5

Control Statements: Part 2

Not everything that can be counted counts, and not every thing that counts can be counted.

— Albert Einstein

Who can control his fate?

— William Shakespeare

The used key is always bright.

— Benjamin Franklin

Intelligence… is the faculty of making artificial objects, especially tools to make tools.

— Henri Bergson

Every advantage in the past is judged in the light of the final issue.

— Demosthenes

OBJECTIVES

In this chapter you will learn:

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

Outline

5.1 Introduction

- **Continue structured programming discussion**
  - **Introduce C++'s remaining control structures**
    - `for`, `do...while`, `switch`

5.2 Essentials of Counter-Controlled Repetition

- **Counter-controlled repetition requires:**
  - **Name of a control variable (loop counter)**
  - **Initial value of the control variable**
  - **Loop-continuation condition that tests for the final value of the control variable**
  - **Increment/decrement of control variable at each iteration**

Outline

fig05_01.cpp

(1 of 1)

```
1  // Fig. 5.1: fig05_01.cpp
2  // Counter-controlled repetition.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int counter = 1; // declare and initialize control variable
10
11    while ( counter <= 10 ) // loop-continuation condition
12    {
13       cout << counter << " ";
14       counter++; // increment control variable by 1
15    } // end while
16
17    cout << endl; // output a newline
18    return 0; // successful termination
19 } // end main
```

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

Control-variable name is **counter** with variable initial value **1**

Condition tests for **counter**'s final value

Increment the value in **counter**

Common Programming Error 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.

Error-Prevention Tip 5.1

Control counting loops with integer values.

Good Programming Practice 5.1

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

Good Programming Practice 5.2

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 5.3

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

5.3 `for` Repetition Statement

- **`for` repetition statement**
  - **Specifies counter-controlled repetition details in a single line of code**

Outline

fig05_02.cpp

(1 of 1)

```
1  // Fig. 5.2: fig05_02.cpp
2  // Counter-controlled repetition with the for statement.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     // for statement header includes initialization,
10    // loop-continuation condition and increment.
11    for ( int counter = 1; counter <= 10; counter++ )
12       cout << counter << " ";
13
14    cout << endl; // output a newline
15    return 0; // indicate successful termination
16 } // end main
```

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

Control-variable name is **counter** with initial value **1**

Condition tests for **counter**'s final value

Increment for **counter**

Common Programming Error 5.2

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.

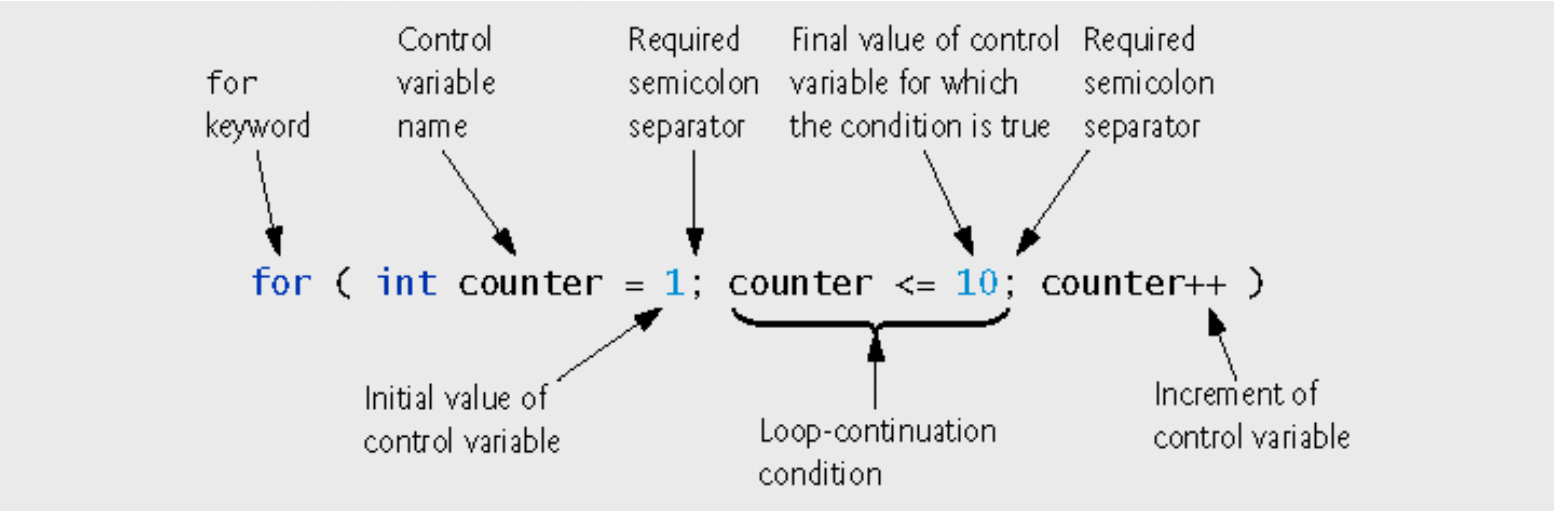


Fig. 5.3 | `for` statement header components.

Good Programming Practice 5.4

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`.

5.3 `for` Repetition Statement (Cont.)

- **General form of the `for` statement**
  - `for ( initialization; loopContinuationCondition; increment )`
    statement`;`
- **Can usually be rewritten as:**
  - *initialization*`;`
    `while ( loopContinuationCondition )`
    `{`
    statement`;`
    increment`;`
    `}`
- **If the control variable is declared in the *initialization* expression**
  - **It will be unknown outside the `for` statement**

Common Programming Error 5.3

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

Portability Tip 5.1

In the C++ standard, the scope of the control variable declared in the initialization section of a `for` statement differs from the scope in older C++ compilers. In pre-standard compilers, the scope of the control variable does not terminate at the end of the block defining the body of the `for` statement; rather, the scope terminates at the end of the block that encloses the `for` statement. C++ code created with prestandard C++ compilers can break when compiled on standard-compliant compilers. If you are working with prestandard compilers and you want to be sure your code will work with standard-compliant compilers, there are two defensive programming strategies you can use: either declare control variables with different names in every `for` statement, or, if you prefer to use the same name for the control variable in several `for` statements, declare the control variable before the first `for` statement.

Good Programming Practice 5.5

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).

5.3 `for` Repetition Statement (Cont.)

- **The *initialization* and *increment* expressions can be comma-separated lists of expressions**
  - **These commas are comma operators**
    - **Comma operator has the lowest precedence of all operators**
  - **Expressions are evaluated from left to right**
  - **Value and type of entire list are value and type of the rightmost expressions**

Common Programming Error 5.4

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

Common Programming Error 5.5

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.

Software Engineering Observation 5.1

Placing a semicolon immediately after a `for` header is sometimes used to create a so-called delay loop. Such a `for` loop with an empty body still loops the indicated number of times, doing nothing other than the counting. For example, you might use a delay loop to slow down a program that is producing outputs on the screen too quickly for you to read them. Be careful though, because such a time delay will vary among systems with different processor speeds.

Error-Prevention Tip 5.2

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.

