Bitwise Operators in C

Uses of Bitwise Operations or Why to Study Bits

1. **Compression**: Occasionally, you may want to implement a large number of Boolean variables, without using a lot of space. A 32-bit int can be used to store 32 Boolean variables. Normally, the minimum size for one Boolean variable is one byte. All types in C must have sizes that are multiples of bytes. However, only one bit is necessary to represent a Boolean value.
2. **Set operations**: You can also use bits to represent elements of a (small) set. If a bit is 1, then element **i** is in the set, otherwise it's not. You can use bitwise AND to implement set intersection, bitwise OR to implement set union.
3. **Encryption**: swapping the bits of a string for e.g. according to a predefined shared key will create an encrypted string.

Generic Bitwise Operations

Bitwise operators only work on a limited number of types: **int** and **char**. This seems restrictive--and it is restrictive, but it turns out we can gain some flexibility by doing some C "tricks".

It turns out there's more than one kind of **int**. In particular, there's **unsigned int**, there's **short int**, there's **long int**, and then unsigned versions of those ints.

The "C" language does not specify the difference between a short int, an int and a long int, except to state that:

```
sizeof( short int ) <= sizeof( int ) <= sizeof( long )
```

Bitwise operators fall into two categories: binary bitwise operators and unary bitwise operators. Binary operators take two arguments, while unary operators only take one.

Bitwise AND

In C, the & operator is bitwise AND. The following is a chart that defines &, defining AND on individual bits.

x_i	y_i	x_i & y_i
0	0	0
0	1	0
1	0	0
1	1	1

We can do an example of bitwise &. It's easiest to do this on 4 bit numbers, however.

Variable	b_3	b_2	b_1	b_0
x	1	1	0	0
y	1	0	1	0
z = x & y	1	0	0	0

Bitwise OR

The | operator is bitwise OR (it's a single vertical bar). The following is a chart that defines |, defining OR on individual bits.

x_i	y_i	$x_i \mid y_i$
0	0	0
0	1	1
1	0	1
1	1	1

We can do an example of bitwise |. It's easiest to do this on 4 bit numbers, however.

Variable	b_3	b_2	b_1	b_0
x	1	1	0	0
y	1	0	1	0
z = x | y	1	1	1	0

Bitwise XOR

The ^ operator is bitwise XOR. The usual bitwise OR operator is *inclusive* OR. XOR is true only if exactly one of the two bits is true.

The following is a chart that defines ^, defining XOR on individual bits.

x_i	y_i	$x_i \wedge y_i$
0	0	0
0	1	1
1	0	1
1	1	0

We can do an example of bitwise ^. It's easiest to do this on 4 bit numbers, however.

Variable	b_3	b_2	b_1	b_0
x	1	1	0	0
y	1	0	1	0
z = x ^ y	0	1	1	0

Bitwise NOT

There's only one unary bitwise operator, and that's bitwise NOT. Bitwise NOT flips all of the bits.

The following is a chart that defines ~, defining NOT on an individual bit.

x_i	$\sim x_i$
0	1
1	0

We can do an example of bitwise ~. It's easiest to do this on 4 bit numbers (although only 2 bits are necessary to show the concept).

Variable	b_3	b_2	b_1	b_0
x	1	1	0	0
z = ~x	0	0	1	1

Facts About Bitwise Operators

Consider the expression **x + y**. Do either **x** or **y** get modified? The answer is no.

Most built-in binary operators do not modify the values of the arguments. This applies to logical operators too. They don't modify their arguments.

There are operators that do assignment such as +=, -=, *=, and so on. They apply to bitwise operators too. For example, |=, &=, ^=. Nearly all binary operators have a version with = after it.

Bitshift Operators

Introduction

The bitshift operators take two arguments, and looks like:

```
x << n
x >> n
```

where **x** can be any kind of int variable or char variable, and **n** can be any kind of int variable.

Restrictions

Like bitwise operators, you can only perform bitshift operations on **x** (the left argument) on certain types: in particular, any kind of **int** and any kind of **char**.

There are sneaky ways to shift bits even if you use other types (say, float). This involves tricks with casting. We'll see this at the end of this chapter.

Operator <<

The operation **x << n** shifts the value of x left by n bits.

Let's look at an example. Suppose **x** is a char and contains the following 8 bits.

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
1	1	0	0	0	1	1	1

If we shift left by 2 bits, the result is:

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	0	0	1	1	1	0	0

When you shift left by **k** bits then, $b_{i+k} = b_i$. If **i + k > N**, then bit b_i fell off the left edge. If **i < K**, then $b_i = 0$. In other words the low K bits are all 0's.

