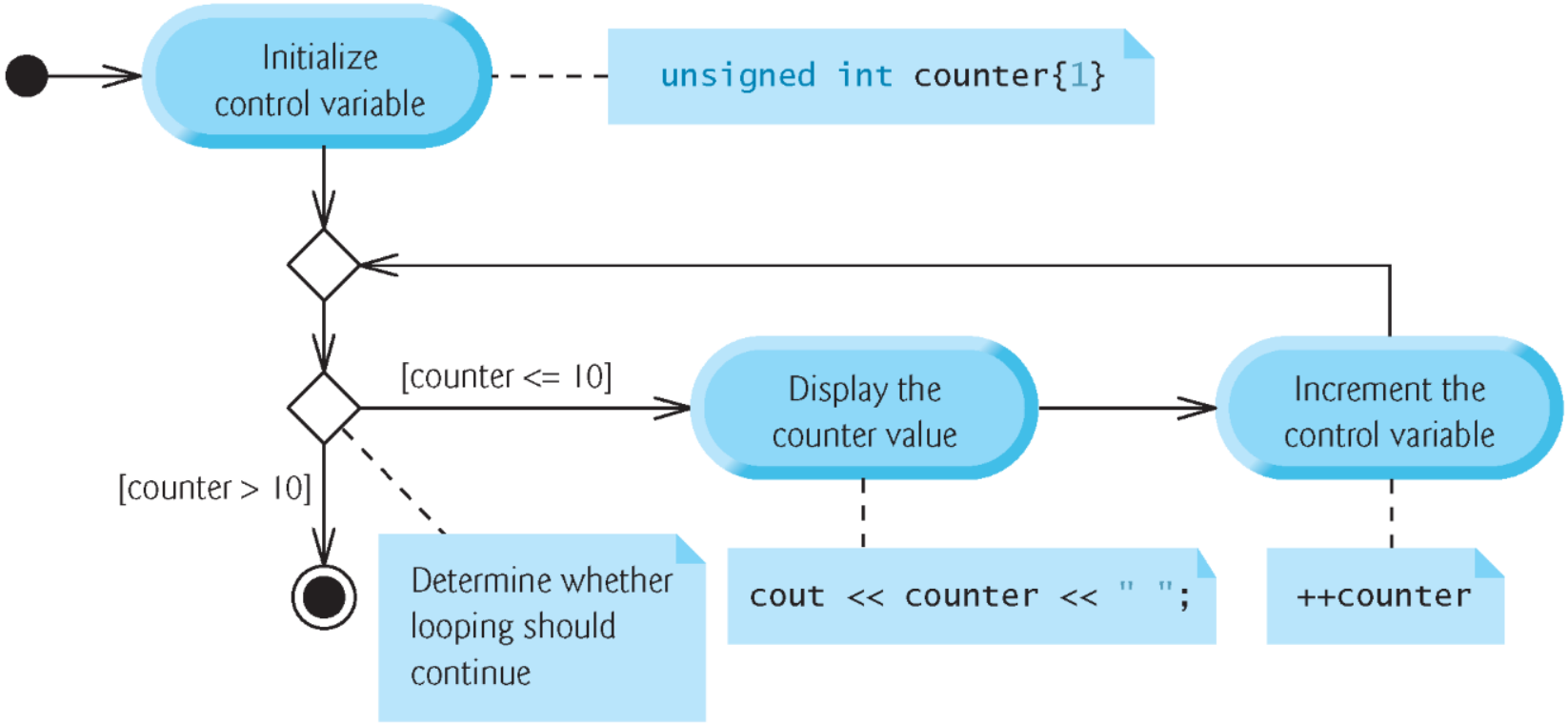

Fig. 5.4 | UML activity diagram for the `for` statement in Fig. 5.2.

5.4 Examples Using the `for` Statement

- Count from 1 to 100 in increments of 1.
  - `for (unsigned int i = 1; i <= 100; ++i)`
- Count from 100 down to 1 in decrements of 1.
  - `for (unsigned int i = 100; i >= 1; --i)`
- Count from 7 to 77 in steps of 7.
  - `for (unsigned int i = 7; i <= 77; i += 7)`
- Count from 20 down to 2 in steps of –2.
  - `for (unsigned int i = 20; i >= 2; i -= 2)`
- Iterate over the sequence 2, 5, 8, 11, 14 , 17 , 20.
  - `for (unsigned int i = 2; i <= 20; i += 3)`
- Iterate over the sequence 99, 88, 77, 66 , 55 , 44 , 33 , 22 , 11, 0. We use `int` rather than `unsigned int` here because the condition does not become false until `i`'s value is `-11`, so the control variable must be able to represent both positive and negative values.
  - `for (int i = 99; i >= 0; i -= 11)`

Common Programming Error 5.4

Using an incorrect relational operator in the loop-continuation condition of a loop that counts downward (e.g., using `i <= 1` *instead of* `i >= 1` *in a loop counting down to 1) is usually a logic error.*

Common Programming Error 5.5

Do not use equality operators (`!=` or `==`) in a loop-continuation condition if the loop's control variable increments or decrements by more than 1. For example, consider the `for` *header*

```
for (unsigned int counter{1}; counter != 10; counter += 2)
```

The loop-continuation test `counter != 10` *never becomes false (resulting in an* infinite loop*) because* `counter` *increments by 2 after each iteration (and never becomes 10).*

5.5 Application: Summing Even Integers

- The program of Fig. 5.5 uses a `for` statement to sum the even integers from 2 to 20.

```
1  // Fig. 5.5: Sum.cpp
2  // Summing integers with the for statement.
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     unsigned int total{0};
8
9     // total even integers from 2 through 20
10    for (unsigned int number{2}; number <= 20; number += 2) {
11       total += number;
12    }
13
14    cout << "Sum is " << total << endl;
15 }
```

```
Sum is 110
```

Fig. 5.5 | Summing integers with the `for` statement.

Good Programming Practice 5.1

Place only expressions involving the control variables in the initialization and increment sections of a `for` *statement.*

Good Programming Practice 5.2

For readability limit the size of control-statement headers to a single line if possible.

5.6 Application: Compound-Interest Calculations

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

    $a = p\ (1 + r)^{n}$

    where

    p is the original amount invested (i.e., the principal),
    r is the annual interest rate,
    n is the number of years and
    a is the amount on deposit at the end of the nth year.
  - This problem involves a loop that performs the indicated calculation for each of the 10 years the money remains on deposit (Fig. 5.6).

```
1  // Fig. 5.6: Interest.cpp
2  // Compound-interest calculations with for.
3  #include <iostream>
4  #include <iomanip>
5  #include <cmath> // for pow function
6  using namespace std;
7
8  int main() {
9     // set floating-point number format
10    cout << fixed << setprecision(2);
11
12    double principal{1000.00}; // initial amount before interest
13    double rate{0.05}; // interest rate
14
15    cout << "Initial principal: " << principal << endl;
16    cout << "    Interest rate:    " << rate << endl;
17
18    // display headers
19    cout << "\nYear" << setw(20) << "Amount on deposit" << endl;
20
```

Fig. 5.6 | Compound-interest calculations with `for`. (Part 1 of 2.)

```
21      // calculate amount on deposit for each of ten years
22      for (unsigned int year{1}; year <= 10; year++) {
23         // calculate amount on deposit at the end of the specified year
24         double amount = principal * pow(1.0 + rate, year);
25
26         // display the year and the amount
27         cout << setw(4) << year << setw(20) << amount << endl;
28      }
29   }
```

```
Initial principal: 1000.00
    Interest rate:    0.05

Year   Amount on deposit
   1             1050.00
   2             1102.50
   3             1157.63
   4             1215.51
   5             1276.28
   6             1340.10
   7             1407.10
   8             1477.46
   9             1551.33
  10             1628.89
```

Fig. 5.6 | Compound-interest calculations with `for`. (Part 2 of 2.)

5.6 Application: Compound-Interest Calculations (cont.)

Using Stream Manipulators to Format Numeric Output

- Parameterized stream manipulators `setprecision` and `setw` and the nonparameterized stream manipulator `fixed`.
- The stream manipulator `setw(4)` specifies that the next value output should appear in a field width of 4—i.e., `cout` prints the value with at least 4 character positions.
  - If less than 4 character positions wide, the value is right justified in the field by default.
  - If more than 4 character positions wide, the field width is extended *rightward* to accommodate the entire value.
- To indicate that values should be output left justified, simply output nonparameterized stream manipulator `left`.
- Right justification can be restored by outputting nonparameterized stream manipulator `right`.