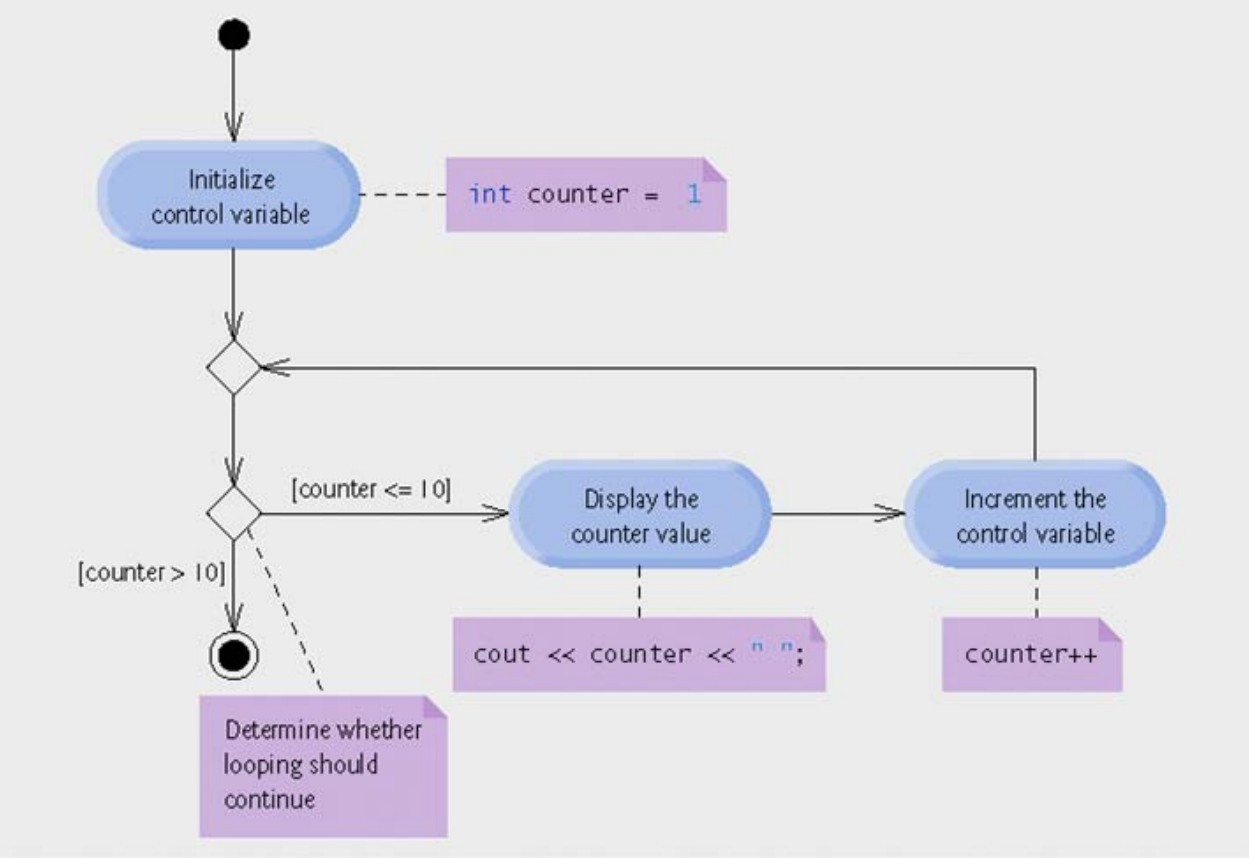


Fig. 5.4 | UML activity diagram for the for statement in Fig. 5.2.

5.4 Examples Using the `for` Statement

- `for` statement examples
  - Vary control variable from `1` to `100` in increments of `1`
    - `for ( int i = 1; i <= 100; i++ )`
  - Vary control variable from `100` to `1` in increments of `-1`
    - `for ( int i = 100; i >= 1; i-- )`
  - Vary control variable from `7` to `77` in steps of `7`
    - `for ( int i = 7; i <= 77; i += 7 )`
  - Vary control variable from `20` to `2` in steps of `-2`
    - `for ( int i = 20; i >= 2; i -= 2 )`
  - Vary control variable over the sequence: `2, 5, 8, 11, 14, 17, 20`
    - `for ( int i = 2; i <= 20; i += 3 )`
  - Vary control variable over the sequence: `99, 88, 77, 66, 55, 44, 33, 22, 11, 0`
    - `for ( int i = 99; i >= 0; i -= 11 )`

Common Programming Error 5.6

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 usually a logic error that yields incorrect results when the program runs.

Outline

fig05_05.cpp

```
1  // Fig. 5.5: fig05_05.cpp
2  // Summing integers with the for statement.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int total = 0; // initialize total
10
11    // total even integers from 2 through 20
12    for ( int number = 2; number <= 20; number += 2 )
13       total += number;
14
15    cout << "Sum is " << total << endl; // display results
16    return 0; // successful termination
17 } // end main
```

```
Sum is 110
```

Vary **number** from **2** to **20** in steps of **2**

Add the current value of **number** to **total**

5.4 Examples Using the `for` Statement (Cont.)

- **Using a comma-separated list of expressions**
  - **Lines 12-13 of Fig. 5.5 can be rewritten as**

```
for ( int number = 2; // initialization
      number <= 20; // loop continuation condition
      total += number, number += 2 ) // total and
                                     // increment
   ;  // empty statement
```

Good Programming Practice 5.6

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 5.7

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

5.4 Examples Using the `for` Statement (Cont.)

- **Standard library function `std::pow`**
  - **Calculates an exponent**
  - **Example**
    - `pow( x, y )`
      - **Calculates the value of `x` raised to the y^{th} power**
  - **Requires header file `<cmath>`**

Outline

fig05_06.cpp

(1 of 2)

```
1  // Fig. 5.6: fig05_06.cpp
2  // Compound interest calculations with for.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::fixed;
7
8  #include <iomanip>
9  using std::setw; // enables program to
10 using std::setprecision;
11
12 #include <cmath> // standard C++ math library
13 using std::pow; // enables program to use function pow
14
15 int main()
16 {
17    double amount; // amount on deposit at end of each year
18    double principal = 1000.0; // initial amount before i
19    double rate = .05; // interest rate
20
21    // display headers
22    cout << "Year" << setw( 21 ) << "Amount on deposit" << endl;
23
24    // set floating-point number format
25    cout << fixed << setprecision( 2 );
26
```

setw stream manipulator will set a field width

standard library function **pow** (in header file **<cmath>**)

C++ treats floating-point values as type **double**

Specify that the next value output should appear in a field width of **21**

```
27    // calculate amount on deposit for each of ten years
28    for ( int year = 1; year <= 10; year++ )
29    {
30       // calculate new amount for specified year
31       amount = principal * pow( 1.0 + rate, year );
32
33       // display the year and the amount
34       cout << setw( 4 ) << year << setw( 21 ) << amount << endl;
35    } // end for
36
37    return 0; // indicate successful termination
38 } // end main
```

Calculate **amount** within **for** statement

Use the **setw** stream manipulator to set field width

```
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
```

Outline

fig05_06.cpp

(2 of 2)

Common Programming Error 5.7

In general, 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 5.8

Do not use variables of type `float` or `double` to perform monetary calculations. The imprecision of floating-point numbers can cause errors that result in incorrect mone-tary values. In the Exercises, we explore the use of integers to perform monetary calculations. [Note: Some third-party vendors sell C++ class libraries that perform precise monetary calculations. We include several URLs in Appendix I.]

5.4 Examples Using the `for` Statement (Cont.)

- **Formatting numeric output**
  - **Stream manipulator `setw`**
    - **Sets field width**
      - **Right justified by default**
        - **Stream manipulator `left` to left-justify**
        - **Stream manipulator `right` to right-justify**
    - **Applies only to the next output value**
  - **Stream manipulators `fixed` and `setprecision`**
    - **Sticky settings**
      - **Remain in effect until they are changed**

Performance Tip 5.1

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 5.2

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

5.5 do…while Repetition Statement

- do…while statement
  - Similar to while statement
  - Tests loop-continuation after performing body of loop
    - Loop body always executes at least once

Good Programming Practice 5.9

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

Outline

fig05_07.cpp

(1 of 1)

```
1  // Fig. 5.7: fig05_07.cpp
2  // do...while repetition statement.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int counter = 1; // initialize counter
10
11    do
12    {
13       cout << counter << " "; // display counter
14       counter++; // increment counter
15    } while ( counter <= 10 ); // end do...while
16
17    cout << endl; // output a newline
18    return 0; // indicate successful termination
19 } // end main
```

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

Declare and initialize control variable **counter**

do…while loop displays **counter**'s value before testing for **counter**'s final value

