3.6 — Logical operators

BY ALEX ON JUNE 15TH, 2007 | LAST MODIFIED BY ALEX ON FEBRUARY 26TH, 2018

While relational (comparison) operators can be used to test whether a particular condition is true or false, they can only test one condition at a time. Often we need to know whether multiple conditions are true at once. For example, to check whether we've won the lottery, we have to compare whether all of the multiple numbers we picked match the winning numbers. In a lottery with 6 numbers, this would involve 6 comparisons, *all* of which have to be true. Other times, we need to know whether any one of multiple conditions is true. For example, we may decide to skip work today if we're sick, or if we're too tired, or if won the lottery in our previous example. This would involve checking whether *any* of 3 comparisons is true.

Logical operators provide us with this capability to test multiple conditions.

C++ provides us with 3 logical operators:

Operator	Symbol	Form	Operation
Logical NOT	!	!x	true if x is false, or false if x is true
Logical AND	&&	x && y	true if both x and y are true, false otherwise
Logical OR	||	x || y	true if either x or y are true, false otherwise

Logical NOT

You have already run across the logical NOT unary operator in section **2.6 -- Boolean values**. We can summarize the effects of logical NOT like so:

Logical NOT (operator !)	
Right operand	**Result**
true	false
false	true

If logical NOT's operand evaluates to true, logical NOT evaluates to false. If logical NOT's operand evaluates to false, logical NOT evaluates to true. In other words, logical NOT flips a boolean value from true to false, and vice-versa.

Logical NOT is often used in conditionals:

```
bool bTooLarge = (x > 100); // bTooLarge is true if x > 100
if (!bTooLarge)
    // do something with x
else
    // print an error
```

One thing to be wary of is that logical NOT has a very high level of precedence. New programmers often make the following mistake:

```
    int x = 5;
    int y = 7;

    if (! x == y)
        cout << "x does not equal y";
    else
        cout << "x equals y";
```

This program prints "x equals y"! But x does not equal y, so how is this possible? The answer is that because the logical NOT operator has higher precedence than the equality operator, the expression `! x == y` actually evaluates as `(!x) == y`. Since x is 5, !x evaluates to 0, and 0 == y is false, so the else statement executes!

Reminder: any non-zero integer value evaluates to *true* when used in a boolean context. Since x is 5, x evaluates to true, and !x evaluates to false (0). Mixing integer and boolean operations like this can be very confusing, and should be avoided!

The correct way to write the above snippet is:

```
    int x = 5;
    int y = 7;

    if (!(x == y))
        cout << "x does not equal y";
    else
        cout << "x equals y";
```

This way, x == y will be evaluated first, and then logical NOT will flip the boolean result.

Rule: If logical NOT is intended to operate on the result of other operators, the other operators and their operands need to be enclosed in parenthesis.
Rule: It's a good idea to always use parenthesis to make your intent clear -- that way, you don't even have to remember the precedence rules.

Simple uses of logical NOT, such as `if (!bValue)` do not need parenthesis because precedence does not come into play.

Logical OR

The logical OR operator is used to test whether either of two conditions is true. If the left operand evaluates to true, or the right operand evaluates to true, the logical OR operator returns true. If both operands are true, then logical OR will return true as well.

Logical OR (operator ||)		
Left operand	**Right operand**	**Result**
false	false	false
false	true	true
true	false	true
true	true	true

For example, consider the following program:

```
#include <iostream>

int main()
{
    std::cout << "Enter a number: ";
    int value;
    std::cin >> value;

    if (value== 0 || value== 1)
        std::cout << "You picked 0 or 1" << std::endl;
    else
        std::cout << "You did not pick 0 or 1" << std::endl;
    return 0;
}
```

In this case, we use the logical OR operator to test whether either the left condition (value == 0) or the right condition (value == 1) is true. If either (or both) are true, the logical OR operator evaluates to true, which means the if statement executes. If neither are true, the logical OR operator evaluates to false, which means the else statement executes.

You can string together many logical OR statements:

```
if (value == 0 || value == 1 || value == 2 || value == 3)
    std::cout << "You picked 0, 1, 2, or 3" << std::endl;
```

New programmers sometimes confuse the logical OR operator (||) with the bitwise OR operator (|). Even though they both have OR in the name, they perform different functions. Mixing them up will probably lead to incorrect results.

Logical AND

The logical AND operator is used to test whether both conditions are true. If both conditions are true, logical AND returns true. Otherwise, it returns false.

Logical AND (operator &&)		
Left operand	**Right operand**	**Result**
false	false	false
false	true	false
true	false	false
true	true	true

For example, we might want to know if the value of variable x is between 10 and 20. This is actually two conditions: we need to know if x is greater than 10, and also whether x is less than 20.

```
#include <iostream>

int main()
{
    std::cout << "Enter a number: ";
    int value;
    std::cin >> value;

    if (value > 10 && value < 20)
        std::cout << "Your value is between 10 and 20" << std::endl;
    else
        std::cout << "Your value is not between 10 and 20" << std::endl;
    return 0;
}
```

In this case, we use the logical AND operator to test whether the left condition (value > 10) AND the right condition (value < 20) are both true. If both are true, the logical AND operator evaluates to true, and the if statement executes. If neither are true, or only one is true, the logical AND operator evaluates to false, and the else statement executes.

