Bitwise Operators

We commonly use the binary operators && and ||, which take the logical and and logical or of two boolean expressions.

Since boolean logic can work with single bits, C/Java/Python (and most languages) provides operators that work on individual bits within a variable.

As we learned earlier in the semester, if we store an int in binary with the value 47, its last eight binary bits are as follows:

00101111

Similarly, 72 in binary is

01001000.

Bitwise operators would take each corresponding bit in the two input numbers and calculate the output of the same operation on each set of bits.

For example, a bitwise AND is represented with a single ampersand sign: &. This operation is carried out by taking the and of two bits. If both bits are one, the answer is one. Otherwise, the answer is zero.

Here is the bitwise and operation on 47 and 72:

```
  0 0 1 0 1 1 1 1
& 0 1 0 0 1 0 0 0
---------------------
  0 0 0 0 1 0 0 0 (which has a value of 8.)
```

Thus, the following code segment has the output 8:

```
int x = 47, y = 72;
int z = x & y;
printf("%d", z);
```

Here is a chart of some other bitwise operators:

<table>
<tr><td>Function</td><td>Operator</td><td>Meaning</td></tr>
<tr><td>and</td><td>&</td><td>1&1 = 1, rest = 0</td></tr>
<tr><td>or</td><td>|</td><td>0 | 0 = 0, rest = 1</td></tr>
<tr><td>xor</td><td>^</td><td>1 ^ 0 = 0 ^ 1 = 1
0 ^ 0 = 1 ^ 1 = 0</td></tr>
<tr><td>bitwise left shift</td><td><<</td><td>a << b adds b 0s to the binary representation of a.</td></tr>
<tr><td>bitwise right shift</td><td>>></td><td>a >> b chops off the last b bits in the binary representation of a</td></tr>
</table>

Now, let’s calculate the other bitwise operations between 47 and 72:

```
   0 0 1 0 1 1 1 1
|  0 1 0 0 1 0 0 0
---------------------
   0 1 1 0 1 1 1 1 (which has a value of 111.)

   0 0 1 0 1 1 1 1
^  0 1 0 0 1 0 0 0
---------------------
   0 1 1 0 0 1 1 1 (which has a value of 103.)
```

Here are a couple examples of bitshifts on 47 and 72:

47 << 2

0 0 1 0 1 1 1 1 << 2 → 1 0 1 1 1 1 0 0 = 188 (effectively multiplying 47 by 2^2, if we look at it Numerically)

47 >> 3

0 0 1 0 1 1 1 1 >> 3 → 0 0 0 0 0 1 0 1 = 5 (effectively doing an integer division of 47 by 2^3)

72 << 1

0 1 0 0 1 0 0 0 << 1 → 1 0 0 1 0 0 0 0 = 144 (effectively multiplying 72 by 2^1)

72 >> 3 → 0 1 0 0 1 0 0 0 >> 3 → 0 0 0 0 1 0 0 1 = 9 (effectively dividing 72 by 2^3.)

For modern cryptography, we think of the plaintext as being a bitstring of some length. Most modern cryptographic schemes have a “block size”, where the input is separated into chunks of n bits, and each chunk of n bits is encrypted. For example, DES (Data Encryption Standard) has a block size of 64 bits, and AES (Advanced Encryption Standard) has versions with input block sizes of 128 bits, 192 bits and 256 bits.

Understanding how bitwise operators work in programming languages is useful because the descriptions of most modern day symmetric ciphers include operations on the bits in blocks of the cipher. In order to code up any of these ideas, it would be best to use bit-wise operators.

The most common bitwise operator used is xor, because xoring some input with a secret key (that is random) effectively randomizes the output.

In dealing with moving bits around, the most common operations would be left or right shifts.