Chapter 4, Control Statements: Part 2

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

OBJECTIVES

In this chapter you'll learn:

- The essentials of counter-controlled repetition.
- To use `for` and `do...while` to execute statements in a program repeatedly.
- To implement multiple selection using the `switch` selection statement.
- How `break` and `continue` alter the flow of control.
- To use the logical operators to form conditional expressions in control statements.
- To avoid confusing the equality and assignment operators.

4.1 Introduction

- `for`, `do...while` and `switch` statements.
- counter-controlled repetition.
- Introduce the `break` and `continue` program control statements.
- Logical operators for more powerful conditional expressions.
- Examine the common error of confusing the equality (==) and assignment (=) operators, and how to avoid it.
- Summarize C++'s control statements.

4.2 Essentials of Counter-Controlled Repetition

- Counter-controlled repetition requires
  - the name of a control variable (or loop counter)
  - the initial value of the control variable
  - the loop-continuation condition that tests for the final value of the control variable (i.e., whether looping should continue)
  - the increment (or decrement) by which the control variable is modified each time through the loop.
- In C++, it's more precise to call a declaration that also reserves memory a definition.

```
1  // Fig. 4.1: fig04_01.cpp
2  // Counter-controlled repetition.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int counter = 1; // declare and initialize control variable
9
10    while ( counter <= 10 ) // loop-continuation condition
11    {
12       cout << counter << " ";
13       counter++; // increment control variable by 1
14    } // end while
15
16    cout << endl; // output a newline
17 } // end main
```

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

Fig. 4.1 | Counter-controlled repetition.

Common Programming Error 4.1

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

Error-Prevention Tip 4.1

Control counting loops with integer values.

Good Programming Practice 4.1

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

Good Programming Practice 4.2

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

Good Programming Practice 4.3

Vertical spacing above and below control statements and indentation of the bodies of control statements give programs a two-dimensional appearance that improves readability.

4.3 `for` Repetition Statement

- The `for` repetition statement specifies the counter-controlled repetition 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.
- If there is more than one statement in the body of the `for`, braces are required to enclose the body of the loop.

```
1  // Fig. 4.2: fig04_02.cpp
2  // Counter-controlled repetition with the for statement.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     // for statement header includes initialization,
9     // loop-continuation condition and increment.
10    for ( int counter = 1; counter <= 10; counter++ )
11       cout << counter << " ";
12
13    cout << endl; // output a newline
14 } // end main
```

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

Fig. 4.2 | Counter-controlled repetition with the `for` statement.

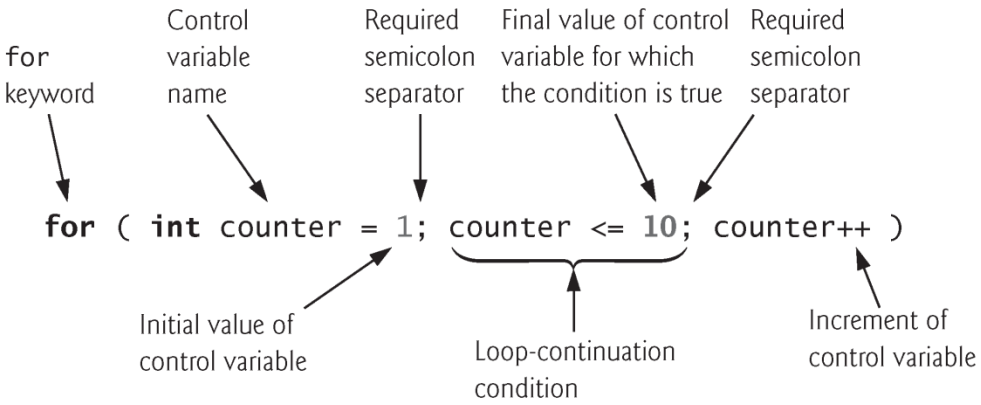

Fig. 4.3 | `for` statement header components.

Common Programming Error 4.2

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

Good Programming Practice 4.4

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

4.3 `for` Repetition 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;
    }
    ```

4.3 `for` Repetition 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 4.3

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

4.3 for Repetition Statement (cont.)

- The *initialization* and *increment* expressions can be comma-separated lists of expressions.
- The commas, as used in these expressions, are comma operators, which guarantee that lists of expressions evaluate from left to right.
  - The lowest precedence of all C++ operators.
- The value and type of a comma-separated list of expressions is the value and type of the rightmost expression.

Good Programming Practice 4.5

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

4.3 `for` Repetition 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.

4.3 `for` Repetition Statement (cont.)

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

Common Programming Error 4.4

Using commas instead of the two required semicolons in a `for` *header is a syntax error.*

Common Programming Error 4.5

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

4.3 `for` Repetition 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 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 4.2

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

4.3 `for` Repetition Statement (cont.)

- The `for` repetition statement's UML activity diagram is similar to that of the `while` statement (Fig. 4.6).
- Figure 4.4 shows the activity diagram of the `for` statement in Fig. 4.2.
- 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.

