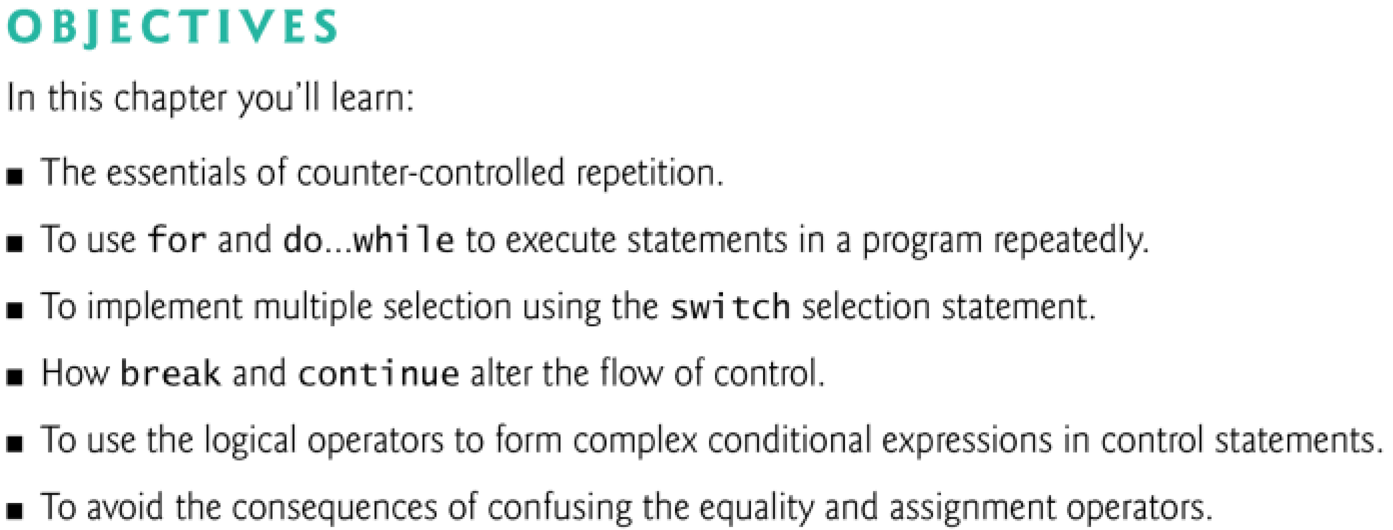
Chapter 5, Control Statements: Part 2

C++ How to Program, 7/e



OBJECTIVES

In this chapter you'll learn:

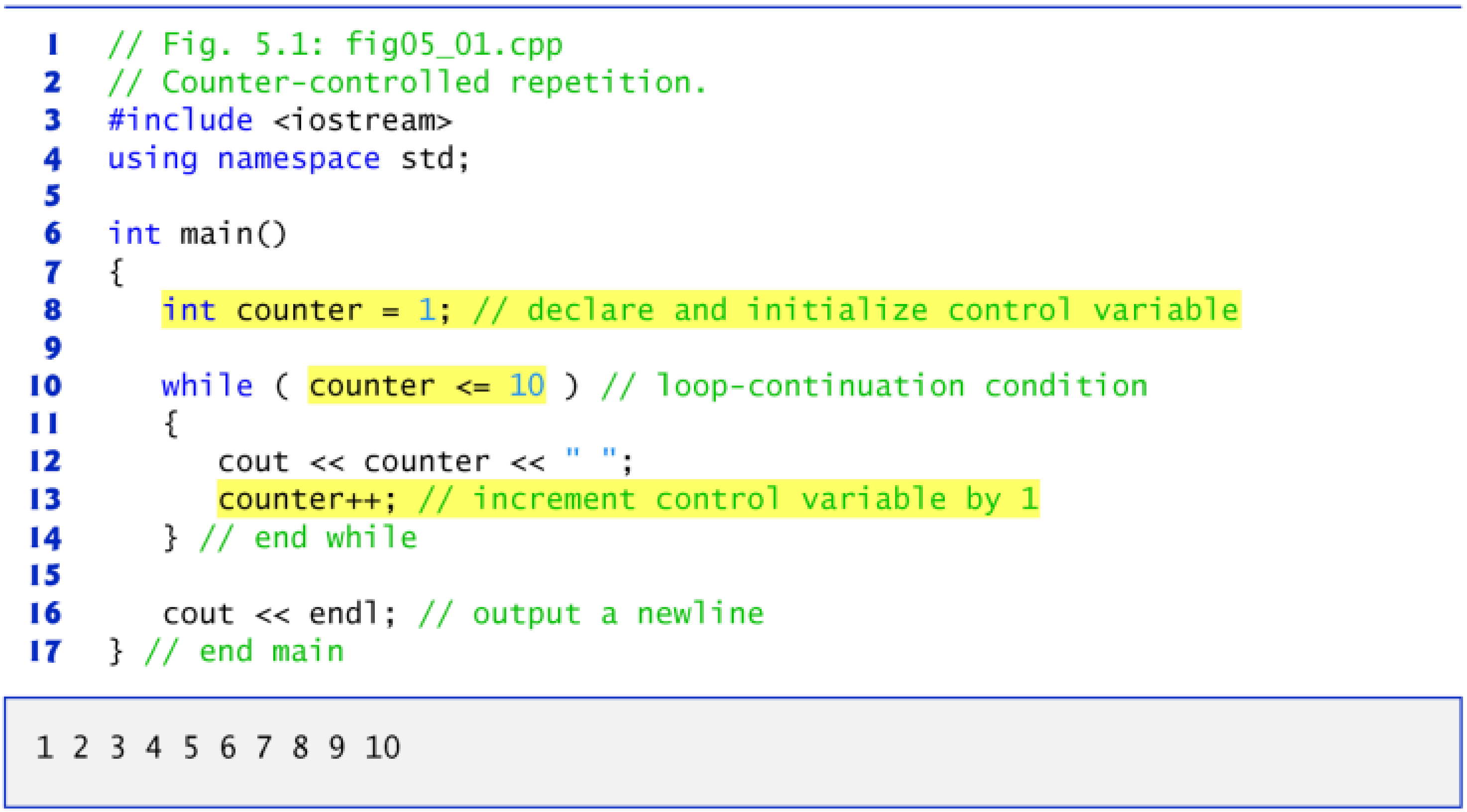
- The essentials of counter-controlled repetition.
- To use `for` and `do…while` to execute statements in a program repeatedly.
- To implement multiple selection using the `switch` selection statement.
- How `break` and `continue` alter the flow of control.
- To use the logical operators to form complex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

5.1 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.

5.2 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.



