

# Using bitwise operators for Booleans in C++

Asked 16 years, 4 months ago    Modified 6 years, 8 months ago

Viewed 94k times



Is there any reason not to use the bitwise operators `&`, `|`, and `^` for "bool" values in C++?

**110**



I sometimes run into situations where I want exactly one of two conditions to be true (XOR), so I just throw the `^` operator into a conditional expression. I also sometimes want all parts of a condition to be evaluated whether the result is true or not (rather than short-circuiting), so I use `&` and `|`. I also need to accumulate Boolean values sometimes, and `&=` and `|=` can be quite useful.



I've gotten a few raised eyebrows when doing this, but the code is still meaningful and cleaner than it would be otherwise. Is there any reason NOT to use these for bools? Are there any modern compilers that give bad results for this?

`c++`

`boolean`

`bitwise-operators`

Share

Improve this question

Follow

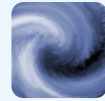
edited Sep 1, 2012 at 22:59



Coding Mash

3,346 ● 5 ● 25 ● 45

asked Aug 23, 2008 at 19:52



Jay Conrod

29.6k ● 20 ● 99 ● 110

If you have learned sufficiently much about C++ since asking this question, you should come back and un-accept the current answer and accept Patrick Johnmeyer's answer for correctly explaining the short-circuiting difference. – [codetaku](#) Apr 4, 2017 at 14:50

General remark because people question if this is a good idea. If you care about the branch coverage of unit tests, then reducing branches is desirable. Bitwise operators are useful for this. – [qznc](#) May 22, 2019 at 11:48

## 9 Answers

Sorted by:

Highest score (default)



72

`||` and `&&` are boolean operators and the built-in ones are guaranteed to return either `true` or `false`. Nothing else.



`|`, `&` and `^` are bitwise operators. When the domain of numbers you operate on is just 1 and 0, then they are exactly the same, but in cases where your booleans are not strictly 1 and 0 – as is the case with the C language – you may end up with some behavior you didn't want. For instance:



```
BOOL two = 2;  
BOOL one = 1;  
BOOL and = two & one;    //and = 0  
BOOL cand = two && one;  //cand = 1
```

In C++, however, the `bool` type is guaranteed to be only either a `true` or a `false` (which convert implicitly to respectively `1` and `0`), so it's less of a worry from this stance, but the fact that people aren't used to seeing such things in code makes a good argument for not doing it. Just say `b = b && x` and be done with it.

Share Improve this answer

edited Apr 28, 2016 at 9:30

Follow



Cheers and hth. - Alf

145k ● 15 ● 214 ● 339

answered Aug 23, 2008 at 20:06



Patrick

92.4k ● 11 ● 52 ● 61

---

no such thing as logical xor operator `^^` in C++. For both arguments as bools `!=` does the same thing, but can be dangerous if either argument is not a bool (as would be using `^` for a logical xor). Not sure how this got accepted...

– [Greg Rogers](#) Oct 6, 2008 at 3:52

---

1 In retrospect, that was kinda dumb. Anyway, changed the example to an `&&` to fix the problem. I swear I've used `^^` sometime in the past, but this must not have been the case.

– [Patrick](#) Oct 7, 2008 at 2:45

---

3 C has a bool type that can only be 0 or 1 since 10 years now. `_Bool` – [Johannes Schaub - litb](#) Dec 23, 2008 at 0:54

---

26 They are still no the same even for bools because they do not short circuit – [aaronman](#) Jul 30, 2013 at 20:44

---

6 Yeah, I strongly believe this should not be the accepted answer until you change "When the domain of numbers you operate on is just 1 and 0, then they are exactly the same" to "When the domain of numbers you operate on is just 1 and

0, the only difference is that the bitwise operators do not short-circuit". The former statement is flatly wrong, so in the meanwhile I have downvoted this answer since it contains that sentence. – [codetaku](#) Apr 4, 2017 at 14:45

---



40



Two main reasons. In short, consider carefully; there could be a good reason for it, but if there is be VERY explicit in your comments because it can be brittle and, as you say yourself, people aren't generally used to seeing code like this.

