```
filename: c_bit-shift-and-bitwise-operations-to-encode-rgb-values-multif_20230413.txt
https://cboard.cprogramming.com/c-programming/71058-bit-shifting-masking.html

Bit Shifting And Masking

     I have two main questions:
     1. What does the '&' actually do to the number in binary? How do I convert the number back?
     Code:
#define RGB16BIT565(r,g,b)  ((b & 31) + ((g & 63) << 5) + ((r & 31)<<11))

     For example, if I put in:
     Code:
color = RGB16BIT565(255, 0, 0);

     what does the 255 change to?
     2. Is this the proper way to get the original value back?
     Code:
#define RGB32BIT(a,r,g,b) ((b) + ((g) << 8) + ((r) << 16) + ((a) << 24)

color = RGB32BIT(0, 255, 0, 0);

red = (color >> 16) & 255;

     Would this code successfully return the number 255?
     Please don't numb down your answers. I'm not stupid. I have just never been taught.

***
     The binary AND (&) operator takes 2 integers and compares each bit in both numbers. It's probably
     easier to see than read. Let's say you had 2 8-bit numbers and you AND them together, you'd get:

     11000100 (number 1)
     01100101 (number 2)
     -------------
     01000100 (result)

     If the bit at a given position in both numbers is 1, that bit position in the result will also be
     1. Bitwise AND is useful for bit masking. Say you have an 8-bit number and you only want the lower
     4 bits of it...you could do:

     10011101
     00001111
     -------------
     00001101

     By doing that, you make it impossible for the upper 4 bits to result in 1, and the lower 4 bits
     will result in whatever they were in the original number.
     So you can encode all 3 8-bit rgb values into a 32-bit int easily and still have 8-bits left over.
     Say you pass in 3 8-bit values that represent the rgb of a color:

     00111001 (red)
     01101010 (green)
     10100111 (blue)

     If you store blue in the lowest 8 bits of the 32-bit int, green in the next 8 bits, and red in the
     8 bits just above that you can encode it by shifting the red color 16 bits to the left and the
     blue color 8 bits to the left, then bitwise ORing them all together. So you've got:

     00000000 00111001 00000000 00000000 (red << 16)
     00000000 00000000 01101010 00000000 (green << 8)
     00000000 00000000 00000000 10100111 (blue)
     ORing them all together you end up with:
     00000000 00111001 01101010 10100111 (color)

     ORing works like ANDing, but if either bit in the specific location is 1, the result is 1. To
     reverse it and break the color back into rgb values you'd use masking:

     red = (color >> 16) & 255;
     green = (color >> 8) & 255;
     blue = color & 255;

     So basically you're just shifting the 8 bits you want into the lowest 8 bits and then masking off
     everything else. Hopefully that all made sense.



---
https://www.fayewilliams.com/2011/09/21/bitwise-rgba-values/

Bitwise RGBA Values
8 June 2016

   Let's take a look at bit shifting in practice.

   Say we have a variable called colour, that contains an RGBA value. If you have never had any
   experience with graphics, all you need to know is that the colours you see on your screen may be
   represented as a combination of four different variables - red, green, blue and alpha. The alpha
```

value is usually a percentage to describe the opacity, while red, green and blue values are combined to describe the final colour.

RGBA values are usually stored in a single 32 bit integer, with 8 bits used for each component:

```
RRRRRRRR GGGGGGGG BBBBBBBB AAAAAAAA
```

All well and good, but imagine we need to know what the green value is independently of everything else. How can we extract this information? And moreover, how do we get a colour encoded into the variable in the first place?

Setting an RGBA value

Imagine we want to set our colour to a bright yellow, fully opaque. This uses the RGBA components:

```
R) 0xFF
G) 0xCC
B) 0x00
A) 0xFF
```

As binary this looks like:

```
11111111 11001100 00000000 11111111
```

OK, we could set the colour variable using a large number:

```
int colour = 4291559679;
```

but that isn't a very intuitive (or re-usable) solution.

Instead we'll add our components in one at a time using a mask, and shift them to the correct positions:

```
unsigned int colour =
0xFF | (0x00 << 8) | (0xCC << 16) | (0xFF << 24);
```

To fully break down what is happening here, let's look at the binary behind the scenes:

The first step is a bitwise OR on 0xFF with 0x00 shifted left by 8 places:

```
0000 0000 1111 1111 // 0xff
0000 0000 0000 0000 // 0x00 << 8
___________________
0000 0000 1111 1111
```

The next step is a bitwise OR on the result with 0xCC shifted left 16 places:

```
0000 0000 0000 0000 1111 1111 // result
1100 1100 0000 0000 0000 0000 // 0xCC << 16
_____________________________
1100 1100 0000 0000 1111 1111
```

And finally a bitwise OR on the result with 0xFF shifted left 24 places:

```
0000 0000 1100 1100 0000 0000 1111 1111 // result
1111 1111 0000 0000 0000 0000 0000 0000 // 0xFF << 24
_______________________________________
1111 1111 1100 1100 0000 0000 1111 1111 // 0xFFCC00FF, or 4291559679
```

The final result is the number we want to assign to the colour integer.

Extracting an RGBA value

Now say we want to extract that green value from our colour integer. We can simply do the following:

```
int green = (colour & 0x00FF0000) >> 16;
```

What's happening here?

First off, we're masking our colour variable using bitwise AND to effectively "turn off" all the components that we aren't interested in:

```
1111 1111 1100 1100 0000 0000 1111 1111
0000 0000 1111 1111 0000 0000 0000 0000
_______________________________________
0000 0000 1100 1100 0000 0000 0000 0000
```

Then we shift 16 places to the right to put our green component in the first byte:

```
0000 0000 0000 0000 0000 0000 1100 1100 // 0xCC
```

Simple! Now we know how to extract any component we choose by adjusting the mask and number of places shifted accordingly.

Bitwise operator summary

A quick guide to which operator to use when.

Bitwise AND
  * Use with a mask to check if bits are on or off
  * Turn off individual bits

```
   Bitwise OR
     * Turn on individual bits

   Bitwise XOR
     * Toggle bits on and off, like a switch

   Bitwise NOT
     * Turn off individual bits with AND

   Bitwise left and right shift
     * Extract bytes from longer variables
     * Insert bytes into longer variables
     * Multiplication and division by powers of 2 (but be cautious with signed integers, remainders and
       overflow)
```

This is not an exhaustive list, but a basic guide. Have fun with bitwise operators, and if you want more examples and ideas, have a look at this fantastic collection of code snippets from Sean Eron Anderson.

---
https://ryanclark.me/rgb-to-hex-via-binary-shifting/

RGB to Hex via Binary Shifting
April 16, 2015

A colleague of mine presented me with this code, baffled as to how it works.

```
function rgbToHex(r, g, b) {
        return "#" + ((1 << 24) + (r << 16) + (g << 8) + b).toString(16).slice(1);
}
```

We know it converts an RGB value to it's hexadecimal equivalent, but how? It's doing some crazy stuff with binary, but when broken down it's actually incredibly simple and very clever.