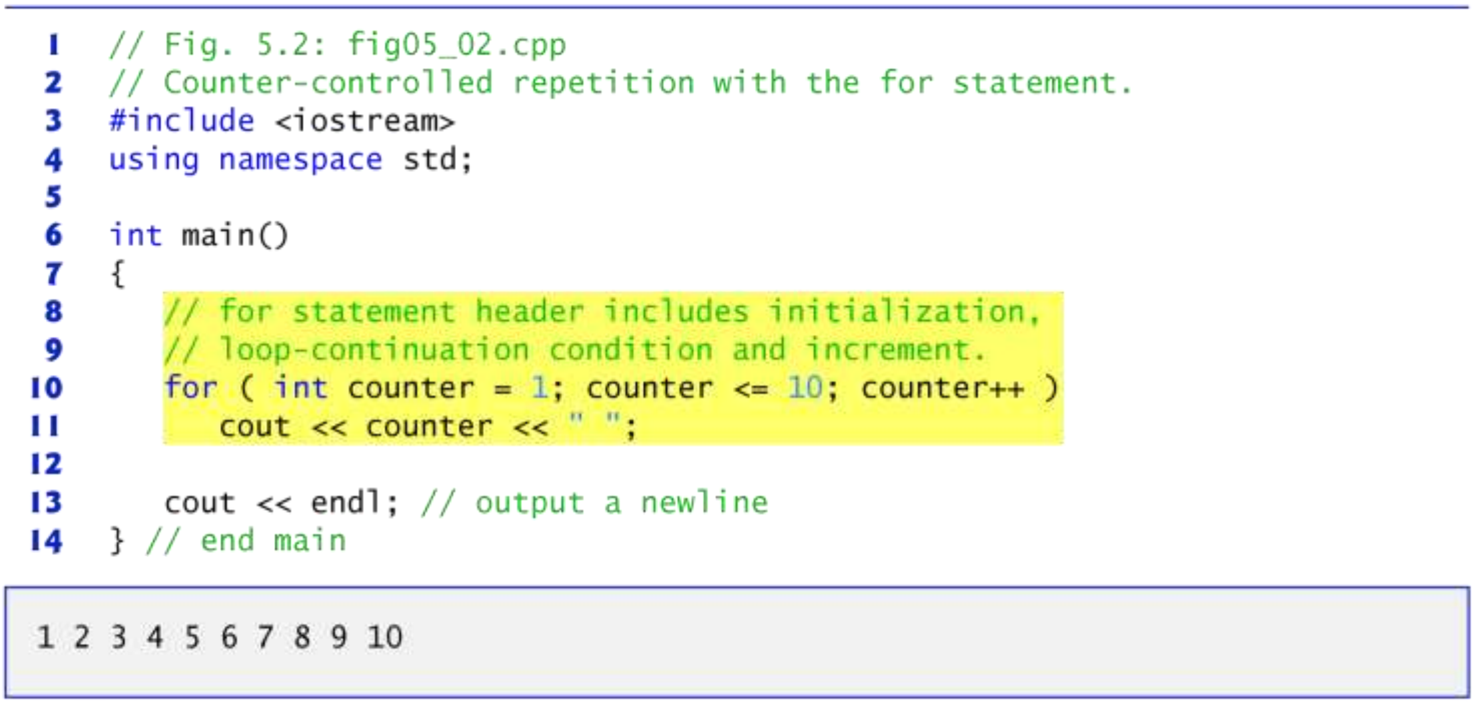
```cpp
// Fig. 5.1: fig05_01.cpp
// Counter-controlled repetition.
#include <iostream>
using namespace std;

int main()
{
   int counter = 1; // declare and initialize control variable

   while ( counter <= 10 ) // loop-continuation condition
   {
      cout << counter << " ";
      counter++; // increment control variable by 1
   } // end while

   cout << endl; // output a newline
} // end main
```

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

Fig. 5.1 | Counter-controlled repetition.

5.3 for Repetition Statement

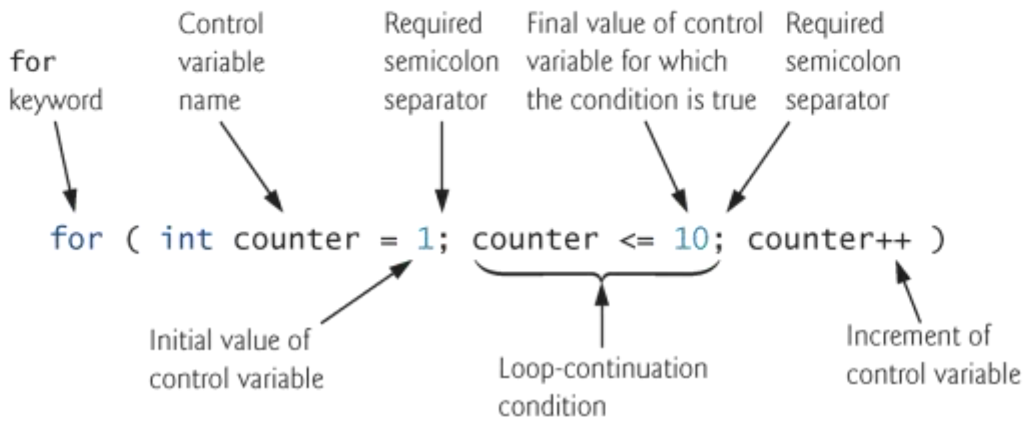
- The `for` repetition statement specifies the counter-controlled repetition details in a single line of code.
- The initialization occurs once when the loop is encountered.
- The condition is tested next and each time the body completes.
- The body executes if the condition is true.
- The increment 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.



```
// Fig. 5.2: fig05_02.cpp
// Counter-controlled repetition with the for statement.
#include <iostream>
using namespace std;

int main()
{
   // for statement header includes initialization,
   // loop-continuation condition and increment.
   for ( int counter = 1; counter <= 10; counter++ )
      cout << counter << " ";

   cout << endl; // output a newline
} // end main
```

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

Fig. 5.2 | Counter-controlled repetition with the for statement.



for keyword

Control variable name

Required semicolon separator

Final value of control variable for which the condition is true

Required semicolon separator

```
for ( int counter = 1; counter <= 10; counter++ )
```

Initial value of control variable

Loop-continuation condition

Increment of control variable

Fig. 5.3 | for statement header components.

5.3 for Repetition Statement (cont.)

- The general form of the `for` statement is

  -
    ```
    for ( initialization; loopContinuationCondition; increment )
        statement
    ```
- In most cases, the `for` statement can be represented by an equivalent `while` statement, as follows:

  -
    ```
    initialization;

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

5.3 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 expres-sions evalu-ate from left to right.
  - The lowest precedence of all C++ opera-tors.
- The value and type of a comma-separated list of expressions is the value and type of the rightmost expression.

5.3 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.*

5.3 for Repetition Statement (cont.)

- The increment expression in the `for` statement acts as a stand-alone 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).

5.3 for Repetition Statement (cont.)

