# Chapter 5, Control Statements: Part 2; Logical Operators

C++ How to Program, 9/e

 2018-06-01

## 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.

 2018-06-01



 2018-06-01

## 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.

 2018-06-01

**Error-Prevention Tip 5.1**

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



## 5.3 for Repetition Statement

- The `for` repetition statement specifies the counter-controlled repetition details in a single line of code.
- To illustrate the power of `for`, let's rewrite the program of Fig. 5.1. The result is shown in Fig. 5.2.
- 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. 5.2: fig05_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 ( unsigned 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. 5.2** | Counter-controlled repetition with the for statement.



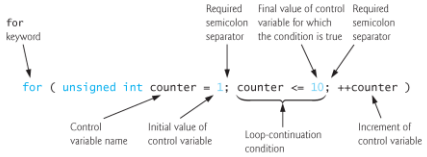


**Fig. 5.3** | for statement header components.

## 5.3 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.



## 5.3 for Repetition Statement (cont.)

***Expressions in the `for` Statement's Header Are Optional***

- 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.)

***Increment Expression Acts Like a Standalone Statement***

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



**Common Programming Error 5.2**

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.

## 5.3 for Repetition Statement (cont.)

***for* Statement: Notes and Observations**

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



**Error-Prevention Tip 5.2**

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



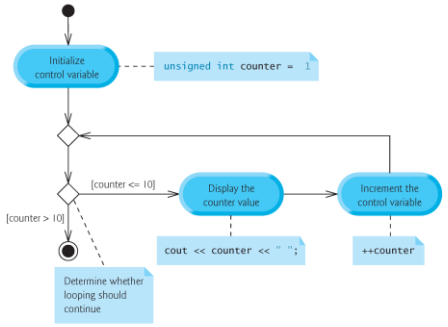


Fig. 5.4 | UML activity diagram for the `for` statement in Fig. 5.2.



## 5.4 Examples Using the for Statement (cont.)

***Application: Compound Interest Calculations***

- 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 (Fig. 5.6).

```
1  // Fig. 5.6: fig05_06.cpp
2  // Compound interest calculations with for.
3  #include <iostream>
4  #include <iomanip>
5  #include <cmath> // standard 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; // annual 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. 5.6 | Compound interest calculations with for. (Part 1 of 2.)

 

```
20    // calculate amount on deposit for each of ten years
21    for ( unsigned 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. 5.6 | Compound interest calculations with for. (Part 2 of 2.)

 

## 5.4 Examples Using the for Statement (cont.)

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

 

## 5.4 Examples Using the for Statement (cont.)

***Using Stream Manipulators to Format Numeric Output***

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

 

## 5.4 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 5.1**

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

 

**Performance Tip 5.2**

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.

 

## 5.5 `do...while` Repetition Statement

- Similar to the `while` statement.
- The `do…while` statement tests the loop-continuation condition *after* the loop body executes; therefore, *the loop body always executes at least once.*
- Figure 5.7 uses a `do…while` statement to print the numbers 1–10.

 

```
1  // Fig. 5.7: fig05_07.cpp
2  // do...while repetition statement.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     unsigned 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. 5.7 | do...while repetition statement.



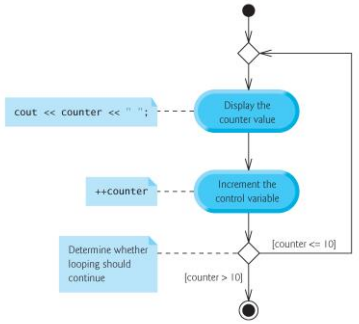


Fig. 5.8 | UML activity diagram for the do...while repetition statement of Fig. 5.7.



## 5.5 do...while Repetition Statement (cont.)

***Braces in a do...while Statement***

- It's not necessary to use braces in the do…while statement if there's only one statement in the body; however, most programmers include the braces to avoid confusion between the `while` and `do...while` statements.
- For example,

```
while ( condition )
```

- normally is regarded as the header of a while statement.



## 5.5 do...while Repetition Statement (cont.)

- A `do...while` with no braces around the single statement body appears as

```
do
   statement
while ( condition );
```

- which can be confusing.
- You might misinterpret the last line—`while( condition );`—as a while statement containing as its body an empty statement.

## 5.5 do…while Repetition Statement (cont.)

- Thus, the do…while with one statement often is written as follows to avoid confusion:

```
do
{
   statement
} while ( condition );
```



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



## 5.6 switch Multiple-Selection Statement (cont.)

### *GradeBook* Class Header

- Class GradeBook (Fig. 5.9) now contains five additional private data members (lines 18–22)—counter variables for each grade category (i.e., A, B, C, D and F).
- The class also contains two additional public member functions—inputGrades and displayGradeReport.



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

Fig. 5.9 | GradeBook class definition that counts letter grades.