Hex Codes

Hex codes are six characters long, the first two being red, middle two being green and the last two being blue. Each character is a hexadecimal number.

```
     R R G G B B
   # 0 0 0 0 0 0
```

Hexadecimal Numbers

A hex value can go from 0 to F - a maximum of 16 different values. When putting two together, we can now have a maximum of 256 (16 multiplied by 16) values (0 through to 255). We can represent 0 through to 255 in 8 bits.

```
   0 represented in 8 bits
   128 64  32  16  8   4   2   1
   0   0   0   0   0   0   0   0

   256 represented in 8 bits
   128 64  32  16  8   4   2   1
   1   1   1   1   1   1   1   1
```

This means that red, green and blue can have 256 different variations each - that's 16,581,375 colours we can reference in just 6 characters!

Double Arrow?

If you don't know about bitwise operators, you might be very confused to see a << in JavaScript. Don't worry, what it does is very easy to understand.

The << operator is also known as a left shift. This will shift the value of r 16 bits to the left. You can, at a maximum, shift a number 31 bits to the left.

Let's do an example

The number one, represented as a decimal, is 1. Represented as binary (in 17 bits for the sake of consistency), is -

```
   65536 32768 16384 8192   4096   2048   1024   512   256   128   64   32   16   8   4   2   1
   0     0     0     0      0      0      0      0     0     0     0    0    0    0   0   0   1
```

When we shift 1 16 bits to the left, we're adding 16 0's to the right of the one.

```
   65536 32768 16384 8192   4096   2048   1024   512   256   128   64   32   16   8   4   2   1
   1     0     0     0      0      0      0      0     0     0     0    0    0    0   0   0   0
```

Which as a decimal, is equal to 65536.

That's all it does! Told you it was simple.

Breaking it down

We're going to start with the middle section - the part that deals with converting the red value to hexadecimal.

Red

The code that does this is (r << 16). The red value is always first in a hex code, so in order to make room for green and blue, we shift it 16 bits to the left. This gives us the red value in binary, as well as 16 bits on the end - 8 bits for green and 8 bits for blue.

If we do (255 << 16), we get the binary

111111110000000000000000

Which, when converted to hexadecimal, is equal to ff0000.

Green

Now that we have converted red to binary and left room for green and blue, we can shift the green value 8 bits to the left. The code that does this is (g << 8).

If we shift 255 to the left by 8 bits and represent it in 24 bits, we get this -

000000001111111100000000

Which, as hexadecimal, is equal to 00ff00.

If you compare that table and the table in the red section, you can see that they line up perfectly - we can add the two values together without them conflicting.

Take the red value

111111110000000000000000

add the green value

000000001111111100000000

and we get

111111111111111100000000

When this is converted to hexadecimal, it is ffff00 - the correct representation of rgb(255, 255, 0) as a hex.

Blue

We don't need to shift our blue value to the left, because we are taking up the last 8 bits with it. If you look at the original code, you can see it just does + b at the end. This is because when adding a binary number and a decimal number in JavaScript, the decimal will be converted over to binary before it's added.

If our blue value is 255, the binary representation of it is (again, in 24 bits)

000000000000000011111111

Which, when added to our red and green value, outputs

111111111111111111111111

Which is equal to "ffffff" when converted to a hexadecimal! Most of you will know that this colour is in fact white, or rgb(255, 255, 255).

Padding

Wait a minute - what's that (1 << 24) at the start? Why is it needed?

The decimal number 1 shifted 24 bits to the left provides us with the necessary padding for our RGB values in binary. When you convert a decimal number to a binary number, you aren't guaranteed 8 bits back - you will get the amount of bits it takes to represent that number (the reason we got 8 bits back for our 255 values is because it takes 8 bits to represent 255 in binary) - if you convert decimal 0 to binary you will only get one bit back - 0, or if you convert 13 to binary you will get four bits back - 1101.

When we shift 1 to the left 24 bits, we get this -

1000000000000000000000000

The number 1 followed by 24 bits - 8 bits for each of our colour values.

This means that whenever we add the results from converting our RGB values to binary, regardless of how many bits the result is, it'll always add into the correct section of bits because we are shifting the values to the left by either 16, 8 or 0 bits.

Binary to Hexadecimal

We can convert our binary number into a hexadecimal string by using toString and passing it 16 as a value. Alternatively, you can use 2 for binary or 8 for octal.

```
function rgbToHex(r, g, b) {
        // r = 255, g = 255, b = 255
        return "#" +
        (
                (1 << 24)
                        // Value: 16777216 or 1000000000000000000000000
                + (r << 16)
                        // Value: 16711680 or 111111110000000000000000
                        // Total: 33488896 or 1111111110000000000000000
                + (g << 8)
                        // Value: 65280    or 1111111100000000
                        // Total: 33554176 or 1111111111111111100000000
                + b
                        // Value: 255      or 11111111
                        // Total: 33554431 or 1111111111111111111111111
        )
        .toString(16) // "1ffffff"
        .slice(1); // "ffffff"
}

rgbToHex(255, 255, 255); // #ffffff
```

Conclusion
   As you can see, the code used above is very clever but also very daunting to look at if you don't
   understand what's happening underneath the hood. Hopefully this post makes it all a lot clearer!

---
https://www.oreilly.com/library/view/actionscript-cookbook/0596004907/ch03s04.html

3.3. Decoding an RGB Value

Problem
   You want to extract the red, green, and blue components from an RGB value returned by Color.getRGB().

Solution
   Use the bitshift right and bitwise AND operators.

Discussion
   You can extract the red, green, and blue components from the single RGB value returned by
   Color.getRGB( ) using the bitshift right (>>) and bitwise AND (&) operators. You can extract one or
   more of the colors individually as follows:

```
// Create the Color object.
my_color = new Color(myMovieClip);

// Get the current RGB color.
rgb = my_color.getRGB(  );

// rgb contains an RGB color value in decimal form, such as 14501017 (rosy pink),
// which is stored internally as its hex equivalent, such as 0xDD4499.
red   = (rgb >> 16);
green = (rgb >> 8) & 0xFF;
blue  =  rgb & 0xFF;
```

   Although displayed as a decimal number, remember that each color is stored internally in its
   hexadecimal form: 0x RRGGBB. For example, the color value 14501017 (which is rosy pink) is stored
   internally as 0xDD4499. In this format, it is easy to see that the red component is DD in hex (221 in
   decimal), the green component is 44 in hex (68 in decimal), and the blue component is 99 in hex (153
   in decimal).

   The preceding transformation effectively separates a 24-bit value into its three 8-bit components
   (the leftmost eight bits represent red, the middle eight bits represent green, and the rightmost
   eight bits represent blue). The bitshift right operator is used to shift the eight bits of interest
   to the rightmost position. ...

   Get Actionscript Cookbook now with the O'Reilly learning platform.

   O'Reilly members experience books, live events, courses curated by job role, and more from O'Reilly
   and nearly 200 top publishers.

