
5.8. Logical Operators

So far we've studied only **simple conditions**, such as `counter <= 10`, `total > 1000` and `number != sentinelValue`. We expressed these conditions in terms of the relational operators `>`, `<`, `>=` and `<=`, and the equality operators `==` and `!=`. Each decision tested precisely one condition. To test multiple conditions while making a decision, we performed these tests in separate statements or in nested `if` or `if...else` statements.

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).

Logical AND (`&&`) Operator

Suppose that we wish to ensure that two conditions are *both* `true` before we choose a certain path of execution. In this case, we can use the `&&` (**logical AND**) operator, as follows:

Click here to view code image

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

This `if` statement contains two simple conditions. The condition `gender == FEMALE` is used here to determine whether a person is a female. The condition `age >= 65` determines whether a person is a senior citizen. The simple condition to the left of the `&&` operator evaluates first. If necessary, the simple condition to the right of the `&&` operator evaluates next. As we'll discuss shortly, the right side of a logical AND expression is evaluated *only* if the left side is `true`. The `if` statement then considers the combined condition

```
gender == FEMALE && age >= 65
```

This condition is `true` if and only if *both* of the simple conditions are `true`. Finally, if this combined condition is indeed `true`, the statement in the `if` statement's body increments the count of `seniorFemales`. If either (or both) of the simple conditions are `false`, then the program skips the incrementing and proceeds to the statement following the `if`. The preceding combined condition can be made more readable by adding redundant parentheses:

```
( gender == FEMALE ) && ( age >= 65 )
```

Common Programming Error 5.10

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++.

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

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

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

Logical OR (||) Operator

Now let's consider the || (**logical OR**) operator. Suppose we wish to ensure that either *or* both of two conditions are `true` before we choose a certain path of execution. In this case, we use the `||` operator, as in the following program segment:

Click here to viewcode image

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

This preceding condition contains two simple conditions. The simple condition `semesterAverage >= 90` evaluates to determine whether the student deserves an "A" in the course because of a solid performance throughout the semester. The simple condition `finalExam >= 90` evaluates to determine whether the student deserves an "A" in the course because of an outstanding performance on the final exam. The `if` statement then considers the combined condition

```
( semesterAverage >= 90 ) || ( finalExam >= 90 )
```

and awards the student an "A" if *either or both* of the simple conditions are `true`. The message "`Student grade is A`" prints unless *both* of the simple conditions are `false`. Figure 5.16 is a truth table for the logical OR operator (`||`).

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

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

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. Thus, evaluation of the expression

```
( gender == FEMALE ) && ( age >= 65 )
```

stops immediately if `gender` is not equal to `FEMALE` (i.e., the entire expression is `false`) and continues if `gender` is equal to `FEMALE` (i.e., the entire expression could still be `true` if the condition `age >= 65` is `true`). This performance feature for the evaluation of logical AND and logical OR expressions is called **short-circuit evaluation**.

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.

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. The unary logical negation operator is placed *before* a condition when we are interested in choosing a path of execution if the original condition (without the logical negation operator) is `false`, such as in the following program segment:

Click here to viewcode image

```
if ( !( grade == sentinelValue ) )
   cout << "The next grade is " << grade << endl;
```

The parentheses around the condition `grade == sentinelValue` are needed because the logical negation operator has a higher precedence than the equality operator.

You can often avoid the `!` operator by using an appropriate relational or equality operator. For example, the preceding `if` statement also can be written as follows:

Click here to view code image

```
if ( grade != sentinelValue )
   cout << "The next grade is " << grade << endl;
```

This flexibility often can help you express a condition in a more "natural" or convenient manner. 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.

Logical Operators Example

Figure 5.18 demonstrates the logical operators by producing their truth tables. The output shows each expression that's 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. We use **stream manipulator `boolalpha`** (a *sticky* manipulator) in line 9 to specify that the value of each `bool` expression should be displayed as either the word "true" or the word "false." For example, the result of the expression `false && false` in line 10 is `false`, so the second line of output includes the word "false." Lines 9–13 produce the truth table for `&&`. Lines 16–20 produce the truth table for `||`. Lines 23–25 produce the truth table for `!`.

Click here to view code image

```
// 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";

   // 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.

Summary of Operator Precedence and Associativity

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

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

Fig. 5.19. Operator precedence and associativity.