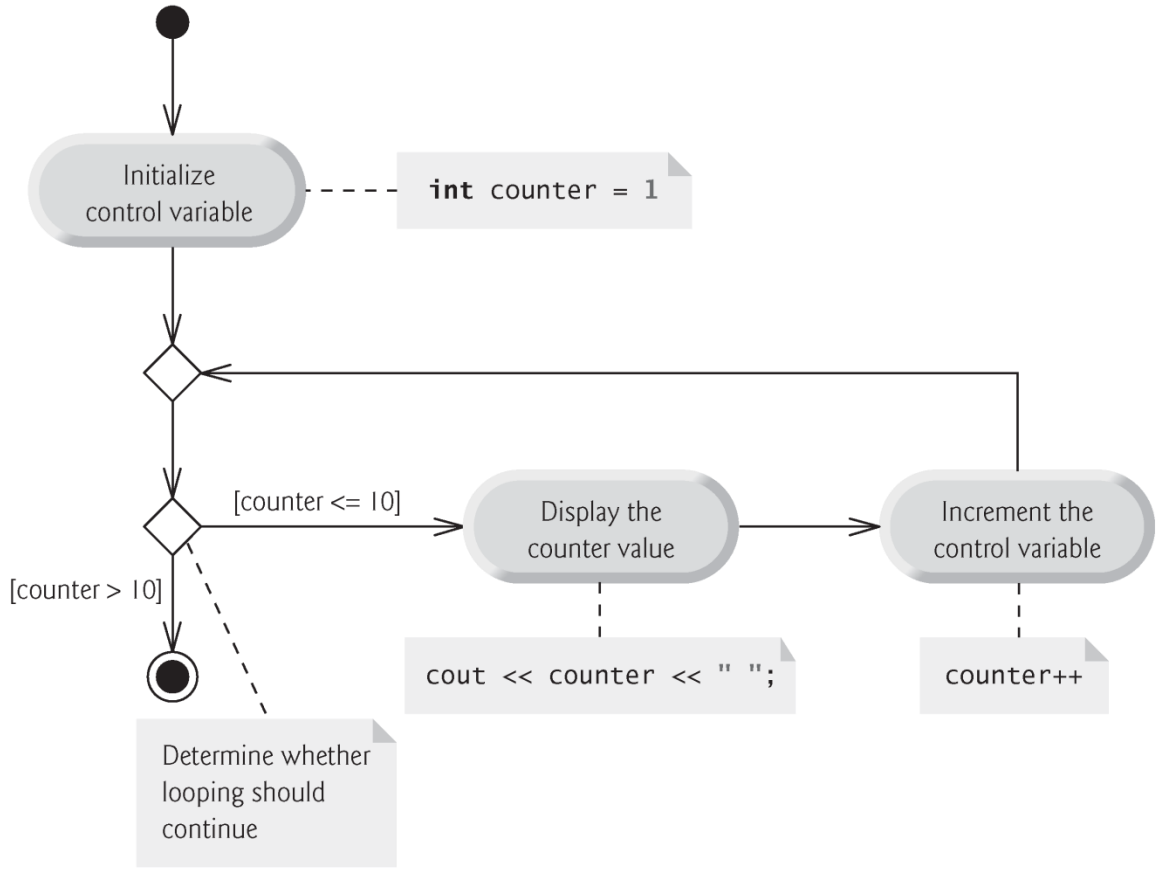

Fig. 4.4 | UML activity diagram for the `for` statement in Fig. 4.2.

4.4 Examples Using the for Statement

- Vary the control variable from 1 to 100 in increments of 1.
  - `for ( int i = 1; i <= 100; i++ )`
- Vary the control variable from 100 down to 1 in decrements of 1.
  - `for ( int i = 100; i >= 1; i-- )`
- Vary the control variable from 7 to 77 in steps of 7.
  - `for ( int i = 7; i <= 77; i += 7 )`
- Vary the control variable from 20 down to 2 in steps of -2.
  - `for ( int i = 20; i >= 2; i -= 2 )`
- Vary the control variable over the following sequence of values: 2, 5, 8, 11, 14, 17.
  - `for ( int i = 2; i <= 17; i += 3 )`
- Vary the control variable over the following sequence of values: 99, 88, 77, 66, 55.
  - `for ( int i = 99; i >= 55; i -= 11 )`

Common Programming Error 4.6

Not using the proper relational operator in the loop-continuation condition of a loop that counts downward (such as incorrectly using `i <= 1` *instead of* `i >= 1` *in a loop counting down to 1) is a logic error that yields incorrect results when the program runs.*

4.4 Examples Using the `for` Statement (cont.)

- The program of Fig. 4.5 uses a `for` statement to sum the even integers from 2 to 20.

```
1  // Fig. 4.5: fig04_05.cpp
2  // Summing integers with the for statement.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int total = 0; // initialize total
9
10    // total even integers from 2 through 20
11    for ( int number = 2; number <= 20; number += 2 )
12       total += number;
13
14    cout << "Sum is " << total << endl; // display results
15 } // end main
```

```
Sum is 110
```

Fig. 4.5 | Summing integers with the `for` statement.

Good Programming Practice 4.6

Although statements preceding a `for` *and statements in the body of a* `for` *often can be merged into the* `for` *header, doing so can make the program more difficult to read, maintain, modify and debug.*

Good Programming Practice 4.7

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

4.4 Examples Using the for Statement (cont.)

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

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

    where

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

```
1  // Fig. 4.6: fig04_06.cpp
2  // Compound interest calculations with for.
3  #include <iostream>
4  #include <iomanip>
5  #include <cmath> // standard C++ math library
6  using namespace std;
7
8  int main()
9  {
10    double amount; // amount on deposit at end of each year
11    double principal = 1000.0; // initial amount before interest
12    double rate = .05; // interest rate
13
14    // display headers
15    cout << "Year" << setw( 21 ) << "Amount on deposit" << endl;
16
17    // set floating-point number format
18    cout << fixed << setprecision( 2 );
19
```

Fig. 4.6 | Compound interest calculations with `for`. (Part 1 of 2.)