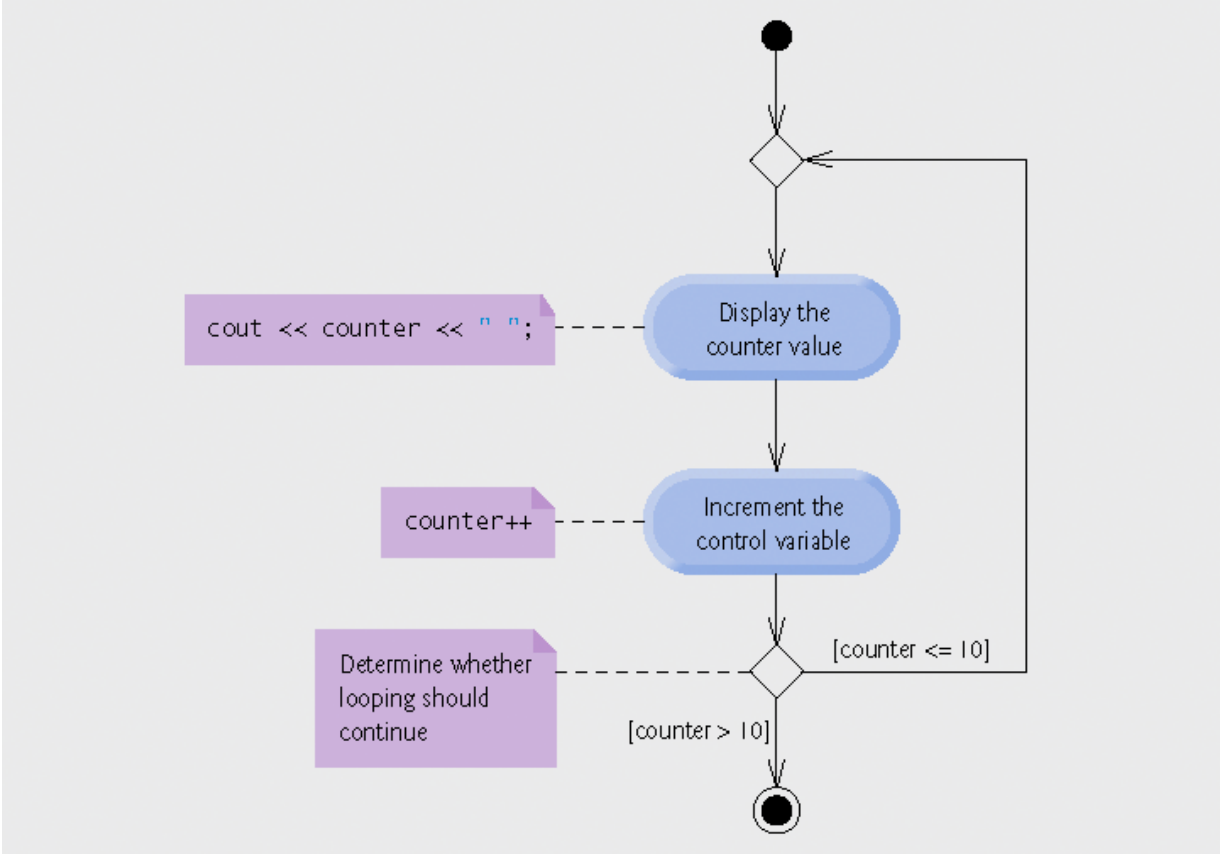


Fig. 5.8 | UML activity diagram for the `do...while` repetition statement of Fig. 5.7.

5.6 `switch` Multiple-Selection Statement

- **`switch` statement**
  - **Used for multiple selections**
  - **Tests a variable or expression**
    - **Compared against constant integral expressions to decide on action to take**
      - **Any combination of character constants and integer constants that evaluates to a constant integer value**

Outline

fig05_09.cpp

(1 of 1)

```
1  // Fig. 5.9: GradeBook.h
2  // Definition of class GradeBook that counts A, B, C, D and F grades.
3  // Member functions are defined in GradeBook.cpp
4
5  #include <string> // program uses C++ standard string class
6  using std::string;
7
8  // GradeBook class definition
9  class GradeBook
10 {
11 public:
12    GradeBook( string ); // constructor initializes course name
13    void setCourseName( string ); // function to set the course name
14    string getCourseName(); // function to retrieve the course name
15    void displayMessage(); // display a welcome message
16    void inputGrades(); // input arbitrary number of grades from user
17    void displayGradeReport(); // display a report based on the grades
18 private:
19    string courseName; // course name for this GradeBook
20    int aCount; // count of A grades
21    int bCount; // count of B grades
22    int cCount; // count of C grades
23    int dCount; // count of D grades
24    int fCount; // count of F grades
25 }; // end class GradeBook
```

Counter variable for each grade category

Outline

fig05_10.cpp

(1 of 5)

```
1  // Fig. 5.10: GradeBook.cpp
2  // Member-function definitions for class GradeBook that
3  // uses a switch statement to count A, B, C, D and F grades.
4  #include <iostream>
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  #include "GradeBook.h" // include definition of class GradeBook
10
11 // constructor initializes courseName with string supplied as argument;
12 // initializes counter data members to 0
13 GradeBook::GradeBook( string name )
14 {
15    setCourseName( name ); // validate and store courseName
16    aCount = 0; // initialize count of A grades to 0
17    bCount = 0; // initialize count of B grades to 0
18    cCount = 0; // initialize count of C grades to 0
19    dCount = 0; // initialize count of D grades to 0
20    fCount = 0; // initialize count of F grades to 0
21 } // end GradeBook constructor
22
```

Initialize each counter variable to **0**

```
23 // function to set the course name; limits name to 25 or fewer characters
24 void GradeBook::setCourseName( string name )
25 {
26    if ( name.length() <= 25 ) // if name has 25 or fewer characters
27       courseName = name; // store the course name in the object
28    else // if name is longer than 25 characters
29    { // set courseName to first 25 characters of parameter name
30       courseName = name.substr( 0, 25 ); // select first 25 characters
31       cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
32          << "Limiting courseName to first 25 characters.\n" << endl;
33    } // end if...else
34 } // end function setCourseName
35
36 // function to retrieve the course name
37 string GradeBook::getCourseName()
38 {
39    return courseName;
40 } // end function getCourseName
41
42 // display a welcome message to the GradeBook user
43 void GradeBook::displayMessage()
44 {
45    // this statement calls getCourseName to get the
46    // name of the course this GradeBook represents
47    cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
48       << endl;
49 } // end function displayMessage
50
```

Outline

fig05_10.cpp

(2 of 5)

```
51 // input arbitrary number of grades from user; update grade counter
52 void GradeBook::inputGrades()
53 {
54    int grade; // grade entered by user
55
56    cout << "Enter the letter grades." << endl
57       << "Enter the EOF character to end input." << endl;
58
59    // loop until user types end-of-file key sequence
60    while ( ( grade = cin.get() ) != EOF )
61    {
62       // determine which grade was entered
63       switch ( grade ) // switch statement nested in while
64       {
65          case 'A': // grade was uppercase A
66          case 'a': // or lowercase a
67             aCount++; // increment aCount
68             break; // necessary to exit switch
69
70          case 'B': // grade was uppercase B
71          case 'b': // or lowercase b
72             bCount++; // increment bCount
73             break; // exit switch
74
75          case 'C': // grade was uppercase C
76          case 'c': // or lowercase c
77             cCount++; // increment cCount
78             break; // exit switch
79
```

Loop condition uses function **`cin.get`** to determine whether there is more data to input

`switch` statement determines which **`case`** label to execute, depending on controlling expression

`grade` is the controlling expression

`case` labels for a grade of A

`break` statement transfers control to after the end of the **`switch`** statement

Outline

`fig05_10.cpp`

(3 of 5)

Outline

fig05_10.cpp

```
80          case 'D': // grade was uppercase D
81          case 'd': // or lowercase d
82             dCount++; // increment dCount
83             break; // exit switch
84
85          case 'F': // grade was uppercase F
86          case 'f': // or lowercase f
87             fCount++; // increment fCount
88             break; // exit switch
89
90          case '\n': // ignore newlines,
91          case '\t': // tabs,
92          case ' ': // and spaces in input
93             break; // exit switch
94
95          default: // catch all other characters
96             cout << "Incorrect letter grade entered."
97                << " Enter a new grade." << endl;
98             break; // optional; will exit switch anyway
99       } // end switch
100   } // end while
101} // end function inputGrades
```

Ignore whitespace characters, do not display an error message

default case for an invalid letter grade

```
102
103// display a report based on the grades entered by user
104void GradeBook::displayGradeReport()
105{
106   // output summary of results
107   cout << "\n\nNumber of students who received each letter grade:"
108      << "\nA: " << aCount // display number of A grades
109      << "\nB: " << bCount // display number of B grades
110      << "\nC: " << cCount // display number of C grades
111      << "\nD: " << dCount // display number of D grades
112      << "\nF: " << fCount // display number of F grades
113      << endl;
114} // end function displayGradeReport
```

Outline

fig05_01.cpp

(5 of 5)

5.6 `switch` Multiple-Selection Statement (Cont.)