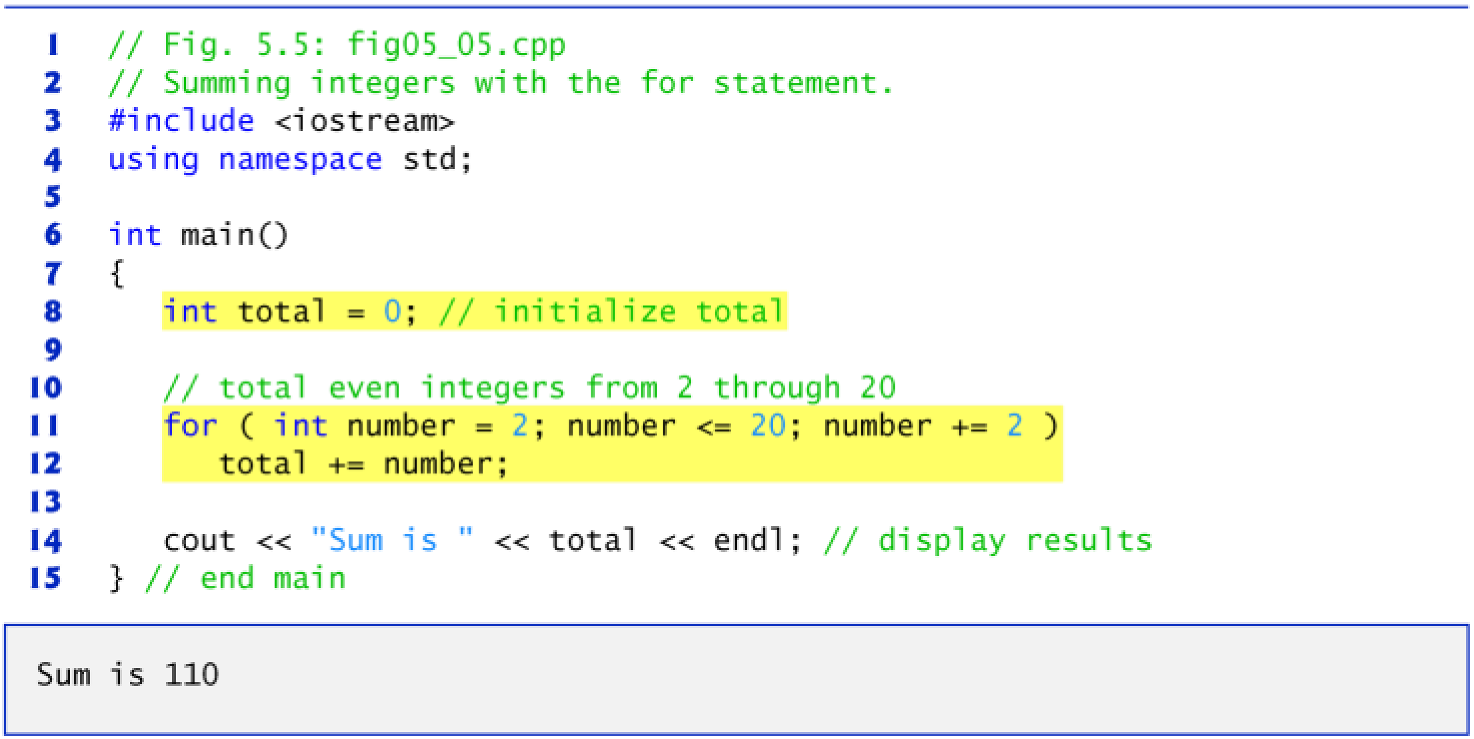
- The initialization, loop-continuation condition and increment expressions of a `for` statement can contain arithmetic expressions.
- The “increment” of a `for` statement can be negative, in which case the loop actually counts downward.
- If the loop-continuation condition is initially false, the body of the `for` statement is not performed.

5.4 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 )`

5.4 Examples Using the for Statement (cont.)

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



```cpp
// Fig. 5.5: fig05_05.cpp
// Summing integers with the for statement.
#include <iostream>
using namespace std;

int main()
{
   int total = 0; // initialize total

   // total even integers from 2 through 20
   for ( int number = 2; number <= 20; number += 2 )
      total += number;

   cout << "Sum is " << total << endl; // display results
} // end main
```

```
Sum is 110
```

Fig. 5.5 | Summing integers with the for statement.

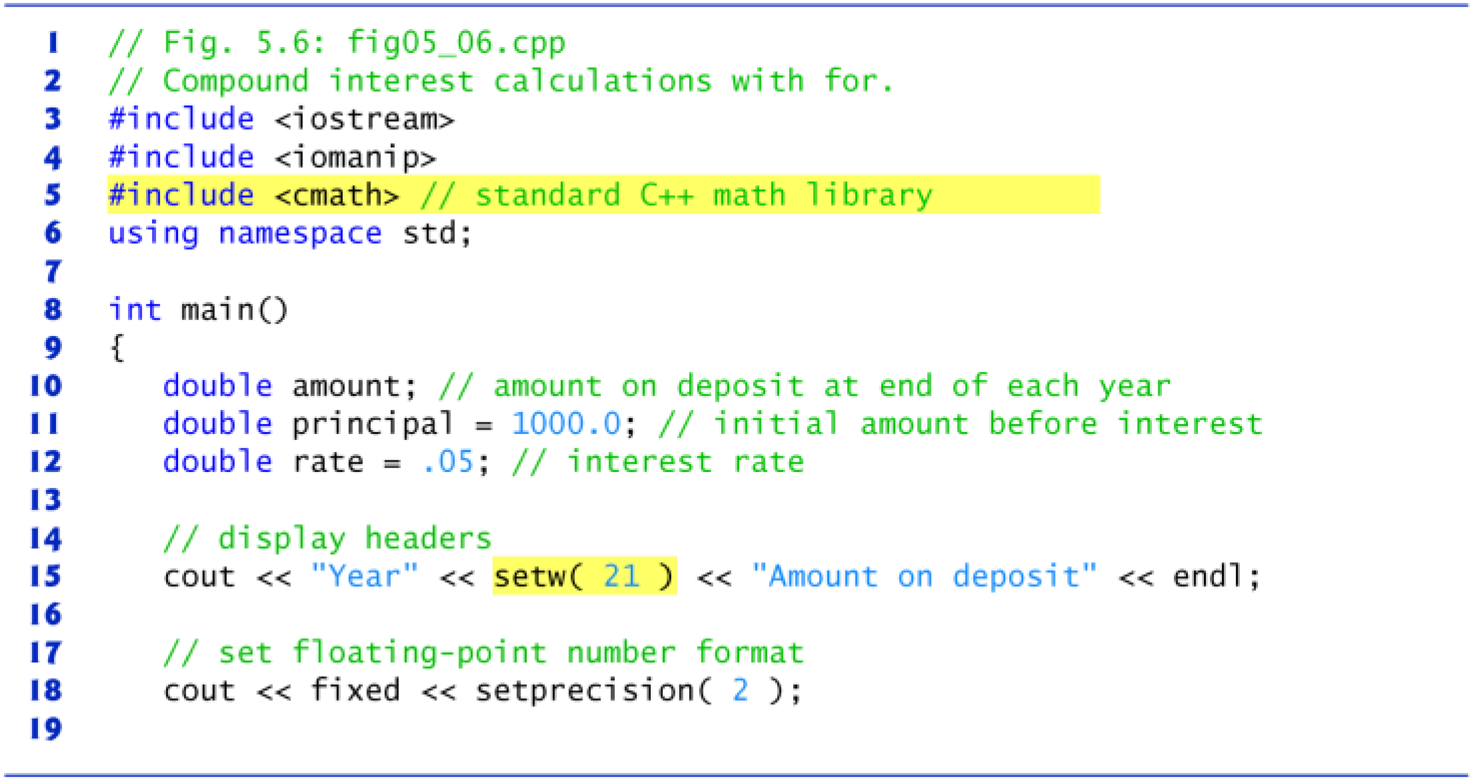
5.4 Examples Using the for Statement (cont.)