```
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  #include "GradeBook.h" // include definition of class GradeBook
6  using namespace std;
7
8  // constructor initializes courseName with string supplied as argument;
9  // initializes counter data members to 0
10 GradeBook::GradeBook( string name )
11    : aCount( 0 ), // initialize count of A grades to 0
12      bCount( 0 ), // initialize count of B grades to 0
13      cCount( 0 ), // initialize count of C grades to 0
14      dCount( 0 ), // initialize count of D grades to 0
15      fCount( 0 )  // initialize count of F grades to 0
16 {
17    setCourseName( name );
18 } // end GradeBook constructor
19
```

Fig. 5.10 | GradeBook class uses switch statement to count letter grades. (Part 1 of 6.)



```
20 // function to set the course name; limits name to 25 or fewer characters
21 void GradeBook::setCourseName( string name )
22 {
23    if ( name.size() <= 25 ) // if name has 25 or fewer characters
24       courseName = name; // store the course name in the object
25    else // if name is longer than 25 characters
26    { // set courseName to first 25 characters of parameter name
27       courseName = name.substr( 0, 25 ); // select first 25 characters
28       cerr << "Name \"" << name << "\" exceeds maximum length (25).\n"
29          << "Limiting courseName to first 25 characters.\n" << endl;
30    } // end if...else
31 } // end function setCourseName
32
33 // function to retrieve the course name
34 string GradeBook::getCourseName() const
35 {
36    return courseName;
37 } // end function getCourseName
38
```

Fig. 5.10 | GradeBook class uses switch statement to count letter grades. (Part 2 of 6.)



```
39 // display a welcome message to the GradeBook user
40 void GradeBook::displayMessage() const
41 {
42    // this statement calls getCourseName to get the
43    // name of the course this GradeBook represents
44    cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
45       << endl;
46 } // end function displayMessage
47
48 // input arbitrary number of grades from user; update grade counter
49 void GradeBook::inputGrades()
50 {
51    int grade; // grade entered by user
52
53    cout << "Enter the letter grades." << endl
54       << "Enter the EOF character to end input." << endl;
55
```

Fig. 5.10 | GradeBook class uses switch statement to count letter grades. (Part 3 of 6.)



```
56    // loop until user types end-of-file key sequence
57    while ( ( grade = cin.get() ) != EOF )
58    {
59       // determine which grade was entered
60       switch ( grade ) // switch statement nested in while
61       {
62          case 'A': // grade was uppercase A
63          case 'a': // or lowercase a
64             ++aCount; // increment aCount
65             break; // necessary to exit switch
66
67          case 'B': // grade was uppercase B
68          case 'b': // or lowercase b
69             ++bCount; // increment bCount
70             break; // exit switch
71
72          case 'C': // grade was uppercase C
73          case 'c': // or lowercase c
74             ++cCount; // increment cCount
75             break; // exit switch
76
```

Fig. 5.10 | GradeBook class uses switch statement to count letter grades. (Part 4 of 6.)

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

Fig. 5.10 | GradeBook class uses switch statement to count letter grades. (Part 5 of 6.)

 

```
99
100 // display a report based on the grades entered by user
101 void GradeBook::displayGradeReport() const
      {
        // output summary of results
102    cout << "\n\nNumber of students who received each letter grade:"
103       << "\nA: " << aCount // display number of A grades
104       << "\nB: " << bCount // display number of B grades
105       << "\nC: " << cCount // display number of C grades
106       << "\nD: " << dCount // display number of D grades
107       << "\nF: " << fCount // display number of F grades
108       << endl;
109 } // end function displayGradeReport
```

Fig. 5.10 | GradeBook class uses switch statement to count letter grades. (Part 6 of 6.)

 

## 5.6 switch Multiple-Selection Statement (cont.)

***Reading Character Input***

- 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`, `long` and `long 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 statement

```
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.