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

```
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. 4.6 | Compound interest calculations with `for`. (Part 2 of 2.)

4.4 Examples Using the `for` Statement (cont.)

- C++ does not include an exponentiation opera-tor, 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.

Common Programming Error 4.7

Forgetting to include the appropriate header file when using standard library functions (e.g., `<cmath>` in a program that uses math library functions) is a compilation error.

Good Programming Practice 4.8

Do not use variables of type `float` *or* `double` *to perform monetary calculations. The imprecision of floating-point numbers can cause incorrect monetary values. In the exercises, we explore the use of integers to perform monetary calculations. Some third-party vendors sell C++ class libraries that perform precise monetary calculations.*

4.4 Examples Using the `for` Statement (cont.)

- 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 to accom-modate 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`.

4.4 Examples Using the `for` Statement (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.

Performance Tip 4.1

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

Performance Tip 4.2

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

4.5 do…while Repetition Statement

- Similar to the `while` statement.
- The `do…while` statement tests the loop-continuation con-dition *after* the loop body executes; therefore, the loop body always executes at least once.
- It's not necessary to use braces in the `do…while` statement if there is only one statement in the body.
  - Most programmers include the braces to avoid confusion between the `while` and `do…while` statements.
- Must end a `do…while` statement with a semicolon.

Good Programming Practice 4.9

Always including braces in a `do...while` *statement helps eliminate ambiguity between the* `while` *statement and the* `do...while` *statement containing one statement.*

```
1  // Fig. 4.7: fig04_07.cpp
2  // do...while repetition statement.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int counter = 1; // initialize counter
9
10    do
11    {
12       cout << counter << " "; // display counter
13       counter++; // increment counter
14    } while ( counter <= 10 ); // end do...while
15
16    cout << endl; // output a newline
17 } // end main
```

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

Fig. 4.7 | do...while repetition statement.

4.5 do…while Repetition Statement

- Figure 4.8 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.

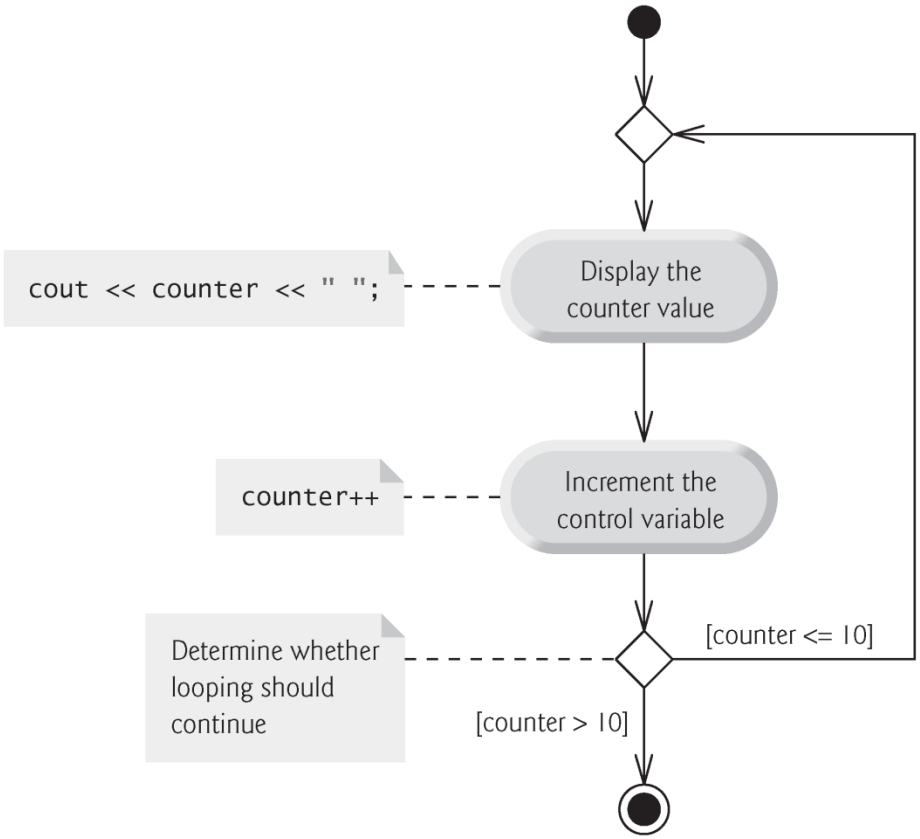

Fig. 4.8 | UML activity diagram for the `do...while` repetition statement of Fig. 4.7.

4.6 `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 a constant integral expression (i.e., any combination of character and integer constants that evaluates to a constant integer value).

```
1  // Fig. 4.9: fig04_09.cpp
2  // Using a switch statement to count A, B, C, D and F grades.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int grade; // letter grade entered by user
9     int aCount; // count of A grades
10    int bCount; // count of B grades
11    int cCount; // count of C grades
12    int dCount; // count of D grades
13    int fCount; // count of F grades
14
15    cout << "Enter the letter grades." << endl
16       << "Enter the EOF character to end input." << endl;
17
```

Fig. 4.9 | Using a `switch` statement to count A, B, C, D and F grades. (Part 1 of 5.)

```
18      // loop until user types end-of-file key sequence
19      while ( ( grade = cin.get() ) != EOF )
20      {
21         // determine which grade was entered
22         switch ( grade ) // switch statement nested in while
23         {
24            case 'A': // grade was uppercase A
25            case 'a': // or lowercase a
26               aCount++; // increment aCount
27               break; // necessary to exit switch
28
29            case 'B': // grade was uppercase B
30            case 'b': // or lowercase b
31               bCount++; // increment bCount
32               break; // exit switch
33
34            case 'C': // grade was uppercase C
35            case 'c': // or lowercase c
36               cCount++; // increment cCount
37               break; // exit switch
38
```

Fig. 4.9 | Using a `switch` statement to count A, B, C, D and F grades. (Part 2 of 5.)