That means that as you shift left, the bits on the high end (to the left) fall off, and 0 is shifted in from the right end.

Left shifting is multiplying by 2^K

If you shift left on an unsigned int by K bits, this is equivalent to multiplying by 2^K.

Shifting left on signed values also works, but overflow occurs when the most significant bit changes values (from 0 to 1, or 1 to 0).

Operator >>

Left shifting generally creates few problems regardless of type. 0's are shifted from the least significant end, and bits fall off the most significant end.

Right shifting causes a few more problems. When shifting to the right for unsigned int, bits fall off the least significant end, and 0's are shifted in from the most significant end. This is also known as *logical right shift* (logical shifts shift in 0's).

However, with signed int, the story is different. What right shifting does depends on the compiler. One possibility is to shift in 0's, just as unsigned int's do. If this occurs, then you divide by 2^K (if you're shifting K bits), but *only* for non-negative values of x. For negative values, you've made it positive, and it no longer makes sense that this operation is equivalent to dividing by 2^K.

Compilers may also shift in the sign bit. Thus, if x is negative, the sign bit is 1, so 1's are shifted from the most significant end. If x is non-negative, 0's are shifted from the most significant end. This is called an *arithmetic right shift* since the sign bit is shifted in. Thus, if you shift right by K bits, then K 1's or K 0's are shifted in.

Let's look at an example of this. Suppose x looks like before:

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
1	1	0	0	0	1	1	1

Let's shift right by 3 bits. If the sign bit is shifted in, the result is:

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
1	1	1	1	1	0	0	0

If 0's are shifted in, the result is:

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	0	0	1	1	0	0	0

Shifting Doesn't Change Values

Here's one of those rather annoying facts about C, that is perfectly consistent with the way C does things. Suppose x = 3 and n = 2. What does the following code do?

```
int x = 3 ;
int n = 2 ;
x << n ;
printf("%d\n", x) ;
```

What does it print? (think before turning the page…).

The answer is 3. The answer is NOT 12.

Why is that? Suppose you were asked to sum x + y. What will happen to x or y? Of course, you know that nothing happens to them. But then where does the sum go? A temporary value is created internally, and that value is typically used to assign to some result or printed.

The same thing happens when you do x << n. It creates a temporary value with a bitshifted version of x, but does not change the value of x.

So, how do you save the change? Simple, just assign it.

```
x = x << n ;
printf("%d\n", x) ;
```

This prints out 12 (if x is 3 and n is 2).

There's an equivalent way to do this. Use the <<=; operator. Nearly every C binary operator has a version with ='s after it.

```
x <<= n ;
printf("%d\n", x) ;
```

Shifting Bits for other types

Suppose you wish to shift left on float variables. The compiler will complain that you can't do this. Bitwise/bitshift operations aren't defined on float variables. So, what can you do?

Trick the compiler. How do you do that? Make it think the float is an int.

Here's the wrong way to do it.

```
float x ;
(int)x <<= n ; // left shift by n bits
```

You can try to cast the float to an int. But if you do this, you have two problems. First, casting creates a temporary value. Even if you save this, the bits will actually change. A float that's cast to an int causes the fractional part to be truncated.

Nevertheless, casting is the right idea. Here's how to do it.

```
float x ;
int * ptr = (int *) (& x) ;
*ptr <<= n ; // left shift by n bits
```

You make a pointer that points to the float with same number of bytes, then you shift based on the dereferenced pointer. At this point, when we manipulate *ptr, we're actually manipulating x.

The Magic of XOR

Various Views of XOR

You can think of XOR in many ways. Assume that **p** and **q** are Boolean variables. Also assume that $\mathbf{p_1}$, $\mathbf{p_2}$, ...$\mathbf{p_n}$ are Boolean variables. Let (+) be the XOR operator (this is a circle with a plus sign inside it). In this case, we assume that **p** and **q** are boolean values, instead of words, and we assume (+) is plain XOR, not bitwise XOR. Later on, we'll use it as bitwise XOR.

- **p (+) q** is true if exactly one of **p** and **q** is true. This is the conventional definition of XOR.
- $\mathbf{p_1}$ **(+)** $\mathbf{p_2}$ **(+) ... (+)** $\mathbf{p_n}$ is true if the number of variables with the value true is odd (and is false if the number of variables with the value true is even). Notice this definition ignores the variables which are assigned to false.

Properties of XOR

Here are several useful properties of XOR. This applies to plain XOR and bitwise XOR.

- **x (+) 0 = x**

  XORing with 0 gives you back the same number. Thus, 0 is the identity for XOR.