 

## 5.6 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 5.1**

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

 

**Portability Tip 5.2**

Testing for the symbolic constant EOF rather than –1 makes programs more portable. The 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.

 

**Error-Prevention Tip 5.3**

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 5.4**

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

 

## 5.6 `switch` Multiple-Selection Statement (cont.)

***Ignoring Newline, Tab and Blank Characters in Input***

- 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.

 

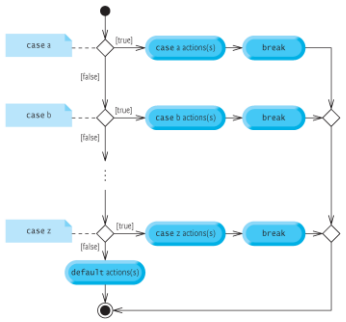


Fig. 5.12 | `switch` multiple-selection statement UML activity diagram with `break` statements.

 

## 5.6 `switch` Multiple-Selection Statement (cont.)

***Notes on Data Types***

- C++ has *flexible data type sizes* (see Appendix C, Fundamental Types).
- C++ provides several integer types.
- The range of integer values for each type is platform dependent.
- In addition to the types `int` and `char`, C++ provides the types `short` (an abbreviation of `short int`), `long` (an abbreviation of `long int`) and `long long` (an abbreviation of `long long int`).
- The minimum range of values for `short` integers is –32,767 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.

 

## 5.6 `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.

 

## 5.6 switch Multiple-Selection Statement (cont.)

***C++11 In-Class Initializers***

- C++11 allows you to provide a default value for a data member when you declare it in the class declaration.
- For example, lines 19–23 of Fig. 5.9 could have initialized data members `aCount`, `bCount`, `cCount`, `dCount` and `fCount` to 0 as follows:

```
unsigned int aCount = 0; // count of A grades
unsigned int bCount = 0; // count of B grades
unsigned int cCount = 0; // count of C grades
unsigned int dCount = 0; // count of D grades
unsigned int fCount = 0; // count of F grades
```

- rather than initializing them in the class's constructor (Fig. 5.10, lines 10–18).



## 5.7 break and continue Statements

***break* Statement**

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



```
1  // Fig. 5.13: fig05_13.cpp
2  // break statement exiting a for statement.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     unsigned 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 count 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. 5.13 | break statement exiting a for statement.



## 5.7 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. 5.14: fig05_14.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 ( unsigned 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. 5.14 | continue statement terminating an iteration of a for statement.

 

**Good Programming Practice 5.5**

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 5.1**

There's 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.

 

## 5.8 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).

 

| expression1 | expression2 | expression1 && expression2 |
|---|---|---|
| false | false | false |
| false | true | false |
| true | false | false |
| true | true | true |

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



| expression1 | expression2 | expression1 \|\| expression2 |
|---|---|---|
| false | false | false |
| false | true | true |
| true | false | true |
| true | true | true |

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



**Performance Tip 5.3**

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***

- 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 (`!`).

| expression | !expression |
|---|---|
| false | true |
| true | false |

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



## 5.8 Logical Operators (cont.)

***Logical Operators Example***

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



```
1  // Fig. 5.18: fig05_18.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";
```

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



```
21
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. 5.18 | Logical operators. (Part 2 of 2.)

| Operators | Associativity | Type |
|---|---|---|
| :: O | left to right *[See caution in Fig. 2.10 regarding grouping parentheses.]* | primary |
| ++ -- static_cast< *type* >() | left to right | postfix |
| ++ -- + - ! | right to left | unary (prefix) |
| * / % | left to right | multiplicative |
| + - | left to right | additive |
| << >> | left to right | insertion/extraction |
| < <= > >= | left to right | relational |
| == != | left to right | equality |
| && | left to right | logical AND |
| \|\| | left to right | logical OR |
| ?: | right to left | conditional |
| = += -= *= /= %= | right to left | assignment |
| , | left to right | comma |

Fig. 5.19 | Operator precedence and associativity.



## 5.9 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`.



## 5.11 Confusing the Equality (==) and Assignment (=) Operators (cont.)

**lvalues *and* rvalues**

- 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.