## Bitwise xor != Logical xor (except for 0 and 1)

Firstly, if you are operating on values other than `false` and `true` (or `0` and `1`, as integers), the `^` operator can introduce behavior not equivalent to a logical xor. For example:

```
int one = 1;
int two = 2;

// bitwise xor
if (one ^ two)
{
    // executes because expression = 3 and any non-zero
}

// logical xor; more correctly would be coded as
// if (bool(one) != bool(two))
// but spelled out to be explicit in the context of th
if ((one && !two) || (!one && two))
{
    // does not execute b/c expression = ((true && false
```

```
// which evaluates to false  
}
```

Credit to user @Patrick for expressing this first.

## Order of operations

Second, `|`, `&`, and `^`, as bitwise operators, do not short-circuit. In addition, multiple bitwise operators chained together in a single statement -- even with explicit parentheses -- can be reordered by optimizing compilers, because all 3 operations are normally commutative. This is important if the order of the operations matters.

In other words

```
bool result = true;  
result = result && a() && b();  
// will not call a() if result false, will not call b()
```

will not always give the same result (or end state) as

```
bool result = true;  
result &= (a() & b());  
// a() and b() both will be called, but not necessarily  
// optimizing compiler
```

This is especially important because you may not control methods `a()` and `b()`, or somebody else may come along and change them later not understanding the

dependency, and cause a nasty (and often release-build only) bug.

Share Improve this answer

edited Oct 6, 2008 at 19:03

Follow


answered Aug 29, 2008 at 5:15



Patrick Johnmeyer

32.5k ● 2 ● 28 ● 26

- 
- 1 Not really a fair comparison if you cast to bool in one case and not in the other case... The cast to bool in both cases is what would make it work, and why it is fragile (because you have to remember it). – [Greg Rogers](#) Oct 6, 2008 at 14:31
- 

The explanation of short-circuiting is why this should absolutely be the accepted answer; Patrick's answer (err... the other Patrick, who is just named Patrick) is completely wrong in saying "When the domain of numbers you operate on is just 1 and 0, then they are exactly the same"  
– [codetaku](#) Apr 4, 2017 at 14:47 

---



15



The raised eyebrows should tell you enough to stop doing it. You don't write the code for the compiler, you write it for your fellow programmers first and then for the compiler. Even if the compilers work, surprising other people is not what you want - bitwise operators are for bit operations not for bools.

I suppose you also eat apples with a fork? It works but it surprises people so it's better not to do it.

Share Improve this answer

answered Aug 23, 2008 at 20:00

Follow



[kokos](#)

43.6k ● 5 ● 37 ● 32

3 Yes, this should be the top answer. Follow the principle of least surprise. Code that does unusual things is harder to read and understand, and code is read far more often than it is written. Don't use cutesy tricks like this. – [Eloff](#) Apr 26, 2014 at 16:07

1 I'm here because my colleague used bitwise "and" operator for bool. And now I'm spending my time trying to understand if this is correct or not. If he didn't do this I would be doing something more useful now `__`. If you value time of your colleagues PLEASE don't use bitwise operators for bool! – [anton\\_rh](#) Dec 21, 2017 at 12:50



I think

12

```
a != b
```



is what you want



Share Improve this answer

answered Aug 29, 2008 at 1:56

Follow




[Mark Borgerding](#)

8,426 ● 4 ● 33 ● 52

1 This is true, +1. But there is no assigning version of this operator ( `!==` so to speak) so if you are computing the XOR of a sequence of `bool` values you would need to write `acc = acc != condition(i);` in a loop body. The compiler probably can handle this as efficiently as if `!==` existed, but some might find that it does not look nice, and prefer the

## 10 Disadvantages of the bitlevel operators.

 You ask:



“Is there any reason not to use the bitwise operators `&`, `|`, and `^` for "bool" values in C++? ”

Yes, the **logical operators**, that is the built-in high level boolean operators `!`, `&&` and `||`, offer the following advantages:

- Guaranteed **conversion of arguments** to `bool`, i.e. to `0` and `1` ordinal value.
- Guaranteed **short circuit evaluation** where expression evaluation stops as soon as the final result is known.  
This can be interpreted as a tree-value logic, with *True*, *False* and *Indeterminate*.
- Readable textual equivalents `not`, `and` and `or`, even if I don't use them myself.

As reader Antimony notes in a comment also the bitlevel operators have alternative tokens, namely



`bitand`, `bitor`, `xor` and `compl`, but in my opinion these are less readable than `and`, `or` and `not`.

Simply put, each such advantage of the high level operators is a disadvantage of the bitlevel operators.

