FEATURE

Techie's Guide to C Programming Part 9

Bytes are fairly easy to cope with. Bits are decidedly nastier. In this installment of the C saga, we'll look at how the bit manipulation facilities of C work.

STEVE RIMMER

Bytes are, of course, composed of bits. Most dogs can tell you this if they're suitably inspired. However, whereas the relationship of bytes to the universe is a fairly obvious one, that of their component bits is not. Working with data at the bit level is tricky. It's not something which BASIC gives you any facilities at all for, and, as such, you may not have tried to do it.

Bitwise manipulation is a tremendously useful tool for all sorts of program applications, but it's very nearly essential for dealing with DOS, which... being basically an assembly language environment... casually flings its bits around with wild abandon.

This month we're going to get a handle on how C copes with bits. Needless to say, more punctuation is in the offing.

C Scapes

Most of the bytes you encounter on the street have eight bits in 'em. While basic arithmetic operators do not allow you to manipulate these directly, they all affect the bits.

Under C on the PC, the *char* data type is effectively a byte. An *int* is a word, or two bytes, and C is very flexible in interchanging these two. Allowing that *a* is a *char* variable, if we say *a = 255*, all the bits in *a* will be set, that is, they'll all be one. It's a lot easier to see this in hex notation, actually. Let's say *a = 0xff*.

The hex number FF consists of two *nybbles*, each of which is F, or sixteen. A nybble is four bits wide. This is how the bits in a nybble work.

Decimal	Hex	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Techie's Guide to C Programming, Part 9

As you can see, the sixteen permutations of a nybble involve all the possible bit patterns. As long as you can remember the bit patterns for the sixteen hex digits from zero through F, you can work out the bit pattern for a byte shown in hexadecimal notation.

Now, there are some basic bit manipulation techniques which C offers us, and they prove to be very useful. Let's start with the simple binary operators.

The most commonly used bitwise operator under C is the AND operator, &. To begin with, you have to be careful not to confuse it with the logical AND operator, &&. If we say something like

```
if(a == 6 && cat == dead) ...
```

we are using the *logical* operator. The bitwise operator is different.

Let's consider the result of six AND three. The kids'll tell you that it's nine. In fact, it's two. No amount of head scratching will work out how that came to pass, however, unless you understand bitwise arithmetic.

If you AND two numbers together, the resulting number will be one in which any bits common to the two numbers are set. As such, six, which has the bit pattern 0110 and three, which has the bit pattern 0011, AND together to form the bit pattern 0010, or two.

Under C would say something like this:

```
a = 6 & 3;
```

The AND operator is very often used for masking off unwanted bits. For example, if you use the *biosequip* function of Turbo C, you'll be able to tell all sorts of useful things about the hardware that your program is running on. One of these very useful things is the type of video card being used. The video card type is held in the fourth and fifth bits of the integer returned by this function. The other bits hold other information, which we aren't concerned with here. Let's start by doing this:

```
a = biosequip() & 0x0030;
```

The number being ANDed with the *biosequip* value masks off all but the fourth and fifth bits. Check out the bit table above if you aren't sure why this is.

Here's what the two remaining bits mean.

Bits45
00Unused (EGA)
0140 Column CGA
1080 Column CGA
11Hercules card

If *a* is 0x0030, that is, if both bits are set, it's a Herc card. If it's 0x0010 or 0x0001 it's a CGA card. If it's 0x0000... no bits set... it's unused or an EGA card.

More Bits

The next most common bitwise operator is OR, which is handled under C by the | symbol. Again, there's a logical OR, ||, which should not be confused with its bitwise cousin.

If you OR two numbers together, the resulting number will have its bits set where set bits existed in either of the two source numbers. Thus, six OR three would be seven... that's 0110 OR 0011, which gives you 0111. In C, we would say

```
a = 6 | 3;
```

The XOR, or "exclusive OR", operator toggles bits. It's handled by the ^ symbol. If we XOR a number with a second number, any set bits which are common to the two will be toggled in the first number. As such, six XOR three gives us four... that's 0110 XOR 0011. The second bit in the second number toggles the second bit in the first, leaving us with 0100.