- 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:

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

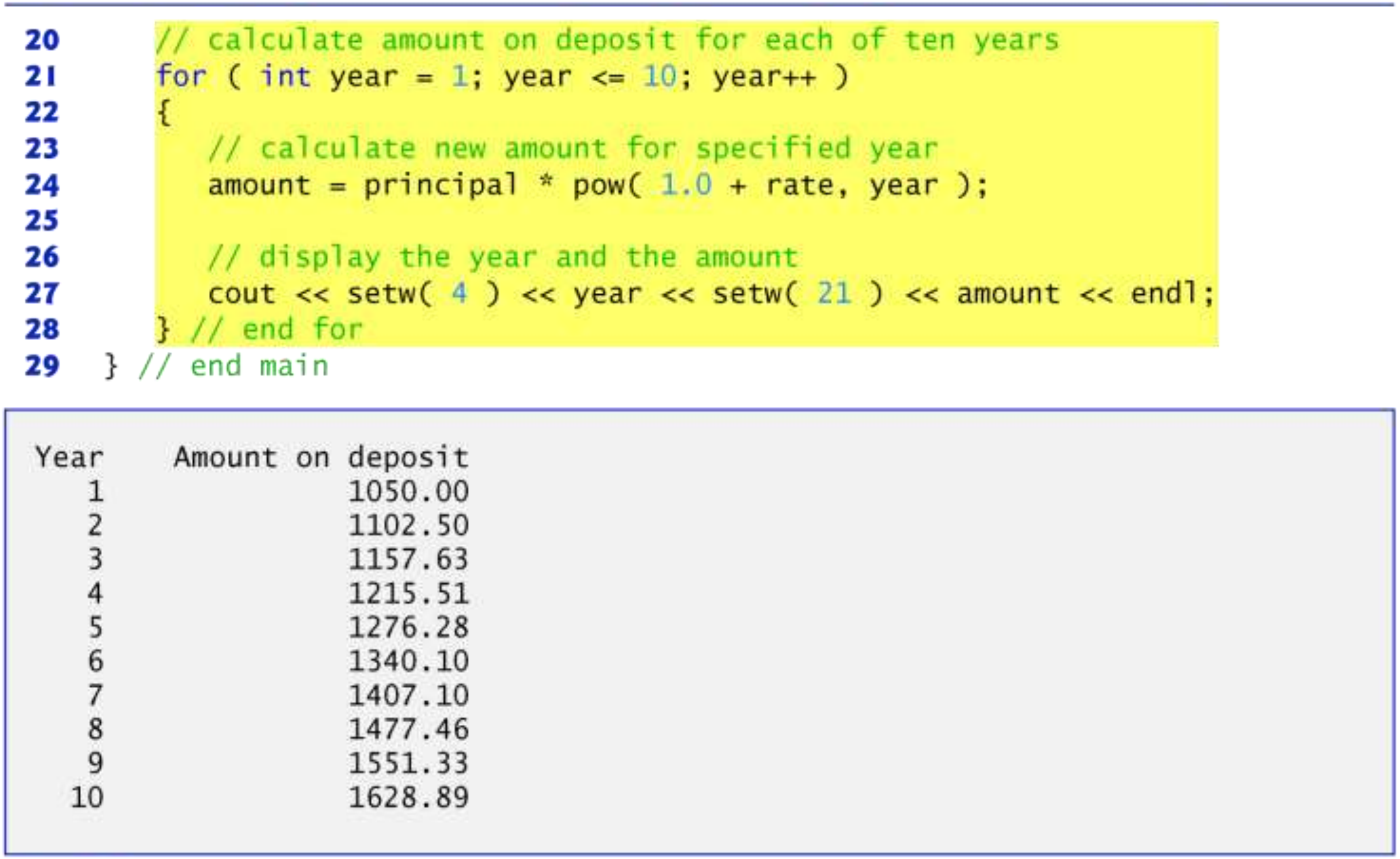
    where

    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 nth year.
  - This problem involves a loop that performs the indicated calculation for each of the 10 years the money remains on deposit.



```cpp
// Fig. 5.6: fig05_06.cpp
// Compound interest calculations with for.
#include <iostream>
#include <iomanip>
#include <cmath> // standard C++ math library
using namespace std;

int main()
{
   double amount; // amount on deposit at end of each year
   double principal = 1000.0; // initial amount before interest
   double rate = .05; // interest rate

   // display headers
   cout << "Year" << setw( 21 ) << "Amount on deposit" << endl;

   // set floating-point number format
   cout << fixed << setprecision( 2 );