- **Reading character input**
  - **Function `cin.get()`**
    - **Reads one character from the keyboard**
  - **Integer value of a character**
    - `static_cast< int >(` *character* `)`
  - **ASCII character set**
    - **Table of characters and their decimal equivalents**
  - **EOF**
    - **<*ctrl*> *d* in UNIX/Linux**
    - **<*ctrl*> *z* in Windows**

Portability Tip 5.2

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

Portability Tip 5.3

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 (but not necessarily –1), so EOF could have different values on different systems.

5.6 `switch` Multiple-Selection Statement (Cont.)

- **`switch` statement**
  - **Controlling expression**
    - **Expression in parentheses after keyword `switch`**
  - **`case` labels**
    - **Compared with the controlling expression**
    - **Statements following the matching `case` label are executed**
      - **Braces are not necessary around multiple statements in a `case` label**
      - **A `break` statements causes execution to proceed with the first statement after the `switch`**
        - **Without a `break` statement, execution will fall through to the next `case` label**

5.6 `switch` Multiple-Selection Statement (Cont.)

- **`switch` statement (Cont.)**
  - **`default` case**
    - **Executes if no matching case label is found**
    - **Is optional**
      - **If no match and no `default` case**
        - **Control simply continues after the `switch`**

Common Programming Error 5.8

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

Common Programming Error 5.9

Omitting the space between the word `case` and the integral value being tested in a `switch` statement can cause a logic error. For example, writing `case3:` instead of writing `case 3:` simply creates an unused label. We will say more about this in Appendix E, C Legacy Code Topics. In this situation, the `switch` statement will not perform the appropriate actions when the `switch`'s controlling expression has a value of 3.

Good Programming Practice 5.10

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 the programmer 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 is common practice to place the `default` clause last.

Good Programming Practice 5.11

In a `switch` statement that lists the `default` clause last, the `default` clause does not require a `break` statement. Some programmers include this `break` for clarity and for symmetry with other cases.

Common Programming Error 5.10

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

```
1  // Fig. 5.11: fig05_11.cpp
2  // Create GradeBook object, input grades and display grade report.
3
4  #include "GradeBook.h" // include definition of class GradeBook
5
6  int main()
7  {
8     // create GradeBook object
9     GradeBook myGradeBook( "CS101 C++ Programming" );
10
11    myGradeBook.displayMessage(); // display welcome message
12    myGradeBook.inputGrades(); // read grades from user
13    myGradeBook.displayGradeReport(); // display report based on grades
14    return 0; // indicate successful termination
15 } // end main
```

Outline

fig05_11.cpp

(1 of 2)

Outline

fig05_11.cpp

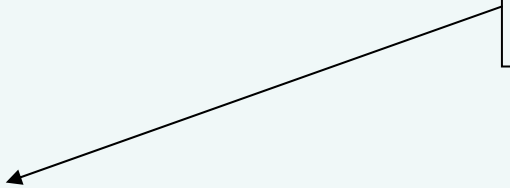
(2 of 2)

```
Welcome to the grade book for
CS101 C++ Programming!

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
```

An error message is shown in response to an invalid grade



Common Programming Error 5.11

Specifying an expression including variables (e.g., `a + b`) in a `switch` statement's `case` label is a syntax error.

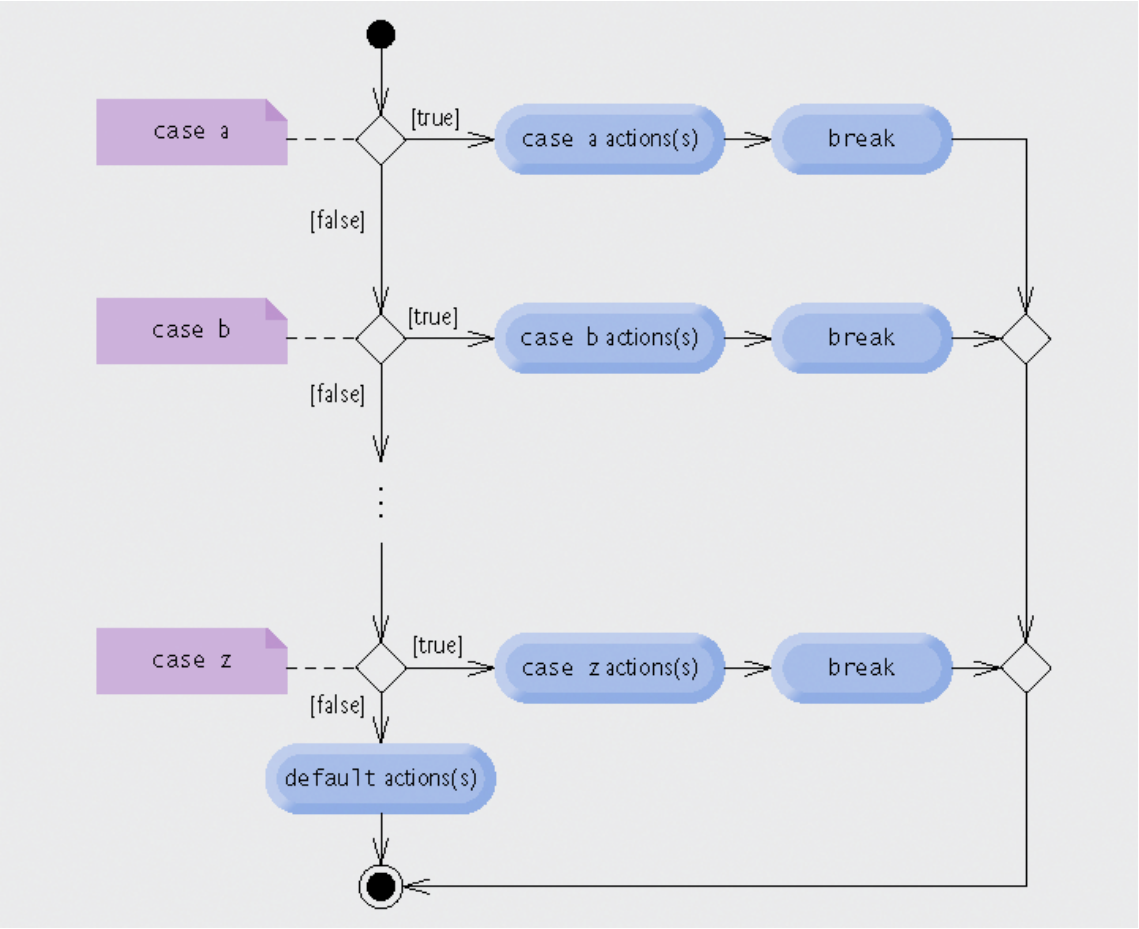


Fig. 5.12 | `switch` multiple-selection statement UML activity diagram with `break` statements.

Common Programming Error 5.12

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:`.

5.6 `switch` Multiple-Selection Statement (Cont.)

- **Integer data types**
  - `short`
    - **Abbreviation of `short int`**
    - **Minimum range is -32,768 to 32,767**
  - `long`
    - **Abbreviation of `long int`**
    - **Minimum range is -2,147,483,648 to 2,147,483,647**
  - `int`
    - **Equivalent to either `short` or `long` on most computers**
  - `char`
    - **Can be used to represent small integers**

Portability Tip 5.4

Because `ints` 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 would like to run the program on several different computer systems.

Performance Tip 5.3

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

Performance Tip 5.4

Using smaller integer sizes can result in a slower program if the machine's instructions for manipulating them are not as efficient as those for the natural-size integers, i.e., integers whose size equals the machine's word size (e.g., 32 bits on a 32-bit machine, 64 bits on a 64-bit machine). Always test proposed efficiency "upgrades" to be sure they really improve performance.

