C and C++ 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 inverse of what you are used to doing - traditionally, you change the value stored in a variable by working with its decimal representation which in turn changes the actual binary representation 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. For example:

```
int x = 1;              // binary representation of x                    0000 0001
x = x + 1;              // result of x after addition                    0000 0010

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

The lines of code "x = x + 1" and "x = x * pow(2, 3)" are abstractions. What's actually going on is the computer is going to the place in memory where the 8 bits are stored for the x and y integer variables. It changes the x from a 1 to a 2 and the y from a 1 to an 8 by changing the actual bits in memory. Working with the actual binary representation of numbers is challenging which is why programming languages developed methods of abstraction so data can be worked with and manipulated in more intuitive and "normal" ways. This is why C, C++, and all other languages allow you to store numbers in variables and use operators to manipulate the numbers like in traditional mathematics. However, C and C++ still allow you to use bitwise operators to actually directly work with a variable's binary representation.

& 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.

```
int x = 178;          // binary representation of x                1011 0010
int 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.

```
int x = 178;          // binary representation of x                1011 0010
int 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.

```
int x = 178;          // binary representation of x                1011 0010
int 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 noted by the second argument. 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. Note that << is also used as an operator with output streams in C++. In this case, it has a completely different meaning.

```
int 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 noted by the second argument. 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. Note that >> is also used as an operator with input streams in C++. In this case, it has a completely different meaning.

```
int 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.

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