- **x (+) 1 = \x**

  XORing with 1 gives you back the negation of the bit. Again, this comes from the truth table. For bitwise XOR, the property is slightly different: **x ^ ~0 = ~x**. That is, if you XOR with all 1's, the result will be the bitwise negation of **x**.

- **x (+) x = 0**

  XORing x with itself gives you 0. That's because **x** is either 0 or 1, and **0 (+) 0 = 0** and **1 (+) 1 = 0**.

- XOR is associative.

  That is: **(x (+) y) (+) z = x (+) (y (+) z)**.

  You can verify this by using truth tables.

- XOR is commutative.

  That is: **x (+) y = y (+) x**.

  You can verify this by using truth tables.

Swapping without "temp"

```
temp = x ;
x = y ;
y = temp ;
```

Now solve this without using a temp variable. This means you can ONLY use **x** and **y**. This does NOT mean that you name the variable **temp2**.

```
x = x ^ y ;
y = x ^ y ;
x = x ^ y ;
```

The key to convincing yourself this works is to keep track of the original value of **x** and **y**. Let **A** be the original value of **x** (that is, the value **x** has just before running these three lines of code). Similarly, let **B** be the original value of **y**.

We can comment each line of code to see what's happening.

```
// x == A, y == B
x = x ^ y ;
 // x == A ^ B, y == B
y = x ^ y ;
 // x == A ^ B
 // y == (A ^ B) ^ B == A ^ (B ^ B)  (by Assoc)
 //   == A ^ 0  (by z ^ z == 0 property)
 //   == A      (by z ^ 0 == z property)
x = x ^ y ;
 // x == ( A ^ B ) ^ A
 //   == ( A ^ A ) ^ B  (by Assoc/Commutativity)
 //   == 0 ^ B            (by z ^ z == 0 property)
 //   == B                (by z ^ 0 == z property)
 // y == A
```

After the second statement has executed, **y** = **A**. After the third statement, **x** = **B**.

Writing bitwise XOR without ^

Suppose you wanted to implement bitwise XOR, but didn't have ^ operator. What would you do? With bitwise AND (&) and bitwise OR (|), you can do this.

```
x ^ y == (~x & y) | (x & ~y)
```

This is the standard definition of XOR as defined in logic books, applied to bitwise operations.

Is Bit *i* Set?

Wishing Doesn't Always Make it So

Suppose you had a character variable, declared like:

```
char ch ;
```

How might you set the least significant bit to 1? You might be tempted to do the following:

```
ch[ 0 ] = 1 ; // WRONG!  Does not set bit 0 of ch
```

After all, aren't arrays wonderful? Why *shouldn't* this work? Why shouldn't C allow us this nice and simple way to manipulate bits? Alas, it doesn't.

The reason has to do with how arrays are defined in C.

arr[i] is really defined as ***(arr + i)** where **arr** is **& arr[0]** and **+ i** does pointer arithmetic. The compiler performs the following computatation.

```
address of arr[ i ] is
  addr( arr ) + ( i * sizeof(type of array element) )
```

Thus, you find out what address **arr** is at, then multiply **i** by the number of bytes for the type of one element in the array. This creates an address. Recall that addresses in memory refer to bytes, not to individual bits. That's where the problem lies. The semantics of array access refer to memory addresses of bytes.

Mask

We're going to consider bitstring pattern called a mask. Here's one example of a mask.

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	0	0	0	1	0	0	0

This is a bitstring with exactly one bit that has a value of 1. In this example, $b_3 = 1$.

Similarly, the bitflipped version of this is also considered a mask.

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
1	1	1	1	0	1	1	1

What's the purpose of a mask? The purpose is to either access an individual bit (or range of bits) or to modify an individual bit (or range of bits).

Creating a Mask

Before we see how to use a mask, let's write code to create a mask. This turns out to be rather simple.

```
unsigned char mask = 1 << i ;
```

What does **1 << i** do? The problem is what type is 1? It's most likely an int, which means that it's mostly like 31 zeroes followed by a 1. So, **1 << i** causes $\mathbf{b_i = 1}$ while the rest of the bits are 0.

To create the second kind, you can do bitwise negation on the mask from the first part, as in:

```
unsigned char mask = 1 << i ;
mask = ~mask ;  // flip the bits to create an inverted mask
```

***the simplest way to create a mask is to assign the relevant hexadecimal value to the mask but this is not always applicable…

Using the Mask

Let's see how we can use this mask to determine whether a bit is set or clear. Before we can do that, perhaps we should *define* what it means to be set or clear.

- To ***set* a bit** means to make the value of the bit 1.
- To ***clear* a bit** means to make the value of the bit 0.

