3.8 — Bitwise operators

BY ALEX ON JUNE 17TH, 2007 | LAST MODIFIED BY ALEX ON FEBRUARY 14TH, 2018

Note: Many people find this lesson challenging. If you get stuck, skip the lesson (and the next one) and come back later. This information is here for your knowledge, but is not required to progress with the tutorials.

Bit manipulation operators manipulate individual bits within a variable.

Why bother with bitwise operators?

In the past, memory was extremely expensive, and computers did not have much of it. Consequently, there were incentives to make use of every bit of memory available. Consider the bool data type -- even though it only has two possible values (true and false), which can be represented by a single bit, it takes up an entire byte of memory! This is because variables need unique addresses, and memory can only be addressed in bytes. The bool uses 1 bit and the other 7 go to waste.

Using bitwise operators, it is possible to write functions that allow us to compact 8 booleans into a single byte-sized variable, enabling significant memory savings at the expense of more complex code. In the past, this was a good trade-off. Today, at least for application programming, it is probably not.

Now memory is significantly cheaper, and programmers have found that it is often a better idea to code what is easiest to understand and maintain than what is most efficient. Consequently, bitwise operators have somewhat fallen out of favor, except in certain circumstances where maximum optimization is needed (eg. scientific programs that use enormous data sets, games where bit manipulation tricks can be used for extra speed, or embedded programming, where memory is still limited). Nevertheless, it is good to at least know about their existence.

There are 6 bit manipulation operators:

Operator	Symbol	Form	Operation
left shift	<<	x << y	all bits in x shifted left y bits
right shift	>>	x >> y	all bits in x shifted right y bits
bitwise NOT	~	~x	all bits in x flipped
bitwise AND	&	x & y	each bit in x AND each bit in y
bitwise OR	|	x | y	each bit in x OR each bit in y
bitwise XOR	^	x ^ y	each bit in x XOR each bit in y

Bit manipulation is one of the few cases where you should unambiguously use *unsigned* integer data types. This is because C++ does not guarantee how signed integers are stored, nor how some bitwise operators apply to signed variables.

Rule: When dealing with bit operators, use unsigned integers.

Bitwise left shift (<<) and bitwise right shift (>>) operators

Note: In the following examples, we will generally be working with 4-bit binary values. This is for the sake of convenience and keeping the examples simple. In C++, the number of bits used will be based on the size of the data type (8 bits per byte).

The bitwise left shift (<<) operator shifts bits to the left. The left operand is the expression to shift, and the right operand is an integer number of bits to shift by. So when we say `3 << 1`, we are saying "shift the bits in the literal 3 left by 1 place".

For example, consider the number 3, which is binary 0011:

3 = 0011
3 << 1 = 0110 = 6
3 << 2 = 1100 = 12
3 << 3 = 1000 = 8

Note that in the third case, we shifted a bit off the end of the number! Bits that are shifted off the end of the binary number are lost forever.

(Reminder: We're working with 4-bit values here. With an 8-bit value, 3 << 3 would be 0001 1000, which is decimal 24. In this case, the 4th bit wouldn't be shifted off the left end of the binary number).

The bitwise right shift (>>) operator shifts bits to the right.

12 = 1100
12 >> 1 = 0110 = 6
12 >> 2 = 0011 = 3
12 >> 3 = 0001 = 1

Note that in the third case we shifted a bit off the right end of the number, so it is lost.

Although our examples above involve shifting literals, you can shift variables as well:

```
unsigned int x = 4;
x = x << 1; // x will be 8
```

Note that the results of applying the bitwise shift operators to a signed integer are compiler dependent.

What!? Aren't operator<< and operator>> used for input and output?

They sure are.

Programs today typically do not make much use of the bitwise left and right shift operators to shift bits. Rather, you tend to see the bitwise left shift operator used with std::cout to output text. Consider the following program:

```
#include <iostream>

int main()
{
    unsigned int x = 4;
    x = x << 1; // use operator<< for left shift
    std::cout << x; // use operator<< for output

    return 0;
}
```

This program prints:

8

In the above program, how does operator<< know to shift bits in one case and output x in another case? The answer is that std::cout has provided its own version of the << operator that gives it a new meaning when used in conjunction with std::cout. This process is called **operator overloading**. When the compiler sees that the left operand of operator<< is std::cout, it knows that it should call the version of operator<< that std::cout overloaded to do output. If the left operand is an integer type, then the standard version of operator<< is called, which does its usual bit-shifting behavior.

We will talk more about operator overloading in a future section, including discussion of how to override operators for your own purposes.

Bitwise NOT

The bitwise NOT operator (~) is perhaps the easiest to understand of all the bitwise operators. It simply flips each bit from a 0 to a 1, or vice versa. Note that the result of a bitwise NOT is dependent on what size your data type is!

Assuming 4 bits:
4 = 0100
~4 = 1011 = 11 (decimal)

Assuming 8 bits:
4 = 0000 0100
~4 = 1111 1011 = 251 (decimal)

Bitwise AND, OR, and XOR

Bitwise AND (&) and bitwise OR (|) work similarly to their logical AND and logical OR counterparts. However, rather than evaluating a single boolean value, they are applied to each bit! For example, consider the expression `5 | 6`. In binary, this is represented as 0101 | 0110. To do (any) bitwise operations, it is easiest to line the two operands up like this:

```
0 1 0 1 // 5
0 1 1 0 // 6
```

and then apply the operation to each *column* of bits. If you remember, logical OR evaluates to true (1) if either the left or the right or both operands are true (1). Bitwise OR evaluates to 1 if either bit (or both) is 1. Consequently, 5 | 6 evaluates like this:

```
0 1 0 1 // 5
0 1 1 0 // 6
-------
0 1 1 1 // 7
```

Our result is 0111 binary (7 decimal).

We can do the same thing to compound OR expressions, such as `1 | 4 | 6`. If any of the bits in a column are 1, the result of that column is 1.

```
0 0 0 1 // 1
0 1 0 0 // 4
0 1 1 0 // 6
--------
0 1 1 1 // 7
```

1 | 4 | 6 evaluates to 7.

Bitwise AND works similarly. Logical AND evaluates to true if both the left and right operand evaluate to true. Bitwise AND evaluates to true only if both bits in the column are 1. Consider the expression `5 & 6`. Lining each of the bits up and applying an AND operation to each column of bits:

```
0 1 0 1 // 5
0 1 1 0 // 6
--------
0 1 0 0 // 4
```

Similarly, we can do the same thing to compound AND expressions, such as `1 & 3 &7`. If all of the bits in a column are 1, the result of that column is 1.

```
0 0 0 1 // 1
0 0 1 1 // 3
0 1 1 1 // 7
--------
0 0 0 1 // 1
```

The last operator is the bitwise XOR (^), also known as *exclusive or*. When evaluating two operands, XOR evaluates to true (1) if one *and only one* of its operands is true (1). If neither or both are true, it evaluates to 0. Consider the expression `6 ^ 3`:

```
0 1 1 0 // 6
0 0 1 1 // 3
-------
0 1 0 1 // 5
```

It is also possible to evaluate compound XOR expression column style, such as `1 ^ 3 ^ 7`. If there are an even number of 1 bits in a column, the result is 0. If there are an odd number of 1 bits in a column, the result is 1.

```
0 0 0 1 // 1
0 0 1 1 // 3
0 1 1 1 // 7
--------
0 1 0 1 // 5
```

Bitwise assignment operators

As with the arithmetic assignment operators, C++ provides bitwise assignment operators in order to facilitate easy modification of variables.

Operator	Symbol	Form	Operation
Left shift assignment	<<=	x <<= y	Shift x left by y bits
Right shift assignment	>>=	x >>= y	Shift x right by y bits
Bitwise OR assignment	|=	x |= y	Assign x | y to x
Bitwise AND assignment	&=	x &= y	Assign x & y to x
Bitwise XOR assignment	^=	x ^= y	Assign x ^ y to x

For example, instead of writing `x = x << 1;`, you can write `x <<= 1;`.

Summary

Summarizing how to evaluate bitwise operations utilizing the column method:

When evaluating bitwise OR, if any bit in a column is 1, the result for that column is 1.
When evaluating bitwise AND, if all bits in a column are 1, the result for that column is 1.
When evaluating bitwise XOR, if there are an odd number of 1 bits in a column, the result for that column is 1.

Quiz

1) What does 0110 >> 2 evaluate to in binary?
2) What does 5 | 12 evaluate to in decimal?
3) What does 5 & 12 evaluate to in decimal?
4) What does 5 ^ 12 evaluate to in decimal?

Quiz answers

1) **Hide Solution**

0110 >> 2 evaluates to 0001

2) **Hide Solution**

```
5 | 12 =
0 1 0 1
1 1 0 0
--------
1 1 0 1 = 13
```

3) **Hide Solution**

```
5 & 12 =
0 1 0 1
1 1 0 0
--------
0 1 0 0 = 4
```

4) **Hide Solution**

5 ^ 12 =
0 1 0 1
1 1 0 0
----
1 0 0 1 = 9


3.8a -- Bit flags and bit masks


Index


3.7 -- Converting between binary and decimal

Share this:

C++ TUTORIAL | PRINT THIS POST

70 comments to 3.8 — Bitwise operators

« Older Comments 1 2

Aleksandr
May 27, 2018 at 4:36 am · Reply

"Note: Many people find this lesson challenging." And yet you explained the concepts so clearly that it became trivial to understand ☺ Thanks!

Eric
April 27, 2018 at 8:34 pm · Reply

I'd like to add something my high school math teacher taught me back in 1975.

OR turns bits *on*.
AND turns bits *off*.
XOR *toggles* bits.

(Edit: Sorry Alex, I see now you say this exact thing in the next chapter!)

Vu Dang
February 10, 2018 at 6:32 pm · Reply

TYPO:

"With an 8-bit value, 3 << 3 would be 24 because the 16-bit wouldn't be shifted off the end of the binary number.", at the "8-bit" and "16-bit"

Alex
February 14, 2018 at 4:06 pm · Reply

Thanks for pointing this out. Fixed!