O.2 — Bitwise operators

BY ALEX ON JUNE 17TH, 2007 | LAST MODIFIED BY NASCARDRIVER ON JANUARY 16TH, 2020

The bitwise operators

C++ provides 6 bit manipulation operators, often called **bitwise** 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

Author's note

In the following examples, we will largely be working with 4-bit binary values. This is for the sake of convenience and keeping the examples simple. In actual programs, the number of bits used is based on the size of the object (e.g. a 2 byte object would store 16 bits).

For readability, we'll also omit the 0b prefix outside of code examples (e.g. instead of 0b0101, we'll just use 0101).

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

The **bitwise left shift** (<<) operator shifts bits to the left. The left operand is the expression to shift the bits of, and the right operator is an integer number of bits to shift left by.

So when we say x << 1, we are saying "shift the bits in the variable x left by 1 place". New bits shifted in from the right side receive the value 0.

0011 << 1 is 0110
0011 << 2 is 1100
0011 << 3 is 1000

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.

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

1100 >> 1 is 0110
1100 >> 2 is 0011
1100 >> 3 is 0001

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

Here's an example of doing some bit shifting:

```
#include <iostream>
#include <bitset>

int main()
{
    std::bitset<4> x { 0b1100 };

    std::cout << x << '\n';
    std::cout << (x >> 1) << '\n'; // shift right by 1, yielding 0110
    std::cout << (x << 1) << '\n'; // shift left by 1, yielding 1000

    return 0;
}
```

This prints:

```
1100
0110
1000
```

Note that the results of applying the bitwise shift operators to a signed integer are compiler dependent prior to C++20.

Warning

Prior to C++20, don't shift a signed integer (and even then, it's probably still better to use unsigned)

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 { 0b0100 };
    x = x << 1; // use operator<< for left shift
    std::cout << std::bitset<4>(x); // use operator<< for output

    return 0;
}
```

This program prints:

```
1000
```

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 **overloaded** (provided an alternate definition for) operator<< that does console output rather than bit shifting.

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 some other type, then operator<< knows it should do its usual bit-shifting behavior.

The same applies for operator>>.

Note that if you're using operator << for both output and left shift, parenthesization is required:

```
#include <iostream>
#include <bitset>

int main()
{
    std::bitset<4> x{ 0b0110 };

    std::cout << x << 1 << '\n'; // print value of x (0110), then 1
    std::cout << (x << 1) << '\n'; // print x left shifted by 1 (1100)

    return 0;
}
```

This prints:

```
01101
1100
```

The first line prints the value of x (0110), and then the literal 1. The second line prints the value of x left-shifted by 1 (1100).

We will talk more about operator overloading in a future section, including discussion of how to overload 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.

Flipping 4 bits:
~0100 is 1011

Flipping 8 bits:
~0000 0100 is 1111 1011

In both the 4-bit and 8-bit cases, we start with the same number (binary 0100 is the same as 0000 0100 in the same way that decimal 7 is the same as 07), but we end up with a different result.

We can see this in action in the following program:

```
#include <iostream>
#include <bitset>

int main()
{
    std::cout << std::bitset<4>(~0b0100u) << ' ' << std::bitset<8>(~0b0100u);

    return 0;
}
```

This prints:
1011 11111011

Bitwise OR

Bitwise OR (|) works much like its *logical OR* counterpart. However, instead of applying the *OR* to the operands to produce a single result, *bitwise OR* applies to each bit! For example, consider the expression `0b0101 | 0b0110`.

To do (any) bitwise operations, it is easiest to line the two operands up like this:

```
0 1 0 1 OR
0 1 1 0
```

and then apply the operation to each *column* of bits.

If you remember, *logical OR* evaluates to *true (1)* if either the left, right, or both operands are *true (1)*, and *0* otherwise. *Bitwise OR* evaluates to *1* if either the left, right, or both bits are *1*, and *0* otherwise. Consequently, the expression evaluates like this:

```
0 1 0 1 OR
0 1 1 0
-------
0 1 1 1
```

Our result is 0111 binary.

```cpp
#include <iostream>
#include <bitset>

int main()
{
    std::cout << (std::bitset<4>(0b0101) | std::bitset<4>(0b0110));

    return 0;
}
```

This prints:

```
0111
```

We can do the same thing to compound OR expressions, such as `0b0111 | 0b0011 | 0b0001`. If any of the bits in a column are *1*, the result of that column is *1*.

```
0 1 1 1 OR
0 0 1 1 OR
0 0 0 1
--------
0 1 1 1
```

Here's code for the above:

```cpp
#include <iostream>
#include <bitset>

int main()
```

```
{
    std::cout << (std::bitset<4>(0b0111) | std::bitset<4>(0b0011) | std::bitset<4>(0b0001));

    return 0;
}
```

This prints:

```
0111
```

Bitwise AND

Bitwise AND (&) works similarly to the above. *Logical AND* evaluates to true if both the left and right operand evaluate to *true*. *Bitwise AND* evaluates to *true (1)* if both bits in the column are *1*. Consider the expression `0b0101 & 0b0110`. Lining each of the bits up and applying an AND operation to each column of bits:

```
0 1 0 1 AND
0 1 1 0
--------
0 1 0 0
```

```
#include <iostream>
#include <bitset>

int main()
{
    std::cout << (std::bitset<4>(0b0101) & std::bitset<4>(0b0110));

    return 0;
}
```

This prints:

```
0100
```

Similarly, we can do the same thing to compound AND expressions, such as `0b0001 & 0b0011 & 0b0111`. If all of the bits in a column are 1, the result of that column is 1.

```
0 0 0 1 AND
0 0 1 1 AND
0 1 1 1
--------
0 0 0 1
```

```
#include <iostream>
#include <bitset>

int main()
{
    std::cout << (std::bitset<4>(0b0001) & std::bitset<4>(0b0011) & std::bitset<4>(0b0111));

    return 0;
}
```

This prints:

```
0001
```

Bitwise XOR

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 `0b0110 ^ 0b0011`:

```
0 1 1 0 XOR
0 0 1 1
-------
0 1 0 1
```

It is also possible to evaluate compound XOR expression column style, such as `0b0001 ^ 0b0011 ^ 0b0111`. 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 XOR
0 0 1 1 XOR
0 1 1 1
--------
0 1 0 1
```

Bitwise assignment operators

Similar to 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;`.

```cpp
#include <bitset>
#include <iostream>

int main()
{
    std::bitset<4> bits { 0b0100 };
    bits >>= 1;
    std::cout << bits;

    return 0;
}
```

This program prints:

```
0010
```

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.

In the next lesson, we'll explore how these operators can be used in conjunction with bit masks to facilitate bit manipulation.

Quiz time

Question #1

a) What does 0110 >> 2 evaluate to in binary?

Show Solution

b) What does the following evaluate to in binary: 0011 | 0101?

Show Solution

c) What does the following evaluate to in binary: 0011 & 0101?

Show Solution

d) What does the following evaluate to in binary (0011 | 0101) & 1001?

Show Solution

Question #2

A bitwise rotation is like a bitwise shift, except that any bits shifted off one end are added back to the other end. For example 0b1001 << 1 would be 0b0010, but a left rotate by 1 would result in 0b0011 instead. Implement a function that does a left rotate on a std::bitset<4>. For this one, it's okay to use std::bitset<4>::test() and std::bitset<4>::set().

The following code should execute:

```cpp
#include <iostream>
#include <bitset>

std::bitset<4> rotl(std::bitset<4> bits)
{
// Your code here
}

int main()
{
    std::bitset<4> bits1{ 0b0001 };
    std::cout << rotl(bits1) << '\n';

    std::bitset<4> bits2{ 0b1001 };
    std::cout << rotl(bits2) << '\n';

    return 0;
}
```

and print the following:

```
0010
0011
```

Show Solution

Question #3

Extra credit: Redo quiz #2 but don't use the test and set functions.

Show Solution

O.3 -- Bit manipulation with bitwise operators and bit masks

Index

O.1 -- Bit flags and bit manipulation via std::bitset

C++ TUTORIAL | PRINT THIS POST

123 comments to O.2 — Bitwise operators

« Older Comments 1 2

Sam
January 15, 2020 at 4:30 pm · Reply

Grammatical error. "Note that if you're using operator << for both output and left shift, parenthisization is required:". Change "parenthisization" to parenthesization.

nascardriver
January 16, 2020 at 2:07 am · Reply

What a weird word, fixed thanks!

kavin
January 3, 2020 at 5:36 am · Reply

Can i do like this too?

```
std::bitset<4> rotl(std::bitset<4> bits)
{
  if (bits.test(3))
  {
    bits=bits << 1; //0010
    bits.set(0);
    return bits;
  }

  else
    bits=bits <<1;
  return bits;
}

int main()
{
  std::bitset<4> bits1{ 0b0001 };
  std::cout << rotl(bits1) << '\n';

  std::bitset<4> bits2{ 0b1001 };
  std::cout << rotl(bits2) << '\n';

  return 0;
}
```

nascardriver
January 3, 2020 at 6:54 am · Reply

Yes, that's good too!
You don't need the assignment in the `else` case:

```
// ...
else
{
  return (bits << 1);
}
```

XpORPID
November 25, 2019 at 3:38 pm · Reply

I was interested in why this code doesn't work for question 2:

```
1  #include <iostream>
2  #include <bitset>
3
4  std::bitset<4> rotl(std::bitset<4> bits)
5  {
6      if (bits.test(3) == 1)
7      {
8          bits <<= 1;
9          return bits.set(0);
10     }
11     else
12         return <<= 1;
13 }
14
15 int main()
16 {
17     std::bitset<4> bits1{ 0b0001 };
18     std::cout << rotl(bits1) << '\n';
19
20     std::bitset<4> bits2{ 0b1001 };
21     std::cout << rotl(bits2) << '\n';
22
23     return 0;
24 }
```

(12, 10): error C2059: syntax error: '<<='

Also getting unexpected results for this:

```
std::bitset<4> b1{ 0b0010 };
    b1 >>= 1;    //should be 0001
    std::cout << b1 << '\n';
    b1 <<= 2;    //should be 0100
    std::cout << b1 << '\n';
    b1 |= 0010;    //should be 0110
    std::cout << b1 << '\n';
    b1 &= 1010;     //should be 0010
    std::cout << b1 << '\n';
    b1 ^= 1011;     //should be 1001
    std::cout << b1 << '\n';
```

Results I get:
0001
0100
1100
0000
0011

Only first 2 operations give expected results

nascardriver
November 26, 2019 at 1:05 am · Reply

```
return <<= 1;
```

`<<=` is a binary operator, it needs two operands, you only gave one. `<<=` modifies the left operand, but you don't need that. I assume you want

```
return (bits << 1);
```

Also note that `test` returns a `bool`, not an integer.

```
if (bits.test(3))
/* ... */
```

```
0010
1010
1011
```

Those aren't binary numbers, they're missing the `0b` prefix.

XpORPID
November 26, 2019 at 5:04 am · Reply

Oh, even after looking through the first code a couple of times I was still looking at the line 8 when the errors was referring to line 12.
Stupid mistakes by me..

Thank you, nascardriver!

knight
November 21, 2019 at 3:37 am · Reply

Can someone explain and analyze the question 2 answer?

nascardriver
November 21, 2019 at 3:41 am · Reply

```
std::bitset<4> rotl(std::bitset<4> bits)
{
  // Assuming bits = 1100

  // Store the leftmost bit
  bool leftbit{ bits.test(3) }; // leftbit = true

  // Shift all bits to the left
  bits <<= 1; // bits = 1000 (The leftmost bit was lost)

  if (leftbit) // If the leftmost bit was a 1, set the rightmost bit to 1
    bits.set(0); // bits = 1001

  return bits;
}
```

knight
November 21, 2019 at 7:05 am · Reply

Thanks nascardriver. But I am confused about the:

```
bool leftbit{ bits.test(3) };
```

and

```
 if (leftbit) bits.set(0);
```

What is 3 meaning, how leftmost bit 1 become rightmost bit 1.

nascardriver
November 21, 2019 at 7:07 am · Reply

Bits are numbered right to left, starting at 0.

```
bit    0 1 0 0
index  3 2 1 0
```

`bits.test(3)` returns true if the leftmost bit is 1, otherwise it returns false.
`bits.set(0)` sets the rightmost bit to 1. We only do this if the leftmost bit was 1 before the shift.

Ged
October 29, 2019 at 11:22 am · Reply

You are using "u" at the end of the bit numbering. Is it because we are using ~ and it requires "u" (it stands for unsigned type) or something else? Cause other examples don't have it.

```
#include <iostream>
#include <bitset>

int main()
{
    std::cout << std::bitset<4>(~0b0100u) << ' ' << std::bitset<8>(~0b0100u);

    return 0;
}
```

nascardriver
October 29, 2019 at 11:29 am · Reply

The first bit of signed integers is used as the sign bit. If it's 1, the integer is considered to be negative, otherwise it's positive. `0b0100`'s first bit is 0, so `~0b0100`'s first bit is 1. That means that the number is negative, but `std::bitset` doesn't work with negative numbers. The bits of unsigned integers don't have a special meaning, because all unsigned integers are non-negative. `0b0100u` is positive and so is `~0b0100u`.

Vlad dude
October 9, 2019 at 8:04 am · Reply

Bitwise XOR explanation,second paragraph,there's an extra bit inside the third value. Thought I wasn't understanding something at first lol

Alex
October 11, 2019 at 1:01 pm · Reply

Thanks! Fixed.

Mike
October 6, 2019 at 10:55 am · Reply

I just completed Ch.O after finishing up Ch.5 before finally noticing the notification regarding the "disjointed lesson numbers". I know it said it was optional, and I was thinking the whole time, these lessons seem out of place.

So where exactly should Ch.O be?

Alex
October 8, 2019 at 2:26 pm · Reply

The lessons should be completed in the order they are listed on the front page of the site, with O slotting in between 5 and 6. Perhaps I should rename O as lesson 6, and renumber 6, so the intended ordering is clearer. I'll think about doing so when I roll out the update to the current chapter 6.

D
October 6, 2019 at 9:12 am · Reply

Concerning best practices:
return 0; statement missing in Question #2.

Alex
October 8, 2019 at 2:23 pm · Reply

Fixed. Thanks!

MorTaZz
October 4, 2019 at 10:31 am · Reply

hi, i know you are a really busy person but would you take a look at this code and tell me why doesnt it work?

```
std::bitset<4> rotl(std::bitset<4> bits)
{
    if(bits.test(3))
        {
            bits << 1;
            bits.set(0);
        }
    else
        bits << 1;
}
```

Alex
October 4, 2019 at 2:01 pm · Reply

You should be using <<=, not <<. Otherwise bits won't get modified.

Aakash
October 4, 2019 at 4:13 am · Reply

Typo Alert!!
"We can do the same thing to compound OR expressions, such as 0b0111 | 0b011 | 0b0001. If any of the bits in a column are 1, the result of that column is 1.",

There should be 0b0011 instead of 0b011.

Alex
October 4, 2019 at 2:00 pm · Reply

Thanks!

Chris
September 30, 2019 at 5:00 pm · Reply

The solution to question 2 seems kind of lengthy. After all, you just need to return (bits<<1)|(bits>>3) from the rotl function.

Alex
October 2, 2019 at 4:23 pm · Reply

You're right. I added this as an extra credit question/solution and gave you a tip o' the hat. Thanks!

giang
January 15, 2020 at 8:29 pm · Reply

Hi, I tried with the real binary & your solution work really great despite its simplicity. But I still can't find or understand the rule behind this solution. Can you explain more clearly??

nascardriver
January 16, 2020 at 2:30 am · Reply

Let's say we have a number
[/code]
ABCD
[/code]
where A, B, C, and D are bits, eg. 0110

First, we perform the left shift (`bits < < 1`)

```
// ABCD << 1
BCD0
```

We lost the A and got a 0 on the right. Now we perform the right shift (`bits >> 3`) (On the original ABCD)

```
// ABCD >> 3
000A
```

We lost everything apart from the A, which is now the least significant bit.

All we have to do now is combine the two results

```
// BCD0 | 000A
```

```
BCDA
```

Mathew
August 23, 2019 at 12:20 pm · Reply

Sorry if it seems like I'm being nitpicky here.

On the first 'Author notes', 0b is mentioned as a suffix, and
it took me a while to understand that it was meant to be a prefix.

"For readability, we'll also omit the 0b suffix outside of code
examples (e.g. instead of 0b0101, we'll just use 0101)."

Thanks for the tutorial!

Alex
August 24, 2019 at 1:14 pm · Reply

-suffix, +prefix. Thanks for the post-fix!

avidlearner
August 17, 2019 at 5:46 pm · Reply

Isn't there a mistake in the solution to Quiz question 1) d)?

You're doing an AND operation instead of an OR.

HD
August 18, 2019 at 11:51 am · Reply

It is correct, just recheck it.
It has both OR and AND operator, but as OR operator is in parenthesis, it is operated first and then its outcome is operated by AND operator with the rightmost operand.

avidlearner
August 18, 2019 at 1:03 pm · Reply

The first operation is not an OR operation at all, it is an AND. It should be an OR. There is a mistake!

Alex
August 18, 2019 at 4:06 pm · Reply

Yup. Mistake fixed. Thanks for pointing that out!

mansbota
July 6, 2019 at 7:03 pm · Reply

```
char a{ 127 };     // 0111 1111  = 127
a <<= 1;           // 1111 1110  = -126
std::cout << +a << "\n";
```

Shouldn't this print -126, since we're dealing with signed char? I get -2.
EDIT: nvm, just realized that it's getting value from two complement method. Using signed and absolute value, it appears that char's range is from -127 to 127 (255), while unsigned the range is from 0 to 255 (256)

Also, why doesn't this compile:

```
char x{ 0b1111'1110 };  // won't compile
char x{ 0b0111'1110 };  // rightfully get 126
```

nascardriver
July 7, 2019 at 12:33 am · Reply

\`0b1111'1110\` doesn't fit into a char.
Despite you entering a binary representation, you're not directly setting the bits of \`x\`.
Writing \`0b1111'1110\` is not different than writing \`254\`.

mansbota
July 7, 2019 at 10:18 am · Reply

Thanks, got it.

Randle
June 10, 2019 at 4:12 pm · Reply

(EDIT) I just looked at the next lesson and realized this is probably more simply done by using an unsigned char, but this still works. :-)

If anyone wants something to try in terminal to see how bitwise operations work on actual c++ datatypes, we can define a short with a binary value (even though they are saved as ints) and follow along with operations.

```
#include <iostream>

int main()
{
    // short is 2 bytes --> 16 bits
    unsigned short x{0b0011};

    std::cout << x << '\n'; // 0000 0000 0000 0011 (3)
    x<<=14;
    std::cout << x << '\n'; // 1100 0000 0000 0000 (49152)
    x<<=1;
    std::cout << x << '\n'; // 1000 0000 0000 0000 (32768)
    x<<=1;
    std::cout << x << '\n'; // 0000 0000 0000 0000 (0)

    unsigned short b1{0b0111};
    unsigned short b2{0b1100};

    // 0000 0000 0000 0111 | 0000 0000 0000 1100 = 0000 0000 0000 1111 (15)
    std::cout << (b1|b2) << '\n';

    // 0000 0000 0000 0111 & 0000 0000 0000 1100 = 0000 0000 0000 0100 (4)
    std::cout << (b1&b2) << '\n';

    // 0000 0000 0000 0111 ^ 0000 0000 0000 1100 = 0000 0000 0000 1011 (11)
    std::cout << (b1^b2) << '\n';

    return 0;
}
```

NooneAtAll
April 30, 2019 at 8:48 pm · Reply

What happens if we bitwise AND between variables of different sizes?

Like

```
    int manyBytes {5}, result;
    long long loadsBytes {127}, result2;
    bool oneByte {true};
    result = manyBytes & oneByte;
    result2 = manyBytes ^ loadsBytes;
```

Does it start from the beginning of both?
Does it loop through the smaller one?

nascardriver
May 1, 2019 at 12:50 am · Reply

The smaller types are promoted to the larger type.
@oneByte -> int
@manyBytes -> long long

If both types are smaller than an int, they're promoted to an int.

```
short s1{};
short s2{};

s1 & s2; // This returns an int.
```

rohit
January 26, 2019 at 11:28 pm · Reply

int is of 32 bits normally.
so why ~3 is giving ans -4 instead of 4294967292. all 32 bits will be flipped if its is an integer?

nascardriver
January 27, 2019 at 4:09 am · Reply

It's a signed integer. Lesson 3.7

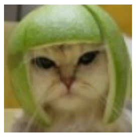
Nathan
January 23, 2019 at 5:01 am · Reply

Did the site go down yesterday?

nascardriver
January 23, 2019 at 9:36 am · Reply

There seems to have been some DNS trouble. If learncpp is down and you want to learn, you can visit snapshots over at archive.org

http://web.archive.org/web/20190119083823/http://www.learncpp.com/



Alex
January 23, 2019 at 9:59 pm · Reply

Yes, it went down quite a few times yesterday. The site apparently ran out of virtual memory and that was causing intermittent issues. I've given the server a reboot and it's been stable since. Sorry for the inconvenience.

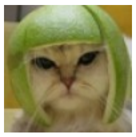
Nathan
January 24, 2019 at 5:25 am · Reply

Thank you, it's running quite well now.

nullptr
December 30, 2018 at 7:05 am · Reply

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

Shouldn't it be overload instead of overrride?



Alex
December 30, 2018 at 11:39 pm · Reply

Yes, thank you. Fixed.

Ajalle Perfej
November 18, 2018 at 6:37 pm · Reply

I'd like to ask your comments on some uses of bitwise operators I found on the internet (link available upon request).

One of them is this form of the postfix initializer and incrementer:

```
int incrementer; //note it's not initialized to any value
-~incrementer //the ~ causes the automatic typing of the uninitialized value as int {-1} and
```

To be fair to the obviously experienced programmer who provides this example for Javascript rather than C++, he calls this use of bitwise operators "evil" because it's compiler abuse just in case you forgot to initialize your incrementer variable.

Another use just punched me in the face, and it's for rounding a float into an integer:

```
auto float_to_int {2.74329}
now_we_have_an_int = float_to_int | 0
```

I figure that shortcut is probably most useful in environments such as embedded programming, but it's my first exposure to memory conservation and I find it stimulating. With regard to C++ or C can you point me to some resources on programming for low-memory environments? I'm happy with just one link where I can learn the basics.

Finally, here's one of my own:

In a very old computer RPG, there were four basic classes: Cleric, Fighter, Magic-User, and Thief. You had a party of six characters, although you could try to win the game with fewer than six. Some races of characters could be two or three classes at the same time. Well, in order to have a reasonable chance at winning the game, you needed 10 character classes among your six characters. There's a way to select those randomly using bitwise logic:

Each character starts with a class value of 0.

Cleric: character | 1
Fighter: character | 2
Magic-User: character | 4
Thief: character | 8

You can code a function that pseudorandomly flips the bits of your six characters until you have your 10 starting character classes, and build your pseudorandom party of six characters that way. It can be really fun just to go with whatever the pseudorandom generator gives you, and there's no way within the game to make pseudorandom character class choices, so this is one way. Neat, eh?

nascardriver
November 21, 2018 at 3:44 am · Reply

Hi Ajalle!

> One of them is this form of the postfix initializer and incrementer [...]
Nope. If @incrementer is uninitialized and not 0, the line after won't magically turn it into a 1. The example works in JS, because uninitialized values in JS are have the value 'undefined'. '~undefined' is -1, so '-~undefined' is 1. Uninitialized variables in C++ have an unspecified value. Always initialize your variables!

> Another use just punched me in the face
Again, this works in JS, but not in C++. floats don't have an @operator|.

Haroon
July 31, 2018 at 1:57 pm · Reply

Is XOR the only bitwise operator that doesn't cause data loss?
Operation reversal can evaluate the original value?

If, yes, I think this should be noted here. If, no, please correct me.

Alex
August 2, 2018 at 9:20 pm · Reply

Bitwise NOT can also be reversed by doing another NOT.

Haroon
August 3, 2018 at 10:21 am · Reply

Yes, logical and simpler.
I think these worth to be noted.
Thanks

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!

Haroon
August 3, 2018 at 10:26 am · Reply

Yes, He did that very well. Thumbs-UP.
I hope Alex can do learnrust.com soon.

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!

« Older Comments 1 2