The XOR operator is useful in doing bit mapped graphics, among other things, although, as most C compiler packages come with graphics libraries, you'll probably never have to use it for this. It's also good for dealing with the machine language interface which C provides for the PC. We'll be looking at this in a future installment of this series.

Finally, there's the NOT operator, handled by the tilde character. That's one of these, ~. This simply inverts all the bits in its destination. For example, the result of NOT 1 is 0xfe. You might write that as:

```
a = ~1;
```

In addition to manipulating bits, C lets you change their positions within a byte by *shifting*. Shifting involves moving all the bits in a byte to the right or the left by a defined number of positions. The shift operators, < to shift left and > to shift right, work like this:

```
a = 1 < 2;
```

In this case, we have shifted the number one, 0001 in binary, left by two positions, making it 0100, or four. If we did this:

```
a = a > 2;
```

it would be one again.

Shifting an integer to the left by one place is equivalent to multiplying it by two. Shifting it to the left by two places multiplies it by four, and so on. This is a good thing to know, as multiplication takes an awful lot longer than bit shifting. Some compilers automatically substitute bit shift operating where they can be used in place of multiplication when they're optimizing your code, but there are a lot of cases wherein this doesn't happen unless you make it happen.

Shifting bits right is equivalent to division in the same way.

But is it Useful?

There are all sorts of practical uses for bitwise operations under C. Let's start with a simple example.

We're going to write a game here. I don't exactly know what the object of it is... it's kind of irrelevant... but it's played on a sixteen by sixteen grid with pennies. Actually, pennies are too small... we'll use loonies, which aren't worth a whole lot more. In this game, any square on the board can be occupied by a loonie or it can be blank. We need a way of storing the condition of each square in memory.

This would be the obvious way.

```
char game_board[16][16];
```

This would create an array of *char*s, with one *char* per square on the board. If a square is occupied, its corresponding element in the array would be non-zero.

This approach is easy, but that array sucks back a quarter of a kilobyte. More to the point, seven eighths of it is redundant. We're using eight bits per element for a binary condition... only one bit is really needed.

Here's a better approach using a bitwise approach.

```
unsigned int game_board[16];
```

This requires thirty two bytes. If the game calls for the board to be initially empty, we would initialize this array as follows.

```
for(i=0;i;++i) game_board[i]=0;
```

Now, let's see how we would place a loonie on the board, that is, how we would set a bit in this array. We'll allow that each *int* in this array represents a vertical row on the board, and the bit position represents the horizontal position in that row. We want to place a loonie on square six of row twelve. Note that both the row and column numbers start at zero. This how it's done.

```
game_board[12] |= (1 < 6);
```

The notation "|=" may be a bit confusing. The above line is equivalent to this:

```
game_board[12] = game_board[12] | (1 < 6);
```

What we've done here is to create a "mask" which represents the bit position in the twelfth element of the array equivalent to the sixth place across. This is easy to see... we've just taken one, which is the first bit set, and marched it across by six places. We then OR it with the integer in question. If the position had already been filled by a previous loonie, nothing would happen. However, as the bit is unset... since we initialized the array to zero... the OR operation sets the bit to one.

Suppose our game called for toggling the status of the squares on the board, such that if one loonie was placed atop another both were removed from the board. We could do this with the XOR operator, like this:

```
game_board[12] ^= (1 < 6);
```

Finally, we could test the status of any element in the array with the AND operator.

```
if(game_board[12] & (1 < 6)) ...
```

This might be a bit obtuse at first. If the bit in question was set, ANDing the integer with our mask would result in an integer with one bit set. This would make it non-zero, and the condition would be true. If the bit was not set, any other bits in the integer would be masked off, and the result would be zero. The condition would be false. Note that here, we do not actually affect the contents of the array *game_board*, but only copy its contents out and manipulate them. In fact, C creates temporary variables to do this in, but we never see them.

Binary Breakdance

In practice, situations which call for bitwise manipulation occur quite frequently in programming. If you're used to writing in BASIC, you'll probably have encountered these and dealt with them in the sorts of convoluted ways that BASIC imposes on its users. C lets you get right down there and meddle with the bits, which is a great deal more flexible.

Assembly language is even more prone to using bitwise approaches to things. When we get into the assembly language interface in C, you'll find bitwise operators flung all over creation. ■