As with logical OR, you can string together many logical AND statements:

```
if (value > 10 && value < 20 && value != 16)
    // do something
else
    // do something else
```

If all of these conditions are true, the if statement will execute. If any of these conditions are false, the else statement will execute.

Short circuit evaluation

In order for logical AND to return true, both operands must evaluate to true. If the first operand evaluates to false, logical AND knows it must return false regardless of whether the second operand evaluates to true or false. In this case, the logical AND operator will go ahead and return false immediately without even evaluating the second operand! This is known as **short circuit evaluation**, and it is done primarily for optimization purposes.

Similarly, if the first operand for logical OR is true, then the entire OR condition has to evaluate to true, and the second operand does not need to be evaluated.

Short circuit evaluation presents another opportunity to show why operators that cause side effects should not be used in compound expressions. Consider the following snippet:

```
if (x == 1 && y++ == 2)
    // do something
```

if x does not equal 1, the conditional must be false, so y++ never gets evaluated! Thus, y will only be incremented if x evaluates to 1, which is probably not what the programmer intended!

As with logical and bitwise OR, new programmers sometimes confuse the logical AND operator (&&) with the bitwise AND operator (&).

Mixing ANDs and ORs

Mixing logical AND and logical OR operators in the same expression often can not be avoided, but it is an area full of potential dangers.

Many programmers assume that logical AND and logical OR have the same precedence (or forget that they don't), just like addition/subtraction and multiplication/division do. However, logical AND has higher precedence than logical OR, thus logical AND operators will be evaluated ahead of logical OR operators (unless they have been parenthesized).

As a result of this, new programmers will often write expressions such as `value1 || value2 && value3`. Because logical AND has higher precedence, this evaluates as `value1 || (value2 && value3)`, not `(value1 || value2) && value3`. Hopefully that's what the programmer wanted! If the programmer was assuming left to right evaluation (as happens with addition/subtraction, or multiplication/division), the programmer will get a result he or she was not expecting!

When mixing logical AND and logical OR in the same expression, it is a good idea to explicitly parenthesize each operator and its operands. This helps prevent precedence mistakes, makes your code easier to read, and clearly defines how you intended the expression to evaluate. For example, rather than writing `value1 && value2 || value3 && value4`, it is better to write `(value1 && value2) || (value3 && value4)`.

De Morgan's law

Many programmers also make the mistake of thinking that `!(x && y)` is the same thing as `!x && !y`. Unfortunately, you can not "distribute" the logical NOT in that manner.

De Morgan's law tells us how the logical NOT should be distributed in these cases:

`!(x && y)` is equivalent to `!x || !y`
`!(x || y)` is equivalent to `!x && !y`

In other words, when you distribute the logical NOT, you also need to flip logical AND to logical OR, and vice-versa!

This can sometimes be useful when trying to make up complex expressions easier to read.

Where's the logical exclusive or (XOR) operator?

Logical XOR is a logical operator provided in some languages that is used to test whether an odd number of conditions is true.

Logical XOR		
Left operand	**Right operand**	**Result**
false	false	false
false	true	true
true	false	true
true	true	false

C++ doesn't provide a logical XOR operator. Unlike logical OR or logical AND, XOR cannot be short circuit evaluated. Because of this, making an XOR operator out of logical OR and logical AND operators is challenging. However, you can easily mimic logical XOR using the not equals operator (!=):

```
1 if (a != b) ... // a XOR b, assuming a and b are booleans
```

This can be extended to multiple operands as follows:

```
1 if (a != b != c != d) ... // a XOR b XOR c XOR d, assuming a, b, c, and d are booleans
```

Note that the above XOR patterns only work if the operands are booleans (not integers). If you need a form of XOR that works with non-boolean operands, you can static_cast them to bool:

```
1 if (static_cast<bool>(a) != static_cast<bool>(b) != static_cast<bool>(c) != static_cast<bool>(d)) ... //
   a XOR b XOR c XOR d, for any type that can be converted to bool
```

Quiz

Evaluate the following:
1) (true && true) || false

2) (false && true) || true
3) (false && true) || false || true
4) (5 > 6 || 4 > 3) && (7 > 8)
5) !(7 > 6 || 3 > 4)

Quiz answers

Note: in the following answers, we “explain our work” by showing you the steps taken to get to the final answer. The steps are separated by a => symbol. For example “(true || false) => true” means we evaluated “(true || false)” to arrive at the value “true”.

1) **Hide Solution**

(true && true) || false =>
true || false =>
true

2) **Hide Solution**

(false && true) || true =>
false || true =>
true

3) **Hide Solution**

(false && true) || false || true =>
false || false || true =>
false || true =>
true

4) **Hide Solution**

(5 > 6 || 4 > 3) && (7 > 8) =>
(false || true) && false =>
true && false =>
false

5) **Hide Solution**

!(7 > 6 || 3 > 4) =>
!(true || false) =>
!true =>
false

 **3.7 -- Converting between binary and decimal**

 **Index**

 **3.5 -- Relational operators (comparisons)**

Share this: