FAQ > Bit manipulation

If you have any questions or comments,
please visit us on the Forums.

FAQ > Prelude's Corner > Bit manipulation

This item was added on: 2004/01/21

Bit Manipulation

In C and C++, values are represented as binary values. The exact values will vary from computer to computer, but the most common sizes at the time of this writing are 127 for char, 32767 for short, and 2147483647 for int and long. The unsigned values are 255, 65535, and 4294967295 respectively. Now, these values are confusing to many programmers, much less non-programmers. They are obviously one less than powers of two, but when seen in source code they appear to be magic numbers. An easier way to represent these values is with the hexadecimal numbering system which shows the binary structure more clearly than decimal values. In hexadecimal, each digit corresponds to four bits of the binary value, with values from 10 to 15 being represented by the letters A through F. So the values 0x7F and 0xFF are the hexadecimal equivalent to the decimal values for char shown above, signed and unsigned respectively.

You'll notice that the hexadecimal number has two digits (the 0x merely states that the number is in hexadecimal format), each of these digits correspond to four bits of the binary value. 255 in binary is

```
1111 1111
```

We know that 1111 in decimal is 15, which is 0xF in hexadecimal. So to convert the binary value to hexadecimal, simply replace every four bits with the corresponding hexadecimal digit:

```
1111 = F
1111 = F
--------
0xFF
```

Signed and Unsigned

Integer values come in two flavors in C and C++, signed and unsigned. Unsigned values are represented by a format where each bit represents a power of two, each position has a weight (1, 2, 4, 8, 16, 32, etc..) and the value of the number is determined by adding the weights of each position whose bit is set to 1. A binary value of 0000 0010 is valued at 2 since the weight of the second position is 2 and no other bits are set to 1.

Signed values are more complicated because they must also be able to represent negative numbers. There are many different ways to go about this, increasing the confusion. The more common ways include one's complement, two's complement, and sign-magnitude. All of these methods use a particular bit to mark the sign of the value, the sign is whether the value is positive or negative, 0 is positive and 1 is negative. Each method goes about marking the sign in different ways:

```
One's complement - This method inverts all of the bits corresponding to the
positive number to create the negative number.
```

```
Ex.
---
1  - 00000001
-1 - 11111110

Two's complement - This method performs a one's complement, but also adds
one to the resulting number.

Ex.
---
1  - 00000001
-1 - 11111111

Sign-magnitude - This method simply toggles the sign bit.

Ex.
---
1  - 00000001
-1 - 10000001
```

Because of the different methods of calculating the signed-ness of a value and other complications when manipulating signed bits, it is highly recommended that unsigned values are used when working with individual bits, all of the code below will be using unsigned values to avoid many of the problems that can occur. We will also restrict ourselves to unsigned int as the smallest type because many of the bit operations promote char and short values to int. Even if the char and short were unsigned to begin with, the promotion could make the value signed, which is just begging for trouble.

Bit Operations

C and C++ programmers have several tools to work with bits effectively, but they appear arcane at first. We will be spending a little bit of time on what each of the operations does and how they can be chained together to manipulate bits in a simple and effective manner. There are six operators that C and C++ support for bit manipulation:

```
&  Bitwise AND
|  Bitwise OR
^  Bitwise Exclusive-OR
<< Bitwise left shift
>> Bitwise right shift
~  Bitwise complement
```

The **bitwise AND** tests two binary numbers and returns bit values of 1 for positions where both numbers had a one, and bit values of 0 where both numbers did not have one:

```
01001011
00010101
&
--------
00000001
```

Notice that a 0,0 combination being tested results in 0, as does a 1,0 combination. Only a 1,1 combination results in a binary 1 in the resulting value. The bitwise AND is often used to mask a set of bits for testing.

The **bitwise OR** tests two binary numbers and returns bit values of 1 for positions where either bit or both bits are one, the result of 0 only happens when both bits are 0:

```
01001011
00010101
|
--------
01011111
```

Notice that a 1,0 combination being tested results in 1, as does a 1,1 combination. Only a 0,0 combination results in a binary 0 in the resulting value. The bitwise OR is used to turn bits on if they were off.