Suppose you're asked to write a function that returns true if bit *i* is set, and false if it's clear. We'll call this function **isBitISet()**.

Its code: (Hereinafter we'll use `typedef unsigned char BYTE` for simplicity)

```
int isBitISet( BYTE ch, int i )
{
   BYTE mask = 1 << i ;
   return mask & ch ;
}
```

Here's why it works (i = 3).

ch	$\mathbf{b_7}$	$\mathbf{b_6}$	$\mathbf{b_5}$	$\mathbf{b_4}$	$\mathbf{b_3}$	$\mathbf{b_2}$	$\mathbf{b_1}$	$\mathbf{b_0}$
mask	0	0	0	0	1	0	0	0
result = ch & mask	0	0	0	0	$\mathbf{b_3}$	0	0	0

We don't particularly care what the character looks like. Just assume it's some 8-bit bitstring, $\mathbf{b_7...b_0}$. To distinguish the character from the mask, let's label the bits of the mask as $\mathbf{m_7...m_0}$.

The mask has bit $\mathbf{m_3}$ set, though 3 was picked arbitrarily. Notice what happens when you bitwise AND the mask and the character.

The bits which are 0 in the mask cause the result to also be 0 (since $\mathbf{0 \ \& \ b_i = 0}$). Bit $\mathbf{b_3}$ is ANDed with the '1' appearing in the mask at position $\mathbf{m_3}$. That results in $\mathbf{b_3}$.

If $\mathbf{b_3 = 1}$, then **result** is non-zero. If $\mathbf{b_3 = 0}$, then **result** is zero.

Is Any Bit Set Within a Range?

Introduction

```
int isBitSetInRange( BYTE ch, int low, int high ) ;
```

In order to implement this function, we need to modify the mask. In particular, suppose **low = 3** and **high = 5**, then we would expect the mask to look like:

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	0	1	1	1	0	0	0

How to Create a Mask with a Range

Method 1 Write a for-loop that checks if bit **i** is set, from the previous section.

```
for ( int i = low ; i <= high ; i++ )
  if ( isBitISet( ch, i )
    return TRUE ;
// If it didn't return true in the for-loop, then return false
return FALSE ;
```

However, the main drawback is using a loop. If the bitstring had been much longer (say, 32 or 64 bits), using a loop is more inefficient than not using one.

Method 2 This is somewhat complicated, so we'll need to break it down into steps.

- First, we need to create a mask with the appropriate number of 1's. Suppose we want **k** 1's. Initially, we'll create a mask with **n - k** 0's, followed by **k** 1's. This is easier (for now) than creating the actual mask we need.

  The question is how to compute **k** given **low** and **high**. This is a classic off-by-one situation. Your first instinct might be **high - low**. But this would be incorrect. If the index **high** == **low**, then **high - low** == **0**.

  That's not what you want. When **high** == **low**, you want to test for a single bit, i.e. $\mathbf{b_{high}}$ (which is the same as $\mathbf{b_{low}}$ since **high** == **low**). Thus, you want the formula to equal 1, when **high** == **low**.

  So, the formula should be **high - low + 1**.

  Here's the code to do this.

  ```
  BYTE mask = ~0 ;   // creates all 1's, using bitwise
                     // negation of 0
  int numOnes = (high - low) + 1 ;
  int numBits = sizeof( BYTE ) * 8 ;

  // creates (numBits - numOnes) 0's followed by numOnes 1's
  mask >>= ( numBits - numOnes ) ;
  ```

- Then, we need to shift left to place the 1's in the correct spot. The question is how much to shift left. The answer is abit tricky. Here's how to figure the answer. Think about the least significant bit. It's at position $\mathbf{b_0}$. It currently has a value of 1. Where should this bit end up? Since it's the rightmost bit, it should be the rightmost 1 when we shift left. And where's the rightmost 1? It's at $\mathbf{b_{low}}$. So we need to shift left by **low** bits. The remaining 1's shift left with it, and the leftmost 1 should land at $\mathbf{b_{high}}$ because we computed the number of 1's ahead of time.

```
mask <<= low ;
```

So, the entire code to create the mask looks like:

```
BYTE mask = ~0 ;   // creates all 1's, using bitwise
                   // negation of 0
int numOnes = (high - low) + 1 ;
int numBits = sizeof( unsigned char ) * 8 ;
// creates (numBits - numOnes) 0's followed by numOnes 1's
mask >>= ( numBits - numOnes ) ;
mask <<= low ; // shift left group of 1's to correct location
```

Notice that no loops were used.

The one possible problem we have is the mask being shifted to the right. If we had used a signed char, instead of an unsigned char, we might have had the sign bit shifted in from the left. That would not create the desired mask (instead, it would leave us with all 1's – think why).