```

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



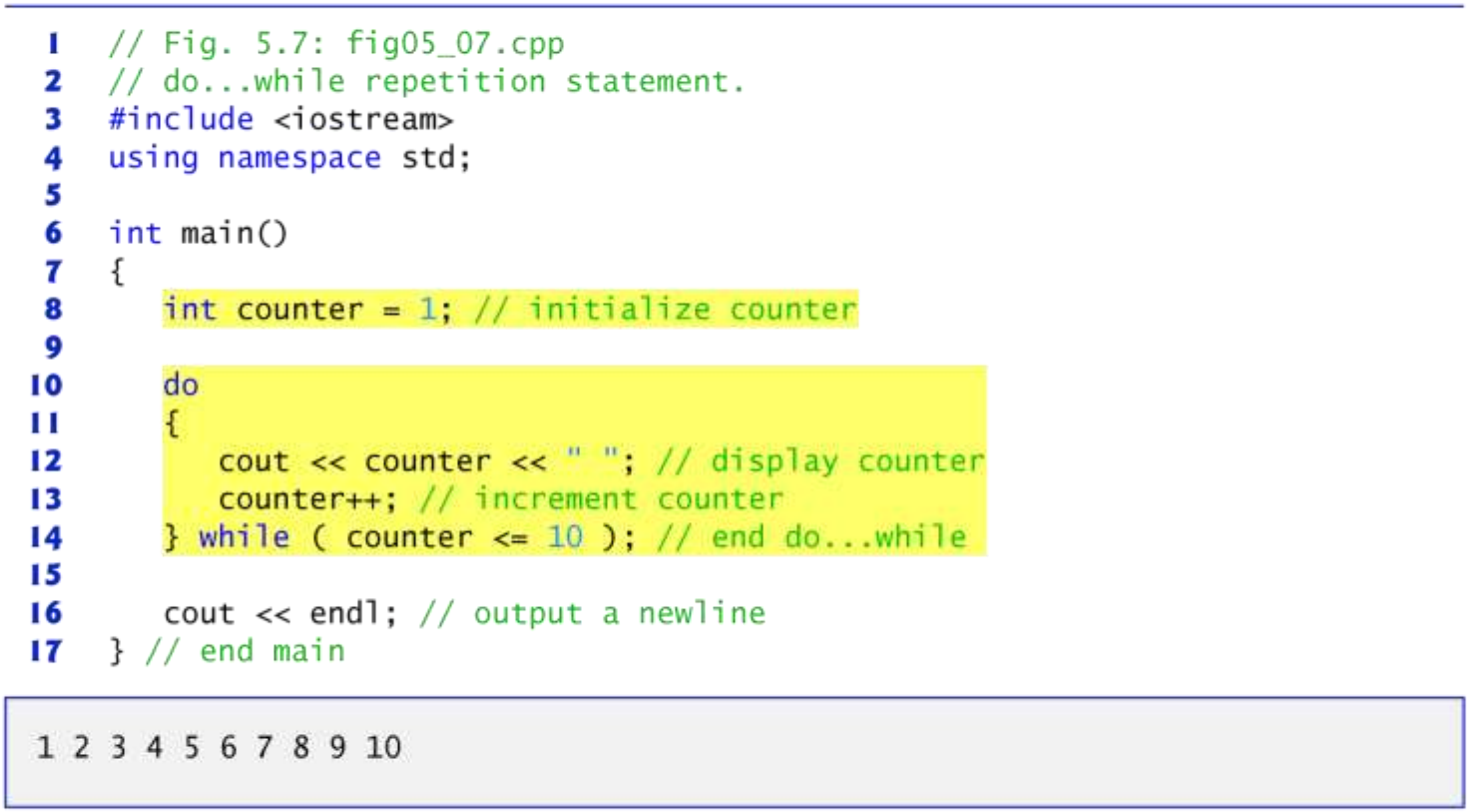
```cpp
   // calculate amount on deposit for each of ten years
   for ( int year = 1; year <= 10; year++ )
   {
      // calculate new amount for specified year
      amount = principal * pow( 1.0 + rate, year );

      // display the year and the amount
      cout << setw( 4 ) << year << setw( 21 ) << amount << endl;
   } // end for
} // 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. 5.6 | Compound interest calculations with for. (Part 2 of 2.)

5.5 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.



```cpp
// Fig. 5.7: fig05_07.cpp
// do...while repetition statement.
#include <iostream>
using namespace std;

int main()
{
   int counter = 1; // initialize counter

   do
   {
      cout << counter << " "; // display counter
      counter++; // increment counter
   } while ( counter <= 10 ); // end do...while

   cout << endl; // output a newline
} // end main
```

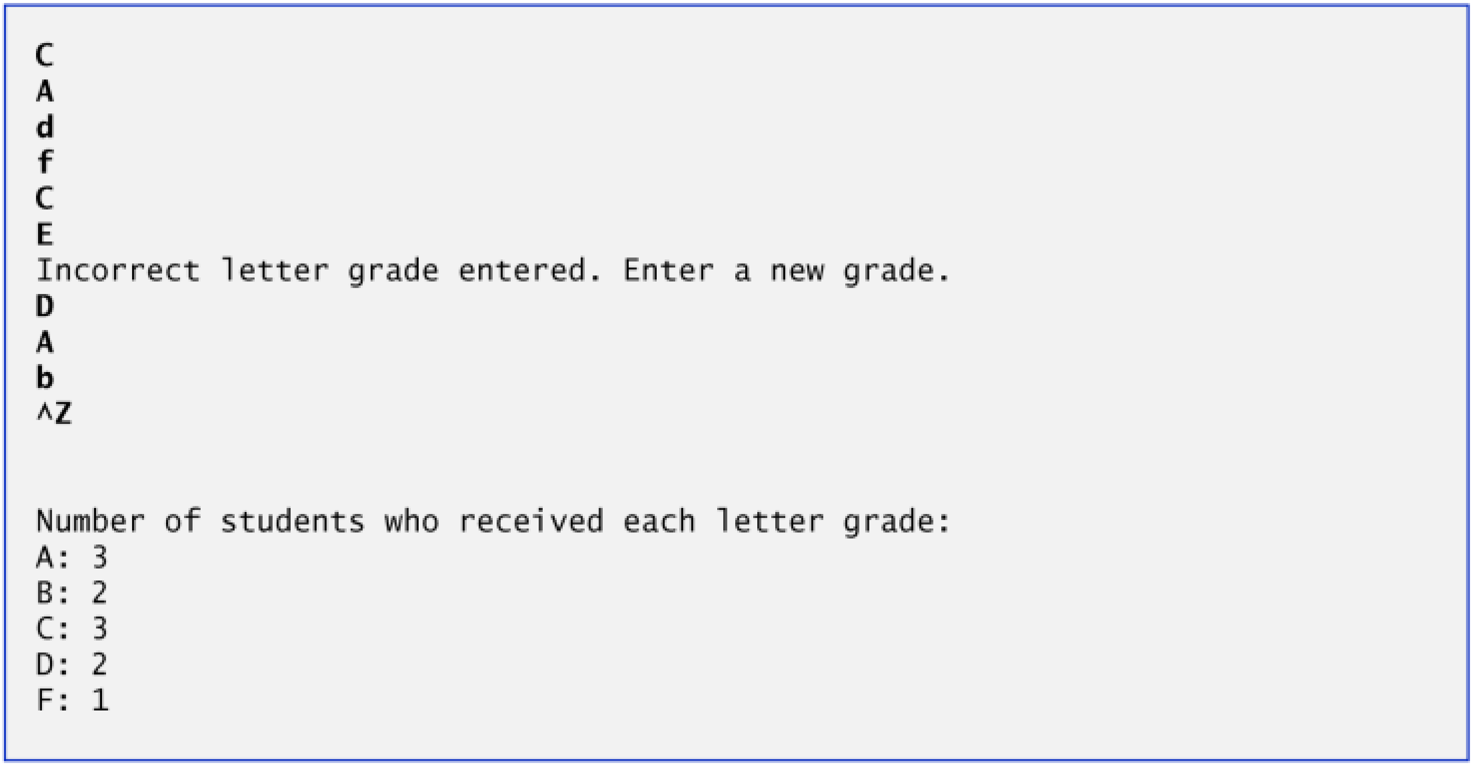
```
1 2 3 4 5 6 7 8 9 10
```

Fig. 5.7 | do...while repetition statement.

5.7 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 con-stants that evaluates to a constant integer value).