The **bitwise Exclusive-OR** tests two binary numbers and returns bit values of 1 for positions where both bits are different, if they are the same then the result is 0:

```
01001011
00010101
^
--------
01011110
```

The **bitwise left shift** moves all bits in the number to the left and fills vacated bit positions with 0.

```
01001011
       2
<<
--------
00101100
```

Shifting is very useful for dealing with individual bits in a binary number. If you want to affect every bit position then instead of working out which bit position with each new bit, simply shift to the next bit and work with the same bit position.

The **bitwise right shift** moves all bits in the number to the right.

```
01001011
       2
>>
--------
??010010
```

Note the use of ? for the fill bits. Where the left shift filled the vacated positions with 0, a right shift will do the same only when the value is unsigned. If the value is signed then a right shift will fill the vacated bit positions with the sign bit or 0, which one is implementation-defined. So the best option is to never right shift signed values.

The **bitwise complement** inverts the bits in a single binary number.

```
~01001011
---------
 10110100
```

The binary complement operator is unary, meaning it is only used on a single number (~num) instead of two numbers like the previous binary operators (num1 & num2, num1 << num2).

The bitwise operators in C and C++ can be chained together and used for a huge number of operations, for example, if you wanted to clear the lowest order 1 bit you would say something like val & (val - 1). To clear all 1 bits except for the lowest bit, the statement could be changed to val & -val. There are many different combinations that can be used to do just about anything with a binary number. Following are two functions which will help in playing around with the operators to figure out just how they work. A tutorial will never be able to explain such operations adequately, so you are encouraged to try things out for yourself. The following two functions will reverse the bits in a number and print all of the bits to an output stream. The print function prints the bits in reverse order, so the reversal function can be used to improve readability of the output:

The rev_bits function is a template function which can be used with any reasonable type for bit manipulation, it works by calculating the number of bits in the type passed to it by multiplying the size of the type by CHAR_BIT from <climits>. It then copies val to ret in reverse simply by copying the lowest order bit of val to ret and then shifting ret left by one, then shifts val right by one. The sequence is as follows:

```
Start:
val  ret
---  ---
1101 0000

copy:
1101 0001
shift:
0110 0010

copy:
0110 0010
shift:
0011 0100

copy:
0011 0101
shift:
0001 1010

copy:
0001 1011
shift:
0000 no_shift

End:
val  ret
---  ---
0000 1011
```

```
template <typename T>
T rev_bits ( T val )
{
  T ret = 0;
  T n_bits = sizeof ( val ) * CHAR_BIT;

  for ( unsigned i = 0; i < n_bits; ++i ) {
    ret = ( ret << 1 ) | ( val & 1 );
    val >>= 1;
  }

  return ret;
}
```

The print_bits function is also a template function which can print the bits of any reasonable data type for bit manipulation, it takes an extra argument which determines which stream to print the bits to (the most common stream would be std::cout). The function is very simple, it merely prints the value of the lowest order bit and then shifts the bits right by one. The result is that the number is actually printed in reverse, but this disadvantage is removed when print_bits is coupled with rev_bits. The only statement which may be confusing is !!(val & 1). What this does is test whether the lowest order bit of val is set, this test will return zero if it is not set and non-zero if it is. Note that I said non-zero and not 1, this is important because you can't be sure that (val & 1) will print the correct value, so the logical negation operator is used once to create a boolean value (which is 0 or 1, nothing else) and once more to set the boolean values correctly (only one negation will result in unset bits printing as 1 and set bits as 0).

```
template <typename T>
void print_bits ( T val, std::ostream& out )
{
  T n_bits = sizeof ( val ) * CHAR_BIT;

  for ( unsigned i = 0; i < n_bits; ++i ) {
    out<< !!( val & 1 );
    val >>= 1;
  }
}
```

Note that both of these functions require that the header <climits> is included for CHAR_BIT. Both functions are generic enough for any reasonable usage. Here is a possible driver the makes use of these functions to test the bitwise AND operator:

```
#include <iostream>
#include <climits>

// The definition of print_bits here

// The definition of rev_bits here

int main()
{
  unsigned a = 0xF, b = 0x6;

  print_bits ( rev_bits ( a ), std::cout );
  std::cout<<'\n';
  print_bits ( rev_bits ( b ), std::cout );
  std::cout<<"\n&\n";
  print_bits ( rev_bits ( a & b ), std::cout );
  std::cout<<'\n';

  return 0;
}
```

Have fun!