5.7 — Logical operators

BY ALEX ON JUNE 15TH, 2007 | LAST MODIFIED BY NASCARDRIVER ON JANUARY 30TH, 2020

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 simultaneously. 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. In other cases, 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 we won the lottery in our previous example. This would involve checking whether *any* of 3 comparisons is true.

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

C++ has 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 lesson **4.9 -- Boolean values**. We can summarize the effects of logical NOT like so:

Logical NOT (operator !)

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:

```
1 bool tooLarge { x > 100 }; // tooLarge is true if x > 100
2 if (!tooLarge)
3     // do something with x
4 else
5     // 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:

```
1 #include <iostream>
2
3 int main()
4 {
5     int x { 5 };
6     int y { 7 };
```

```
7
8       if (! x > y)
9           std::cout << "x is not greater than y\n";
10      else
11          std::cout << "x is equal to or greater than y\n";
12
13      return 0;
14  }
```

This program prints:

```
x is equal to or greater than y
```

But *x* is not equal to or greater than *y*, so how is this possible? The answer is that because the *logical NOT* operator has higher precedence than the greater than 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!

The correct way to write the above snippet is:

```
1   #include <iostream>
2
3   int main()
4   {
5       int x { 5 };
6       int y { 7 };
7
8       if (!(x > y))
9           std::cout << "x is not greater than y\n";
10      else
11          std::cout << "x is equal to or greater than y\n";
12
13      return 0;
14  }
```

This way, `x > y` will be evaluated first, and then logical NOT will flip the Boolean result.

Best practice

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.

Simple uses of *logical NOT*, such as `if (!value)` 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, or both are true, then the *logical OR* operator returns true. Otherwise it will return false.

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\n";
    else
        std::cout << "You did not pick 0 or 1\n";
    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\n";
```

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 operands are true. If both operands 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\n";
    else
        std::cout << "Your value is not between 10 and 20\n";
    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 won't be 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 whole condition 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!

Warning

Short circuit evaluation may cause *Logical OR* and *Logical AND* to not evaluate one operand. Avoid using expressions with side effects in conjunction with these operators.

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).

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)`.

Best practice

When mixing *logical AND* and *logical OR* in a single expression, explicitly parenthesize each operation to ensure they evaluate how you intend.

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 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*, *logical XOR* cannot be short circuit evaluated. Because of this, making an *logical XOR* operator out of *logical OR* and *logical AND* operators is challenging. However, you can easily mimic *logical XOR* using the *inequality* operator (!=):

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

This can be extended to multiple operands as follows:

```
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 *logical XOR* that works with non-Boolean operands, you can static_cast them to bool:

```
if (static_cast<bool>(a) != static_cast<bool>(b) != static_cast<bool>(c) != static_cast<bool>(d
```

Quiz time

Question #1

Evaluate the following expressions.

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. Expressions that were ignored due to the short circuit rules are placed in square brackets. For example
(1 < 2 || 3 != 3) =>
(true || [3 != 3]) =>
(true) =>
true
means we evaluated (1 < 2 || 3 != 3) to arrive at (true || [3 != 3]) and evaluated that to arrive at "true". The 3 != 3 was never executed due to short circuiting.

a) (true && true) || false

Show Solution

b) (false && true) || true

Show Solution

c) (false && true) || false || true

Show Solution

d) (5 > 6 || 4 > 3) && (7 > 8)

Show Solution

e) !(7 > 6 || 3 > 4)

Show Solution

5.x -- Chapter 5 summary and quiz

Index

5.6 -- Relational operators and floating point comparisons

C++ TUTORIAL | PRINT THIS POST

132 comments to 5.7 — Logical operators

« Older Comments 1 2

Raton
February 4, 2020 at 9:51 am · Reply

Hi, I wrote a function to simulate a logical XOR:

```
1 bool logicXor(bool op1, bool op2)
2 {
3     return (op1 ? (!op2) : op2);
4 }
```

I think it makes it easier to read in expressions than using operator!=. Is it okay to use it or should I use operator!=?

Christopher Springer
January 29, 2020 at 8:52 am · Reply

For the quiz questions in this lesson it may be useful to indicate in the step which values were never evaluated due to short-circuiting of the logical operators. As written, the answers seem to imply that all of the values were evaluated...which is not the case. Just a suggestion! :)

nascardriver
January 30, 2020 at 7:34 am · Reply

I updated to quiz to respect short circuiting. I wasn't sure about which syntax to use, please let me know if what I went with is understandable.
Thanks for the suggestion!

pali7
January 28, 2020 at 2:17 am · Reply

You have one small typo in first column:
"if we're sick, or if we're too tired, or if won the lottery in our previous example"
should be
"if we're sick, or if we're too tired, or if WE won the lottery in our previous example"
As a side note: Thank you very much for the best tutorials for cpp I ever encountered! :)

nascardriver
January 28, 2020 at 2:44 am · Reply

Lesson updated, thanks!

5irKawfycups
November 5, 2019 at 7:02 pm · Reply

"Don't assume the compiler will evaluate in the order you think it will" seems to be a high takeaway in these sections!

I.e., be explicit and use parentheses!

Louis Cloete
August 19, 2019 at 1:20 pm · Reply

Hi Alex!

In the snippet just after "Logical NOT is often used in conditionals:" why is the expression in parenthesis? I thought it could be without them?

Also, I think the correct way to fix the precedence issue example is to use != instead of ! and ==, not to use parenthesis. If you want to keep the parenthesis example, I think one of the operators with < or > in them would be less contrived.

Alex
August 21, 2019 at 8:13 pm · Reply

I removed the offending parenthesis.

I also updated the example per your feedback. Thanks for the suggestion!

quiz 5
August 9, 2019 at 8:21 am · Reply

result of quiz 5 is not true??

Louis Cloete
August 19, 2019 at 1:01 pm · Reply

No. The expression inside the parenthesis evaluate to true, but then that is negated because of the ! in front, so it becomes a false.

daniel jo
June 13, 2019 at 11:49 pm · Reply

On logical OR you should also say that if left operand is true that it WONT evaluate the right operand

Alex
June 18, 2019 at 3:31 pm · Reply

Thanks for the feedback. I've updated the language to make it clear that the evaluation of the right operand doesn't happen.

Michael
April 18, 2019 at 12:26 am · Reply

In the last example, why is casting to bool needed? Wouldn't casting integers to bool be redundant since any non-zero integers evaluates to true and false otherwise already?

nascardriver
April 18, 2019 at 1:41 am · Reply

Assume
a = 1
b = 2
c = 3
d = 4

Without cast

```
1 != 2 != 3 != 4
(((1 != 2) != 3) != 4)
((true != 3) != 4)
// Promotion to int
((1 != 3) != 4)
(true != 4)
// Promotion to int
(1 != 4)
true
```

With cast

```
1 != 2 != 3 != 4
(((true != true) != true) != true)
((false != true) != true)
(true != true)
false
```

Michael
April 18, 2019 at 4:20 am · Reply

I see, but why would we ever want to cast int to bool? I feel like it doesn't make sense at all.

Anoop

May 26, 2019 at 11:54 pm · Reply

Because there are a lot of programmers who use integers to denote boolean variables, by making integer 0 be boolean false and any non-zero integer be boolean true. This is a remnant from older C programming, because unlike C++, C doesn't have a built-in boolean datatype by default, so all boolean logic was executed with ints.

Alireza
February 7, 2019 at 6:03 am · Reply

Greeting,

I want to know learn where Non-logical operators can be used ?
I mean these operators ' & , | ' .

nascardriver
February 7, 2019 at 7:07 am · Reply

Lesson O.3.8

Alireza
February 7, 2019 at 7:43 am · Reply

Thank you so much

NoviceN00b
December 19, 2018 at 9:36 am · Reply

In the first code snippit of Logical OR, on line 9, it is written as:

```
1 if (value== 0 || value== 1)
```

Whereas, in other examples of code it would be written as:

```
1 if (value == 0 || value == 1)
```

Just a small little inconsistency I noticed.

Alex
December 24, 2018 at 11:53 am · Reply

Thanks, formatting fixed!

Louis Cloete
December 5, 2018 at 2:52 pm · Reply

Just above the "Where's the logical exclusive or (XOR) operator?" heading:

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

Shouldn't it be "... trying to make complex expressions ..." ?

Benur21

November 10, 2018 at 1:44 pm · Reply

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

I would like to add the situation where bValue is a preprocessor macro. bValue could be replaced by "5 == 7", which would break the thing.

Alex
November 13, 2018 at 10:35 pm · Reply

You shouldn't be using preprocessor macros with substitution text. :)

Clapfish
November 9, 2018 at 11:26 am · Reply

Hi Alex!

Since you appreciate inconsistencies being flagged up, the second and third code snippets in this lesson feature 'cout' as opposed to 'std::cout'.

Hope that helps!

Alex
November 10, 2018 at 9:21 am · Reply

Thanks! Fixed.

Fateh Chadha
October 1, 2018 at 12:17 am · Reply

Hi this was a really great tutorial. So simplified. Though 1 question. Would you recommend to use logical operators in conditional operators?.. Although the idea sounds kinda complicated to me. Or should I practice?

Alex
October 1, 2018 at 9:32 am · Reply

Using logical operators in conditionals is very common, so better to practice than avoid. :)

Gabe
April 11, 2018 at 12:52 pm · Reply

Hi,
Is there a way of telling the compiler 1 through 25 or you have to use logical AND every time?

nascardriver
April 12, 2018 at 7:07 am · Reply

Hi Gabe!

What do you mean by "1 through 25"? Can you give an example of what you have in mind?

Gabe
April 15, 2018 at 7:47 am · Reply

I make a variable that gets assigned by an rng, then make an if statement that prints a string based on the value the rng comes up with. So, what I mean is can I write "if rngoutcome = 1 through 25
std::cout << string;"

nascardriver
April 15, 2018 at 8:07 am · Reply

Do you mean this?

```
1 if ((rngOutcome >= 1) && (rngOutcome <= 25))
2 {
3   std::cout << "Hello World!" << std::endl;
4 }
```

References
Lesson 3.1 - Operator precedence and associativity

Gabe
April 16, 2018 at 2:38 am · Reply

oh yea makes sense
thanks

Baljinder
February 23, 2018 at 7:23 am · Reply

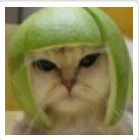
This is just a minor observation. The code snippet immediately under Logical AND writes (on line 7)

```
1 std::cin >> value ;
```

instead of

```
1 std::cin >> value;
```

as has been done in other examples.



Alex
February 26, 2018 at 8:05 pm · Reply

Thanks. Fixed!

Farshid
February 8, 2018 at 4:18 am · Reply

C++ provides a logical xor operator in the form of '^', for example:

```
1 #include <iostream>
2
3 int main(void){
4 bool value = true;
```

```
5  value = value^true; //false because of true XOR true = false
6  std::cout<<"true XOR true: "<<value<<std::endl; // 0 corresponds to false
7
8  value = true;
9  value = value^false; // true XOR false = true
10 std::cout<<"true XOR false: "<<value<<std::endl; // 1 corresponds to true
11
12 value = false;
13 value = value^false; //false XOR false = false
14 std::cout<<"false XOR false: "<<value<<std::endl; // 0 corresponds to false
15
16 value = false;//
17 value = value^true; //false XOR true = true
18 std::cout<<"false XOR true: "<<value<<std::endl; // 1 corresponds to true
19
20 return 0;
21 }
```

Which is the truth table for XOR.

nascardriver
February 8, 2018 at 4:35 am · Reply

Hi Farshid!

^ is the binary xor operator and therefore covered in lesson 3.8 (Bitwise operators).
!= is used as a logical xor

Farshid
February 8, 2018 at 5:52 am · Reply

Alright, thanks for clarification. I am yet to arrive at 3.8 :D

Trevor
January 15, 2018 at 7:45 pm · Reply

The short circuit evaluation would also apply to the AND operator - if the first operand is false, the result is false without having to evaluate the second operand. I would be concerned that the optimiser might also evaluate just the second operand (particularly if it is already a boolean variable) and skip evaluating the first operand if the second operand is true (for OR) or false (for AND), so does C++ guarantee the order of evaluation?
Thanks

nascardriver
January 16, 2018 at 2:50 am · Reply

Hi Trevor!
Conditions will be executed left to right unless the && or || operator have been overloaded to behave differently (Chapter 9).
I've never come across a class doing this, so I'd say you're safe to assume the default execution order.

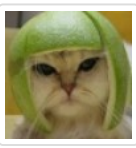
Alex
January 17, 2018 at 6:06 pm · Reply

(Initial incorrect response removed to confusion -- see the next response for correct answer)

nascardriver
January 18, 2018 at 3:15 am · Reply

ISO N4659 §8.14
"Unlike &, && guarantees left-to-right
evaluation: the second operand is not evaluated if the first operand is false."
(§8.15 is analogical for the logical or)

Correct me if I'm wrong, I think this means (a && b) will execute a first guaranteed.



Alex
January 18, 2018 at 6:15 pm · Reply

I stand corrected.

It appears that && (and ||) do guarantee left to right order of evaluation, but only for non-overloaded functions. If you overload && or ||, no short circuit evaluation or guaranteed ordering will take place.

Sooi Shanghei
October 30, 2017 at 9:27 am · Reply

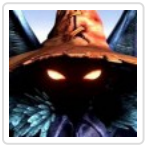
can we solve fifth question as:

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

?

Alex
October 31, 2017 at 10:58 am · Reply

Logically yes. But I'd argue it's less correct because when evaluating an expression, things inside the parenthesis should take precedence.



William
October 29, 2017 at 2:53 pm · Reply

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

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

Isn't the above code the same as the code below?

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

```
3
4       if (x != y)
5           cout << "x does not equal y";
6       else
7           cout << "x equals y";
```

Sooi Shanghei
October 30, 2017 at 9:06 am · Reply

yes,

```
1   if (!(x == y))
```

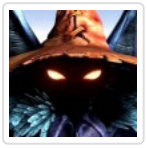
produces the same result as

```
1   if (x != y)
```

but be careful:

in first example you use "logical not" operator that has level 3 right to left (R->L) precedence,
but in second example you use "inequality" operator that has level 9 left to right (L->R) precedence.

They produce the same result but they are not the same thing.



William
October 31, 2017 at 3:18 pm · Reply

Ok that's good to know, thanks for the reply. :)

Alex
October 31, 2017 at 10:33 am · Reply

Yes.

jack
September 30, 2017 at 1:52 pm · Reply

HI Alex, Great tutorials you are such a Great teacher. are these expressions equivalent?
if not which is better?

```
1   x >4 && x < 10
2         //and
3   4<x<10
```

Alex
October 2, 2017 at 9:35 pm · Reply

Yes, they're equivalent, but C++ doesn't understand the latter, so we have to use the former.

programer
August 5, 2017 at 2:49 pm · Reply

Hi Alex,

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!

I was trying to better understand de morgans law then i came up with this solution to try to understand how it works is it correct ?

!(x and y) is equivalent to (!x and !y or !y and !x or !y and !y or !x and !x)

!(x or y) is equivalent to (!x or !y and !y or !x and !y or !y and !x or !x)

simone
March 28, 2017 at 1:15 am · Reply

Hi Alex,

There is one thing I don't get. You wrote

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

I assume that the XOR of more booleans is true whenever (and only when) exactly one of the booleans are true. Am I right?

But this dos not seem true to me, even with only three booleans. In fact if