5.6 Application: Compound-Interest Calculations (cont.)

- Stream manipulator `fixed` indicates that floating-point values should be output as fixed-point values with decimal points.
- Stream manipulator `setprecision` specifies the number of digits to the right of the decimal point.
- Stream manipulators `fixed` and `setprecision` remain in effect until they're changed—such settings are called sticky settings.
- The field width specified with `setw` applies only to the next value output.

5.6 Application: Compound-Interest Calculations (cont.)

- C++ does *not* include an exponentiation operator, so we use the standard library function `pow`.
  - `pow(x, y)` calculates the value of `x` raised to the y^{th} power.
  - Takes two arguments of type `double` and returns a `double` value.
- This program will not compile without including header file `<cmath>`.
  - Includes information that tells the compiler to convert the value of `year` to a temporary `double` representation before calling the function.
  - Contained in `pow`'s function prototype.

Performance Tip 5.1

In loops, avoid calculations for which the result never changes—such calculations should typically be placed before the loop. Many of today's sophisticated optimizing compilers will place such calculations before loops in the compiled code.

5.6 Application: Compound-Interest Calculations (cont.)

- Variables of type `float` represent single-precision floating-point numbers and have approximately seven significant digits on most of today's systems.
- Variables of type `double` represent double-precision floating-point numbers.
  - These require twice as much memory as `float` variables and provide approximately 15 significant digits on most of today's systems—approximately double the precision of `float` variables.

5.6 Application: Compound-Interest Calculations (cont.)

- Most programmers represent floating-point numbers with type `double`.
- C++ treats all floating-point numbers you type in a program's source code (such as 7.33 and 0.0975) as `double` values by default.
- Such values in the source code are known as floating-point literals.

5.6 Application: Compound-Interest Calculations (cont.)

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

Common Programming Error 5.6

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.

5.6 Application: Compound-Interest Calculations (cont.)

- Floating-point numbers have numerous applications, especially for measured values.
  - E.g., a "normal" body temperature of 98.6 degrees Fahrenheit does not need to be precise to a large number of digits. 98.6 may actually be 98.5999473210643. Calling this number simply 98.6 is fine for most applications involving body temperatures.
- Due to the imprecise nature of floating-point numbers, type `double` is preferred over type `float`, because double variables can represent floating-point numbers more precisely.

5.6 Application: Compound-Interest Calculations (cont.)

- Simple explanation of what can go wrong when using `double` (or `float`) to represent dollar amounts (assuming that dollar amounts are displayed with two digits to the right of the decimal point):
  - Two calculated double dollar amounts stored in the machine could be 14.234 (which would normally be rounded to 14.23 for display purposes) and 18.673 (which would normally be rounded to 18.67 for display purposes).
  - When these amounts are added, they produce the internal sum 32.907, which would normally be rounded to 32.91 for display purposes. Thus, your output could appear as

```
  14.23
+ 18.67
  32.91
```

  - A person adding the individual numbers as displayed would expect the sum to be 32.90.

Error-Prevention Tip 5.6

Do not use variables of type `double` *(or* `float`*) to perform precise monetary calculations. The imprecision of floating-point numbers can lead to errors.*

5.6 Application: Compound-Interest Calculations (cont.)

- Even simple dollar amounts can have representational errors when they're stored as `double`s.
- We created a simple program with the declaration

```
double d = 123.02;
```

- Then displayed variable `d`'s value with many digits of precision to the right of the decimal.
- The resulting output showed 123.02 as 123.0199999..., which is another example of a representational error.

5.7 Integer-Based Monetary Calculations with Class DollarAmount

- In Chapter 3, we developed an `Account` class.
- A typical dollar `Account` balance—such as $437.19—has a whole number of dollars (e.g., 437) to the left of the decimal point and a whole number of cents (e.g., 19) to the right.
- For simplicity, class `Account` represented its balance with type `int`, which of course limited balances to whole dollar amounts.

5.7 Integer-Based Monetary Calculations with Class DollarAmount

- Fig. 5.6's compound-interest example processed dollar amounts as type `double`
- Representational errors, because we stored precise decimal dollar amounts and interest rates as `double`s.
  - Can avoid these by performing all calculations using integer arithmetic—even interest calculations.
- Rounding of the floating-point values.
  - We cannot avoid rounding on interest calculations, because they result in fractional pennies when working with dollar amounts. Those fractional pennies must be rounded to the hundredths place.
  - With integer arithmetic we can exercise precise control over rounding without suffering the representational errors associated with floating-point calculations.

5.7 Integer-Based Monetary Calculations with Class DollarAmount

- Many points in monetary calculations in which rounding may occur.
  - For example, on a restaurant bill, the tax could be calculated on each individual item, resulting in many separate rounding operations, or it could be calculated only once on the total bill amount—these alternate approaches could yield different results.
- Performing monetary calculations with `doubles` can cause problems for organizations that require precise dollar amounts—such as banks, insurances companies and businesses in general.

5.7 Integer-Based Monetary Calculations with Class DollarAmount

- We'll create a `DollarAmount` class for precise control over monetary amounts.
- Represents dollar amounts in whole numbers of pennies—for example, $1,000 is stored as 100000.
  - A key benefit of this approach is that integer values are represented exactly in memory.

5.7 Integer-Based Monetary Calculations with Class DollarAmount

- `DollarAmount` capabilities
  - a constructor to initialize a `DollarAmount` object to a whole number of pennies
  - an `add` member function that adds `DollarAmounts`
  - a `subtract` member function that subtracts `DollarAmounts`
  - an `addInterest` member function that calculates annual interest on the amount in the `DollarAmount` object on which the function is called and adds the interest to the amount in that object
  - a `toString` member function that returns a `DollarAmount`'s `string` representation.

5.7 Integer-Based Monetary Calculations with Class DollarAmount

- All calculations use integer arithmetic
  - no floating-point calculations are used, so DollarAmounts always represent their values precisely.
- Reimplement Fig. 5.6's compound-interest example using `DollarAmounts`.
- Then we'll present the class and walk through the code.
- Class `DollarAmount` will control precisely how that rounding occurs using integer arithmetic.

5.7 Integer-Based Monetary Calculations with Class DollarAmount

- Figure 5.7 adds `DollarAmounts`, subtracts `DollarAmounts` and calculates compound interest with `DollarAmounts` to demonstrate the class's capabilities.

```
1  // Fig. 5.7: Interest.cpp
2  // Compound-interest calculations with class DollarAmount and integers.
3  #include <iostream>
4  #include <iomanip>
5  #include <string>
6  #include "DollarAmount.h"
7  using namespace std;
8
9  int main() {
10    DollarAmount d1{12345}; // $123.45
11    DollarAmount d2{1576}; // $15.76
12
13    cout << "After adding d2 (" << d2.toString() << ") into d1 ("
14       << d1.toString() << "), d1 = ";
15    d1.add(d2); // modifies object d1
16    cout << d1.toString() << "\n";
17
18    cout << "After subtracting d2 (" << d2.toString() << ") from d1 ("
19       << d1.toString() << "), d1 = ";
20    d1.subtract(d2); // modifies object d1
21    cout << d1.toString() << "\n";
22
```

Fig. 5.7 | Compound-interest calculations with class `DollarAmount` and integers. (Part 1 of 5.)

```
23      cout << "After subtracting d1 (" << d1.toString() << ") from d2 ("
24         << d2.toString() << "), d2 = ";
25      d2.subtract(d1); // modifies object d2
26      cout << d2.toString() << "\n\n";
27
28      cout << "Enter integer interest rate and divisor. For example:\n"
29         << "for     2%, enter:    2 100\n"
30         << "for   2.3%, enter:   23 1000\n"
31         << "for  2.37%, enter:  237 10000\n"
32         << "for 2.375%, enter: 2375 100000\n> ";
33      int rate; // whole-number interest rate
34      int divisor; // divisor for rate
35      cin >> rate >> divisor;
36
37      DollarAmount balance{100000}; // initial principal amount in pennies
38      cout << "\nInitial balance: " << balance.toString() << endl;
39
```

Fig. 5.7 | Compound-interest calculations with class `DollarAmount` and integers. (Part 2 of 5.)

```
40      // display headers
41      cout << "\nYear" << setw(20) << "Amount on deposit" << endl;
42
43      // calculate amount on deposit for each of ten years
44      for (unsigned int year{1}; year <= 10; year++) {
45         // increase balance by rate % (i.e., rate / divisor)
46         balance.addInterest(rate, divisor);
47
48         // display the year and the amount
49         cout << setw(4) << year << setw(20) << balance.toString() << endl;
50      }
51   }
```