5.7 break and continue Statements

- break/continue statements
  - Alter flow of control
- break statement
  - Causes immediate exit from control structure
  - Used in while, for, do…while or switch statements
- continue statement
  - Skips remaining statements in loop body
    - Proceeds to increment and condition test in for loops
    - Proceeds to condition test in while/do…while loops
  - Then performs next iteration (if not terminating)
  - Used in while, for or do…while statements

Outline

fig05_13.cpp

(1 of 1)

```
1  // Fig. 5.13: fig05_13.cpp
2  // break statement exiting a for statement.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int count; // control variable also used after loop terminates
10
11    for ( count = 1; count <= 10; count++ ) // loop 10 times
12    {
13       if ( count == 5 )
14          break; // break loop only if x is 5
15
16       cout << count << " ";
17    } // end for
18
19    cout << "\nBroke out of loop at count = " << count << endl;
20    return 0; // indicate successful termination
21 } // end main
```

Loop **10** times

Exit **for** statement (with a **break**) when **count** equals **5**

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

Outline

fig05_14.cpp

(1 of 1)

```
1  // Fig. 5.14: fig05_14.cpp
2  // continue statement terminating an iteration of a for statement.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     for ( int count = 1; count <= 10; count++ ) // loop 10 times
10    {
11       if ( count == 5 ) // if count is 5,
12          continue;      // skip remaining code in loop
13
14       cout << count << " ";
15    } // end for
16
17    cout << "\nUsed continue to skip printing 5" << endl;
18    return 0; // indicate successful termination
19 } // end main
```

Loop **10** times

Skip line **14** and proceed to line **9** when **count** equals **5**

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

Good Programming Practice 5.12

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.

Performance Tip 5.5

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

Software Engineering Observation 5.2

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 rule of thumb: First, make your code simple and correct; then make it fast and small, but only if necessary.

5.8 Logical Operators

- **Logical operators**
  - **Allows for more complex conditions**
    - **Combines simple conditions into complex conditions**
- **C++ logical operators**
  - **&& (logical AND)**
  - **|| (logical OR)**
  - **! (logical NOT)**

5.8 Logical Operators (Cont.)

- **Logical AND (&&) Operator**
  - **Consider the following `if` statement**

```
if ( gender == 1 && age >= 65 )
   seniorFemales++;
```

  - **Combined condition is `true`**
    - **If and only if both simple conditions are `true`**
  - **Combined condition is `false`**
    - **If either or both of the simple conditions are `false`**

Common Programming Error 5.13

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++.

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

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

5.8 Logical Operators (Cont.)

- **Logical OR (||) Operator**
  - **Consider the following `if` statement**

```
if ( ( semesterAverage >= 90 ) || ( finalExam >= 90 )
   cout << "Student grade is A" << endl;
```

  - **Combined condition is `true`**
    - **If either or both of the simple conditions are `true`**
  - **Combined condition is `false`**
    - **If both of the simple conditions are `false`**

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

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

5.8 Logical Operators (Cont.)

- **Short-Circuit Evaluation of Complex Conditions**
  - **Parts of an expression containing && or || operators are evaluated only until it is known whether the condition is true or false**
  - **Example**
    - `( gender == 1 ) && ( age >= 65 )`
      - **Stops immediately if `gender` is not equal to `1`**
        - **Since the left-side is `false`, the entire expression must be `false`**

Performance Tip 5.6

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.

5.8 Logical Operators (Cont.)

- **Logical Negation (`!`) Operator**
  - **Unary operator**
  - **Returns `true` when its operand is `false`, and vice versa**
  - **Example**
    - ```
      if ( !( grade == sentinelValue ) )
         cout << "The next grade is " << grade << endl;
      ```
      is equivalent to:
      ```
      if ( grade != sentinelValue )
         cout << "The next grade is " << grade << endl;
      ```
- **Stream manipulator `boolalpha`**
  - **Display `bool` expressions in words, "true" or "false"**


      ```

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

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

```
1  // Fig. 5.18: fig05_18.cpp
2  // Logical operators.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::boolalpha; // causes bool values to print as "true" or "false"
7
8  int main()
9  {
10    // create truth table for && (logical AND) operator
11    cout << boolalpha << "Logical AND (&&)"
12       << "\nfalse && false: " << ( false && false )
13       << "\nfalse && true: " << ( false && true )
14       << "\ntrue && false: " << ( true && false )
15       << "\ntrue && true: " << ( true && true ) << "\n\n";
16
17    // create truth table for || (logical OR) operator
18    cout << "Logical OR (||)"
19       << "\nfalse || false: " << ( false || false )
20       << "\nfalse || true: " << ( false || true )
21       << "\ntrue || false: " << ( true || false )
22       << "\ntrue || true: " << ( true || true ) << "\n\n";
23
24    // create truth table for ! (logical negation) operator
25    cout << "Logical NOT (!)"
26       << "\n!false: " << ( !false )
27       << "\n!true: " << ( !true ) << endl;
28    return 0; // indicate successful termination
29 } // end main
```

Stream manipulator **boolalpha** causes **bool** values to display as the words “true” or “false”

Use **boolalpha** stream manipulator in **cout**

Output logical AND truth table

Output logical OR truth table

Output logical NOT truth table

fig05_18.cpp

(1 of 2)

```
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
```

Outline

fig05_18.cpp

(2 of 2)

Operators						Associativity	Type
`()`						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. 5.19 | Operator precedence and associativity.

5.9 Confusing Equality (==) and Assignment (=) Operators

- **Accidentally swapping the operators == (equality) and = (assignment)**
  - **Common error**
    - **Assignment statements produce a value (the value to be assigned)**
    - **Expressions that have a value can be used for decision**
      - **Zero = `false`, nonzero = `true`**
  - **Does not typically cause syntax errors**
    - **Some compilers issue a warning when = is used in a context normally expected for ==**

5.9 Confusing Equality (==) and Assignment (=) Operators (Cont.)

- **Example**

```
if ( payCode == 4 )
   cout << "You get a bonus!" << endl;
```

  - If `paycode` is 4, bonus is given
- **If == was replaced with =**

```
if ( payCode = 4 )
   cout << "You get a bonus!" << endl;
```

  - `paycode` is set to 4 (no matter what it was before)
  - Condition is `true` (since 4 is non-zero)
    - Bonus given in every case

Common Programming Error 5.14

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

Error-Prevention Tip 5.3

Programmers normally write conditions such as `x == 7` with the variable name on the left and the constant on the right. By reversing these so that the constant is on the left and the variable name is on the right, as in `7 == x`, the programmer who accidentally replaces the `==` operator with `=` will be protected by the compiler. 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.

5.9 Confusing Equality (==) and Assignment (=) Operators (Cont.)

- *Lvalues*
  - Expressions that can appear on left side of equation
  - Can be changed (i.e., variables)
    - `x = 4;`
- *Rvalues*
  - Only appear on right side of equation
  - Constants, such as numbers (i.e. cannot write `4 = x;`)
- *Lvalues* can be used as *rvalues*, but not vice versa

Error-Prevention Tip 5.4

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.

5.10 Structured Programming Summary

- **Structured programming**
  - **Produces programs that are easier to understand, test, debug and modify**
- **Rules for structured programming**
  - **Only use single-entry/single-exit control structures**
  - **Rules (Fig. 5.21)**
    - **Rule 2 is the stacking rule**
    - **Rule 3 is the nesting rule**