In particular, since the bitwise operators lack argument conversion to 0/1 you get e.g. `1 & 2` → `0`, while `1 && 2` → `true`. Also `^`, bitwise exclusive or, can misbehave in this way. Regarded as boolean values 1 and 2 are the same, namely `true`, but regarded as bitpatterns they're different.

---

## How to express logical *either/or* in C++.

You then provide a bit of background for the question,

“I sometimes run into situations where I want exactly one of two conditions to be true (XOR), so I just throw the `^` operator into a conditional expression.”

Well, the bitwise operators have **higher precedence** than the logical operators. This means in particular that in a mixed expression such as

```
a && b ^ c
```

you get the perhaps unexpected result `a && (b ^ c)`.

Instead write just

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

expressing more concisely what you mean.

For the multiple argument *either/or* there is no C++ operator that does the job. For example, if you write `a ^ b ^ c` than that is not an expression that says “either `a`, `b` or `c` is true”. Instead it says, “An odd number of `a`, `b` and `c` are true”, which might be 1 of them or all 3...

To express the general either/or when `a`, `b` and `c` are of type `bool`, just write

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

or, with non-`bool` arguments, convert them to `bool`:

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

---

## Using `&=` to accumulate boolean results.

You further elaborate,

“I also need to accumulate Boolean values sometimes, and `&=` and `|=?` can be quite useful.”

Well, this corresponds to checking whether respectively *all* or *any* condition is satisfied, and **de Morgan's law** tells you how to go from one to the other. I.e. you only need one of them. You could in principle use `*=` as a `&&=` - operator (for as good old George Boole discovered, logical AND can very easily be expressed as multiplication), but I think that that would perplex and perhaps mislead maintainers of the code.

Consider also:

```
struct Bool
{
    bool    value;

    void operator&=( bool const v ) { value = value &&
operator bool() const { return value; }
};

#include <iostream>

int main()
{
    using namespace std;

    Bool a  = {true};
    a &= true || false;
    a &= 1234;
    cout << boolalpha << a << endl;

    bool b = {true};
    b &= true || false;
```

```
b &= 1234;  
cout << boolalpha << b << endl;  
}
```

Output with Visual C++ 11.0 and g++ 4.7.1:

```
true  
false
```

The reason for the difference in results is that the `bitlevel` `&=` does not provide a conversion to `bool` of its right hand side argument.

So, which of these results do you desire for your use of `&=` ?

If the former, `true`, then better define an operator (e.g. as above) or named function, or use an explicit conversion of the right hand side expression, or write the update in full.

Share Improve this answer

edited May 23, 2014 at 12:19

Follow

answered Dec 21, 2012 at 7:56



Cheers and hth. - Alf

145k ● 15 ● 214 ● 339

---

bitwise operators also have text equivalents – Antimony May 23, 2014 at 9:31

---

@Antimony: I didn't understand that comment at first but yes, the bitlevel operations have alternative tokens `bitand`,

`bitor`, `xor` and `compl`. I think that's why I used the qualification "readable". Of course, the readability of e.g. `compl 42` is subjective. ;-) – [Cheers and hth. - Alf](#) May 23, 2014 at 11:29

---

As far as I can tell logical operators *can* be overloaded with other argument types (though they usually are not), so the first point "guaranteed conversion of arguments to `bool`" is just a consequence of convention, not a guarantee that the C++ language actually gives you. – [Marc van Leeuwen](#) Apr 28, 2016 at 9:18

---

@MarcvanLeeuwen: Possibly your visual subsystem failed to notice "the built-in high level boolean operators `!`, `&&` and `||`". You're right that these operators can be overloaded, and then a main problem is that the semantics change subtly, no longer short circuit evaluation. But that's a problem with overloads, not with the built-in operators.  
– [Cheers and hth. - Alf](#) Apr 28, 2016 at 9:25 ✎

---

That is right (I did miss that). But then the statement is still misleading, because you are just saying that those operators take boolean arguments; if (and only if) the built-in versions of the operators get selected, then the compiler will have inserted a conversion of the arguments. Similarly the built-in (high level) unsigned integer addition operator `+` guarantees conversion of its arguments to `unsigned`, but that does not prevent `unsigned int n=-1; n+=3.14;` from actually using the `double` (or is it `float` ?) addition operation. (Compare your `&=` example.) – [Marc van Leeuwen](#) Apr 28, 2016 at 9:47 ✎

---



3

Contrary to Patrick's answer, C++ has no `^^` operator for performing a short-circuiting exclusive or. If you think about it for a second, having a `^^` operator wouldn't make sense anyway: with exclusive or, the result always



depends on both operands. However, Patrick's warning about non-`bool` "Boolean" types holds equally well when comparing `1 & 2` to `1 && 2`. One classic example of this is the Windows [GetMessage\(\)](#) function, which returns a tri-state `BOOL`: nonzero, `0`, or `-1`.

