Binary Numbers

Representing Binary Numbers

Computers operate on 1's and 0's; these are called 'bits'. A byte is a group of 8 bits, like this: 00110101. A computer word on my computer ('int') is 4 bytes, 32 bits: 10011010110101011010001010101011. Other computers have different word sizes.

As you can see 32 ones and zeroes is a bit cumbersome to write down (or even to read). Thus, conventionally people break the number down into groups of 3 or 4 bits:

```
  1001.1010.1101.0101.1010.0010.1010.1011
  10.011.010.110.101.011.010.001.010.101.011 <-- note that the count of
3 starts on the right
```

These grouped sets of bits are then mapped onto digits, either four bits per hexadecimal (base 16) digit or three bits per octal (base 8) digit. Obviously, hexadecimal needs some new digits (since decimal digits only go 0..9 and we need 6 more). These days, the letters 'A'..'F' are used for the 'digits' that represent 10..15. Here's the map; the correspondence is obvious:

```
   OCTAL:                    HEXADECIMAL:
000 -> 0  100 -> 4      0000 -> 0  0100 -> 4  1000 -> 8  1100 -> C
001 -> 1  101 -> 5      0001 -> 1  0101 -> 5  1001 -> 9  1101 -> D
010 -> 2  110 -> 6      0010 -> 2  0110 -> 6  1010 -> A  1110 -> E
011 -> 3  111 -> 7      0011 -> 3  0111 -> 7  1011 -> B  1111 -> f
```

So now we can write the hex and octal representations of those integers above quite quickly along with their notations for C and other languages:

```
       1001.1010.1101.0101.1010.0010.1010.1011
   ->     9    A    D    5    A    2    A    B  --> 0x9AD5A2AB
                          (that's 0x in front of the hex number)
```

and

```
       10.011.010.110.101.011.010.001.010.101.011
        2   3   2   6   5   3   2   1   2   5   3 -> 023265321253
                          (that's a numeric '0' in front)
```

Octal is easier to write down quickly, but hexadecimal has the nice properties of breaking easily into bytes (which are pairs of digits).

Operating on Binary Numbers in Programs

Sometimes it is handy to work with the bits stored in numbers rather than just treating them as integers. Examples of such times include remembering choices (each bit slot can be a 'yes'/'no' indicator), keeping track of option flags (same idea, really, each bit slot is a 'yes'/'no' indicator for a flag's presence), or keeping track of a number of small integers (e.g., successive pairs of bit slots can remember numbers from 0..3). Of course, occasionally problems actually contain 'bit strings'.

In C/C++ and others, assigning a binary number is easy if you know its octal or hexadecimal representation:

```
i = 0x9AD5A2AB;
```

or

```
i = 023265321253;
```

More often, a set of single-bit values is combined to create an integer of interest. One might think the statement below would do that:

```
i = 0x10000 + 0x100;
```

and it will -- until the sign bit enters the picture or the same bit is combined twice:

```
i = 0x100 + 0x100;
```

In that case, a 'carry' occurs and then i contains 0x200 instead of 0x100 as probably desired. The 'or' operation -- denoted as '|' in C/C++ and others -- does the right thing. It combines successive bits using these four rules:

```
0|0 -> 0
0|1 -> 1
1|0 -> 1
1|1 -> 1
```

The '|' operation is called 'bitwise or' in C so as not to be confused with it's cousin '||' called 'logical or' or 'orif'. The '||' operator evaluates the arithmetic value its left side and, if false (exactly 0), evaluates its right side. If either evaluation is nonzero, then '||' evaluates to true (exactly 1 in C).

It is the final rule that distinguishes the '|' operator from '+'. Sometimes operators like this are displayed as a 'truth table':

|	0	1
0	0	1
1	1	1

It's easy to see that the 'bitwise or' operation is a way to set bits inside an integer. A '1' results with either or both of the input bits are '1'.

The easy way to query bits is using the 'logical and' (also known as 'andif') operator, denoted as '&' that has this truth table:

&	0	1
0	0	0
1	0	1

A '1' results only when *both* input bits are '1'. So, if one wants to know if the 0x100 bit is '1' in an integer, the if statement is simple:

```
if (a & 0x100) { printf("yes, 0x100 is on\n"); }
```

C/C++ (and others) contain additional operators, including 'exclusive or' (denoted '^') with this truth table:

^	0	1
0	0	1
1	1	0

The 'exclusive or' operator is sometimes called 'xor', for easy of typing. Xor yields a '1' either exactly *one* of its inputs is one: either one or the other, but not both. This operator is very hand for 'toggling' (flipping) bits, changing them from '1' to '0' or vice-versa. Consider this statement:

```
a = a ^ 0x100;    /* same as a ^= 0x100; */
```

The 0x100 bit will be changed from 0->1 or 1->0, depending on its current value.

Switching off a bit requires two operators. The new one is the unary operator that toggles every bit in a word, creating what is called the 'bitwise complement' or just 'complement' of a word. Sometimes this is called 'bit inversion' or just 'inversion' and is denoted by the tilde: '~'. Here's a quick example:

```
char a, b;      /* eight bits, not 32 */
a = 0x4A;       /* 0100.1010 */
b = ~a;         /* flip every bit: 1011.0101 */
printf("b is 0x%X\n", b);
```

which yields something like:

```
b is 0xB5
```

Thus, if we have a single bit switched on (e.g., 0x100) then ~0x100 has all but one bit switched on: 0xFFFFFEFF (note the 'E' as the third 'digit' from the right).

These two operators combine to create a scheme for switching off bits:

```
a = a & (~0x100);       /* swtch off the 0x100 bit */
                        /* same as a &= ~0x100;
```

since all but one bit in ~0x100 is on, all the bits except the 0x100 bits appear in the result. Since the 0x100 bit is 'off' in ~0x100, that bit is guaranteed to be '0' in the result. This operation is universally called 'masking' as in 'mask off the 0x100 bit'.

Summary

In summary, these operators enable setting, clearing, toggling, and testing any bit or combination of bits in an integer:

```
a |= 0x20;              /* turn on bit 0x20 */
a &= ~0x20;             /* turn off bit 0x20 */
a ^= 0x20;              /* toggle bit 0x20 */
if (a & 0x20) {
        /* then the 0x20 bit is on */
}
```