Fig. 5.7 | Compound-interest calculations with class `DollarAmount` and integers. (Part 3 of 5.)

```
After adding d2 (15.76) into d1 (123.45), d1 = 139.21
After subtracting d2 (15.76) from d1 (139.21), d1 = 123.45
After subtracting d1 (123.45) from d2 (15.76), d2 = -107.69

Enter integer interest rate and divisor. For example:
for     2%, enter:    2 100
for   2.3%, enter:   23 1000
for  2.37%, enter:  237 10000
for 2.375%, enter: 2375 100000
> 5 100

Initial balance: 1000.00

Year   Amount on deposit
   1             1050.00
   2             1102.50
   3             1157.63
   4             1215.51
   5             1276.29
   6             1340.10
   7             1407.11
   8             1477.47
   9             1551.34
  10             1628.91
```

Fig. 5.7 | Compound-interest calculations with class `DollarAmount` and integers. (Part 4 of 5.)

```
After adding d2 (15.76) into d1 (123.45), d1 = 139.21
After subtracting d2 (15.76) from d1 (139.21), d1 = 123.45
After subtracting d1 (123.45) from d2 (15.76), d2 = -107.69

Enter integer interest rate and divisor. For example:
for     2%, enter:    2 100
for   2.3%, enter:   23 1000
for  2.37%, enter:  237 10000
for 2.375%, enter: 2375 100000
> 525 10000

Initial balance: 1000.00

Year   Amount on deposit
   1             1052.50
   2             1107.76
   3             1165.92
   4             1227.13
   5             1291.55
   6             1359.36
   7             1430.73
   8             1505.84
   9             1584.90
  10             1668.11
```

Fig. 5.7 | Compound-interest calculations with class `DollarAmount` and integers. (Part 5 of 5.)

5.7 Integer-Based Monetary Calculations with Class DollarAmount

- `addInterest` member function receives two arguments:
  - an integer representation of the interest rate and
  - an integer divisor (a power of 10)
- To determine the interest amount, `addInterest` multiplies the `DollarAmount` object's number of pennies by the integer representation of the interest rate, then divides the result by the integer divisor.

5.7 Integer-Based Monetary Calculations with Class DollarAmount

- For example:
  - To calculate 5% interest, enter the integers 5 and 100. In integer arithmetic, multiplying a DollarAmount by 5, then dividing the result by 100 calculates 5% of the DollarAmount's value.
  - To calculate 5.25% interest, enter the integers 525 and 10000. In integer arithmetic, multiplying a DollarAmount by 525, then dividing the result by 10000 calculates 5.25% of the DollarAmount's value.

5.7 Integer-Based Monetary Calculations with Class DollarAmount

- Figure 5.8 defines class `DollarAmount` with data member amount (line 41) representing an integer number of pennies.

```
1  // Fig. 5.8: DollarAmount.h
2  // DollarAmount class stores dollar amounts as a whole numbers of pennies
3  #include <string>
4  #include <cmath>
5
6  class DollarAmount {
7  public:
8     // initialize amount from an int64_t value
9     explicit DollarAmount(int64_t value) : amount{value} { }
10
11    // add right's amount to this object's amount
12    void add(DollarAmount right) {
13       // can access private data of other objects of the same class
14       amount += right.amount;
15    }
16
17    // subtract right's amount from this object's amount
18    void subtract(DollarAmount right) {
19       // can access private data of other objects of the same class
20       amount -= right.amount;
21    }
22
```

Fig. 5.8 | `DollarAmount` class stores dollar amounts as whole numbers of pennies. (Part 1 of 2.)

```
23      // uses integer arithmetic to calculate interest amount,
24      // then calls add with the interest amount
25      void addInterest(int rate, int divisor) {
26         // create DollarAmount representing the interest
27         DollarAmount interest{
28            (amount * rate + divisor / 2) / divisor
29         };
30
31         add(interest); // add interest to this object's amount
32      }
33
34      // return a string representation of a DollarAmount object
35      std::string toString() const {
36         std::string dollars{std::to_string(amount / 100)};
37         std::string cents{std::to_string(std::abs(amount % 100))};
38         return dollars + "." + (cents.size() == 1 ? "0" : "") + cents;
39      }
40   private:
41      int64_t amount{0}; // dollar amount in pennies
42   };
```

Fig. 5.8 | `DollarAmount` class stores dollar amounts as whole numbers of pennies. (Part 2 of 2.)

5.7 Integer-Based Monetary Calculations with Class DollarAmount

- On most computers, an `int`'s range is
  - –2,147,483,647 to 2,147,483,647
- This would limit a `DollarAmount` to approximately ±$21 million
  - too small for many monetary applications
- C++11's `long long` type
  - –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 as a minimum
  - ±$92 quadrillion, likely more than enough for every known monetary application.

5.7 Integer-Based Monetary Calculations with Class DollarAmount

- The range of `long long` (and C++'s other integer types) can vary across platforms.
- For portability, C++11 introduced new integer-type names so you can choose the appropriate range of values for your program.
- `int64_t` supports the exact range
  - –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
  - For a list of C++11's other new integer-type names, see the header <cstdint>.

5.7 Integer-Based Monetary Calculations with Class DollarAmount

- The `add` member function (line 12–15) receives another `DollarAmount` object as an argument and adds its value to the `DollarAmount` object on which add is called.
- Line 14 uses the `+=` operator to add `right.amount` (that is, the amount in the argument object) to the current object's amount, thus modifying the object.

5.7 Integer-Based Monetary Calculations with Class DollarAmount

- The expression `right.amount` accesses the `private` `amount` data member in the object right.
- This is a special relationship among objects of the same class—a member function of a class can access both the `private` data of the object on which that function is called and the `private` data of other objects of the same class that are passed to the function.
- `subtract` (lines 18–21) works similarly to `add`, but uses the `-=` operator to subtract `right.amount` from the current object's amount.

5.7 Integer-Based Monetary Calculations with Class DollarAmount

- `addInterest` (line 25–32) performs the interest calculation using its `rate` and `divisor` parameters, then adds the interest to the `amount`.
- Interest calculations normally yield fractional results that require rounding.
- We perform only integer arithmetic calculations with integer results here, so we completely avoid the representational error of `double`, but as you'll see, rounding is still required.

5.7 Integer-Based Monetary Calculations with Class DollarAmount

- For this example, we use half-up rounding.
- First, consider rounding the floating-point values 0.75 and 0.25 to the nearest integer. Using half-up rounding, values .5 and higher should round up and everything else should round down, so 0.75 rounds up to 1 and 0.25 rounds down to 0.

5.7 Integer-Based Monetary Calculations with Class DollarAmount

- Consider rounding an integer interest calculation. Assume that we're calculating 5% interest on $10.75.
  - We treat 10.75 as 1075, and to calculate 5% interest, we multiply by 5, then divide by 100.
  - In the interest calculation, we first multiply 1075 by 5, yielding the integer value 5375, which represents 53 whole pennies and 75 hundredths of a penny.
  - In this case, the 75 hundredths of a penny should round up to a whole penny, resulting in 54 whole pennies.
  - More generally:
    - 50 to 99 hundredths should round up to the next higher whole penny
    - 1 to 49 hundredths should round down to the next lower whole penny.

5.7 Integer-Based Monetary Calculations with Class DollarAmount

- If we divide 5375 by 100 to complete the interest calculation, the result is 53, which is incorrect
  - integer arithmetic truncates
- To fix this, add 50 to ensure that 50 to 99 hundredths round up. For example:
  - 5350 + 50 yields 5400—dividing that by 100 yields 54
  - 5375 + 50 yields 5425—dividing that by 100 yields 54
  - 5399 + 50 yields 5449—dividing that by 100 yields 54
- Similarly:
  - 5301 + 50 yields 5351—dividing that by 100 yields 53
  - 5325 + 50 yields 5375—dividing that by 100 yields 53
  - 5349 + 50 yields 5399—dividing that by 100 yields 53

5.7 Integer-Based Monetary Calculations with Class DollarAmount

- The following calculation performs the half-up rounding described above:
  - `(amount * rate + divisor / 2) / divisor`
- Rather than adding 50, we add `divisor / 2`.
  - Adding 50 is correct when the divisor is 100, but for other divisors, this would not round to the correct digit.
  - Consider 5.25% interest on $10.75. In integer arithmetic, we treat 10.75 as 1075, and to calculate 5.25% interest, we multiply by 525, then divide by 10000. We first multiply 1075 by 525, yielding the integer value 564375, which represents 56 whole pennies and 4375 ten-thousandths of a penny. This should round down to 56 whole pennies. To round correctly, in this case, we need to add 5000—that is, half of the divisor 10000.

