Bitwise Operators in C and C++

Generally, as a programmer you don't need to concern yourself about operations at the bit level. You're free to think in bytes, or ints and doubles, or even higher level data types composed of a combination of these. But there are times when you'd like to be able to go to the level of an individual bit. Exclusive-or encryption is one example when you need bitwise operations.

Another example comes up when dealing with data compression: what if you wanted to compress a file? In principle, this means taking one representation and turning it into a representation that takes less space. One way of doing this is to use an encoding that takes less than 8 bits to store a byte. (For instance, if you knew that you would only be using the 26 letters of the Roman alphabet and didn't care about capitalization, you'd only need 5 bits to do it.) In order to encode and decode files compressed in this manner, you need to actually extract data at the bit level.

Finally, you can use bit operations to speed up your program or perform neat tricks. (This isn't always the best thing to do.)

Thinking about Bits

The byte is the lowest level at which we can access data; there's no "bit" type, and we can't ask for an individual bit. In fact, we can't even perform operations on a single bit -- every bitwise operator will be applied to, at a minimum, an entire byte at a time. This means we'll be considering the whole representation of a number whenever we talk about applying a bitwise operator. (Note that this doesn't mean we can't ever change only one bit at a time; it just means we have to be smart about how we do it.) Understanding what it means to apply a bitwise operator to an entire string of bits is probably easiest to see with the shifting operators. By convention, in C and C++ you can think about binary numbers as starting with the most significant bit to the left (i.e., 10000000 is 128, and 00000001 is 1). Regardless of underlying representation, you may treat this as true. As a consequence, the results of the left and right shift operators are not implementation dependent for unsigned numbers (for signed numbers, the right shift operator is implementation defined).

The leftshift operator is the equivalent of moving all the bits of a number a specified number of places to the left:

```
[variable]<<[number of places]
```

For instance, consider the number 8 written in binary 00001000. If we wanted to shift it to the left 2 places, we'd end up with 00100000; everything is moved to the left two places, and zeros are added as padding. This is the number 32 -- in fact, left shifting is the equivalent of multiplying by a power of two.

```
int mult_by_pow_2(int number, int power)
{
    return number<<power;
}
```

Note that in this example, we're using integers, which are either 2 or 4 bytes, and that the operation gets applied to the entire sequence of 16 or 32 bits.

But what happens if we shift a number like 128 and we're only storing it in a single byte: 10000000? Well, 128 * 2 = 256, and we can't even store a number that big in a byte, so it shouldn't be surprising that the result is 00000000.

It shouldn't surprise you that there's a corresponding right-shift operator: >> (especially considering that I mentioned it earlier). Note that a bitwise right-shift will be the equivalent of integer division by 2.

Why is it integer division? Consider the number 5, in binary, 00000101. 5/2 is 2.5, but if you are performing integer division, 5/2 is 2. When you perform a right shift by one: (unsigned int) 5>>1, you end up with 00000010, as the rightmost 1 gets shifted off the end; this is the representation of the number 2. Note that this only holds true for unsigned integers; otherwise, we are not guaranteed that the padding bits will be all 0s.

Generally, using the left and write shift operators will result in significantly faster code than calculating and then multiplying by a power of two. The shift operators will also be useful later when we look at how to manipulating individual bits.

For now, let's look at some of the other binary operators to see what they can do for us.

Bitwise AND

The bitwise AND operator is a single ampersand: &. A handy mneumonic is that the small version of the boolean AND, &&, works on smaller pieces (bits instead of bytes, chars, integers, etc). In essence, a binary AND simply takes the logical AND of the bits in each position of a number in binary form.

For instance, working with a byte (the char type):

```
01001000 &
10111000 =
--------
00001000
```

The most significant bit of the first number is 0, so we know the most significant bit of the result must be 0; in the second most significant bit, the bit of second number is zero, so we have the same result. The only time where both bits are 1, which is the only time the result will be 1, is the fifth bit from the left. Consequently,

```
72 & 184 = 8
```

Bitwise OR

Bitwise OR works almost exactly the same way as bitwise AND. The only difference is that only one of the two bits needs to be a 1 for that position's bit in the result to be 1. (If both bits are a 1, the result will also have a 1 in that position.) The sybmol is a pipe: |. Again, this is similar to boolean logical operator, which is ||.

```
01001000 |
10111000 =
--------
11111000
```

and consequently

```
72 | 184 = 248
```

Let's take a look at an example of when you could use just these four operators to do something potentially useful. Let's say that you wanted to keep track of certain boolean attributes about something -- for instance, you might have eight cars (!) and want to keep track of which are in use. Let's assign each of the cars a number from 0 to 7.

