Bitwise Operators

For the purposes of demonstration, 8-bit integers are being used in place of more traditional integers like 32-bit integers.

Bitwise operators do the opposite of what you are used to doing - traditionally, you change the value stored in a variable by working with its decimal representation (base 10) which in turn changes the actual binary representation (base 2) of a variable in the computer's memory whereas bitwise operators directly manipulate the binary representation of a variable in the computer's memory which in turn changes its decimal representation. In the following examples, you can see how the mathematical operations are actually changing the binary representation of the variables in memory.

```
unsigned x = 1;        // binary representation of x                  0000 0001
x = x * 2;             // result of x after multiplying by 2          0000 0010

unsigned y = 1;        // binary representation of y                  0000 0001
y = y * pow(2, 3);     // result of y after multiplying by 2 three times  0000 1000
```

In practice, bitwise operations should only be performed on unsigned data types because the results of bitwise operations on signed data types are implementation-defined. This website has a more in depth explanation.
https://wiki.sei.cmu.edu/confluence/display/c/INT13-C.+Use+bitwise+operators+only+on+unsigned+operands

& The **AND** operator - performs the bitwise AND operation on every bit in two numbers. For the result of the operation to be a 1 both bits must also be a 1. Note that **&** is also used as the address of operator. In this case, it has a completely different meaning.

```
unsigned x = 178;       // binary representation of x             1011 0010
unsigned y = 201;       // binary representation of y             1100 1001
x = x & y;              // result of x after the AND operation    1000 0000
```

| The **OR** operator - performs the bitwise OR operation on every bit in two numbers. For the result of the operation to be a 1 only one of the bits needs to be a 1.

```
unsigned x = 178;       // binary representation of x             1011 0010
unsigned y = 201;       // binary representation of y             1100 1001
x = x | y;              // result of x after the OR operation     1111 1011
```

^ The **XOR** operator - performs the bitwise XOR operation on every bit in two numbers. For the result of the operation to be a 1 both of the bits need to be different.

```
unsigned x = 178;       // binary representation of x             1011 0010
unsigned y = 201;       // binary representation of y             1100 1001
x = x ^ y;              // result of x after the XOR operation     0111 1011
```

<< The **Left Shift** operator - performs the left shift operation on every bit in a single number. All of the bits are shifted left by the number of positions denoted by the operand on the right side. During every shift, the leftmost bit "falls off the cliff" and goes away, and the rightmost bit becomes a 0. The left shift operator is equivalent to multiplying a number by 2. It is also important to know that it is undefined behavior to left shift an operand by a value that is greater than or equal to its width in bits. For example, if x is a 32 bit integer then x << 40 is undefined because 40 >= 32. Lastly, note that **<<** is also used as an operator with output streams in C++. In this case, it has a completely different meaning.

```
unsigned x = 178;       // binary representation of x             1011 0010
x = x << 1;             // result after left shifting x by 1       0110 0100
```

>> The **Right Shift** operator - performs the right shift operation on every bit in a single number. All of the bits are shifted right by the number of positions denoted by the operand on the right side. During every shift, the leftmost bit becomes a 0, and the rightmost bit "falls off the cliff" and goes away. The right shift operator is equivalent to dividing a number by 2. It is also important to know that it is undefined behavior to right shift an operand by a value that is greater than or equal to its width in bits. For example, if x is a 32 bit integer then x >> 40 is undefined because 40 >= 32. Lastly, note that **>>** is also used as an operator with input streams in C++. In this case, it has a completely different meaning.

```
unsigned x = 178;       // binary representation of x             1011 0010
x = x >> 1;             // result after right shifting x by 1      0101 1001
```

~ The **NOT** operator - performs the NOT operation on every bit in a single number. Every bit is inverted meaning all of the 1s becomes 0s and all of the 0s become 1s. As a consequence of this, it is also colloquially referred to as the bitwise "inverse" operator. Note that this bitwise operator is a unary operator unlike the previous bitwise operators which were binary operators.

```
unsigned x = 178;       // binary representation of x             1011 0010
x = ~x;                 // result after inverting the bits of x    0100 1101
```