```
#include <iostream>
using namespace std;

int main()
{
    char grade;
    int aCount = 0, bCount = 0, cCount = 0, dCount = 0, fCount = 0;

    cout<< "enter grade (EOF to end)" << endl;
    grade = cin.get();
    while (grade != EOF)
    {
        switch (grade)
        {
        case 'A':
        case 'a':
            aCount++;
            break;
        case 'B':
        case 'b':
            bCount++;
            break;
        case 'C':
        case 'c':
            cCount++;
            break;
        case 'D':
        case 'd':
            dCount++;
            break;
        case 'F':
        case 'f':
            fCount++;
            break;
        case '\n':
        case '\t':
        case ' ':
            break;
        default:
            cout << "incorrect input. enter another input" << endl;
            break;
        }
        grade = cin.get();
    }
    cout << "Number of students who received each letter grade : " << endl;
    cout << "A = " << aCount << endl << "B = " << bCount << endl
         << "C = " << cCount << endl << "D = " << dCount << endl
         << "F = " << fCount << endl;
    return 0;
}
```

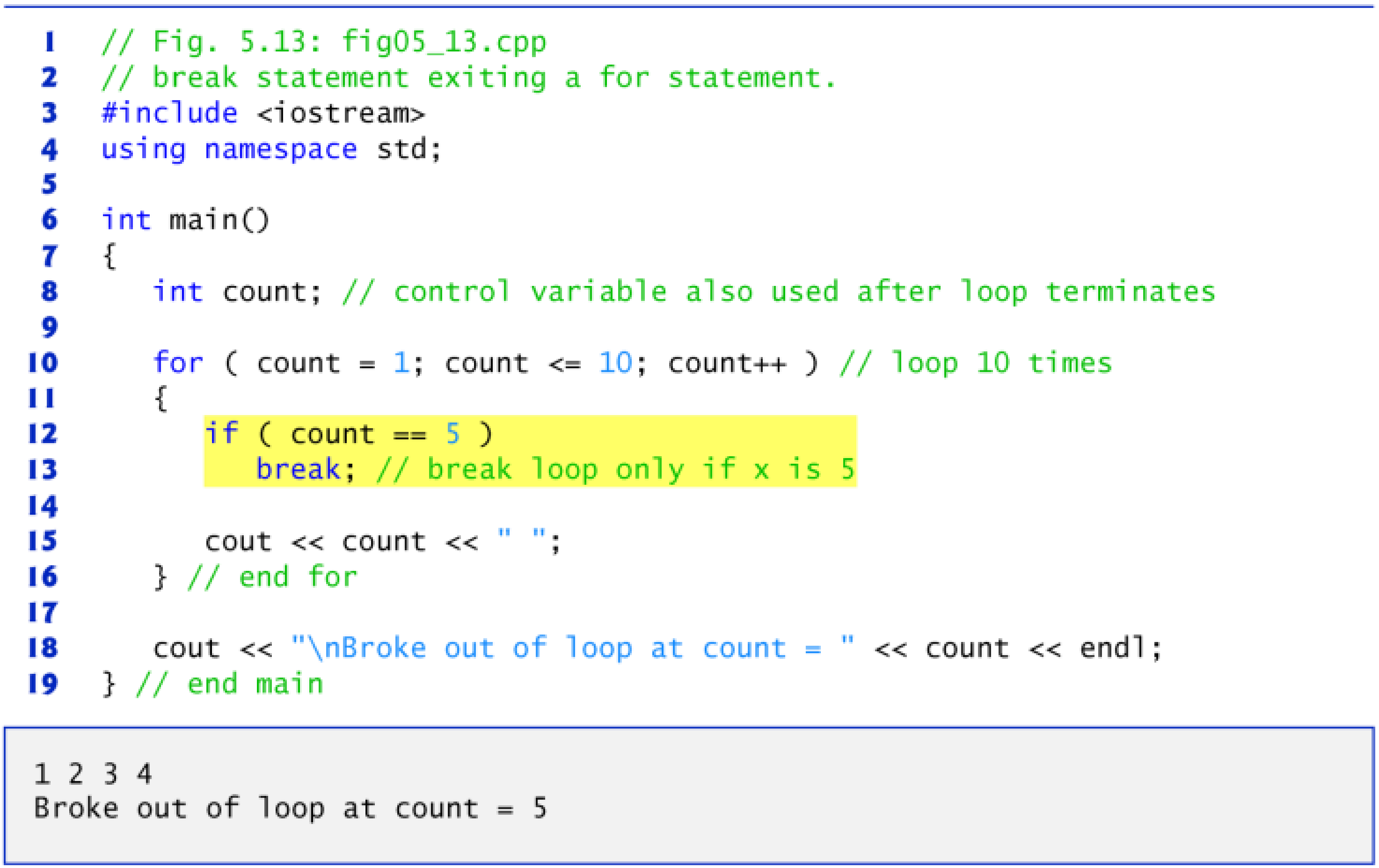


```
C
A
d
f
C
E
Incorrect letter grade entered. Enter a new grade.
D
A
b
^Z


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

Fig. 5.11 | Creating a GradeBook object and calling its member functions. (Part 2 of 2.)

5.9 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.



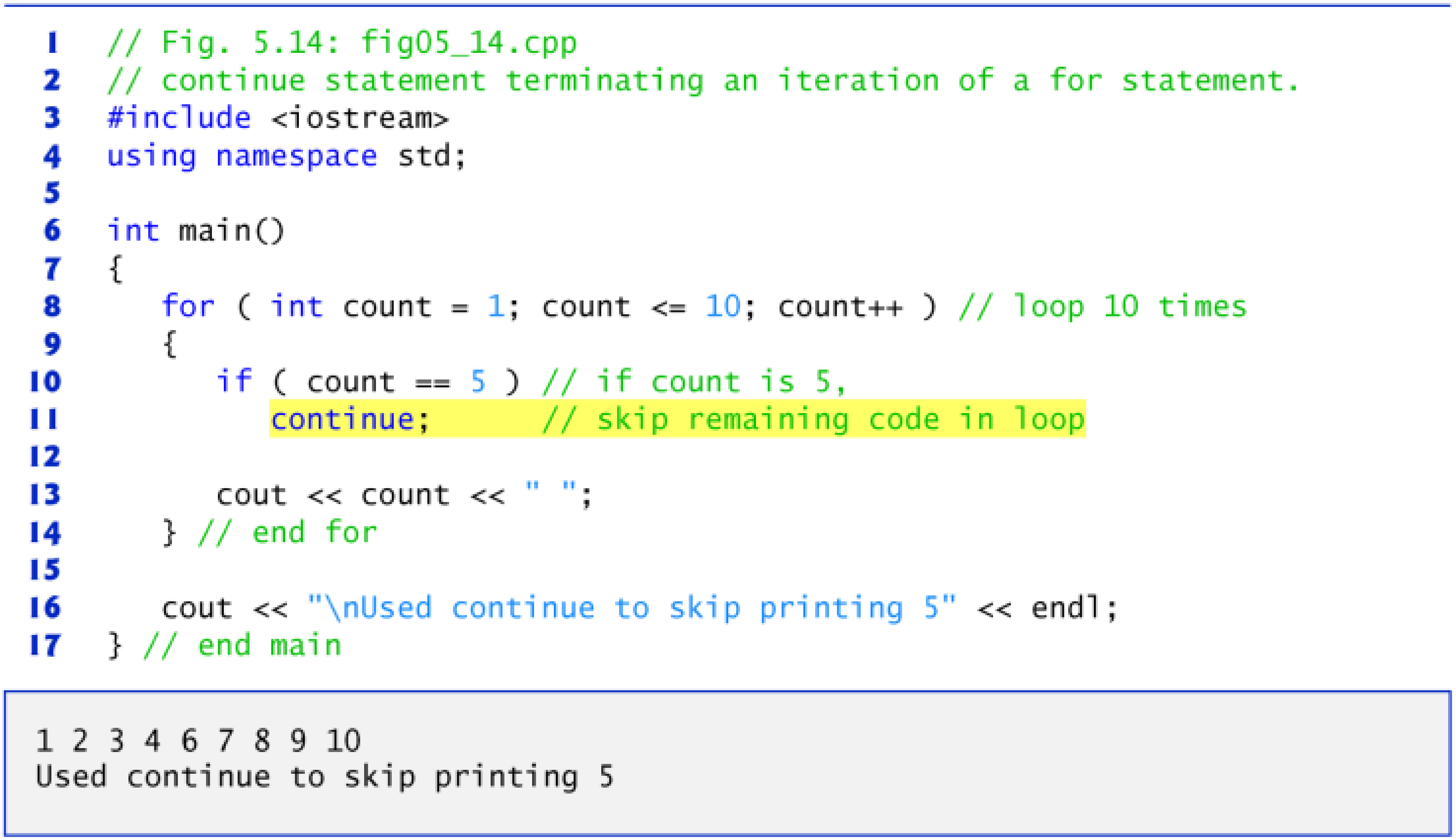
```cpp
// Fig. 5.13: fig05_13.cpp
// break statement exiting a for statement.
#include <iostream>
using namespace std;

int main()
{
   int count; // control variable also used after loop terminates

   for ( count = 1; count <= 10; count++ ) // loop 10 times
   {
      if ( count == 5 )
         break; // break loop only if x is 5

      cout << count << " ";
   } // end for

   cout << "\nBroke out of loop at count = " << count << endl;
} // end main
```