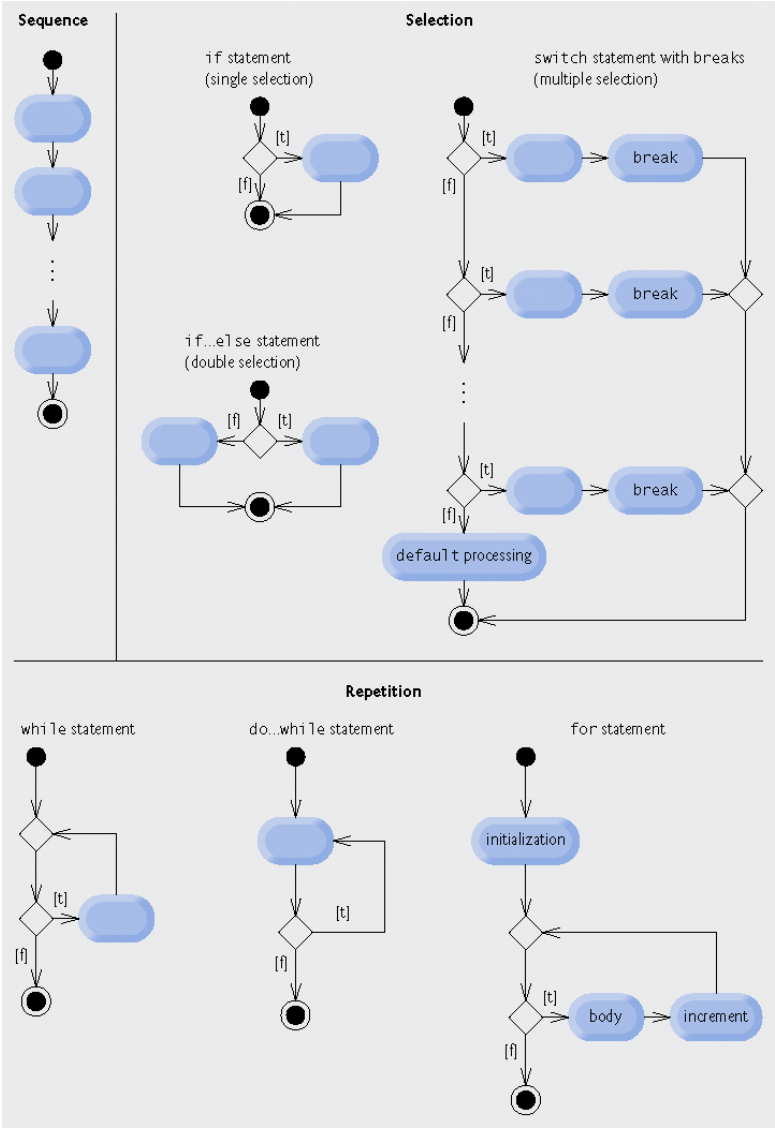


Fig. 5.20 | C++'s single-entry/single-exit sequence, selection and repetition statements.

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, `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.

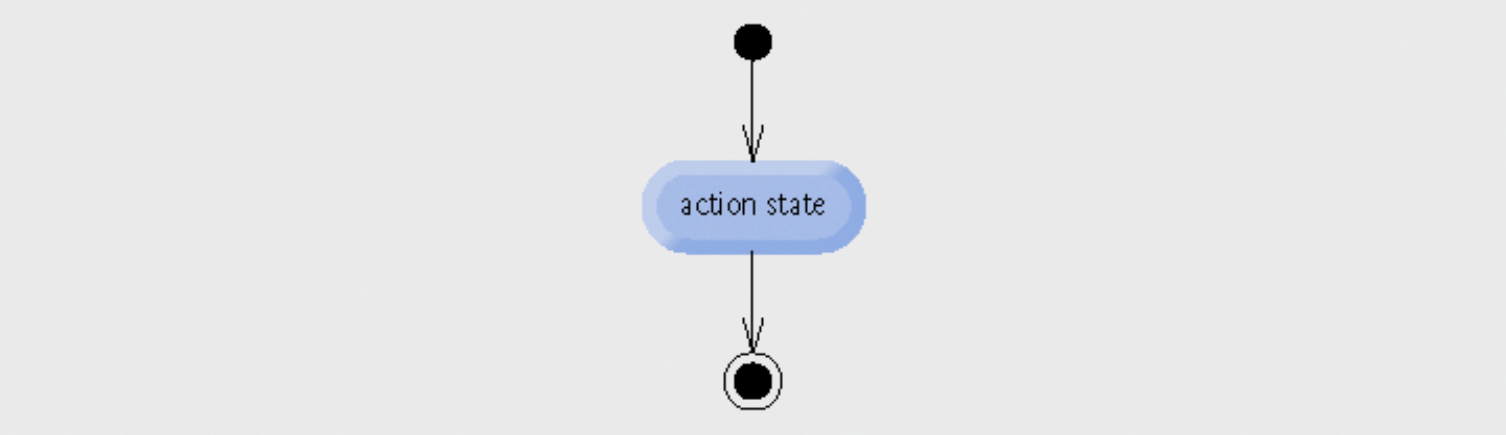


Fig. 5.22 | Simplest activity diagram.

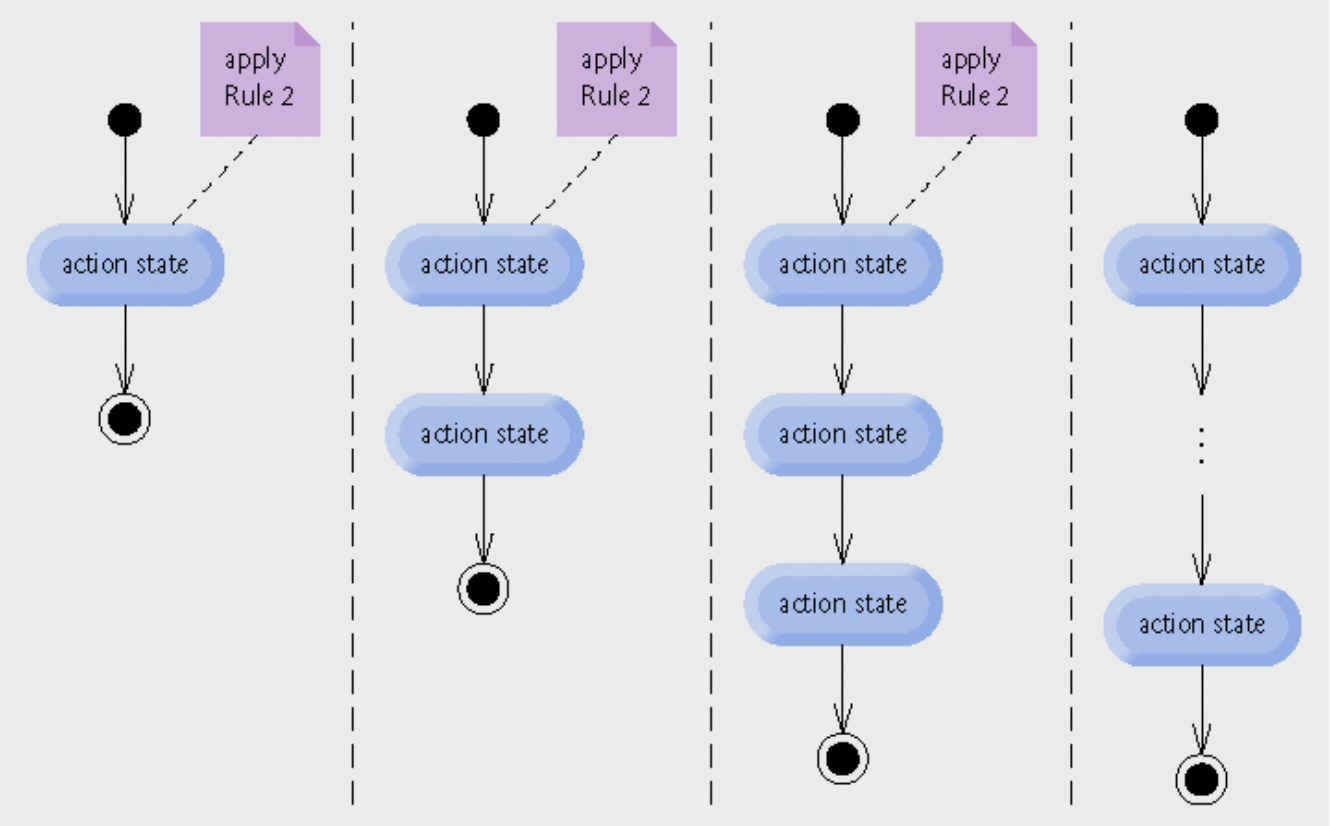


Fig. 5.23 | Repeatedly applying Rule 2 of Fig. 5.21 to the simplest activity diagram.