5.7 Integer-Based Monetary Calculations with Class DollarAmount

- After the interest is calculated, line 31 calls member function `add`, passing the new `DollarAmount` object interest as an argument—this updates the amount in the `DollarAmount` object on which `addInterest` was called.
- Any member function of a class can call any other directly to perform operations on the same object of the class.

5.7 Integer-Based Monetary Calculations with Class DollarAmount

- The `toString` member function (line 35–39) returns a `DollarAmount`'s `string` representation
- C++ Standard Library function `to_string` (from header `<string>`) converts a numeric value to a `string` object.
- C++ Standard Library function `abs` (from header `<cmath>`) gets the absolute value of a number.
- "adding" `string` literals and `string` objects using the `+` operator is known as `string` concatenation.
- `string` member function `size` returns the number of characters in a `string` cents.

5.7 Integer-Based Monetary Calculations with Class DollarAmount

- Half-up rounding is a biased technique—fractional amounts of .1, .2, .3 and .4 round down, and .5, .6, .7, .8 and .9 round up.
  - Four values round down and five round up.
  - Because more values round up than down, this can lead to discrepancies in monetary calculations.
- Banker's rounding fixes this problem by rounding .5 to the nearest even integer
  - 0.5 rounds to 0, 1.5 and 2.5 round to 2, 3.5 and 4.5 round to 4, etc.

5.8 do...while Iteration Statement

- Similar to the `while` statement.
- The `do...while` statement tests the loop-continuation condition *after* the loop body executes; therefore, *the loop body always executes at least once.*
- Figure 5.9 uses a `do...while` statement to print the numbers 1–10.

```
1  // Fig. 5.9: DoWhileTest.cpp
2  // do...while iteration statement.
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     unsigned int counter{1};
8
9     do {
10       cout << counter << "  ";
11       ++counter;
12    } while (counter <= 10); // end do...while
13
14    cout << endl;
15 }
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 5.9 | do...while iteration statement.

5.8 do...while Iteration Statement (cont.)

***do...while* Statement UML Activity Diagram**

- Figure 5.10 contains the `do...while` statement's UML activity diagram, which makes it clear that the loop-continuation condition is not evaluated until after the loop performs its body at least once.

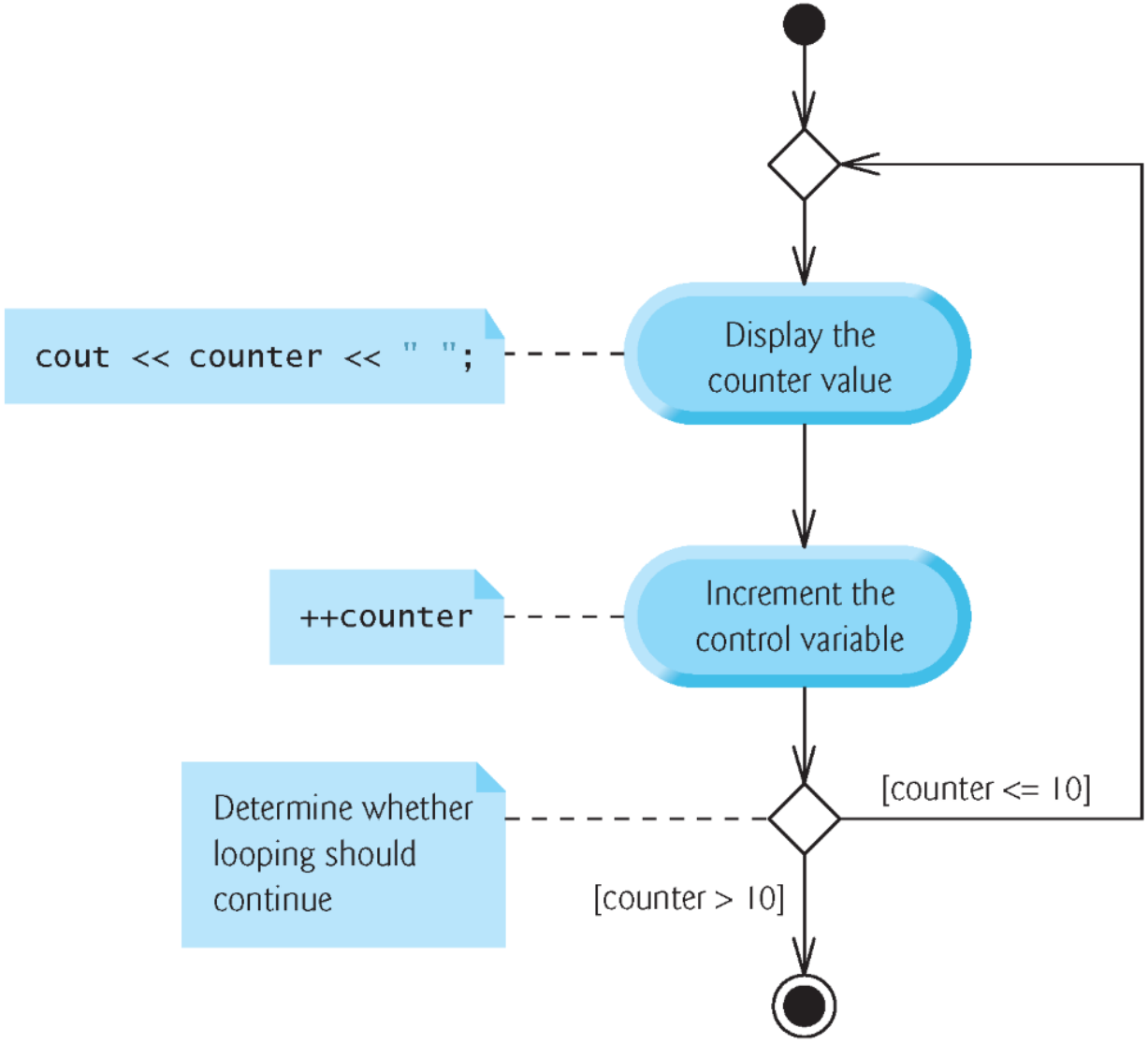

Fig. 5.10 | `do...while` iteration statement UML activity diagram.

5.9 `switch` Multiple-Selection Statement

- The `switch` multiple-selection statement performs many different actions based on the possible values of a variable or expression.
- Each action is associated with the value of an integral constant expression (i.e., any combination of character and integer constants that evaluates to a constant integer value).
- Figure 5.11 calculates the class average of a set of numeric grades entered by the user, and uses a switch statement to determine whether each grade is the equivalent of an A, B, C, D or F and to increment the appropriate grade counter.

```
1  // Fig. 5.11: LetterGrades.cpp
2  // Using a switch statement to count letter grades.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main() {
8     int total{0}; // sum of grades
9     unsigned int gradeCounter{0}; // number of grades entered
10    unsigned int aCount{0}; // count of A grades
11    unsigned int bCount{0}; // count of B grades
12    unsigned int cCount{0}; // count of C grades
13    unsigned int dCount{0}; // count of D grades
14    unsigned int fCount{0}; // count of F grades
15
16    cout << "Enter the integer grades in the range 0-100.\n"
17       << "Type the end-of-file indicator to terminate input:\n"
18       << "   On UNIX/Linux/Mac OS X type <Ctrl> d then press Enter\n"
19       << "   On Windows type <Ctrl> z then press Enter\n";
20
21    int grade;
22
```

Fig. 5.11 | Using a `switch` statement to count letter grades. (Part 1 of 5.)

```
23      // loop until user enters the end-of-file indicator
24      while (cin >> grade) {
25         total += grade; // add grade to total
26         ++gradeCounter; // increment number of grades
27
28         //  increment appropriate letter-grade counter
29         switch (grade / 10) {
30            case 9:  // grade was between 90
31            case 10: // and 100, inclusive
32               ++aCount;
33               break; // exits switch
34
35            case 8: // grade was between 80 and 89
36               ++bCount;
37               break; // exits switch
38
39            case 7: // grade was between 70 and 79
40               ++cCount;
41               break; // exits switch
42
43            case 6: // grade was between 60 and 69
44               ++dCount;
45               break; // exits switch
46
```

Fig. 5.11 | Using a `switch` statement to count letter grades. (Part 2 of 5.)

```
47          default: // grade was less than 60
48             ++fCount;
49             break; // optional; exits switch anyway
50          } // end switch
51       } // end while
52
53       // set floating-point number format
54       cout << fixed << setprecision(2);
55
56       // display grade report
57       cout << "\nGrade Report:\n";
58
59       // if user entered at least one grade...
60       if (gradeCounter != 0) {
61          // calculate average of all grades entered
62          double average = static_cast<double>(total) / gradeCounter;
63
64          // output summary of results
65          cout << "Total of the " << gradeCounter << " grades entered is "
66             << total << "\nClass average is " << average
67             << "\nNumber of students who received each grade:"
68             << "\nA: " << aCount << "\nB: " << bCount << "\nC: " << cCount
69             << "\nD: " << dCount << "\nF: " << fCount << endl;
70       }
```

Fig. 5.11 | Using a `switch` statement to count letter grades. (Part 3 of 5.)