```
39          case 'D': // grade was uppercase D
40          case 'd': // or lowercase d
41             dCount++; // increment dCount
42             break; // exit switch
43
44          case 'F': // grade was uppercase F
45          case 'f': // or lowercase f
46             fCount++; // increment fCount
47             break; // exit switch
48
49          case '\n': // ignore newlines,
50          case '\t': // tabs,
51          case ' ': // and spaces in input
52             break; // exit switch
53
54          default: // catch all other characters
55             cout << "Incorrect letter grade entered."
56                << " Enter a new grade." << endl;
57             break; // optional; will exit switch anyway
58       } // end switch
59    } // end while
60
```

Fig. 4.9 | Using a `switch` statement to count A, B, C, D and F grades. (Part 3 of 5.)

```
61      // output summary of results
62      cout << "\n\nNumber of students who received each letter grade:"
63         << "\nA: " << aCount // display number of A grades
64         << "\nB: " << bCount // display number of B grades
65         << "\nC: " << cCount // display number of C grades
66         << "\nD: " << dCount // display number of D grades
67         << "\nF: " << fCount // display number of F grades
68         << endl;
69   } // end function main
```

Fig. 4.9 | Using a `switch` statement to count A, B, C, D and F grades. (Part 4 of 5.)

```
Enter the letter grades.
Enter the EOF character to end input.
a
B
c
C
A
d
f
C
E
Incorrect letter grade entered. Enter a new grade.
D
A
b
^Z

Number of students who received each letter grade:
A: 3
B: 2
C: 3
D: 2
F: 1
```

Fig. 4.9 | Using a `switch` statement to count A, B, C, D and F grades. (Part 5 of 5.)

4.6 switch Multiple-Selection Statement (cont.)

- The `cin.get()` function reads one character from the keyboard.
- Normally, characters are stored in variables of type `char`; however, characters can be stored in any integer data type, because types `short`, `int` and `long` are guaranteed to be at least as big as type `char`.
- Can treat a character either as an integer or as a character, depending on its use.
- For example, the stat-ment
  - ```
    cout << "The character (" << 'a' << ") has the value "
       << static_cast< int > ( 'a' ) << endl;
    ```
- prints the character `a` and its integer value as follows:
  - ```
    The character (a) has the value 97
    ```
- The integer 97 is the character's numerical representation in the computer.

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

- Generally, assignment statements have the value that is assigned to the variable on the left side of the =.
- `EOF` stands for "end-of-file". Commonly used as a sentinel value.
  - *However, you do not type the value –1, nor do you type the letters EOF as the sentinel value.*
  - You type a system-dependent keystroke combination that means "end-of-file" to indicate that you have no more data to enter.
- `EOF` is a symbolic integer constant defined in the `<iostream>` header file.

Portability Tip 4.1

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

Portability Tip 4.2

Testing for the symbolic constant `EOF` *rather than* `-1` *makes programs more portable. The ANSI/ISO C standard, from which C++ adopts the definition of* `EOF`*, states that* `EOF` *is a negative integral value, so* `EOF` *could have different values on different systems.*

4.6 `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 expression in the parentheses.
  - The controlling expression.
- The `switch` statement 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 af-ter the `switch`.

4.6 `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 4.8

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

Common Programming Error 4.9

Omitting the space between the word `case` *and the integral value being tested in a* `switch` *statement—e.g., writing* `case3:` *instead of* `case 3:`*—is a logic error. The* `switch` *statement will not perform the appropriate actions when the controlling expression has a value of 3.*

4.6 `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`.

Good Programming Practice 4.10

Provide a `default` *case in* `switch` *statements. Cases not explicitly tested in a* `switch` *statement without a* `default` *case are ignored. Including a* `default` *case focuses you on the need to process exceptional conditions. There are situations in which no* `default` *processing is needed. Although the* `case` *clauses and the* `default` *case clause in a* `switch` *statement can occur in any order, it's common practice to place the* `default` *clause last.*

Good Programming Practice 4.11

The last `case` *in a* `switch` *statement does not require a* `break` *statement. Some programmers include this* `break` *for clarity and for symmetry with other cases.*

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

- Reading characters one at a time can cause some problems.
- To have the program read the characters, we must send them to the computer by pressing the *Enter key*.
- This places a newline character in the input after the character we wish to process.
- Often, this newline character must be specially processed.

Common Programming Error 4.10

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

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

- Figure 4.12 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 4.12 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.

Common Programming Error 4.11

Specifying a nonconstant integral expression in a `switch`*'s* `case` *label is a syntax error.*

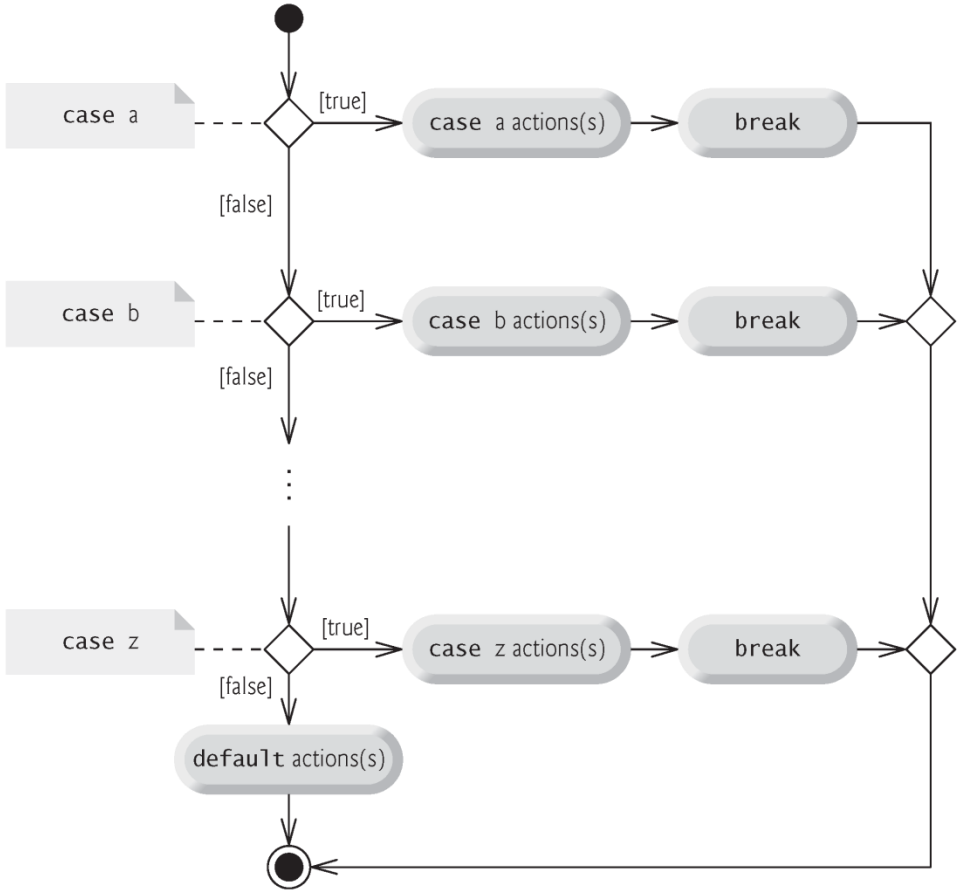

Fig. 4.10 | `switch` multiple-selection statement UML activity diagram with `break` statements

Common Programming Error 4.12

Providing identical case labels in a `switch` *statement is a compilation error. Providing case labels containing different expressions that evaluate to the same value also is a compilation error. For example, placing* `case 4 + 1:` *and* `case 3 + 2:` *in the same* `switch` *statement is a compilation error, because these are both equivalent to* `case 5:`.

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

- C++ has flexible data type sizes (see Appendix C, Fundamental Types).
- C++ provides several integer types.
- The range of integer values for each type depends on the particular computer's hardware.
- In addition to the types `int` and `char`, C++ provides the types `short` (an abbreviation of `short int`) and `long` (an abbreviation of `long int`).
- The minimum range of values for `short` integers is –32,768 to 32,767.
- For the vast majority of integer calculations, `long` integers are sufficient.
- The minimum range of values for `long` integers is –2,147,483,648 to 2,147,483,647.

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

- On most computers, `int`s are equivalent either to `short` or to `long`.
- The range of values for an `int` is at least the same as that for `short` integers and no larger than that for `long` integers.
- The data type `char` can be used to represent any of the characters in the computer's character set.
- It also can be used to represent small integers.

Portability Tip 4.3

Because `int`*s can vary in size between systems, use* `long` *integers if you expect to process integers outside the range –32,768 to 32,767 and you'd like to run the program on several different computer systems.*

Performance Tip 4.3

If memory is at a premium, it might be desirable to use smaller integer sizes.

4.7 break and continue Statements

- 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.