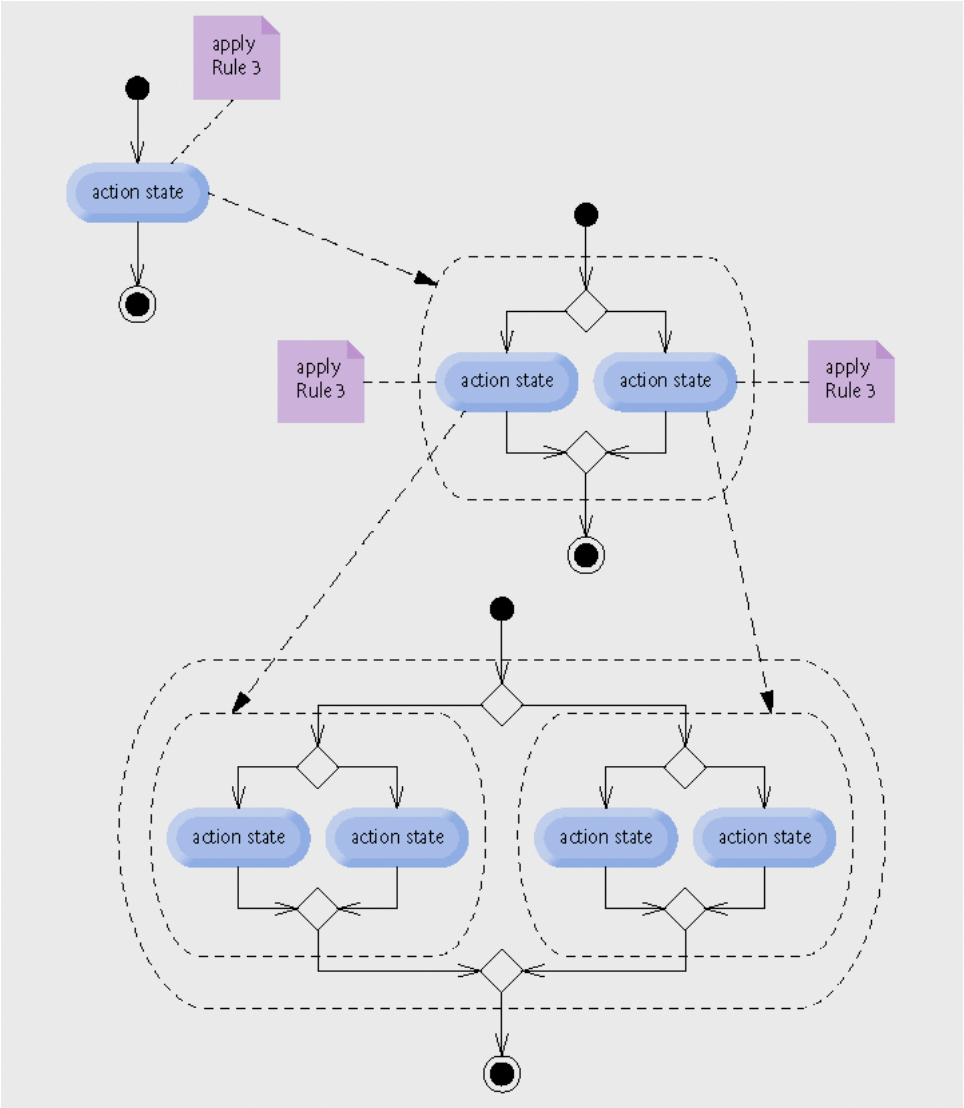


Fig. 5.24 | Applying Rule 3 of Fig. 5.21 to the simplest activity diagram several times.

5.10 Structured Programming Summary (Cont.)

- **Sequence structure**
  - **"built-in" to C++**
- **Selection structure**
  - **`if`, `if…else` and `switch`**
- **Repetition structure**
  - **`while`, `do…while` and `for`**

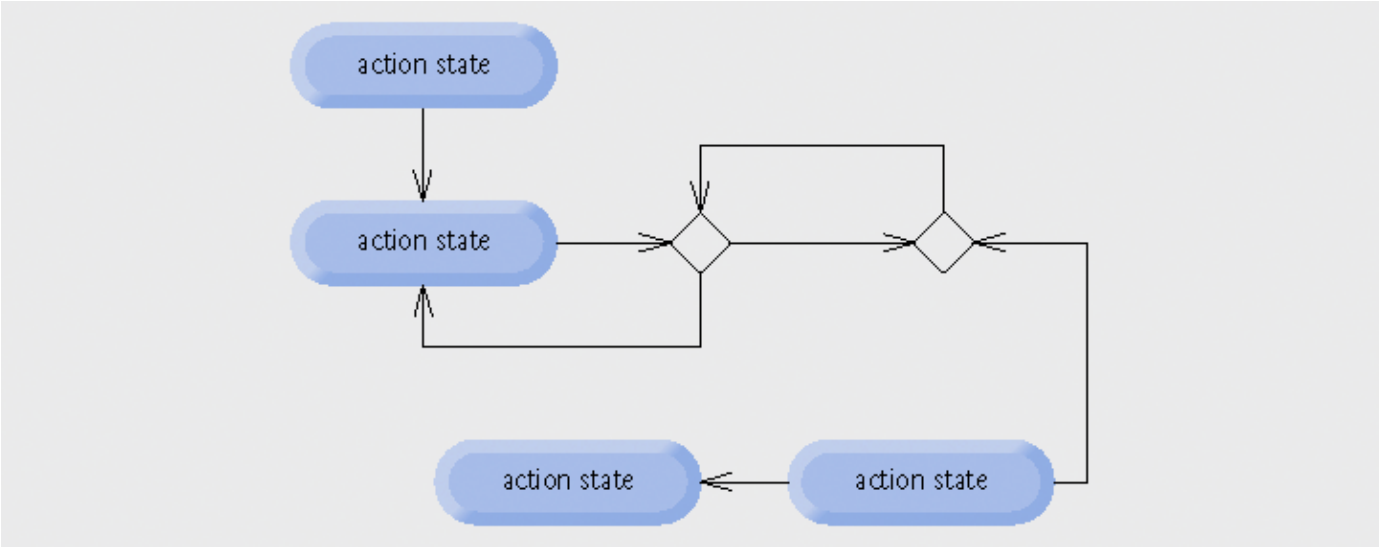


Fig. 5.25 | Activity diagram with illegal syntax.

5.11 (Optional) Software Engineering Case Study: Identifying Object's State and Activities in the ATM System

- **State Machine Diagrams**
  - **Commonly called state diagrams**
  - **Model several states of an object**
    - **Show under what circumstances the object changes state**
    - **Focus on system behavior**
  - **UML representation**
    - **Initial state**
      - **Solid circle**
    - **State**
      - **Rounded rectangle**
    - **Transitions**
      - **Arrows with stick arrowheads**

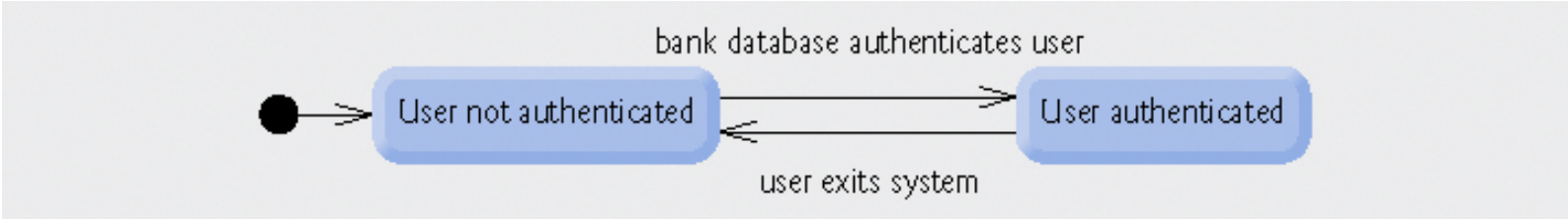


Fig. 5.26 | State diagram for the ATM object.

Software Engineering Observation 5.3

Software designers do not generally create state diagrams showing every possible state and state transition for all attributes—there are simply too many of them. State diagrams typically show only the most important or complex states and state transitions.

5.11 (Optional) Software Engineering Case Study : Identifying Object's State and Activities in the ATM System (Cont.)