```
71      else { // no grades were entered, so output appropriate message
72         cout << "No grades were entered" << endl;
73      }
74   }
```

```
Enter the integer grades in the range 0-100.
Type the end-of-file indicator to terminate input:
   On UNIX/Linux/Mac OS X type <Ctrl> d then press Enter
   On Windows type <Ctrl> z then press Enter
99
92
45
57
63
71
76
85
90
100
^Z

Grade Report:
Total of the 10 grades entered is 778
Class average is 77.80
```

Fig. 5.11 | Using a `switch` statement to count letter grades. (Part 4 of 5.)

```
Number of students who received each grade:
A: 4
B: 1
C: 2
D: 1
F: 2
```

Fig. 5.11 | Using a `switch` statement to count letter grades. (Part 5 of 5.)

5.9 `switch` Multiple-Selection Statement (cont.)

- Lines 16–19 prompt the user to enter integer grades or type the end-of-file indicator to terminate the input.
- The end-of-file indicator is a system-dependent keystroke combination used to indicate that there's no more data to input.
- In Chapter 14, File Processing, you'll see how the end-of-file indicator is used when a program reads its input from a file.

5.9 switch Multiple-Selection Statement (cont.)

- On UNIX/Linux/Mac OS X systems, end-of-file is entered by typing the following sequence on a line by itself
  - <Ctrl> d
  - This notation means to simultaneously press both the Ctrl key and the d key.
- On Windows systems, end-of-file can be entered by typing
  - <Ctrl> z
  - Windows typically displays the characters ^Z on the screen in response to this key combination

Portability Tip 5.1

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

5.9 `switch` Multiple-Selection Statement (cont.)

- The `switch` statement consists of a series of `case` labels and an optional `default` case.
- When the flow of control reaches the `switch`, the program evaluates the controlling expression in the parentheses.
- Compares the value of the controlling expression with each `case` label.
- If a match occurs, the program executes the statements for that `case`.
- The `break` statement causes program control to proceed with the first statement after the `switch`.

5.9 `switch` Multiple-Selection Statement (cont.)

- Listing `case`s consecutively with no statements between them enables the `case`s to perform the same set of statements.
- Each `case` can have multiple statements.
  - The `switch` selection statement does not require braces around multiple statements in each `case`.
- Without `break` statements, each time a match occurs in the `switch`, the statements for that `case` *and* subsequent `case`s execute until a `break` statement or the end of the `switch` is encountered.
  - Referred to as "falling through" to the statements in subsequent `case`s.

Common Programming Error 5.7

Forgetting a `break` *statement when one is needed in a* `switch` *is a logic error.*

5.9 `switch` Multiple-Selection Statement (cont.)

- If *no* match occurs between the controlling expression's value and a `case` label, the `default` case executes.
- If no match occurs in a `switch` statement that does not contain a `default` case, program control continues with the first statement after the `switch`.

Error-Prevention Tip 5.7

In a `switch`*, ensure that you test for all possible values of the controlling expression.*

5.9 `switch` Multiple-Selection Statement (cont.)

- Figure 5.12 shows the UML activity diagram for the general `switch` multiple-selection statement.
- Most `switch` statements use a `break` in each `case` to terminate the `switch` statement after processing the `case`.
- Figure 5.12 emphasizes this by including `break` statements in the activity diagram.
- Without the `break` statement, control would not transfer to the first statement after the `switch` statement after a `case` is processed.
- Instead, control would transfer to the next `case`'s actions.
- The diagram makes it clear that the `break` statement at the end of a `case` causes control to exit the `switch` statement immediately.

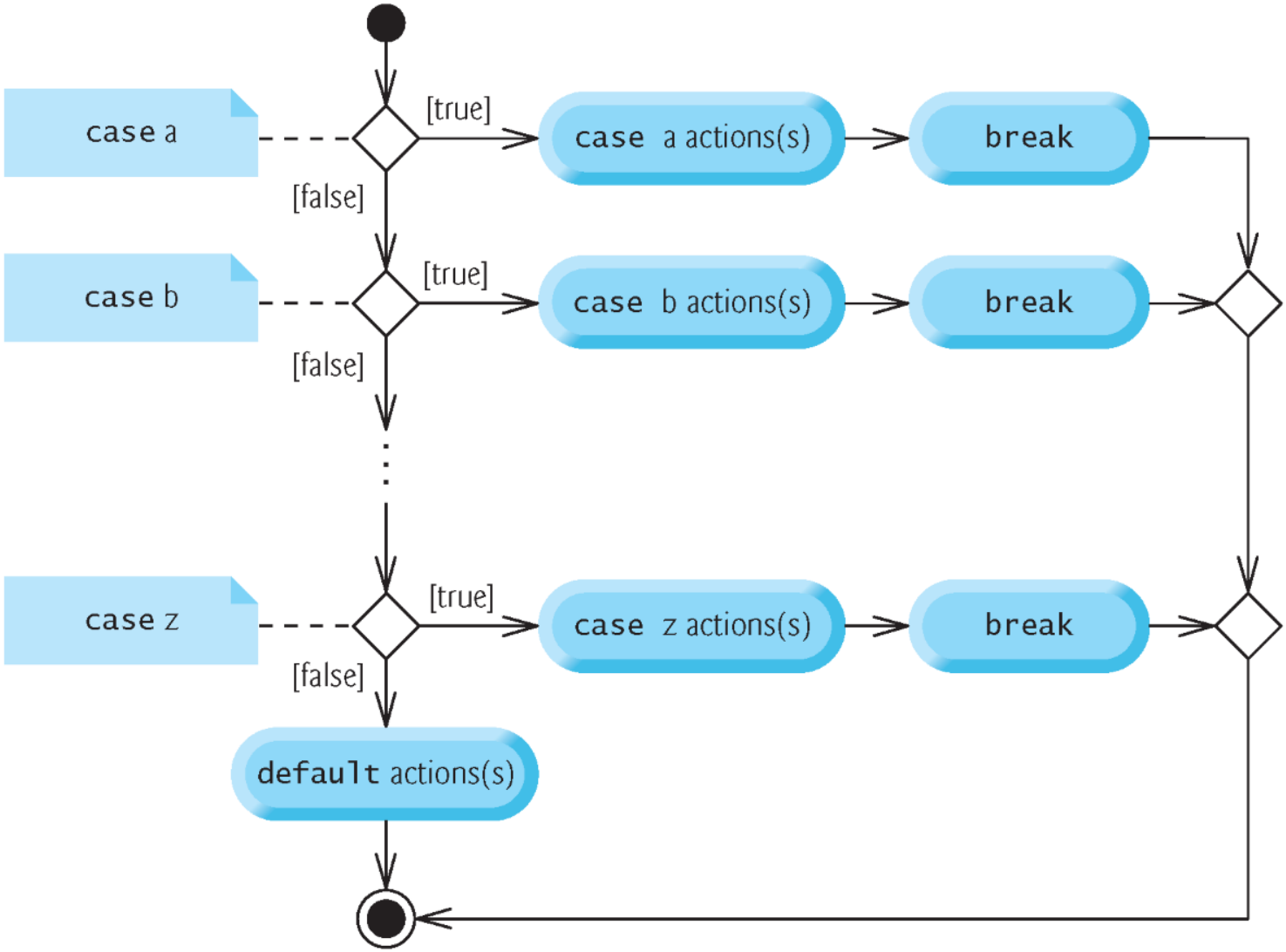

Fig. 5.12 | `switch` multiple-selection statement UML activity diagram with `break` statements.

Error-Prevention Tip 5.8

Provide a `default` *case in* `switch` *statements. This focuses you on the need to process exceptional conditions.*

Good Programming Practice 5.3

Although each `case` *and the* `default` *case in a* `switch` *can occur in any order, place the* `default` *case last. When the* `default` *case is listed last, the* `break` *for that case is not required.*

5.9 `switch` Multiple-Selection Statement (cont.)

- `case` values
  - An integer constant is simply an integer value.
  - In addition, you can use character constants—specific characters in single quotes, such as 'A', '7' or '$'—which represent the integer values of characters and `enum` constants (introduced in Section 6.8). (Appendix B shows the integer values of the characters in the ASCII character set, which is a subset of the Unicode® character set.)
  - The expression in each `case` also can be a constant variable—a variable containing a value which does not change for the entire program.
    - Such a variable is declared with keyword `const`

5.10 `break` and `continue` Statements

- In addition to selection and iteration statements, C++ provides statements `break` (which we discussed in the context of the `switch` statement) and `continue` to alter the flow of control.

5.10.1 break Statement

- The break statement, when executed in a while, for, do…while or switch statement, causes immediate exit from that statement.
- Program execution continues with the next statement.
- Common uses of the break statement are to escape early from a loop or to skip the remainder of a switch statement.
- Figure 5.13 demonstrates the break statement (line 13) exiting a for iteration statement.