```
1  // Fig. 4.11: fig04_11.cpp
2  // break statement exiting a for statement.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int count; // control variable also used after loop terminates
9
10    for ( count = 1; count <= 10; count++ ) // loop 10 times
11    {
12       if ( count == 5 )
13          break; // break loop only if x is 5
14
15       cout << count << " ";
16    } // end for
17
18    cout << "\nBroke out of loop at count = " << count << endl;
19 } // end main
```

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

Fig. 4.11 | `break` statement exiting a `for` statement.

4.7 break and continue Statements (cont.)

- 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. 4.12: fig04_12.cpp
2  // continue statement terminating an iteration of a for statement.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     for ( int count = 1; count <= 10; count++ ) // loop 10 times
9     {
10       if ( count == 5 ) // if count is 5,
11          continue;      // skip remaining code in loop
12
13       cout << count << " ";
14    } // end for
15
16    cout << "\nUsed continue to skip printing 5" << endl;
17 } // end main
```

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

Fig. 4.12 | `continue` statement terminating a single iteration of a `for` statement.

Performance Tip 4.4

The `break` *and* `continue` *statements, when used properly, perform faster than do the corresponding structured techniques.*

Good Programming Practice 4.12

Some programmers feel that `break` *and* `continue` *violate structured programming. The effects of these statements can be achieved by structured programming techniques we soon will learn, so these programmers do not use* `break` *and* `continue`*. Most programmers consider the use of* `break` *in* `switch` *statements acceptable.*

Software Engineering Observation 4.1

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

4.8 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).

4.8 Logical Operators (cont.)

- 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`.

Common Programming Error 4.13

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

4.8 Logical Operators (cont.)

- Figure 4.15 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 in-clude 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. 4.13 | && (logical AND) operator truth table.

4.8 Logical Operators (cont.)

- The || (logical OR) operator determines if either *or both* of two conditions are `true` before we choose a certain path of execution.
- Figure 4.16 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.
- 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.