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

Fig. 5.13 | break statement exiting a for statement.

5.9 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.



```cpp
// Fig. 5.14: fig05_14.cpp
// continue statement terminating an iteration of a for statement.
#include <iostream>
using namespace std;

int main()
{
   for ( int count = 1; count <= 10; count++ ) // loop 10 times
   {
      if ( count == 5 ) // if count is 5,
         continue;      // skip remaining code in loop

      cout << count << " ";
   } // end for

   cout << "\nUsed continue to skip printing 5" << endl;
} // end main
```

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

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

5.10 Logical Operators

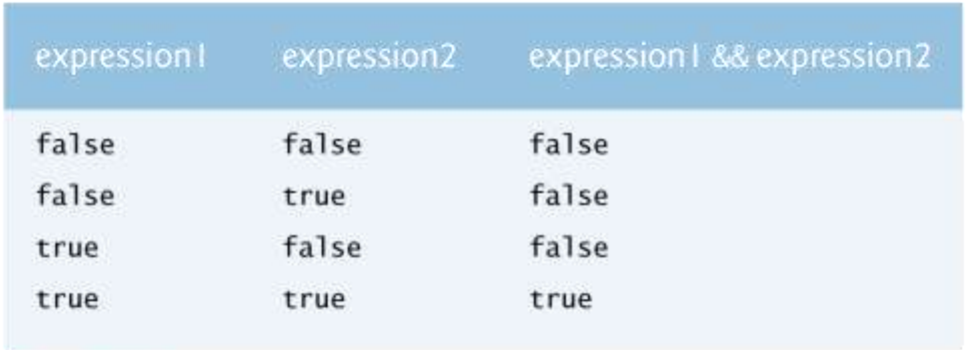
- C++ provides logical operators that are used to form more complex conditions by combining simple conditions.
- The logical operators are && (logical AND), || (logical OR) and ! (logical NOT, also called logical negation).

5.10 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`.

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 in-clude relational operators, equality operators and/or logical operators.

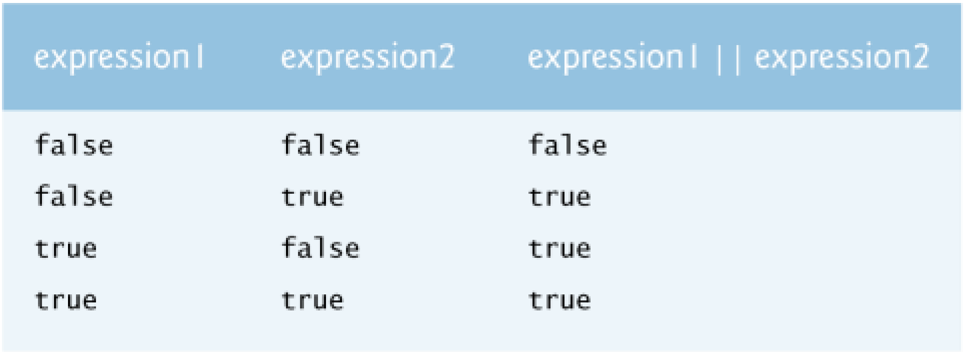


expression1	expression2	expression1 && expression2
false	false	false
false	true	false
true	false	false
true	true	true

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

5.10 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 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.
- 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`

