O.2 — Bitwise operators

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).

The bitwise operators are defined for integral types and std::bitset. We'll use std::bitset in our examples because it's easier to print the output in binary.

Avoid using the bitwise operators with signed operands, as many operators will return implementation-defined results prior to C++20 or have other potential gotchas that are easily avoided by using unsigned operands (or std::bitset).

Best practice

To avoid surprises, use the bitwise operators with unsigned operands or std::bitset.

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 operand 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 <bitset>
#include <iostream>

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
```

For advanced readers

Bit-shifting in C++ is endian-agnostic. Left-shift is always towards the most significant bit, and right-shift towards the least significant bit.

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 (or other stream objects) to output text. Consider the following program:

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

int main()
{
    unsigned int x { 0b0100 };
    x = x << 1; // use operator<< for left shift
    std::cout << std::bitset<4>{ x } << '\n'; // 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 is an integral 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 <bitset>
#include <iostream>

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 chapter, 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 <bitset>
#include <iostream>

int main()
{
	std::cout << ~std::bitset<4>{ 0b0100 } << ' ' << ~std::bitset<8>{ 0b0100 } <<
'\n';

	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.

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

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

	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:

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

int main()
{
	std::cout << (std::bitset<4>{ 0b0111 } | std::bitset<4>{ 0b0011 } |
std::bitset<4>{ 0b0001 }) << '\n';

	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 <bitset>
#include <iostream>

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

        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 <bitset>
#include <iostream>

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

        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;`.

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

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

    return 0;
}
```

This program prints:

```
0010
```

As an aside…

There is no bitwise NOT assignment operator. This is because the other bitwise operators are binary, but bitwise NOT is unary (so what would go on the right-hand side of a `~=` operator?). If you want to flip all of the bits, you can use normal assignment here: `x = ~x;`

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 test() and set().

The following code should execute:

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

// "rotl" stands for "rotate left"
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 (use bitwise operators).

Show Hint

Show Hint

Show Solution