---
https://www.appsloveworld.com/java/100/159/bit-shift-and-bitwise-operations-to-encode-rgb-values

Bit Shift and Bitwise operations to encode RGB values-Java

   score:11
   Accepted answer

```
        int blueMask = 0xFF0000, greenMask = 0xFF00, redMask = 0xFF;
        int r = 12, g = 13, b = 14;
        int bgrValue = (b << 16) + (g << 8) + r;
        System.out.println("blue:" + ((bgrValue & blueMask) >> 16));
        System.out.println("red:" + ((bgrValue & redMask)));
        System.out.println("green:" + ((bgrValue & greenMask) >> 8));
```

   score:1
   If you simply want to/from RGB conversion and don't care how I would suggest using java.awt.Color

```
int r = 255; //red
int g = 255; //green
int b = 255; //blue
int a = 255; //alpha
Color c = new Color(r,g,b,a);
```

The using the getRGB method and getRed, getBlue, getGreen methods

```
int RGB = c.getRGB();
int red = c.getRed();
int blue = c.getBlue();
int green = c.getGreen();
```

Alternatively you can construct a color object using the Color(r,g,b) constructor, it will have default 255 alpha.

With bit operations (ARGB, 32 bit colorspace). Constructing the RGB color:

```
int alpha = 255;
int red = 128;
int green = 128;
int blue = 128;
int RGB = (alpha << 24);
RGB = RGB | (red << 16);
RGB = RGB | (green << 8);
RGB = RGB | (blue);

System.out.println(Integer.toBinaryString(RGB));
```

```
Out 11111111 10000000 10000000 10000000
```

Decoding is done as in the link in the comment.

score:1
Here is a mock program that I've done up that might assist you. I approached the conversion much like Dev Blanked based off an old program I did, but he answered while I was putting the program together. Since I did the work anyways, figured I'd share in case it helped in any way.

```
import java.util.Scanner;
import java.math.*;

public class RGB{

        public static void main(String[]args){
                Scanner scan = new Scanner(System.in);
                int code; //Code for the color
                int red, green, blue; //Individual colors
                int rMask = 0xFF0000, gMask = 0xFF00, bMask = 0xFF; //Masks for the colors

                //Take input
                System.out.println("Please enter the red color. Range [0, 255] only please.");
                red = scan.nextInt();
                System.out.println("Please enter the green color. Range [0, 255] only please.");
                green = scan.nextInt();
                System.out.println("Please enter the blue color. Range [0, 255] only please.");
                blue = scan.nextInt();

                //Generate code based on Behnil's way.
                code = 0;
                code += (int) (red * Math.pow(2, 16));
                code += (int) (green * Math.pow(2, 8));
                code += (int) (blue * Math.pow(2,0));
                System.out.println("The code is " + code + ".");

                //Clear values
                red = 0;
                green = 0;
                blue = 0;

                //Obtain values.
                red = (code & rMask) >> 16;
                green = (code & gMask) >> 8;
                blue = (code & bMask);

                System.out.println("Your red value is: " + red);
                System.out.println("Your green value is: " + green);
                System.out.println("Your blue value is: " + blue);
        }

}
```

score:1

```
public static void main(String[] args){
        int red = 111;
        int green = 222;
        int blue = 121;

        int code = red*256*256 + green*256 + blue;

        blue = code%256;
        green = (code%(256*256) - blue)/256;
        red = (code - blue - green*256)/(256*256);

        System.out.println("" + red + green + blue);
}
```

this outputs 111222121 as intended. This is the way i fixed it but i am not sure if the pro's agree with this as it might be slower than using bitshifts

---
filename: c_bitwise-operators-in-c-cpp_20220304.txt
https://www.geeksforgeeks.org/bitwise-operators-in-c-cpp/

Bitwise Operators in C/C++
15 Nov, 2021

In C, the following 6 operators are bitwise operators (work at bit-level)

Operators in C

	Operator	Type
Unary uperator ------->	++, --	Unary operator
Binary operator --+	+,-,*,/,%	Arithmetic operator
	<,<=,>,>=,==, !=	Relational operator
	&&,||,!	Logical operator
	&,|,<<,>>,⍰,^	Bitwise operator
	=,+=,-=,*=,/=,%=	Assignement operator
Ternary operator ----->	?:	Ternary or conditional operator

1. The & (bitwise AND) in C or C++ takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.

2. The | (bitwise OR) in C or C++ takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.

3. The ^ (bitwise XOR) in C or C++ takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.

4. The << (left shift) in C or C++ takes two numbers, left shifts the bits of the first operand, the second operand decides the number of places to shift.

5. The >> (right shift) in C or C++ takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift.

6. The ⍰ (bitwise NOT) in C or C++ takes one number and inverts all bits of it

Example:

C++
```
#include <iostream>
using namespace std;

int main() {
        // a = 5(00000101), b = 9(00001001)
        int a = 5, b = 9;

        // The result is 00000001
        cout<<"a = " << a <<","<< " b = " << b <<endl;
        cout << "a & b = " << (a & b) << endl;

        // The result is 00001101
        cout << "a | b = " << (a | b) << endl;

        // The result is 00001100
        cout << "a ^ b = " << (a ^ b) << endl;

        // The result is 11111010
        cout << "⍰(" << a << ") = " << (⍰a) << endl;

        // The result is 00010010
        cout<<"b << 1" <<" = "<< (b << 1) <<endl;

        // The result is 00000100
        cout<<"b >> 1 "<<"= " << (b >> 1 )<<endl;

        return 0;
}

// This code is contributed by sathiyamoorthics19
```

C
```
// C Program to demonstrate use of bitwise operators
#include <stdio.h>

int main() {
        // a = 5(00000101), b = 9(00001001)
        unsigned char a = 5, b = 9;

        // The result is 00000001
        printf("a = %d, b = %d\n", a, b);
        printf("a&b = %d\n", a & b);

        // The result is 00001101
        printf("a|b = %d\n", a | b);

        // The result is 00001100
        printf("a^b = %d\n", a ^ b);

        // The result is 11111010
        printf("⍰a = %d\n", a = ⍰a);

        // The result is 00010010
        printf("b<<1 = %d\n", b << 1);

        // The result is 00000100
        printf("b>>1 = %d\n", b >> 1);

        return 0;
}
```

Output:
```
a = 5, b = 9
a&b = 1
a|b = 13
a^b = 12
⍰a = 250
b<<1 = 18
b>>1 = 4
```

Interesting facts about bitwise operators

1. The left shift and right shift operators should not be used for negative numbers. If the second operand(which decides the number of shifts) is a negative number, it results in undefined behaviour in C. For example results of both 1 <<- 1 and 1 >> -1 is undefined. Also, if the number is shifted more than the size of the integer, the behaviour is undefined. For example, 1 << 33 is undefined if integers are stored using 32 bits. Another thing is, NO shift operation is performed if additive-expression(operand that decides no of shifts) is 0. See this for more details. Note: In C++, this behavior is well-defined.
2. The bitwise XOR operator is the most useful operator from a technical interview perspective. It is used in many problems. A simple example could be "Given a set of numbers where all elements occur even a number of times except one number, find the odd occurring number" This problem can be efficiently solved by just doing XOR of all numbers.