Using `&` instead of `&&` and `|` instead of `||` is not an uncommon typo, so if you are deliberately doing it, it deserves a comment saying why.

Share Improve this answer

answered Sep 16, 2008 at 5:54

Follow



[bk1e](#)

24.3k ● 6 ● 57 ● 65

- 
- 7 A `^^` operator would still be useful regardless of the short-circuit consideration. It would a) evaluate the operands in a boolean context and b) guarantee to return a 1 or 0.  
– [Craig McQueen](#) Feb 21, 2011 at 0:37

---

@CraigMcQueen. It is simple to define `inline bool XOR(bool a, bool b) { return a!=b; }` and get what you want, except that it is a function rather than an (infix) operator. Or you could directly use `!=` or overload some other operator with this meaning, but then of course you need to be very careful to not accidentally end up using an unintended overload of the same name. And incidentally `||` and `&&` return `true` or `false`, not `1` or `0`.  
– [Marc van Leeuwen](#) Apr 28, 2016 at 8:50

---

And it seems that the fact that `!`, `||` and `&&` evaluate their arguments in a Boolean context is only because these operators are rarely overloaded to accept other types; as far as I can tell the language does allow such overloads, and therefore does not guarantee evaluation of arguments in a



2



Patrick made good points, and I'm not going to repeat them. However might I suggest reducing 'if' statements to readable english wherever possible by using well-named boolean vars. For example, and this is using boolean operators but you could equally use bitwise and name the bools appropriately:

```
bool onlyAIsTrue = (a && !b); // you could use bitwise
bool onlyBIsTrue = (b && !a); // and not need this sec
if (onlyAIsTrue || onlyBIsTrue)
{
    .. stuff ..
}
```

You might think that using a boolean seems unnecessary, but it helps with two main things:

- Your code is easier to understand because the intermediate boolean for the 'if' condition makes the intention of the condition more explicit.
- If you are using non-standard or unexpected code, such as bitwise operators on boolean values, people can much more easily see why you've done this.

EDIT: You didnt explicitly say you wanted the conditionals for 'if' statements (although this seems most likely), that was my assumption. But my suggestion of an intermediate boolean value still stands.

Share Improve this answer

answered Aug 24, 2008 at 17:42

Follow



2

Using bitwise operations for bool helps save unnecessary branch prediction logic by the processor, resulting from a 'cmp' instruction brought in by logical operations.



Replacing the logical with bitwise operations (where all operands are bool) generates more efficient code offering the same result. The efficiency ideally should outweigh all the short-circuit benefits that can be leveraged in the ordering using logical operations.

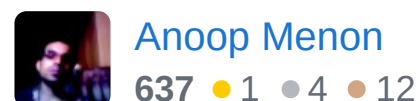


This can make code a bit un-readable albeit the programmer should comment it with reasons why it was done so.

Share Improve this answer

answered Apr 26, 2018 at 9:34

Follow



0

IIRC, many C++ compilers will warn when attempting to cast the result of a bitwise operation as a bool. You would have to use a type cast to make the compiler happy.



Using a bitwise operation in an if expression would serve the same criticism, though perhaps not by the compiler.



Any non-zero value is considered true, so something like "if (7 & 3)" will be true. This behavior may be acceptable





in Perl, but C/C++ are very explicit languages. I think the Spock eyebrow is due diligence. :) I would append "==" 0" or "!= 0" to make it perfectly clear what your objective was.

But anyway, it sounds like a personal preference. I would run the code through lint or similar tool and see if it also thinks it's an unwise strategy. Personally, it reads like a coding mistake.

Share Improve this answer

answered Aug 29, 2008 at 2:08

Follow



[spoulson](#)

21.6k ● 16 ● 78 ● 102

---

Your post motivated me to test this and gcc did not warn! Bummer, because I was going to use that warning to justify letting me run `&=` to accumulate bool results, trusting that others would see the warning if they later change my test. My code would fail for even integers, since they evaluate to 0/false - without warnings! Time to refactor... – [sage](#) Jun 22, 2013 at 18:37

---