```
1  // Fig. 5.13: BreakTest.cpp
2  // break statement exiting a for statement.
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     unsigned int count; // control variable also used after loop
8
9     for (count = 1; count <= 10; count++) { // loop 10 times
10       if (count == 5) {
11          break; // terminates loop if count is 5
12       }
13
14       cout << count << " ";
15    }
16
17    cout << "\nBroke out of loop at count = " << count << endl;
18 }
```

```
1 2 3 4
Broke out of loop at count = 5
```

Fig. 5.13 | break statement exiting a for statement.

5.10.2 `continue` Statement

- The `continue` statement, when executed in a `while`, `for` or `do...while` statement, skips the remaining statements in the body of that statement and proceeds with the next iteration of the loop.
- In `while` and `do...while` statements, the loop-continuation test evaluates immediately after the `continue` statement executes.
- In the `for` statement, the increment expression executes, then the loop-continuation test evaluates.

```
1  // Fig. 5.14: ContinueTest.cpp
2  // continue statement terminating an iteration of a for statement.
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     for (unsigned int count{1}; count <= 10; count++) { // loop 10 times
8        if (count == 5) {
9           continue; // skip remaining code in loop body if count is 5
10       }
11
12       cout << count << " ";
13    }
14
15    cout << "\nUsed continue to skip printing 5" << endl;
16 }
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing 5
```

Fig. 5.14 | `continue` statement terminating an iteration of a `for` statement.

Software Engineering Observation 5.1

Some programmers feel that `break` *and* `continue` *violate structured programming. Since the same effects are achievable with structured-programming techniques, these programmers do not use* `break` *or* `continue`.

Software Engineering Observation 5.2

There's a tension between achieving quality software engineering and achieving the best-performing software. Sometimes one of these goals is achieved at the expense of the other. For all but the most performance-intensive situations, apply the following rule of thumb: First, make your code simple and correct; then make it fast and small, but only if necessary.

5.11 Logical Operators

- C++ provides logical operators that are used to form more complex conditions by combining simple conditions.
- The logical operators are && (logical AND), || (logical OR) and ! (logical NOT, also called logical negation).

5.11.1 Logical AND (&&) Operator

- The && (logical AND) operator is used to ensure that two conditions are *both* `true` before we choose a certain path of execution.
- The simple condition to the left of the && operator evaluates first.
- If necessary, the simple condition to the right of the && operator evaluates next.
- The right side of a logical AND expression is evaluated only if the left side is `true`.

5.10 Logical Operators (cont.)

- Figure 5.15 summarizes the && operator.
- The table shows all four possible combinations of `false` and `true` values for *expression1* and *expression2.*
- Such tables are often called truth tables.
- C++ evaluates to `false` or `true` all expressions that include relational operators, equality operators and/or logical operators.

expression1	expression2	expression1 && expression2
false	false	false
false	true	false
true	false	false
true	true	true

Fig. 5.15 | && (logical AND) operator truth table.

5.11.2 Logical OR (||) Operator

- The || (logical OR) operator determines if either *or* both of two conditions are `true` before we choose a certain path of execution.
- Figure 5.16 is a truth table for the logical OR operator (||).
- The && operator has a higher precedence than the || operator.
- Both operators associate from left to right.

expression1	expression2	expression1 || expression2
false	false	false
false	true	true
true	false	true
true	true	true

Fig. 5.16 | || (logical OR) operator truth table.

5.11.3 Short-Circuit Evaluation

- An expression containing && or || operators evaluates only until the truth or falsehood of the expression is known.
- This performance feature for the evaluation of logical AND and logical OR expressions is called short-circuit evaluation.

Common Programming Error 5.8

In expressions using operator &&, *a condition—we'll call this the* dependent condition*—may require another condition to be true for the evaluation of the dependent condition to be meaningful. In this case, the dependent condition should be placed* after *the* && *operator to prevent errors. Consider the expression* `(i != 0) && (10 / i == 2)`. *The dependent condition* `(10 / i == 2)` must *appear after the* && *operator to prevent the possibility of division by zero.*

5.11.4 Logical Negation (!) Operator

- The ! (logical negation, also called logical NOT or logical complement) operator "reverses" the meaning of a condition.
- Unlike the logical operators && and ||, which are binary operators that combine two conditions, the logical negation operator is a unary operator that has only one condition as an operand.
- Figure 5.17 is a truth table for the logical negation operator.

expression	!expression
`false`	`true`
`true`	`false`

Fig. 5.17 | `!` (logical negation) operator truth table.

5.11.5 Logical Operators Example

- Figure 5.18 demonstrates the logical operators by producing their truth tables.
- The output shows each expression that is evaluated and its `bool` result.
- By default, `bool` values `true` and `false` are displayed by `cout` and the stream insertion operator as `1` and `0`, respectively.
- Stream manipulator `boolalpha` (a sticky manipulator) specifies that the value of each `bool` expression should be displayed as either the word "true" or the word "false."

```
1  // Fig. 5.18: LogicalOperators.cpp
2  // Logical operators.
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     // create truth table for && (logical AND) operator
8     cout << boolalpha << "Logical AND (&&)"
9        << "\nfalse && false: " << (false && false)
10       << "\nfalse && true: " << (false && true)
11       << "\ntrue && false: " << (true && false)
12       << "\ntrue && true: " << (true && true) << "\n\n";
13
14    // create truth table for || (logical OR) operator
15    cout << "Logical OR (||)"
16       << "\nfalse || false: " << (false || false)
17       << "\nfalse || true: " << (false || true)
18       << "\ntrue || false: " << (true || false)
19       << "\ntrue || true: " << (true || true) << "\n\n";
```

Fig. 5.18 | Logical operators. (Part 1 of 2.)

```
20
21      // create truth table for ! (logical negation) operator
22      cout << "Logical negation (!)"
23         << "\n!false: " << (!false)
24         << "\n!true: " << (!true) << endl;
25   }
```

```
Logical AND (&&)
false && false: false
false && true: false
true && false: false
true && true: true

Logical OR (||)
false || false: false
false || true: true
true || false: true
true || true: true