C++
```
#include <iostream>
using namespace std;

// Function to return the only odd
// occurring element
int findOdd(int arr[], int n) {
        int res = 0, i;
        for (i = 0; i < n; i++) {
                res ^= arr[i];
        }
        return res;
}

// Driver Method
int main(void) {
        int arr[] = { 12, 12, 14, 90, 14, 14, 14 };
        int n = sizeof(arr) / sizeof(arr[0]);
        cout << "The odd occurring element is  "<< findOdd(arr, n);
        return 0;
}

// This code is contributed by shivanisinghss2110
```

C
```
#include <stdio.h>

        // Function to return the only odd
        // occurring element
        int findOdd(int arr[], int n) {
                int res = 0, i;
                for (i = 0; i < n; i++) {
                        res ^= arr[i];
                }
                return res;
        }

        // Driver Method
        int main(void) {
                int arr[] = { 12, 12, 14, 90, 14, 14, 14 };
                int n = sizeof(arr) / sizeof(arr[0]);
                printf("The odd occurring element is %d ", findOdd(arr, n));
                return 0;
        }
```

   Output:

```
The odd occurring element is 90
```

1. The following are many other interesting problems using XOR operator.
    1. Find the Missing Number
    2. swap two numbers without using a temporary variable
    3. A Memory Efficient Doubly Linked List
    4. Find the two non-repeating elements.
    5. Find the two numbers with odd occurences in an unsorted-array.
    6. Add two numbers without using arithmetic operators.
    7. Swap bits in a given number/.
    8. Count number of bits to be flipped to convert a to b .
    9. Find the element that appears once.
    10. Detect if two integers have opposite signs.
2. The bitwise operators should not be used in place of logical operators. The result of logical operators (&&, || and !) is either 0 or 1, but bitwise operators return an integer value. Also, the logical operators consider any non-zero operand as 1. For example, consider the following program, the results of & and && are different for same operands.

C++

```
#include <iostream>
using namespace std;

int main() {
        int x = 2, y = 5;
        (x & y) ? cout <<"True " : cout <<"False ";
        (x && y) ? cout <<"True " : cout <<"False ";
        return 0;
}

// This code is contributed by shivanisinghss2110
```

C

```
#include <stdio.h>

int main() {
        int x = 2, y = 5;
        (x & y) ? printf("True ") : printf("False ");
        (x && y) ? printf("True ") : printf("False ");
        return 0;
}
```

   Output:

```
False True
```

   1.The left-shift and right-shift operators are equivalent to multiplication and division by 2 respectively. As mentioned in point 1, it works only if numbers are positive.

C++

```
#include <iostream>
using namespace std;

int main() {

        int x = 19;
        cout<<"x << 1 = "<< (x << 1) <<endl;
        cout<<"x >> 1 = "<< (x >> 1) <<endl;
        return 0;
}

// This code is contributed by sathiyamoorthics19
```

C

```
#include <stdio.h>

int main() {
        int x = 19;
        printf("x << 1 = %d\n", x << 1);
        printf("x >> 1 = %d\n", x >> 1);
        return 0;
}
```

Output:

```
x << 1 = 38
x >> 1 = 9
```

2.The & operator can be used to quickly check if a number is odd or even. The value of expression (x & 1) would be non-zero only if x is odd, otherwise the value would be zero.

C++

```
#include <iostream>
using namespace std;

int main() {

        int x = 19 ;
        (x & 1) ? cout<<"Odd" : cout<< "Even" ;

        return 0;
}

// This code is contributed by sathiyamoorthics19
```

C

```
#include <stdio.h>

int main() {
        int x = 19;
        (x & 1) ? printf("Odd") : printf("Even");
        return 0;
}
```

Output:

```
Odd
```

3.The ~ operator should be used carefully. The result of ~ operator on a small number can be a big number if the result is stored in an unsigned variable. And the result may be a negative number if the result is stored in a signed variable (assuming that the negative numbers are stored in 2's complement form where the leftmost bit is the sign bit)

C++

```
#include <iostream>
using namespace std;

int main() {

        unsigned int x = 1;
        signed int a = 1;
        cout<<"Signed Result "<< ~a <<endl ;
        cout<<"Unsigned Result "<< ~x ;
        return 0;
}

// This code is contributed by sathiyamoorthics19
```

C

```
// Note that the output of the following
// program is compiler dependent
#include <stdio.h>

int main() {
        unsigned int x = 1;
        printf("Signed Result %d \n", ~x);
        printf("Unsigned Result %ud \n", ~x);
        return 0;
}
```

Output:

```
Signed Result -2
Unsigned Result 4294967294d
```

---
filename: c_bitwise-operators_20220304.txt
https://www.guru99.com/c-bitwise-operators.html

Bitwise Operators in C: AND, OR, XOR, Shift & Complement
December 25, 2021

What are Bitwise Operators?

Bitwise Operators are used for manipulating data at the bit level, also called bit level programming. Bitwise operates on one or more bit patterns or binary numerals at the level of their individual bits. They are used in numerical computations to make the calculation process faster.

Following is the list of bitwise operators provided by 'C' programming language:

Operator	Meaning
&	Bitwise AND operator
|	Bitwise OR operator
^	Bitwise exclusive OR operator
⍰	Binary One's Complement Operator is a unary operator
<<	Left shift operator
>>	Right shift operator

Bitwise operators cannot be directly applied to primitive data types such as float, double, etc. Always remember one thing that bitwise operators are mostly used with the integer data type because of its compatibility.

The bitwise logical operators work on the data bit by bit, starting from the least significant bit, i.e. LSB bit which is the rightmost bit, working towards the MSB (Most Significant Bit) which is the leftmost bit.

The result of the computation of bitwise logical operators is shown in the table given below.

x	y	x & y	x | y	x ^ y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Bitwise AND

This is one of the most commonly used logical bitwise operators. It is represented by a single ampersand sign (&). Two integer expressions are written on each side of the (&) operator.
The result of the bitwise AND operation is 1 if both the bits have the value as 1; otherwise, the result is always 0.

Let us consider that we have 2 variables op1 and op2 with values as follows:

```
Op1 = 0000 1101
Op2 = 0001 1001
```

The result of the AND operation on variables op1 and op2 will be

```
Result = 0000 1001
```

As we can see, two variables are compared bit by bit. Whenever the value of a bit in both the variables is 1, then the result will be 1 or else 0.

Bitwise OR

It is represented by a single vertical bar sign (|). Two integer expressions are written on each side of the (|) operator.

The result of the bitwise OR operation is 1 if at least one of the expression has the value as 1; otherwise, the result is always 0.

Let us consider that we have 2 variables op1 and op2 with values as follows:

```
Op1 = 0000 1101
Op2 = 0001 1001
```

The result of the OR operation on variables op1 and op2 will be

```
Result = 0001 1101
```

As we can see, two variables are compared bit by bit. Whenever the value of a bit in one of the variables is 1, then the result will be 1 or else 0.

Bitwise Exclusive OR

It is represented by a symbol (^). Two integer expressions are written on each side of the (^) operator.