Since we have eight items, all we really need is a single byte, and we'll use each of its eight bits to indicate whether or not a car is in use. To do this, we'll declare a char called in_use, and set it to zero. (We'll assume that none of the cars are initially "in use".)

```
char in_use = 0;
```

Now, how can we check to make sure that a particular car is free before we try to use it? Well, we need to isolate the one bit that corresponds to that car. The strategy is simple: use bitwise operators to ensure every bit of the result is zero except, possibly, for the bit we want to extract.

Consider trying to extract the fifth bit from the right of a number: XX?XXXXX We want to know what the question mark is, and we aren't concerned about the Xs. We'd like to be sure that the X bits don't interfere with our result, so we probably need to use a bitwise AND of some kind to make sure they are all zeros. What about the question mark? If it's a 1, and we take the bitwise AND of XX?XXXXX and 00100000, then the result will be 00100000:

```
XX1XXXXX &
00100000 =
--------
00100000
```

Whereas, if it's a zero, then the result will be 00000000:

```
XX0XXXXX &
00100000 =
--------
00000000
```

So we get a non-zero number if, and only if, the bit we're interested in is a 1.

This procedure works for finding the bit in the nth position. The only thing left to do is to create a number with only the one bit in the correct position turned on. These are just powers of two, so one approach might be to do something like:

```
int is_in_use(int car_num)
{
    // pow returns an int, but in_use will also
be promoted to an int
    // so it doesn't have any effect; we can
think of this as an operation
    // between chars
    return in_use & pow(2, car_num);
}
```

While this function works, it can be confusing. It obscures the fact that what we want to do is shift a bit over a certain number of places, so that we have a number like 00100000 -- a couple of zeros, a one, and some more zeros. (The one could also be first or last -- 10000000 or 00000001.)

We can use a bitwise leftshift to accomplish this, and it'll be much faster to boot. If we start with the number 1, we are guaranteed to have only a single bit, and we know it's to the far-right. We'll keep in mind that car 0 will have its data stored in the rightmost bit, and car 7 will be the leftmost.

```
int is_in_use(int car_num)
{
```

```
    return in_use & 1<<car_num;
}
```

Note that shifting by zero places is a legal operation -- we'll just get back the same number we started with.

All we can do right now is check whether a car is in use; we can't actually set the in-use bit for it. There are two cases to consider: indicating a car is in use, and removing a car from use. In one case, we need to turn a bit on, and in the other, turn a bit off.

Let's tackle the problem of turning the bit on. What does this suggest we should do? If we have a bit set to zero, the only way we know right now to set it to 1 is to do a bitwise OR. Conveniently, if we perform a bitwise OR with only a single bit set to 1 (the rest are 0), then we won't affect the rest of the number because anything ORed with zero remains the same (1 OR 0 is 1, and 0 OR 0 is 0).

Again we need to move a single bit into the correct position: void set_in_use(int car_num) { in_use = in_use | 1<<car_num; } What does this do? Take the case of setting the rightmost bit to 1: we have some number 0XXXXXXX | 10000000; the result, 1XXXXXXX. The shift is the same as before; the only difference is the operator and that we store the result.

Setting a car to be no longer in use is a bit more complicated. For that, we'll need another operator.

The Bitwise Complement

The bitwise complement operator, the tilde, ~, flips every bit. A useful way to remember this is that the tilde is sometimes called a twiddle, and the bitwise complement twiddles every bit: if you have a 1, it's a 0, and if you have a 0, it's a 1.

This turns out to be a great way of finding the largest possible value for an unsigned number:

```
unsigned int max = ~0;
```

0, of course, is all 0s: 00000000 00000000. Once we twiddle 0, we get all 1s: 11111111 11111111. Since max is an unsigned int, we don't have to worry about sign bits or twos complement. We know that all 1s is the largest possible number.

Note that ~ and ! cannot be used interchangeably. When you take the logical NOT of a non-zero number, you get 0 (FALSE). However, when you twiddle a non-zero number, the only time you'll get 0 is when every bit is turned on. (This non-equivalence principle holds true for bitwise AND too, unless you know that you are using strictly the numbers 1 and 0. For bitwise OR, to be certain that it would be equivalent, you'd need to make sure that the underlying representation of 0 is all zeros to use it interchangeably. But don't do that! It'll make your code harder to understand.)

Now that we have a way of flipping bits, we can start thinking about how to turn off a single bit. We know that we want to leave other bits unaffected, but that if we have a 1 in the given position, we want it to be a 0. Take some time to think about how to do this before reading further.

We need to come up with a sequence of operations that leaves 1s and 0s in the non-target position unaffected; before, we used a

bitwise OR, but we can also use a bitwise AND. 1 AND 1 is 1, and 0 AND 1 is 0. Now, to turn off a bit, we just need to AND it with 0: 1 AND 0 is 0. So if we want to indicate that car 2 is no longer in use, we want to take the bitwise AND of XXXXX1XX with 11111011.