So, here's a solution that avoids this problem.

```
BYTE mask = ~0 ;   // creates all 1's, using bitwise
                   // negation of 0
int numOnes = (high - low) + 1 ;

// creates (numBits - numOnes) 1's followed by numOnes 0's
mask <<= numOnes ;

// creates (numBits - numOnes) 0's followed by numOnes 1's
mask = ~mask ;
mask <<= low ; // shift left the group of 1's to correct location
```

By left shifting, we guarantee that 0's are shifted in from the right, regardless of whether the type is signed or unsigned. However, we need the additional step of negating the mask to get the pattern we want. This solution avoids using right shifts, thus should be more portable.

Method 3 The previous solution is moderately complicated to think about, though once you get used to the idea, it isn't all that bad.

A third way to produce the mask is to not only use bitshift operators, but also to use subtraction.

Consider the following two masks.

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
maskHigh	0	0	1	1	1	1	1	1
maskLow	0	0	0	0	0	1	1	1
maskHigh - maskLow	0	0	1	1	1	0	0	0

So, if we could create a mask with **high** 1's and a mask with **low** 1's (as shown above), then subtract, we'd get the desired mask.

How can we get a mask with **k** 1's? To answer this, answer the following question. Suppose you have 1000 in base 10. This is 1, followed by 3 zeroes. If you subtract 1 from 1000, what do you get? You get 999. Suppose you have 100000. This is 1 followed by 5 zeroes. If you subtract 1 from that, you get 99999.

If you have 1, followed by **k** zeroes and you subtract 1 from it, then what do you get? You get **k** 9's.

Now, if this were binary, and you have 1 followed by **k** zeroes, and you subtract 1, what would you get? You'd get **k** 1's.

So, here's how you get **maskHigh** and **maskLow**.

```
BYTE maskHigh = ( 1 << ( high + 1 ) ) - 1 ;
BYTE maskLow = ( 1 << low ) - 1 ;
BYTE mask = maskHigh - maskLow ;
```

There is a slight awkwardness when computing **maskHigh**. We want the leftmost 1 to appear at $\mathbf{b_{high}}$. This means when we create the 1, followed by k zeroes, the "1" has to be shifted to bit position $\mathbf{b_{high + 1}}$. On the other hand, no such problems occur with **maskLow**. The string of 1's we want appear in the correct location by shifting 1 to $\mathbf{b_{low}}$.

If you're observant, you'll notice that when we compute **mask**, we're performing a computation that's of the form **(x - 1) - (y - 1)**. Notice the 1's cancel, and you get **(x - y)**.

So, we can get the mask without subtracting 1, as in:

```
BYTE mask = ( 1 << ( high + 1 ) ) - ( 1 << low ) ;
```

Putting it All Together

Once you create the mask, the code is really simple to check the range.

```
int isBitSetInRange( BYTE ch, int low, int high )
{
   BYTE mask = .... ;
   // code to create mask

   return ch & mask ;
}
```

How to Set a Bit

So far, we've written functions to determine the value of individual bits of a number, or to determine if, within some range of bits, whether any bit was set.

I'm sure you're interested in actually modifying bits. Once you understand masks, it's easy to modify individual bits.

Here's the prototype of the function we wish to implement.

```
BYTE setBit( BYTE ch, int i ) ;
```

This will take a character as input, set $\mathbf{b_i}$ to 1, and return that new character, while leaving the original character untouched (which happens because the original character is passed by value, which means it's a copy of the argument).

```
BYTE setBit( BYTE ch, int i )
{
  BYTE mask = 1 << i ;  // we could cast to BYTE, just to be safe
  return ch | mask ;  // using bitwise OR
}
```

Let's see how this works on an example:

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
ch	1	0	0	0	0	0	1	1
mask	0	0	0	0	1	0	0	0
ch | mask	1	0	0	0	1	0	1	1

Instead of bitwise AND, we use bitwise OR. If you perform bitwise OR on a bit **b** with 0, you get back bit **b**. Thus, for all the bit positions in the mask with zero (in the example above, it's every bit except bit $\mathbf{b_3}$ in the mask), the corresponding bit position of **ch & mask** is just the value of the corresponding bit position in **ch**.

Setting or clearing a bit according to val (0/1):

```
BYTE setBit( BYTE ch, int i, int val )
{
  BYTE mask = 1 << i ;
  if( val == 1) // setting the bit
        return ch | mask ;  // using bitwise OR
  else{
        mask = ~mask;  // clearing the bit
        return ch & mask ;  // using bitwise AND
  }
}
```