- **Activity Diagrams**
  - **Focus on system behavior**
  - **Model an object's workflow during program execution**
    - **Actions the object will perform and in what order**
  - **UML representation**
    - **Initial state**
      - **Solid circle**
    - **Action state**
      - **Rectangle with outward-curving sides**
    - **Action order**
      - **Arrow with a stick arrowhead**
    - **Final state**
      - **Solid circle enclosed in an open circle**

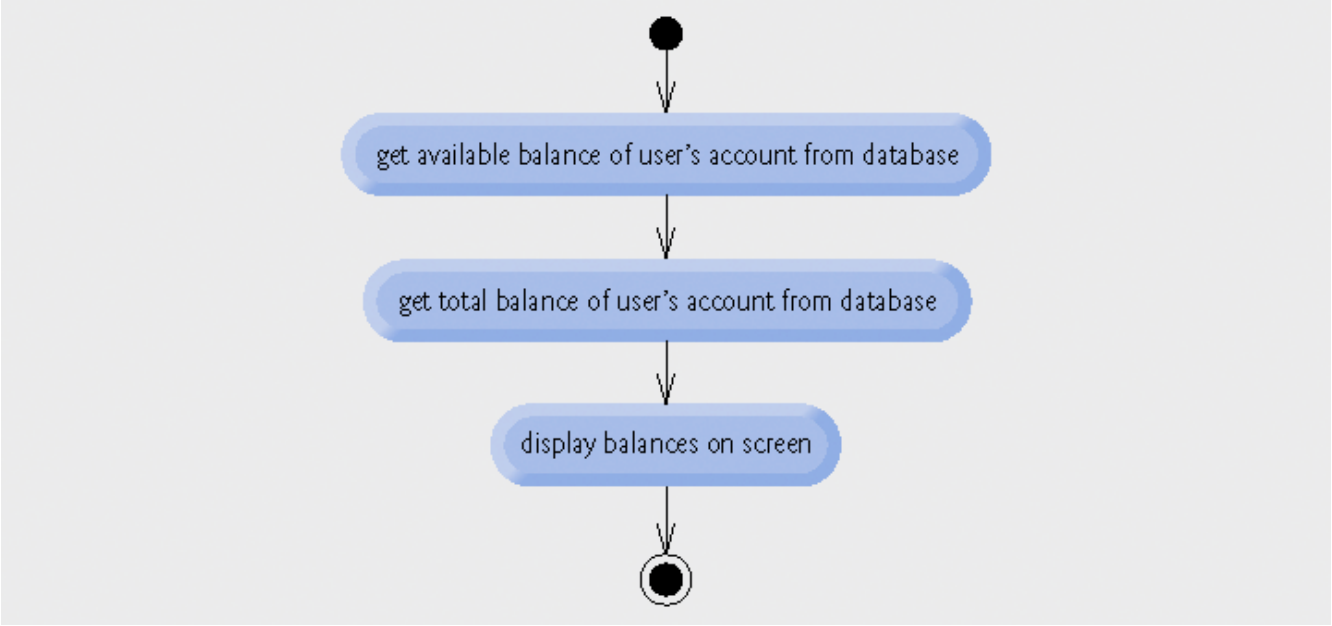


Fig. 5.27 | Activity diagram for a `BalanceInquiry` transaction.

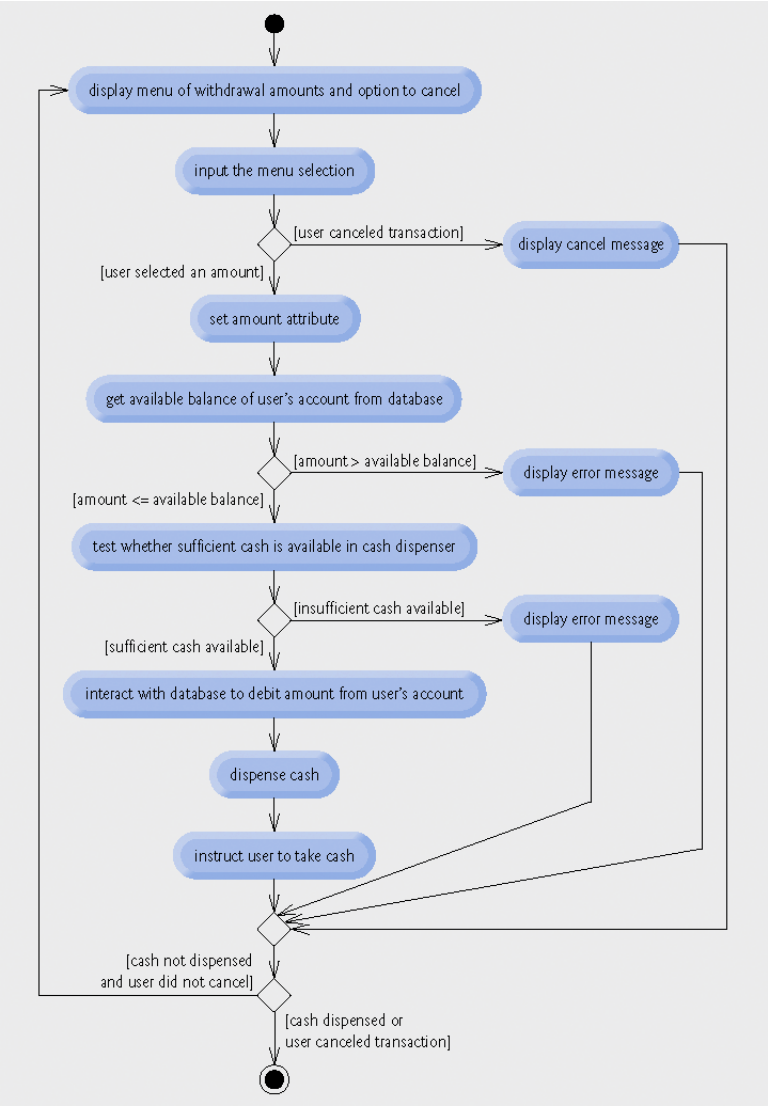


Fig. 5.28 | Activity diagram for a `Withdrawal` transaction.

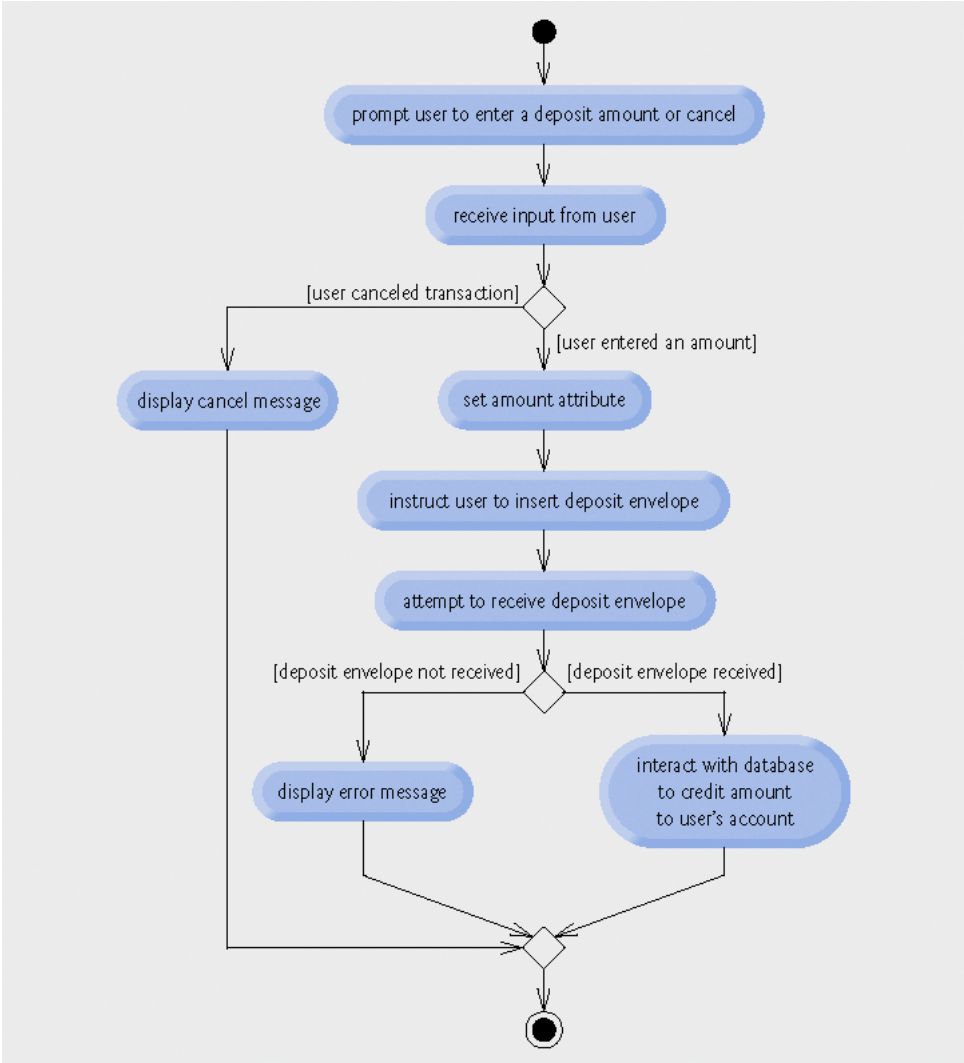


Fig. 5.29 | Activity diagram for a `Deposit` transaction.