Bitwise operators

- Three major bitwise operators:
  - **AND**, **OR** and **XOR**
- Used to combine two numbers:
- *Bitwise:* The operation is applied individually to each bit of the two values being combined.
- Example:

z = x AND y

  - means that bit# o of z is actually bit# **0** of x ANDed with bit# **0** of y.

The AND Operator

- There are **two kinds** of ***AND*** operators in the C language:
  - the logical AND → ***&&***
  - the bitwise AND → ***&***

```
if ( (x == 5) && (y == 7) )   Do_Something();
```

- In this case, you would expect that the function will only be called

  if x is 5 **and** y is 7.

- The bitwise AND works very much the same way, except that it works by doing bitwise operations.

The AND Operator

In other words, we have the following truth table for the bitwise AND:

0 AND 0 = 0

0 AND 1 = 0

1 AND 0 = 0

1 AND 1 = 1

x AND 0 = 0

x AND 1 = x

It is interesting to note that if you AND any bit with 0, the result is 0;

Also if you AND any bit with 1, the result is the original bit.

AND...

- 12 = 00001100 (In Binary)
- 25 = 00011001 (In Binary)
- Bit Operation of 12 and 25

```
  00001100
& 00011001
----------
  00001000  =  8 (In decimal)
```

AND...

```
#include <stdio.h>
int main()
{
    int a = 12, b = 25;
    printf( "Output = %d",  a&b );
    return 0;
}
```

Output = 8

The OR Operator

- Similar to the AND, there are two different types of OR in the C language:
  - **The logical OR uses the `||` operator**
  - **The bitwise OR uses the `|` operator**
- The use of the logical OR might look something like this:

```
if ((x == 5) || (y == 7))     Do_Something();
```

  - The function will be called **if x is 5 or if y is 7, or both.**
  - The only way the function is not called is if both of the conditions are false.

OR ...

- The bitwise OR is very similar, in that it returns **0** if and only if ***both*** of its operands are **0**.
- To illustrate this, we have the following truth table:

0 OR 0 = 0	
0 OR 1 = 1	x OR 0 = x
1 OR 0 = 1	x OR 1 = 1
1 OR 1 = 1	

Note that whenever you OR a bit with 0, the result is the original bit and whenever you OR a bit with 1, the result will always be 1.

OR ...

- 12 = 00001100 (In Binary)
- 25 = 00011001 (In Binary)
- Bitwise OR Operation of 12 and 25

```
  00001100
| 00011001
----------
  00011101    = 29 (In decimal)
```

OR...

```
#include <stdio.h>
int main()
{
    int a = 12, b = 25;
    printf( "Output = %d", a|b );
    return 0;
}
```

Output = 29

The XOR Operator

- XOR is a bit of an out-of-the-way operator and has no logical equivalent for it in C.
- The XOR operation is symbolized by the ^ character in C.
- The term XOR stands for "exclusive OR" and means "***one or the other, but not both***."
- In other words, XOR returns 1 if and **only if *exactly one* of its operands is 1**.
- If both operands are 0, or both are 1, then XOR returns 0.

The XOR Operator: Points to note

- Anything XORed with itself returns o.
- Any bit XORed with o yields the original bit.
- Any bit XORed with 1 yields the complement of the original bit.

$0 \text{ XOR } 0 = 0$

$0 \text{ XOR } 1 = 1$

$1 \text{ XOR } 0 = 1$

$1 \text{ XOR } 1 = 0$

$x \text{ XOR } 0 = x$

$x \text{ XOR } 1 = \sim x$

XOR...

- 12 = 00001100 (In Binary) 25 = 00011001 (In Binary)
- Bitwise XOR Operation of 12 and 25

```
  00001100
^ 00011001
----------
  00010101  = 21 (In decimal)
```

XOR...

```
#include <stdio.h>
int main()
{
  int a = 12, b = 25;
  printf("Output = %d", a^b );
  return 0;
}
```

Output = 21

Swapping using XOR

```
                    // Value of x                    Value of y
                    //-----------------------------------------
int x, y;           //      o                          o
x = A;              //      A                          o
y = B;              //      A                          B
x = x ^ y;          //    A ^ B                        B
y = x ^ y;          //    A ^ B                        A ^ B ^ B == A ^ o == A
x = x ^ y;          // A ^ A ^ B                      A
                    // == o ^ B == B                  A
                    //       B                        A
```