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

Fig. 4.14 | || (logical OR) operator truth table.

Performance Tip 4.5

In expressions using operator `&&`*, if the separate conditions are independent of one another, make the condition most likely to be* `false` *the leftmost condition. In expressions using operator* `||`*, make the condition most likely to be* `true` *the leftmost condition. This use of short-circuit evaluation can reduce a program's execution time.*

4.8 Logical Operators (cont.)

- C++ provides the ! (logical NOT, also called logical negation) operator to "reverse" a condition's meaning.
- The unary logical negation operator has only a single condition as an operand.
- You can often avoid the ! operator by using an appropriate relational or equality operator.
- Figure 4.17 is a truth table for the logical negation operator (!).

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

Fig. 4.15 | ! (logical negation) operator truth table.

4.8 Logical Operators (cont.)

- Figure 4.18 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. 4.16: fig04_16.cpp
2  // Logical operators.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     // create truth table for && (logical AND) operator
9     cout << boolalpha << "Logical AND (&&)"
10       << "\nfalse && false: " << ( false && false )
11       << "\nfalse && true: " << ( false && true )
12       << "\ntrue && false: " << ( true && false )
13       << "\ntrue && true: " << ( true && true ) << "\n\n";
14
15    // create truth table for || (logical OR) operator
16    cout << "Logical OR (||)"
17       << "\nfalse || false: " << ( false || false )
18       << "\nfalse || true: " << ( false || true )
19       << "\ntrue || false: " << ( true || false )
20       << "\ntrue || true: " << ( true || true ) << "\n\n";
21
```

Fig. 4.16 | Logical operators. (Part 1 of 2.)

```
22      // create truth table for ! (logical negation) operator
23      cout << "Logical NOT (!)"
24         << "\n!false: " << ( !false )
25         << "\n!true: " << ( !true ) << endl;
26   } // end main
```

```
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 NOT (!)
!false: true
!true: false
```

Fig. 4.16 | Logical operators. (Part 2 of 2.)

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

Fig. 4.17 | Operator precedence and associativity.

4.9 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`

Common Programming Error 4.14

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

Error-Prevention Tip 4.3

Programmers normally write conditions such as `x == 7` *with the variable name on the left and the constant on the right. By placing the constant on the left, as in* `7 == x`*, you'll be protected by the compiler if you accidentally replace the* `==` *operator with* `=`*. The compiler treats this as a compilation error, because you can't change the value of a constant. This will prevent the potential devastation of a runtime logic error.*

4.9 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 4.4

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

4.10 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 4.20 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.

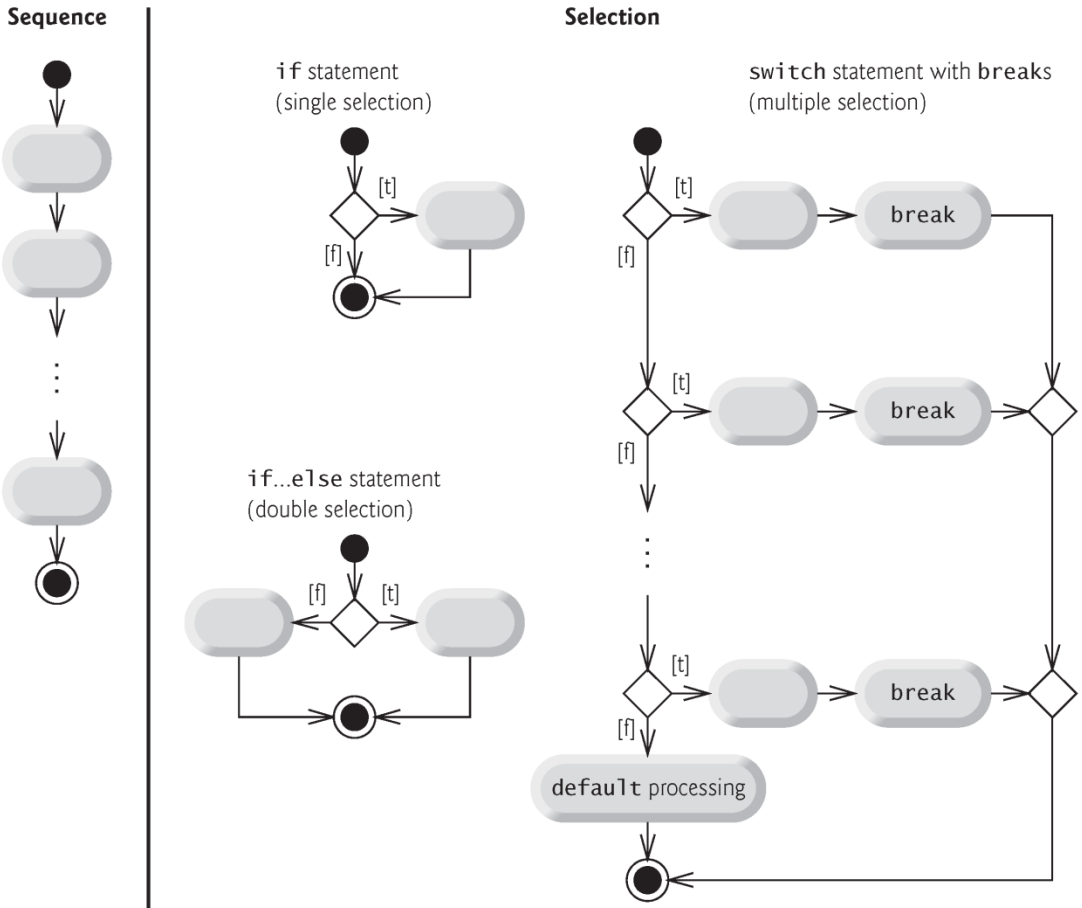