How can we get that number? This is where the ability to take the complement of a number comes in handy: we already know how to turn a single bit on. If we turn one bit on and take the complement of the number, we get every bit on except that bit:

```
~(1<<position)
```

Now that we have this, we can just take the bitwise AND of this with the current field of cars, and the only bit we'll change is the one of the car_num we're interested in.

```
int set_unused(int car_num)
{
    in_use = in_use & ~(1<<position);
}
```

You might be thinking to yourself, but this is kind of clunky. We actually need to know whether a car is in use or not (if the bit is on or off) before we can know which function to call. While this isn't necessarily a bad thing, it means that we do need to know a little bit about what's going on. There is an easier way, but first we need the last bitwise operator: exclusive-or.

Bitwise Exclusive-Or (XOR)

There is no boolean operator counterpart to bitwise exclusive-or, but there is a simple explanation. The exclusive-or operation takes two inputs and returns a 1 if either one or the other of the inputs is a 1, but not if both are. That is, if both inputs are 1 or both inputs are 0, it returns 0. Bitwise exclusive-or, with the operator of a carrot, ^, performs the exclusive-or operation on each pair of bits. Exclusive-or is commonly abbreviated XOR.

For instance, if you have two numbers represented in binary as 10101010 and 01110010 then taking the bitwise XOR results in 11011000. It's easier to see this if the bits are lined up correctly:

```
01110010 ^
10101010
--------
11011000
```

You can think of XOR in the following way: you have some bit, either 1 or 0, that we'll call A. When you take A XOR 0, then you always get A back: if A is 1, you get 1, and if A is 0, you get 0. On the other hand, when you take A XOR 1, you flip A. If A is 0, you get 1; if A is 1, you get 0.

So you can think of the XOR operation as a sort of selective twiddle: if you apply XOR to two numbers, one of which is all 1s, you get the equivalent of a twiddle.

Additionally, if you apply the XOR operation twice -- say you have a bit, A, and another bit B, and you set C equal to A XOR B, and then take C XOR B: you get A XOR B XOR B, which essentially either flips every bit of A twice, or never flips the bit, so you just get back A. (You can also think of B XOR B as cancelling out.) As

an exercise, can you think of a way to use this to exchange two integer variables without a temporary variable? (Once you've figured it out, check the solution.)

How does that help us? Well, remember the first principle: XORing a bit with 0 results in the same bit. So what we'd really like to be able to do is just call one function that flips the bit of the car we're interested in -- it doesn't matter if it's being turned on or turned off -- and leaves the rest of the bits unchanged.

This sounds an awful lot like the what we've done in the past; in fact, we only need to make one change to our function to turn a bit on. Instead of using a bitwise OR, we use a bitwise XOR. This leaves everything unchanged, but flips the bit instead of always turning it on:

```
void flip_use_state(int car_num)
{
    in_use = in_use ^ 1<<car_num;
}
```

When should you use bitwise operators?

Bitwise operators are good for saving space -- but many times, space is hardly an issue. And one problem with working at the level of the individual bits is that if you decide you need more space or want to save some time -- for instance, if we needed to store information about 9 cars instead of 8 -- then you might have to redesign large portions of your program. On the other hand, sometimes you can use bitwise operators to cleverly remove dependencies, such as by using ~0 to find the largest possible integer. And bit shifting to multiply by two is a fairly common operation, so it doesn't affect readability in the way that advanced use of bit manipulation can in some cases (for instance, using XOR to switch the values stored in two variables).

There are also times when you need to use bitwise operators: if you're working with compression or some forms of encryption, or if you're working on a system that expects bit fields to be used to store boolean attributes.

Summary

You should now be familiar with six bitwise operators:

Works on bits for left argument, takes an integer as a second argument

```
bit_arg<<shift_arg
```

Shifts bits to of bit_arg shift_arg places to the left -- equivalent to mulitplication by 2^shift_arg

```
bit_arg>>shift_arg
```

Shifts bits to of bit_arg shift_arg places to the right -- equivalent to integer division by 2^shift_arg

Works on the bits of both arguments

```
left_arg & right_arg
```

Takes the bitwise AND of left_arg and right_arg

```
left_arg ^ right_arg
```

Takes the bitwise XOR of left_arg and right_arg

```
left_arg | right_arg
```

Works on the bits of only argument

```
~arg
```

Reverses the bits of arg

Skills and knowledge You also know a couple of neat tricks that you can use when performance is critical, or space is slow, or you just need to isolate and manipulate individual bits of a number.

And you now should have a better sense of what goes on at the lowest levels of your computer.

A Parting Puzzle

One final neat trick of bitwise operators is that you can use them, in conjunction with a bit of math, to find out whether an integer is a power of two. Take some time to think about it, then check out the solution.