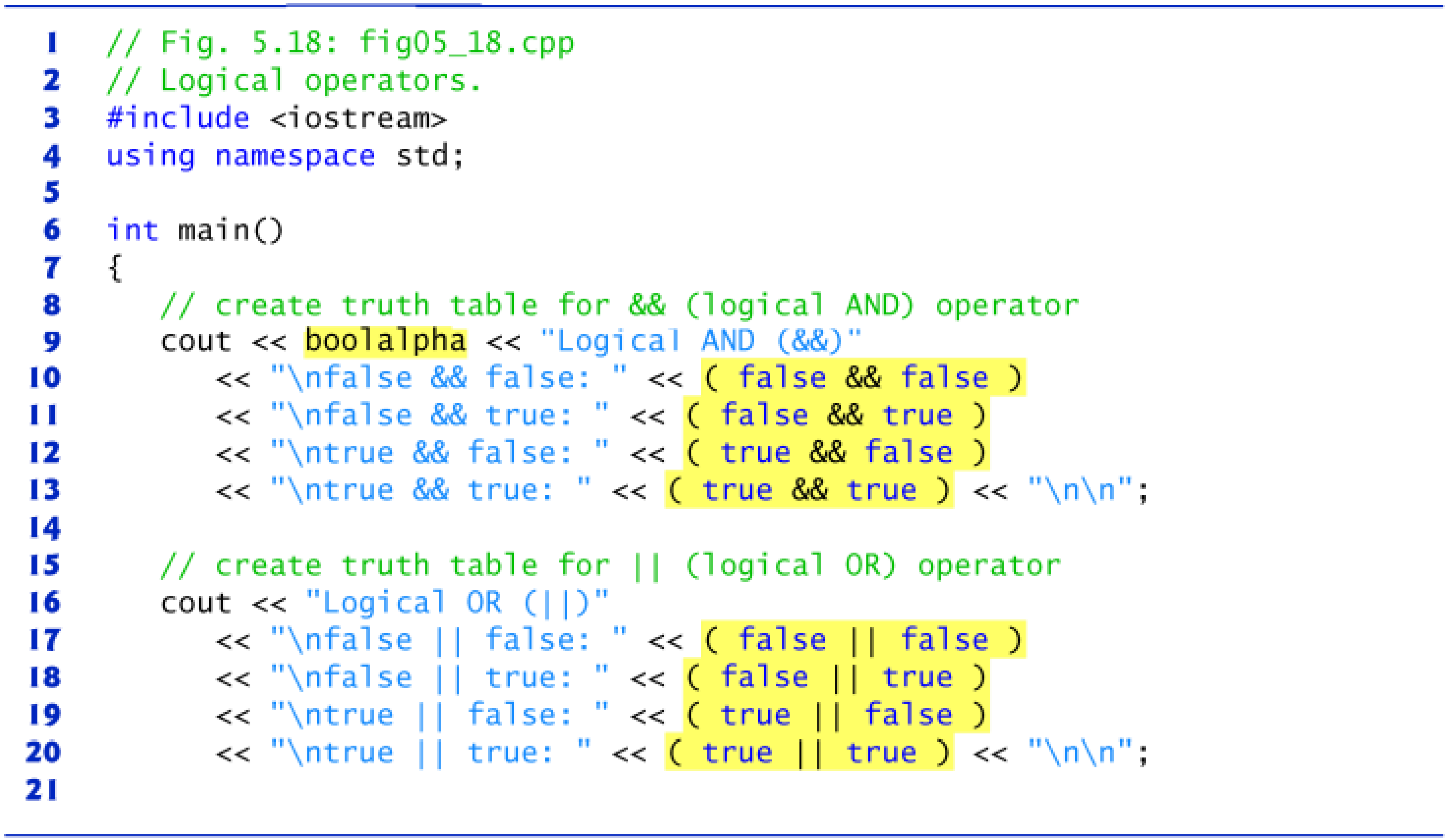
Fig. 5.16 | `||` (logical OR) operator truth table.

5.10 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 5.17 is a truth table for the logical negation operator (!).

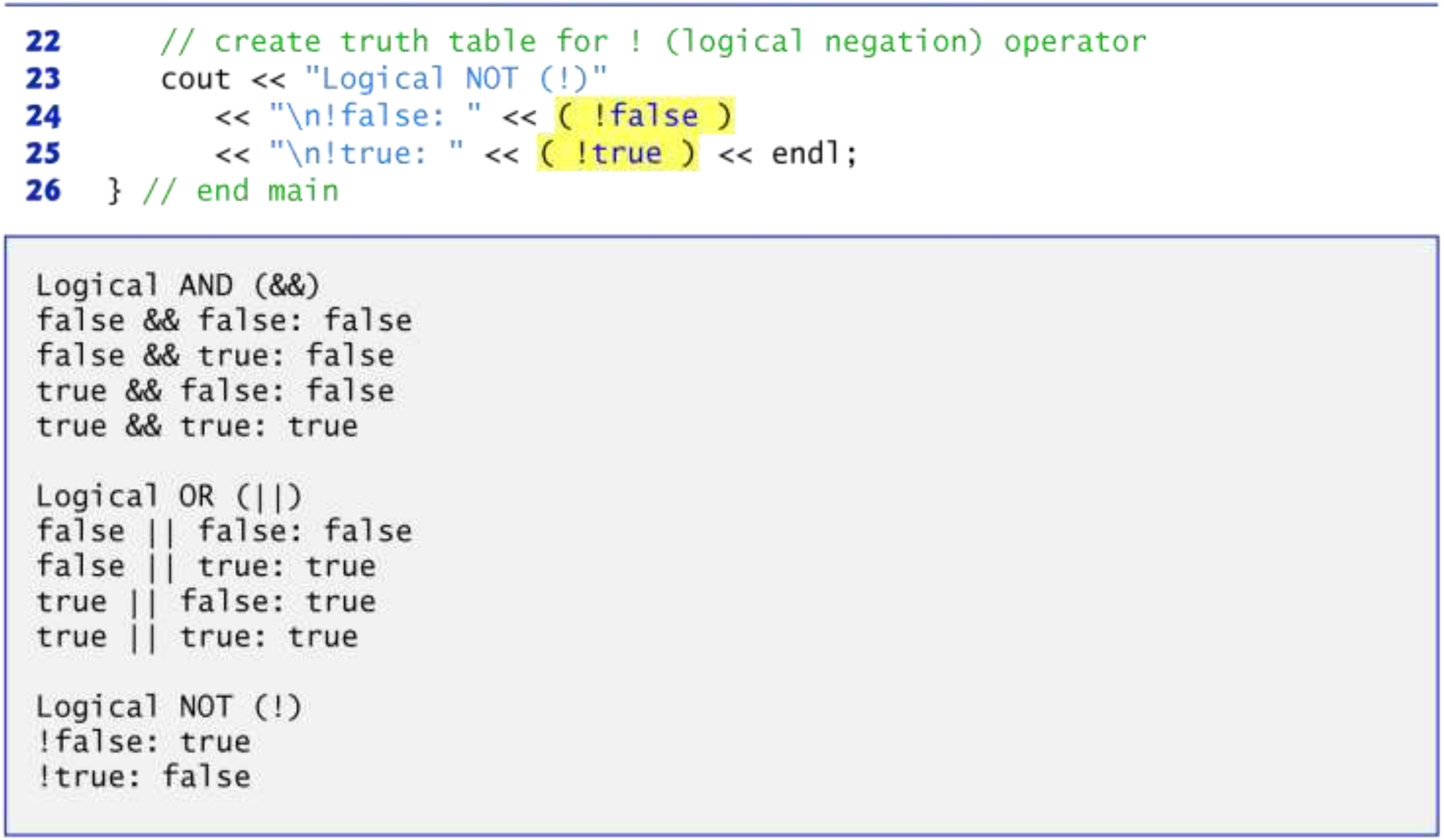
5.10 Logical Operators (cont.)

- 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."



```cpp
// Fig. 5.18: fig05_18.cpp
// Logical operators.
#include <iostream>
using namespace std;

int main()
{
   // create truth table for && (logical AND) operator
   cout << boolalpha << "Logical AND (&&)"
      << "\nfalse && false: " << ( false && false )
      << "\nfalse && true: " << ( false && true )
      << "\ntrue && false: " << ( true && false )
      << "\ntrue && true: " << ( true && true ) << "\n\n";

   // create truth table for || (logical OR) operator
   cout << "Logical OR (||)"
      << "\nfalse || false: " << ( false || false )
      << "\nfalse || true: " << ( false || true )
      << "\ntrue || false: " << ( true || false )
      << "\ntrue || true: " << ( true || true ) << "\n\n";

```

Fig. 5.18 | Logical operators.



```
   // create truth table for ! (logical negation) operator
   cout << "Logical NOT (!)"
      << "\n!false: " << ( !false )
      << "\n!true: " << ( !true ) << endl;
} // 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. 5.18 | Logical operators.