Fig. 4.18 | C++'s single-entry/single-exit sequence, selection and repetition statements. (Part 1 of 2.)

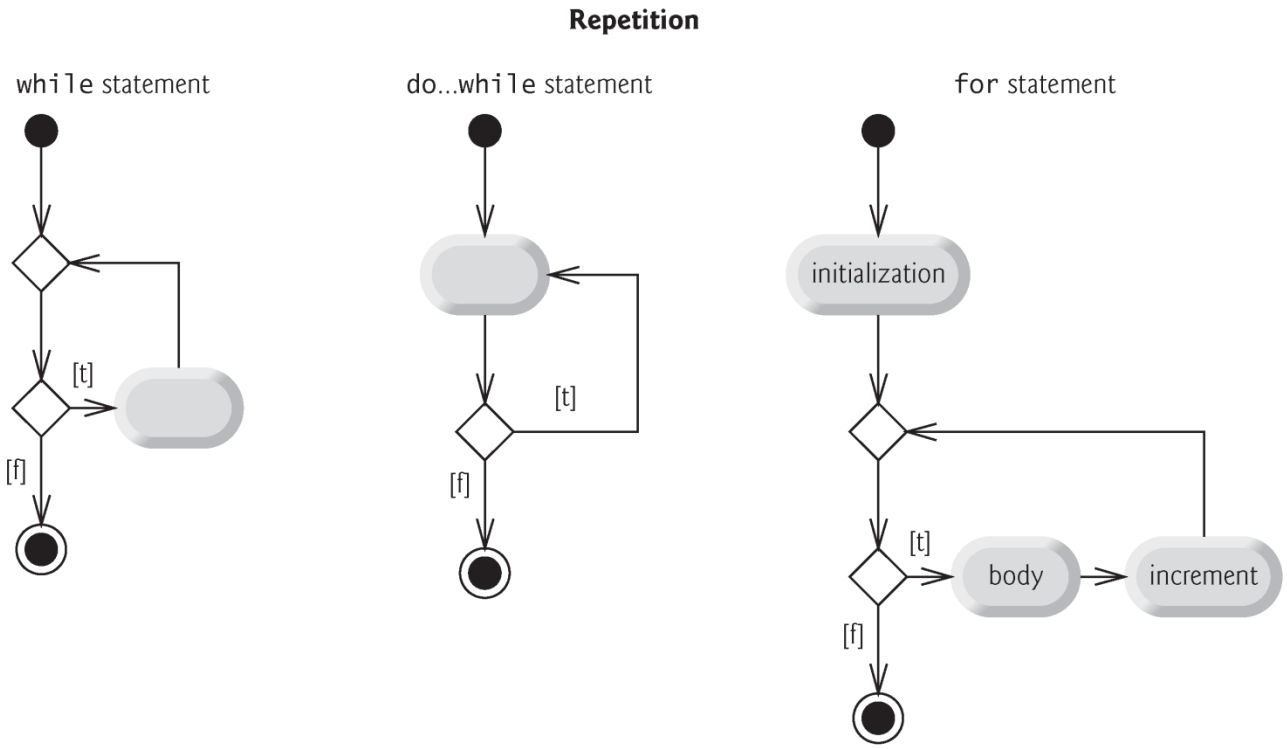

Fig. 4.18 | C++'s single-entry/single-exit sequence, selection and repetition statements. (Part 2 of 2.)

4.10 Structured Programming Summary (cont.)

- Figure 4.21 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. 4.22), consisting of only an initial state, an action state, a final state and transition arrows.
- Applying the rules of Fig. 4.21 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. 4.20).
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, `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. 4.19 | Rules for forming structured programs.

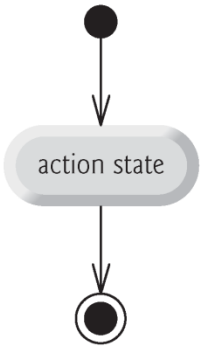

Fig. 4.20 | Simplest activity diagram.

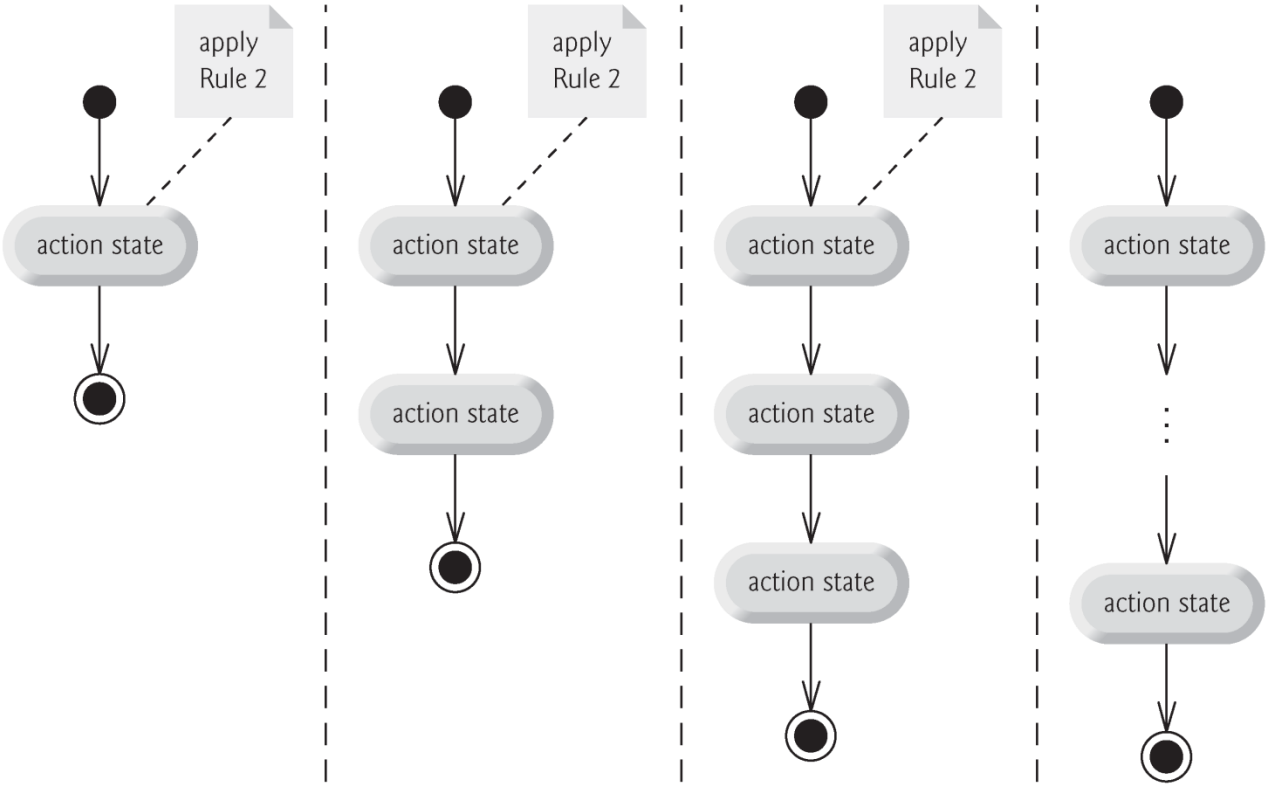

Fig. 4.21 | Repeatedly applying Rule 2 of Fig. 4.19 to the simplest activity diagram.

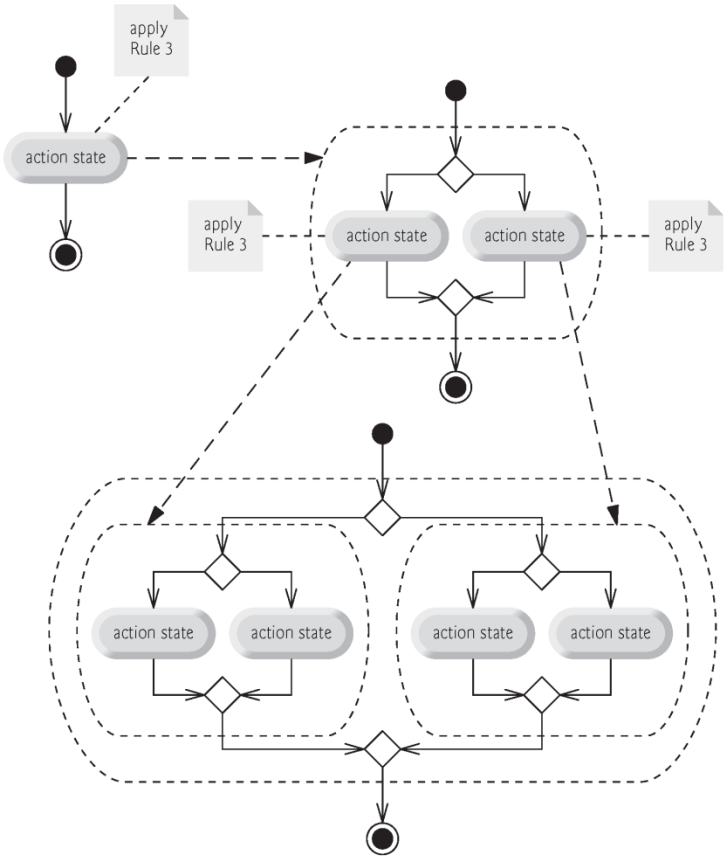

Fig. 4.22 | Applying Rule 3 of Fig. 4.19 to the simplest activity diagram several times.

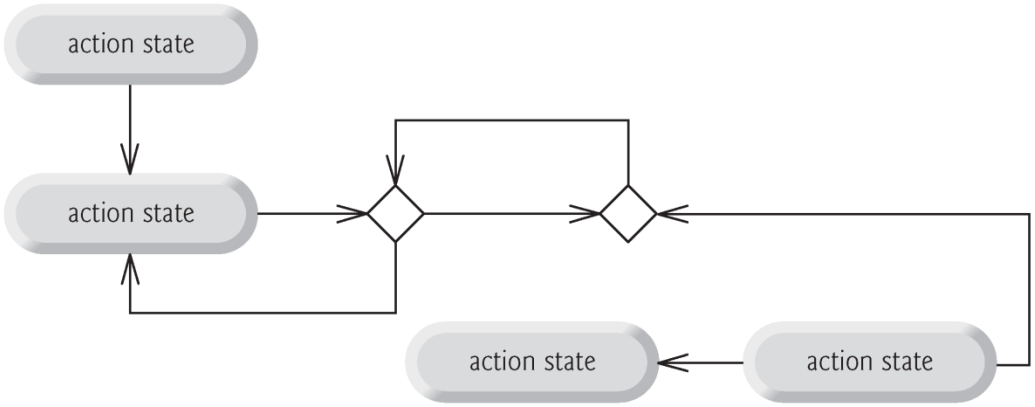

Fig. 4.23 | Activity diagram with illegal syntax.