```
1 a=b=c=true
```

then

```
1 ((a!=b)!c)=((true!=true)!=true)=(false!=true)=true
```

But, perhaps, I just got wrong what is the XOR of multi booleans. So,

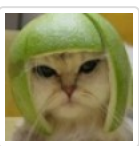
```
1 ((a!=b)!c)
```

would just mean

```
1  ((a XOR b) XOR c)
```

which means: if c is true then a and b are either both false or either both true, if c is false then one of a and b is true and the other is false (in order for the statement to be true), which in turn means that exactly one of the three is false or they are all true. I don't know in which situations this expression might be useful. Especially with more than three variables the logical table becomes quite complex and hard to understand.

Thanks.



Alex
March 28, 2017 at 4:43 pm · Reply

XOR of multiple booleans evaluates to true whenever an odd number of the inputs are true, and false whenever an even number of inputs are true. XOR is rarely used.

simone
March 29, 2017 at 12:09 am · Reply

Cool! Thank you.

Qaiser Abbas
March 20, 2017 at 5:35 am · Reply

Hi,I am a beginner of C++.How do I start..? What Basics should a beginner must know before his coding..?

Alex
March 20, 2017 at 8:51 am · Reply

Start at the beginning of the tutorial. It will tell you everything you need to know.

felipe
March 14, 2017 at 2:45 pm · Reply

Is

```
1 if (a != b != c != d)
```

evaluated to

```
1 if ((a != b) != (c != d))
```

or to

```
1 if (a != b then if b != c then if c != d)
```

? thanks.

Alex
March 14, 2017 at 3:05 pm · Reply

Operator != evaluates from left to right, so (a != b != c != d) evaluates as (((a != b) != c) != d).

Igna
January 31, 2017 at 2:38 pm · Reply

Typo... you put "then" instead of "than" in the statement "and also whether x is less then 20"
And sorry about my English but I'm Argentinian so I speak Spanish.
Have a nice day and your tutorial is amazing!!!

Alex
February 1, 2017 at 12:56 pm · Reply

Thanks, I've fixed the typo. Appreciate you pointing it out.

maria
December 11, 2016 at 6:06 am · Reply

i want to write a program where i just want to pick out cars which are black and has a brand name of toyota... now i am facing the problem that where will i declare the flag of black and toyota.. although i know this can be also done with out flag... just want to know
[/code]

```
#include <iostream>
int main()
{


  std::cout<< "enter colour";
  int colour;
  std::cin>>colour;

   std::cout<< "enter brand";
  int brand;
  std::cin>>brand;




  if(colour==1 && brand==0)

  {


    black= 1;    //colour flag
  brand toyota=0;   //brand flag

     std::cout<<"pick yhe car"<< std::endl;
  }
  else
  {
     std::cout<<"pick another"<<std::endl;

  }
  return 0;
}
```

Mike
November 22, 2016 at 4:11 am · Reply

Alex and others please help

```
1     while (ch != 'g' || 'c' || 'p' || 't')
2     {
3
4         cout << "Please enter a   c ,p,t or g\n";
5         cin >> ch;
6     }
7  // This code was supposed to run until ch is g or c or p or t  , but even if I put the vales
```

Alex
November 22, 2016 at 9:16 am · Reply

This evaluates as

```
while (ch != ( 'g' || 'c' || 'p' || 't' ))
```

which isn't what you want. You need to "distribute the variable", like so:

```
while (ch != 'g' && ch != 'c' && ch != 'p' && ch != 't' ))
```

(Note: I changed your logical ORs to logical ANDs as well)

Mike
November 22, 2016 at 4:33 pm · Reply

Thanks again Alex

Nyap
May 2, 2016 at 8:02 am · Reply

Might be a good idea to say how you actually type a logical or operator (Press alt gr + that three character key underneath esc)

Alex
May 2, 2016 at 3:22 pm · Reply

How you type | depends on the layout of your keyboard. In the US it's shift-\.

« Older Comments 1 2