Logical negation (!)
!false: true
!true: false
```

Fig. 5.18 | Logical operators. (Part 2 of 2.)

5.11.5 Logical Operators (cont.)

- Figure 5.19 adds the logical and comma operators to the operator precedence and associativity chart.
- The operators are shown from top to bottom, in decreasing order of precedence.

Operators						Associativity	Type
`::`	`()`					left to right *[See caution in Fig. 2.10 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
`&&`						left to right	logical AND
`||`						left to right	logical OR
`?:`						right to left	conditional
`=`	`+=`	`-=`	`*=`	`/=`	`%=`	right to left	assignment
`,`						left to right	comma

Fig. 5.19 | Operator precedence and associativity.

5.12 Confusing the Equality (==) and Assignment (=) Operators

- Accidentally swapping the operators == (equality) and = (assignment).
- Damaging because they ordinarily do not cause syntax errors.
- Rather, statements with these errors tend to compile correctly and the programs run to completion, often generating incorrect results through runtime logic errors.
- [*Note: Some compilers issue a warning when = is used in a context where == typically is expected.]*
- Two aspects of C++ contribute to these problems.
  - One is that *any expression that produces a value can be used in the decision portion of any control statement.*
  - The second is that assignments produce a value—namely, the value assigned to the variable on the left side of the assignment operator.
- *Any nonzero value is interpreted as* `true`.

5.12 Confusing the Equality (==) and Assignment (=) Operators (cont.)

- Variable names are said to be ***lvalues*** (for "left values") because they can be used on the *left* side of an assignment operator.
- Constants are said to be ***rvalues*** (for "right values") because they can be used on only the *right* side of an assignment operator.
- *Lvalues* can also be used as *rvalues,* but not vice versa.

Error-Prevention Tip 5.9

Programmers normally write conditions such as `x == 7` *with the variable name (an* lvalue*) on the left and the literal (an* rvalue*) on the right. Placing the literal on the left, as in* `7 == x`*, enables the compiler to issue an error if you accidentally replace the* `==` *operator with* `=`*. The compiler treats this as a compilation error, because you can't change a literal's value.*

Common Programming Error 5.9

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

Error-Prevention Tip 5.10

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

5.13 Structured Programming Summary

- We've learned that structured programming produces programs that are easier than unstructured programs to understand, test, debug, modify, and even prove correct in a mathematical sense.
- Figure 5.20 uses activity diagrams to summarize C++'s control statements.
- The initial and final states indicate the *single entry point* and the *single exit point* of each control statement.

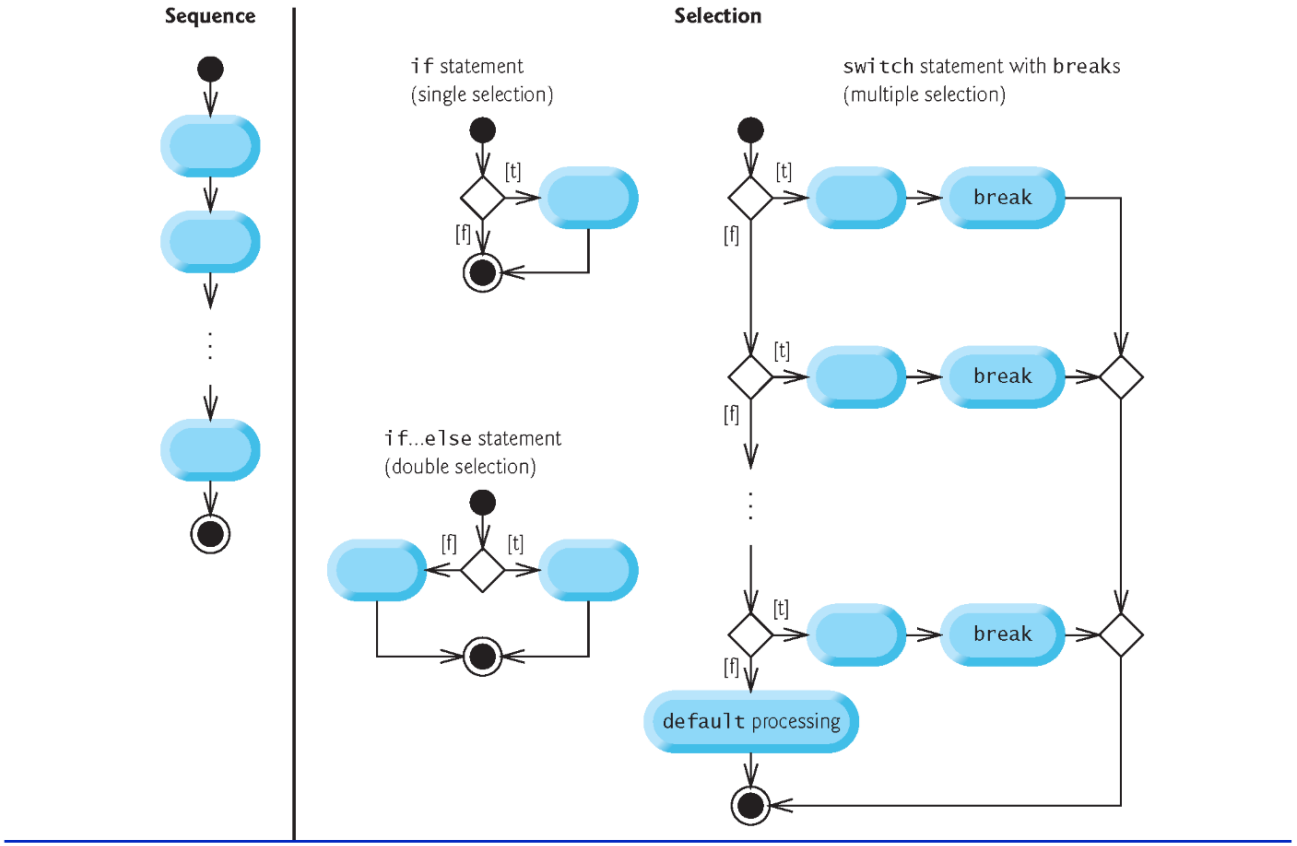

Fig. 5.20 | C++'s single-entry/single-exit sequence, selection and iteration statements. (Part 1 of 2.)

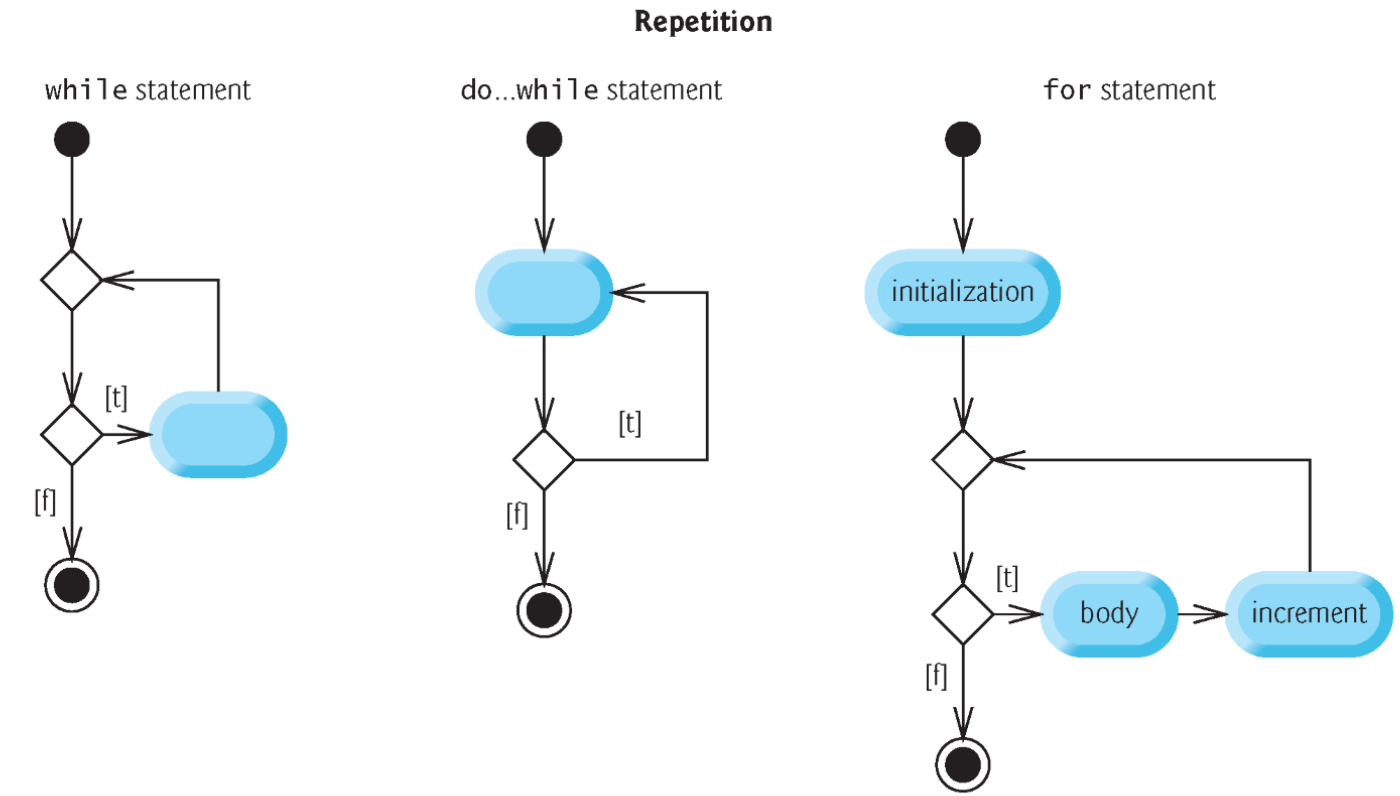

Fig. 5.20 | C++'s single-entry/single-exit sequence, selection and iteration statements. (Part 2 of 2.)

5.13 Structured Programming Summary (cont.)

- Figure 5.21 shows the rules for forming structured programs.
- The rules assume that action states may be used to indicate any action.
- The rules also assume that we begin with the so-called *simplest activity diagram* (Fig. 5.22), consisting of only an initial state, an action state, a final state and transition arrows.
- Applying the rules of Fig. 5.21 always results in an activity diagram with a neat, building-block appearance.
- Rule 2 generates a stack of control statements, so let's call Rule 2 the stacking rule.
- Rule 3 is the nesting rule.

Rules for forming structured programs
1. Begin with the simplest activity diagram (Fig. 5.22).
2. Any action state can be replaced by two action states in sequence.
3. Any action state can be replaced by any control statement (sequence of action states, `if`, `if...else`, `switch`, `while`, `do...while` or `for`).
4. Rules 2 and 3 can be applied as often as you like and in any order.

Fig. 5.21 | Rules for forming structured programs.

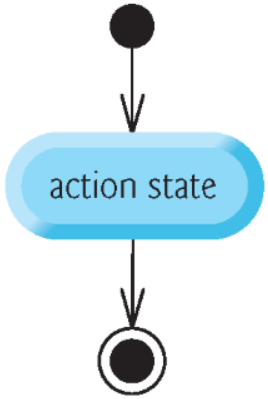

Fig. 5.22 | Simplest activity diagram.

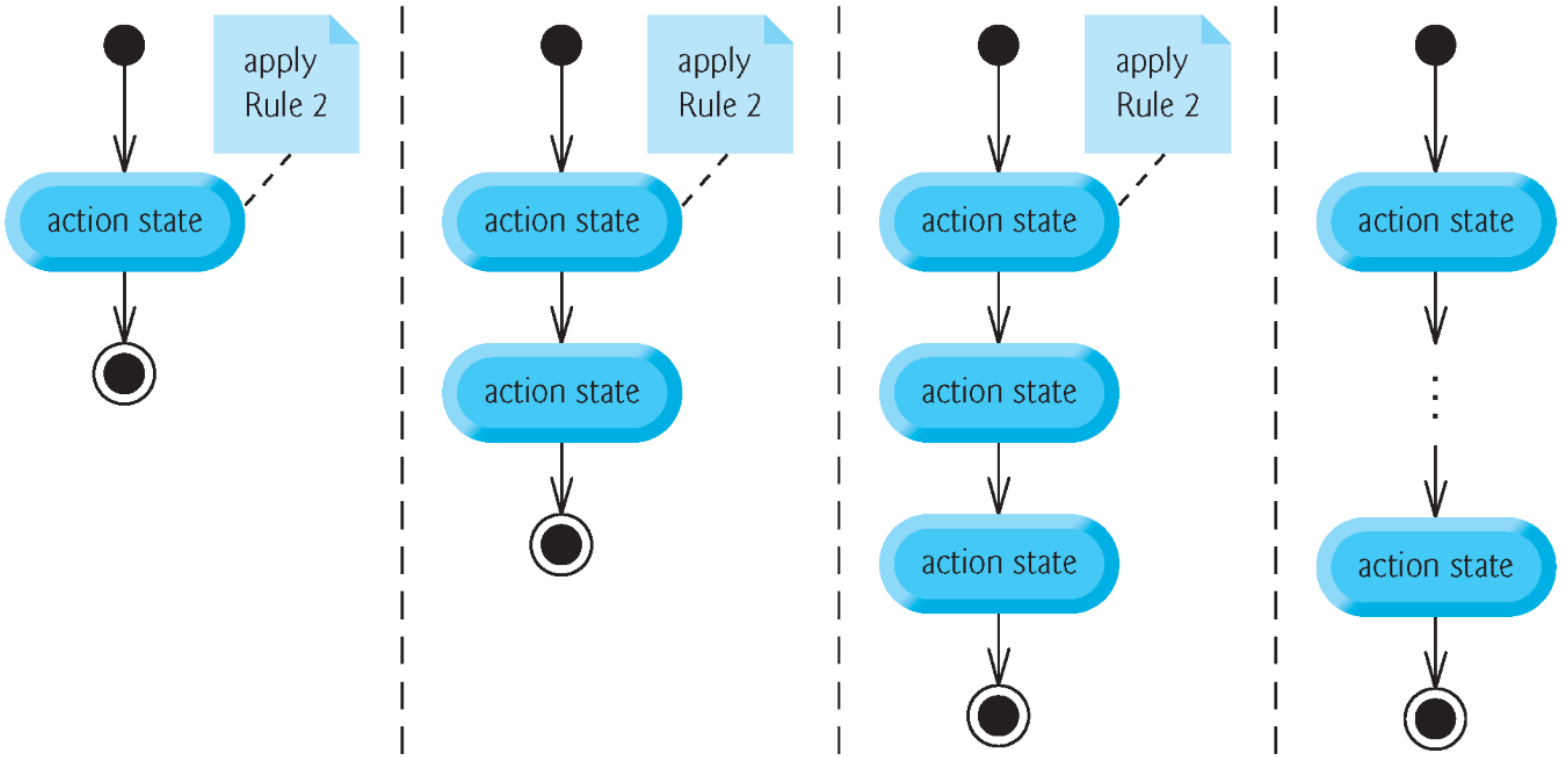

Fig. 5.23 | Repeatedly applying rule 2 of Fig. 5.21 to the simplest activity diagram.

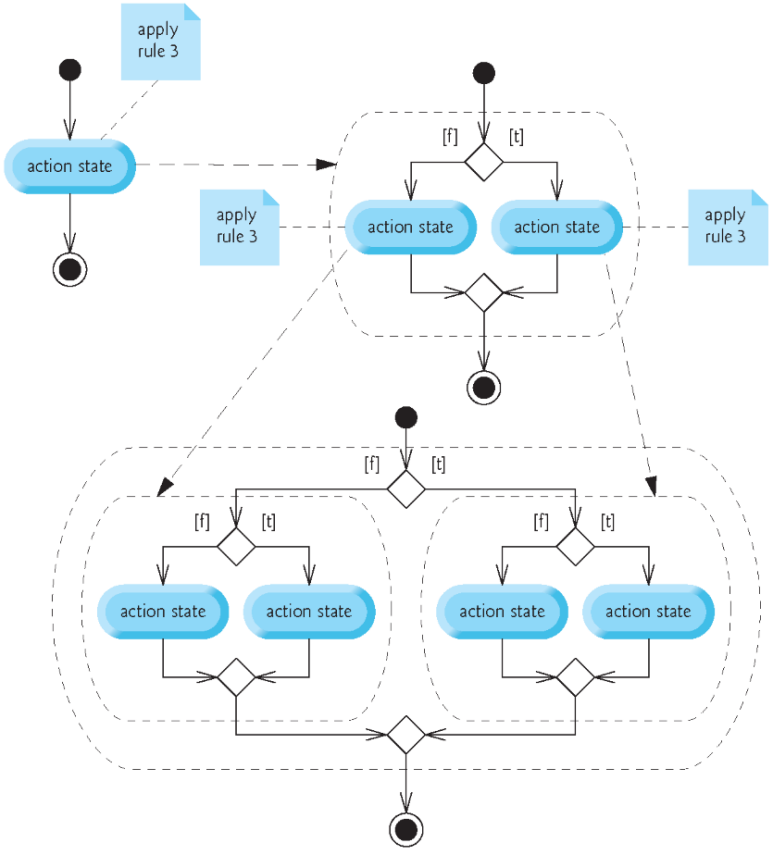

Fig. 5.24 | Repeatedly applying rule 3 of Fig. 5.21 to the simplest activity diagram.

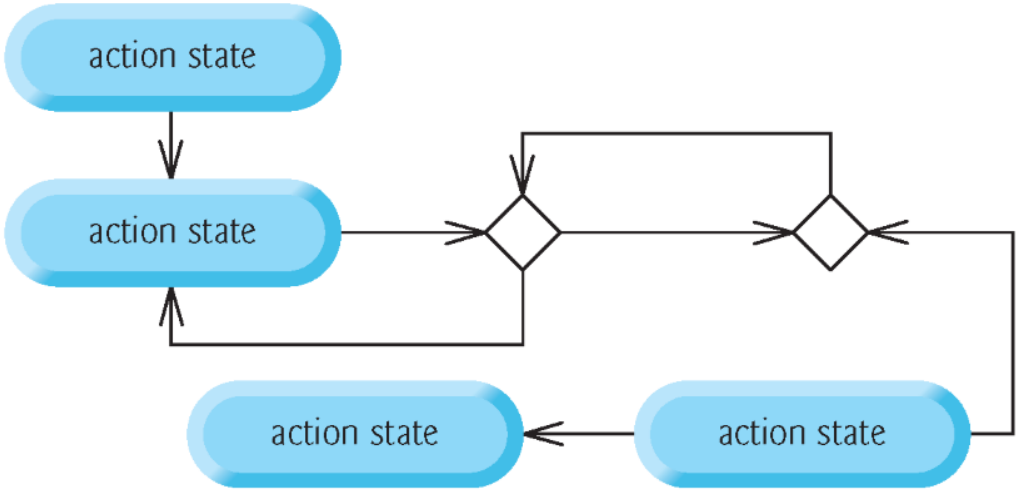

Fig. 5.25 | "Unstructured" activity diagram.