The result of the bitwise Exclusive-OR operation is 1 if only one of the expression has the value as 1; otherwise, the result is always 0.

Let us consider that we have 2 variables op1 and op2 with values as follows:

```
Op1 = 0000 1101
Op2 = 0001 1001
```

The result of the XOR operation on variables op1 and op2 will be

```
Result = 0001 0100
```

As we can see, two variables are compared bit by bit. Whenever only one variable holds the value 1 then the result is 0 else 0 will be the result.

Let us write a simple program that demonstrates bitwise logical operators.

```
#include <stdio.h>
int main() {
        int a = 20;      /* 20 = 010100 */
        int b = 21;      /* 21 = 010101 */
        int c = 0;

        c = a & b;       /* 20 = 010100 */
        printf("AND - Value of c is %d\n", c );

        c = a | b;       /* 21 = 010101 */
        printf("OR - Value of c is %d\n", c );

        c = a ^ b;       /* 1 = 0001 */
        printf("Exclusive-OR - Value of c is %d\n", c );

        getch();
}
```

Output:

```
AND - Value of c is 20
OR - Value of c is 21
Exclusive-OR - Value of c is 1
```

Bitwise shift operators

The bitwise shift operators are used to move/shift the bit patterns either to the left or right side. Left and right are two shift operators provided by 'C' which are represented as follows:

```
Operand << n (Left Shift)
Operand >> n (Right Shift)
```

Here,
  * an operand is an integer expression on which we have to perform the shift operation.
  * 'n' is the total number of bit positions that we have to shift in the integer expression.

The left shift operation will shift the 'n' number of bits to the left side. The leftmost bits in the expression will be popped out, and n bits with the value 0 will be filled on the right side.

The right shift operation will shift the 'n' number of bits to the right side. The rightmost 'n' bits in the expression will be popped out, and the value 0 will be filled on the left side.

Example: x is an integer expression with data 1111. After performing shift operation the result will be:

```
x << 2 (left shift) = 1111<<2 = 1100
x>>2 (right shift) = 1111>>2 = 0011
```

Shifts operators can be combined then it can be used to extract the data from the integer expression. Let us write a program to demonstrate the use of bitwise shift operators.

```
#include <stdio.h>

int main() {
        int a = 20;    /* 20 = 010100 */
        int c = 0;

        c = a << 2;    /* 80 = 101000 */
        printf("Left shift - Value of c is %d\n", c );

        c = a >> 2;    /*05 = 000101 */
        printf("Right shift - Value of c is %d\n", c );
        return 0;
}
```

Output:

```
Left shift - Value of c is 80
Right shift - Value of c is 5
```

After performing the left shift operation the value will become 80 whose binary equivalent is 101000.

After performing the right shift operation, the value will become 5 whose binary equivalent is 000101.

Bitwise complement operator

The bitwise complement is also called as one's complement operator since it always takes only one value or an operand. It is a unary operator.

When we perform complement on any bits, all the 1's become 0's and vice versa.

If we have an integer expression that contains 0000 1111 then after performing bitwise complement operation the value will become 1111 0000.

Bitwise complement operator is denoted by symbol tilde (⍰).

Let us write a program that demonstrates the implementation of bitwise complement operator.

```
#include <stdio.h>
```

```
int main() {
        int a = 10;         /* 10 = 1010 */
        int c = 0;
        c = ⍰(a);
        printf("Complement - Value of c is %d\n", c );
        return 0;
}
```

Output:

```
Complement - Value of c is -11
```

Here is another program, with an example of all the operatoes discussed so far:

```
#include <stdio.h>

main() {
        unsigned int x = 48; /* 48 = 0011 0000 */
        unsigned int y = 13; /* 13 = 0000 1101 */
        int z = 0;

        z =x & y;            /* 0 = 0000 0000 */
        printf("Bitwise AND Operator - x & y = %d\n", z );

        z = x | y;           /* 61 = 0011 1101 */
        printf("Bitwise OR Operator - x | y = %d\n", z );

        z= x^y;              /* 61 = 0011 1101 */
        printf("Bitwise XOR Operator- x^y= %d\n", z);

        z = ⍰x;              /*-49 = 11001111 */
        printf("Bitwise One's Complement Operator - ⍰x = %d\n", z);

        z = x << 2;          /* 192 = 1100 0000 */
        printf("Bitwise Left Shift Operator x << 2= %d\n", z );

        z= x >> 2;           /* 12 = 0000 1100 */
        printf ("Bitwise Right Shift Operator x >> 2= %d\n", z );
}
```

After we compile and run the program, it produces the following result:

```
Bitwise AND Operator - x & y = 0
Bitwise OR Operator - x | y = 61
Bitwise XOR Operator- x^y= 61
Bitwise One's Complement Operator - ⍰x = -49
Bitwise Left Shift Operator x << 2= 192
Bitwise Right Shift Operator x >> 2= 12
```

Summary

* Bitwise operators are special operator set provided by 'C.'
* They are used in bit level programming.
* These operators are used to manipulate bits of an integer expression.
* Logical, shift and complement are three types of bitwise operators.
* Bitwise complement operator is used to reverse the bits of an expression.

---
filename: c_bitwise-operators_2_20230324.txt
https://www.programiz.com/c-programming/bitwise-operators

Bitwise Operators in C Programming

In this tutorial you will learn about all 6 bitwise operators in C programming with examples.

In the arithmetic-logic unit (which is within the CPU), mathematical operations like: addition, subtraction, multiplication and division are done in bit-level. To perform bit-level operations in C programming, bitwise operators are used.

Operators	Meaning of operators
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
⍰	Bitwise complement
<<	Shift left
>>	Shift right

Bitwise AND Operator &

The output of bitwise AND is 1 if the corresponding bits of two operands is 1. If either bit of an operand is 0, the result of corresponding bit is evaluated to 0.

In C Programming, the bitwise AND operator is denoted by &.

Let us suppose the bitwise AND operation of two integers 12 and 25.

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bit Operation of 12 and 25
  00001100
& 00011001
  ________
  00001000  = 8 (In decimal)
```

Example 1: Bitwise AND

```
#include <stdio.h>

int main() {

        int a = 12, b = 25;
        printf("Output = %d", a & b);

        return 0;
}
```

   Output
```
Output = 8
```

Bitwise OR Operator |
   The output of bitwise OR is 1 if at least one corresponding bit of two operands is 1. In C
   Programming, bitwise OR operator is denoted by |.

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bitwise OR Operation of 12 and 25
  00001100
| 00011001
  ________
  00011101 = 29 (In decimal)
```

Example 2: Bitwise OR

```
#include <stdio.h>

int main() {

        int a = 12, b = 25;
        printf("Output = %d", a | b);

        return 0;
}
```

   Output
```
Output = 29
```

Bitwise XOR (exclusive OR) Operator ^
   The result of bitwise XOR operator is 1 if the corresponding bits of two operands are opposite. It is
   denoted by ^.

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bitwise XOR Operation of 12 and 25
  00001100
^ 00011001
  ________
  00010101 = 21 (In decimal)
```

Example 3: Bitwise XOR

```
#include <stdio.h>

int main() {

        int a = 12, b = 25;
        printf("Output = %d", a ^ b);

        return 0;
}
```

   Output
```
Output = 21
```

Bitwise Complement Operator ⍰

   Bitwise complement operator is a unary operator (works on only one operand). It changes 1 to 0 and 0
   to 1. It is denoted by ⍰.

```
35 = 00100011 (In Binary)
```

```
Bitwise complement Operation of 35
~ 00100011
  ________
  11011100 = 220 (In decimal)
```

Twist in Bitwise Complement Operator in C Programming

The bitwise complement of 35 (~35) is -36 instead of 220, but why?

For any integer n, bitwise complement of n will be -(n + 1). To understand this, you should have the knowledge of 2's complement.

2's Complement

Two's complement is an operation on binary numbers. The 2's complement of a number is equal to the complement of that number plus 1. For example:

Decimal	Binary	2's complement
0	00000000	-(11111111+1) = -00000000 = -0 (decimal)
1	00000001	-(11111110+1) = -11111111 = -256 (decimal)
12	00001100	-(11110011+1) = -11110100 = -244 (decimal)
220	11011100	-(00100011+1) = -00100100 = -36 (decimal)

Note: Overflow is ignored while computing 2's complement.

The bitwise complement of 35 is 220 (in decimal). The 2's complement of 220 is -36. Hence, the output is -36 instead of 220.

Bitwise Complement of Any Number N is -(N+1). Here's how:
bitwise complement of N = ~N (represented in 2's complement form)
2'complement of ~N= -(~(~N)+1) = -(N+1)

Example 4: Bitwise complement

```
#include <stdio.h>

int main() {

        printf("Output = %d\n", ~35); printf("Output = %d\n", ~-12);

        return 0;
}
```

Output

```
Output = -36
Output = 11
```

Shift Operators in C programming

There are two shift operators in C programming:

* Right shift operator
* Left shift operator.

Right Shift Operator

Right shift operator shifts all bits towards right by certain number of specified bits. It is denoted by >>.

```
212 = 11010100 (In binary)
212 >> 2 = 00110101 (In binary) [Right shift by two bits]
212 >> 7 = 00000001 (In binary)
212 >> 8 = 00000000
212 >> 0 = 11010100 (No Shift)
```

Left Shift Operator

Left shift operator shifts all bits towards left by a certain number of specified bits. The bit positions that have been vacated by the left shift operator are filled with 0. The symbol of the left shift operator is <<.

```
212 = 11010100 (In binary)
212<<1 = 110101000    (In binary) [Left shift by one bit]
212<<0 = 11010100     (Shift by 0)
212<<4 = 110101000000 (In binary) =3392(In decimal)
```

Example #5: Shift Operators

```
#include <stdio.h>

int main() {

        int num=212, i;
        for (i = 0; i <= 2; ++i) {
                printf("Right shift by %d: %d\n", i, num >> i);
        }
        printf("\n");

        for (i = 0; i <= 2; ++i) {
                printf("Left shift by %d: %d\n", i, num << i);
   }
   return 0;
}

Right Shift by 0: 212
Right Shift by 1: 106
Right Shift by 2: 53

Left Shift by 0: 212
Left Shift by 1: 424
Left Shift by 2: 848
```

---
filename: c_howto-use-bitwise-operators_20220304.txt
https://betterprogramming.pub/how-to-use-bitwise-operators-90cd7a3a0fd7

How to Use Bitwise Operators

How Do Computers Represent Data?

If you've ever programmed before, you have almost definitely used some kind of numeric data type, such as an integer.

But have you ever thought about how your computer actually stores these numbers?

Instead of using powers of 10 as we normally would to represent numbers, computers store values using powers of 2.

Binary representation

This means that numbers are represented using only strings of zeros and ones - a binary system. Each digit in that binary string is called a bit, and each group of eight bits is one byte.

Typically, we read these binary strings from right to left in increasing powers of 2. This is just like we would read numbers normally, with the one's place being the rightmost digit, followed by the ten's place to its left, and so on.

So, if we wanted to represent the (unsigned) number 10 in binary, it would be 1010, since 1(2^0) + 0(2^1) + 1(2^2) + 0(2^3) = 10.

What Can We Do With Binary Data?

Now that we understand how these numbers are stored under the hood, we can leverage this knowledge and use operations that directly act on the binary representations of numbers.

First, think of the operations that you already know about - this probably includes addition (+), subtraction (-), multiplication (*), and division (/). These are all operators that can be used on numbers of any base - so it does not matter whether or not we think of numbers as binary or base-10.

However, there are also operators that specifically act on the bits of the numbers in binary form, aptly named bitwise operators.

Before I cover why these are useful, let's go over the main ones and explain how they work and what they look like.

Negation (□)

The negation operator simply reverses all of the bits in a number. So, if x == 0101, then □x == 1010.

Left shift (<<)

The left shift operator moves all of the bits to the left by a specified number of bits. Any bits that go past the leftmost position are ignored. So, if x == 101010, then x << 2 == 101000.

Right shift (>>)

Similarly, the right shift operator moves all of the bits to the right by a specified amount of bits. So, if x == 101010, then x >> 2 == 001010.

AND (&)

The AND operator, used with two binary numbers, returns a new binary number with 1s in every position where both numbers had a 1 and 0 in every other position.

So, if x == 1100 and y == 1011, then x & y == 1000, since the two numbers both only had a 1 in their leftmost bit.

OR (|)

The OR operator, used with two binary numbers, returns a new binary number with 1s in every position where either number had a 1, and 0 in every other position.

So, if x == 1100 and y == 1011, then x | y == 1111, since at least one of the two numbers had a 1 in each position.

XOR (^)

The XOR (or exclusive OR) operator, used with two binary numbers, returns a new binary number with 1s in every position in which the two numbers disagreed - so one had a 1 in that position and the other had a 0 - and 0 elsewhere.

So, if x == 1100 and y == 1011, then x ^ y == 0111, since the two numbers disagreed in all but the leftmost bit.

Why Is This Useful?

If it's not obvious why you would use these operators - you're not alone. When I first learned about them, they didn't seem all that useful.

But, in many fields, such as embedded programming, systems programming, and networking, they can be seen in solutions to many different problems.

Now, let's go over some use cases of bitwise operators and hopefully you'll begin to appreciate them a bit more.

Data representation

Bitwise operators are a great way to make very efficient use of space when representing data.

Imagine the following situation:

> You're a game developer for a computer game and you want to store players' mouse positions periodically while they play the game. To do this, you send the players' cursor positions every minute to your game's server. The game runs in a fixed window size of 1000 x 1000.

The naive way to do this would simply be to send back two integers of data to your server (one for the x-coordinate and one for the y-coordinate) for each snapshot. Typically, integers are 32 bits, so this would mean sending back 64 bits of data.

However, we can make this much more space-efficient by using bitwise operators.

Since each coordinate is at most 1000, that means we only need 10 bits to store each number, since binary 1111111111 is 1023, which is larger than 1000. This means that we actually only need 20 bits in total.

We can store these 20 bits in a single 32-bit integer - the rightmost 10 bits will represent the x-coordinate, the next 10 bits will represent the y-coordinate, and the remaining 12 bits will just be zero.

Once the data reaches the back-end server, we can then extract each number with the right shift and OR operators.

Useful properties of base-2

Being in base-2 has some handy properties that make some common questions very easy to answer.

For example, we can very easily determine if a number is a power of 2.

Any number that is a power of 2 will have only its leftmost bit set to 1 and all others will be 0.

We can quickly determine if a number, x, follows this pattern by checking if (x & (x - 1)) == 0. This works because if x is a power of 2, then x-1 will have every bit except its leftmost bit set to 1, so when we use the AND operator, they have no bits in agreement at any position.

Algorithms/interview questions

Finally, bitwise operators can be found sporadically in algorithmic problems that one may encounter in technical interviews. These examples often take advantage of the fact that x ^ x == 0.

Swap two numbers without a temporary variable

To swap two integers, x and y, without using a third temporary value, we can use XOR three times.

If you trace it out, the following assignments will result in the two variables' values being swapped in the end: x = x ^ y, y = x ^ y, and x = x ^ y.

Find the single integer in an array that does not appear twice

We XOR every element in the array. Since a number XOR'd with itself is 0, the value of that running total, once we go through the entire array, will be the integer that does not appear twice.

---
filename: c_real-world-uses-of-bitwise-operators_20220304.txt
https://blog.tarkalabs.com/real-world-uses-of-bitwise-operators-c41429df507f

Real World Uses of Bitwise Operators
Sep 3, 2020

As a developer, you definitely would've learned about bitwise operators at some point and would've thought to yourself. "huh, that's neat. but where would I actually use it?". That's what we'll be looking at today.

The modern world of computing and software engineering has extracted the complexities away to an extent that you really don't need to know about how the code you write is executed.

While this is okay for most uses where your code is executed on servers and powerful computers, you'll run into major problems while working on limited resource devices like microcontrollers and FPGAs. You will need to know how your code is executed on these devices in order to optimize it to run on limited resources. (Kilobytes of RAM & KHz of clock speed).

Understanding bitwise operators will bring you a step closer to optimization. Let's first understand what bitwise operators are. Please note that this article will cover usage of bitwise operators in C, but the logic and syntax remains common across most languages. It's also expected that you know and understand binary to decimal and decimal to binary conversions. If you don't know that already, there are plenty resources available online for you to learn it.

What are bitwise Operators?

Bitwise operators are operators that perform operations on data at a bit level.

What do I mean by that? They help you manipulate the bits that make up the piece of data which is represented by a datatype. This will start making more sense when we dive deeper into the topic. Let's now take a look at what are the different bitwise operators available to us.

operator	name	example	variant	associativity
⍰	bitwise NOT	⍰5	-	Right to left
>>	Right Shift	5 >> 2	x >>= 2	Left to right
<<	Left Shift	5 << 2	x <<= 2	Left to right
&	bitwise AND	5 & 6	x &= 6	Left to right
^	bitwise XOR	5 ^ 6	x ^= 6	Left to right
|	bitwise OR	5 | 6	x |= 6	Left to right

Using the sizeof() operator, we can find the number of bytes each datatype represents and since each byte it made up of 8 bits, we can find out the total number of bits taken up by each datatype with some multiplication magic.

Let's now take a look at each operator in detail.

Different Bitwise Operators and Their Functions

& Bitwise AND

The bitwise AND operator (&) takes two operands and compares the operands bit by bit and sets the corresponding output bit to 1 if and only if both input bits are 1. Here's the truth table for the bitwise AND operator:

input bit(1)	input bit(2)	output bit
1	1	1
1	0	0
0	1	0
0	0	0

Let's take the following code for example

```
#include <stdio.h>
#include <stdlib.h>

int main() {
        int x=5, y=6;
        printf("x & y = %d\n", (x & y));
        // "x & y = 4"
        return EXIT_SUCCESS;
}
```

so the value of the variable x is 5 and the value of variable y is 6. The computer that I'm working on is a 64-bit machine so the sizeof(int) is 4 bytes so the binary representation of 5 world be:
00000000 00000000 00000000 00000101

and the binary representation of 6 would be:
00000000 00000000 00000000 00000110

For the sake of simplicity, let's just take the last byte into consideration and let's perform the bitwise AND operation on them.

```
00000101 & 00000110 = 00000100 // 4
```

As you can see, the bits of 5 & 6 were compared and the output byte was set to 00000100 as a result which is the binary representation of the number 4. This is how we arrived at "5 & 6 = 4".

| Bitwise OR

The bitwise OR operator take in two operands just like the bitwise AND and compares them bit by bit but instead of setting the output bit to 1 only when both input bits are 1, it follows this truth table:

input bit(1)	input bit(2)	output bit

1	1	1
1	0	1
0	1	1
0	0	0

Let's look at the below example:

```
#include <stdlib.h>
#include <stdio.h>

void main() {
        printf("5 | 6 = %d\n", (5 | 6));
        return EXIT_SUCCESS;
}
```

The above program would print "5 | 6 = 7". The following bit view of the operation would tell you how the program arrived at that output:

```
00000101 & 00000110 = 00000111 // 7
```

^ Bitwise XOR

The bitwise XOR operator works similar to the bitwise OR operator. The only difference is that the output bit will be set to 1 when both input bits are different. The following is the logic table for bitwise XOR:

input bit(1)	input bit(2)	output bit
1	1	0
1	0	1
0	1	1
0	0	0

Here is an example program that would show how the bitwise XOR works:

```
#include <stdlib.h>
#include <stdio.h>

int main() {
        printf("5^6 = %d\n", (5^6));
        return EXIT_SUCCESS;
}
```

The above program would generate the following output:

```
5^6 = 3
```

How you ask? let's take a look at the bits

```
00000101 & 00000110 = 00000011 // 3
```

Comparing the input bits according to the bitwise XOR truth table. We arrive at 00000011 which is the binary representation of 3.

□ Bitwise NOT

The bitwise NOT operator is a little different from the other operators as it accepts only one operand. The bitwise NOT operator flips the bits of it's operand. Let us take a look at the following example to understand it better,

```
#include <stdlib.h>
#include <stdio.h>

int main() {
        printf("□5=%d\n", □5);
        return EXIT_SUCCESS;
}
```

The above program would produce "□5=-6" as output. Let's look at how the program arrived at the value of -6 when the bitwise NOT operator was applied to 5.

```
□00000101 = 11111010 // -6
```

Since negative numbers are stored as 2's complement, "11111010" would be -6 in it's decimal representation.

<< Bitwise Left Shift Operator

The left shift operator like the name suggests, moves the bits in it's left operand to the right by the number of places specified in the right operand.

So, if the binary representation of 5 is 00000101 then applying the shift operator like so "5 << 2" would move the bits to the left by two places striping the 2 most significant bits and adding 2 zeros as the least significant bits. The binary representation of the result would be 00010100 which is 20 in decimal representation. Let's take a look this in code.

As long as your set bits don't overflow and get dropped off, each left shift by 1 place has the effect of multiplying the left operand by 2.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
        printf("5 << 2 = %d\n",(5 << 2));
        return EXIT_SUCCESS;
}
```

As expected, the program prints an output of "5 << 2 = 20".

>> Bitwise Right Shift Operator
The bitwise right shift operator is very similar to the left shift operator. Instead of shifting the bits to the left, believe it or not, it shifts the bits to the right. Yes, you didn't see that one coming did you?

As the bits are shifted to the right, the least significant bits are dropped and an equivalent number of zeros are added as most significant bits.

Here's some code.

```
#include <stdlib.h>
#include <stdio.h>

int main() {
        printf("5 >> 2 = %d\n", (5 >> 2));
        // 00000101 >> 2 == 00000001
        return EXIT_SUCCESS;
}
```

Simple enough, when we shift the bits of 5 (00000101) 2 places to the right, we get (00000001) which is basically 1 in decimal and binary which is why the above code would print "5 >> 2 = 1".

As long as your set bits don't overflow and get dropped off, each right shift by 1 place has the effect of dividing the left operand by 2.

Now that you've finally mastered bitwise operators, you're probably thinking "What on earth would I do with this abundance of information?".

I have got you covered. In the next section, we will discuss some of the practical uses of bitwise operators.

Practical Uses of Bitwise Operators

1. Storing Multiple Boolean Flags
When working on limited memory devices, you can't really afford to have a thousand boolean flags assigned to variables. This is a really inefficient way to store your flags because Boolean values in C are basically a byte long. We don't need an entire byte to store just a zero or a one. The rest of the bits are going to remain zero throughout. You cannot afford to throw away precious memory when you're working with just KBs of RAM.

So what is the solution? Let's look at the code.

```
#include <stdlib.h>
#include <stdbool.h>
#include <stdio.h>

#define SECURITY_CHECK_FLAG  0b00000001
#define TWO_FACTOR_AUTH_FLAG 0b00000010
#define JUST_A_FLAG_FLAG     0b00000100

int main() {
        // THE BULKY WAY
        // MEMORY CONSUMED: 3 * sizeof(bool) = 3 bytes
        bool security_check   = true;
        bool two_factor_auth  = false;
        bool just_a_flag_flag = true;


        // THE OPTIMIZED BITWISE WAY
        // MEMORY CONSUMED: sizeof(unsigned short int) =  2 bytes

        unsigned short int flags = 0;           /* 0b00000000 */
        // ENABLING FLAGS
        flags |= TWO_FACTOR_AUTH_FLAG;          /* 0b00000010 */
        flags |= JUST_A_FLAG_FLAG;              /* 0b00000110 */

        printf("flags = %d\n", flags);          /* 6 == 0b00000110*/
        // CHECKING FLAG STATES
        if (flags&JUST_A_FLAG_FLAG) {
                printf("JUST_A_FLAG_FLAG is Enabled\n");
        }

        if (flags&SECURITY_CHECK_FLAG) {
                printf("SECURITY_CHECK_FLAG is enabled\n");
        }

        // Disabling Flags
        flags ^= TWO_FACTOR_AUTH_FLAG;          /* 0b00000100 */
        printf("flags = %d\n", flags);          /* 4 == 0b00000100 */

        // will not print because of flag being displabled
        if (flags&TWO_FACTOR_AUTH_FLAG) {
                printf("TWO_FACTOR_AUTH_FLAG is Enabled\n");
        }

        return EXIT_SUCCESS;
}
```

You must have definitely wondered, all of this just to save 1 byte? well that's not entirely true. The increase in memory consumption per boolean flag is one byte, but while using the bitwise flags, you only need an additional byte every 8 flags you add to the program. To increase or decrease the number of flags you just need to change the datatype. if you need only less than 8 flags, you can use "unsigned char ". If you need more than 8 flags, you can use "unsigned long int" or "unsigned long long int" depending on your requirement.

2. Checking for Odd and Even Numbers
   Another trick you can achieve using bitwise operators is checking if an integer is even or odd. Let's see how we can do it.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
        int x=3;
        if (!(x&1)) {
                printf("x is even");
        } else {
                printf("x is odd!");
        }
}
```

So the logic behind !(x&1) is that the binary representation of 1 is 0001 (considering only 4 bits for simplicity) and the place value of the least significant bit is 1. All odd numbers can be represented as 2n+1. So the least significant bit is always 1 for odd numbers. So when compare any odd number with 0001 using the bitwise AND, the output bytes will be 0001. so by using the logical NOT operator, we are able to check for even numbers. For odd numbers you just need to remove the logical NOT and just check (x&1).

3. Swapping Variables Without Using a Third Variable
   So the swapping of two variables without involving a third variable can be achieved without the bitwise operators, but since you already know bitwise operators, why not?

```
#include <stdlib.h>
#include <stdio.h>

int main() {
        int a=3, b=4;
        printf("a: %d; b: %d\n");

            /* a == 0011; b == 0100 */
        a ^= b; /* a == 0111; b == 0100 */
        b ^= a; /* a == 0111; b == 0011 */
        a ^= b; /* a == 0100; b == 0011 */

        printf("a: %d; b: %d\n");
        return EXIT_SUCCESS;
}
```

4. Converting text casing (Lowercase & Uppercase)

```
#include <stdlib.h>
#include <stdio.h>

int main() {
        char word[8] = "sREedEv";
        char *wordptr = &word[0];

        while (wordptr < &word[7]) {
                printf("UPPERCASE: %c\n", *wordptr & '_'); // converts the char into uppercase regardless of th
e current casing
                printf("LOWERCASE: %c\n", *wordptr | ' '); // converts the char into lowercase regardless of th
e current casing
                wordptr++;
        }

        return EXIT_SUCCESS;
}
```

So in the above example, we are using the bitwise AND with '_' char as the right operand to convert each character in the string to uppercase. And to convert to a lowercase character, we are using the bitwise OR operator with the space ASCII character as the right operand.

5. Checking if a number is a power of 2

```
#include <stdio.h>
#include <stdlib.h>

int main() {
        int a=32;
        if (a > 0 && (a & (a - 1)) == 0) {
                printf("%d is a power of 2", a);
        }
        return EXIT_SUCCESS;
}
```

To understand how the above condition works, we need to understand a property of powers of two. Powers of two when represented in binary have only one set bit i.e., only one bit has the value of 1. Any power of 2, minus 1 would have binary representation where all bits are set except the bit that represents the next number (which is a power of 2).

Using the bitwise AND with the number n and n-1, we can conclude that the number in question is a power of two if it returns 0.

To understand how the above condition works, we need to understand a property of powers of two. Powers of two when represented in binary have only one set bit i.e., only one bit has the value of 1. Any power of 2, minus 1 would have binary representation where all bits are set except the bit that represents the next number (which is a power of 2).

Using the bitwise AND with the number n and n-1, we can conclude that the number in question is a power of two if it returns 0.

These are just some of the many use cases of bitwise operators. The more you understand the properties of binary numbers, datatypes and encoding the better you'll be able to utilize these operators. That's all for now. Byte Byte.
