Playing with bits

Lecture 10.01

Outline

- Shifting bits
- Bitwise operators: &, |, ~,^
- Using bits for yes/no flags
- Applications
- Bit puzzles

Recap: numeric information

- *Numerals* and *numeral systems*: symbols and collections of symbols used to represent small numbers, together with rules for representing larger numbers
- Most famous numeral system: *decimal* – basis of all modern math

Symbols:

0,1,2,3,4,5,6,7,8,9

Rules for bigger numbers

2	1	0	Positions
2	**3**	**1**	
hundreds	tens	ones	
$2*10^2$	$3*10^1$	$1*10^0$	
200+30+1=**231**			

On and off: 2 states

- Computers use correspondence of current in a digital circuit (on) and absence of it (off) to represent only two digits: 0 and 1
- This is called a *binary* numeral system
- Basic numerals are 0 and 1, but the **rules** of creating larger numbers **are the same** as for the decimal system:

Symbols:

0,1

Rules for bigger numbers

7	6	5	4	3	2	1	0
1	**1**	**1**	**0**	**0**	**1**	**1**	**1**
128s	64s	32s	16s	8s	4s	2s	1s
$1*2^7$	$1*2^6$	$1*2^5$	$0*2^4$	$0*2^3$	$1*2^2$	$1*2^1$	$1*2^0$
128+64+32+4+2+1=**231**							

Binary digits (bits) and bytes

One byte: 8 bits

Single bit: on or off

7	6	5	4	3	2	1	0
1	**1**	**1**	**0**	**0**	**1**	**1**	**1**
128s	64s	32s	16s	8s	4s	2s	1s
$1*2^7$	$1*2^6$	$1*2^5$	$0*2^4$	$0*2^3$	$1*2^2$	$1*2^1$	$1*2^0$
128+64+32+4+2+1=**231**							

What is the largest number we can represent with 8 bits (1 byte)?

Try binary addition

```
 111      101
+110     +111
```

Or subtraction

```
 100      101
-  1      -11
```

Binary numbers: multiplication

- Multiplication in the binary system works the same way as in the decimal system:

```
      101
*      11
      101
     101
     1111
```

The same:
1*1=1
1*0=0
0*1=0

- Note that *multiplying by two* is extremely easy. To multiply by two, just add a 0 on the end (same as multiplying by 10 in a decimal system)

We think in bytes (8 bits at a time)

- We think about memory in ***bytes***, or ***ints*** and ***doubles***, or even in ***structs*** composed of multiple bytes
- The ***byte*** is the lowest level at which we can access data in C: there's no "bit" type, and we can't ask for an individual bit

When do we want individual bits

- *Compress*: take one representation and turn it into a representation that takes less space:
  - How many bits do we need to represent any of 26*2 letters of English alphabet?
  - Of DNA alphabet?
- *Speedup*: bit operations are extremely fast
- *Encrypt*: fast and simple XOR encryption

Thinking about Bits

- The minimum unit of memory is *byte* => we can't even perform operations on a single bit
- This means we'll be considering the whole representation of a number when applying a bitwise operator
- But the goal is to be able to access individual bit: to get and set its value

unsigned

- We apply bitwise operators to unsigned integral values only, because some operations for signed numbers are hardware and system-dependent
- In case of unsigned char you can think about binary numbers as starting with the most significant bit to the left:

10000000 is 128

00000001 is 1

- We do not care about endianness: all bitwise operators are implemented to read the numbers from left to right

The left-shift operator <<

- Shifting 1-bits in *variable n_places* to the left:

[variable]<<[n_places]

00001000 << 2

↓

00100000

- Left shifting is equivalent to multiplying by a power of two:

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

Shifting away

unsigned char c = 128 //(1 byte)

c << 1 = ?

- 128 * 2 = 256, we can't even store a number that big in a byte

1000 0000 << 1

↓

00000000

The right-shift operator >>

- Shifting 1-bits in *variable n_places* to the right:

[variable]>>[n_places]

unsigned char c = 8;

- 00001000

b = c>>2

- 00000010

- A bitwise right-shift is equivalent to integer division by 2
- Note that this only holds for unsigned integers; otherwise, we are not guaranteed that the padding bits will be all 0s

Speedup trick: dividing by 2^n - multiplying by 2^n

- Using the left and right shift operators will result in significantly faster code than calculating 2^n and then multiplying or dividing:

```
void mult_power_2(unsigned int *num, int pow){
    *num = *num << pow;
}

void divide_power_2(unsigned int *num, int pow){
    *num = *num >> pow;
}
```

Bitwise AND &

- The small version of the Boolean AND (&&) works on smaller pieces (bits instead of bytes, chars, integers…)
- A binary AND & takes the logical AND of two bits in the same position of two numbers

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

72 & 184 = 8

- The result is 1 only when both bits are 1 (the fifth bit from the left)

Bitwise OR |

- Bitwise OR takes a Boolean OR for each separate bit in the corresponding position of two numbers
- Only one of the two bits needs to be a 1 for the bit in the result to be 1.

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

72 | 184 = 248

Sample application 1: Bit flags

- You have eight cars (!)
- You want to keep track of which are in use
- Let's assign each of the cars a number from 0 to 7

- To store the state of each car we need a single byte, where we use each of its eight bits to indicate whether or not a car is in use
- We'll assume that none of the cars are initially "in use"

```
unsigned char in_use = 0;                    //00000000
```

Checking whether the car at index 5 is in use

- We need to isolate the one bit that corresponds to that car
- Extract the fifth bit from the right of a number: XX?XXXXX
- If we take the bitwise AND of XX?XXXXX and 00100000, then the result will be 0 if car is not in use, and >0 otherwise

```
XX1XXXXX &          XX0XXXXX &
00100000 =          00100000 =
--------            --------
00100000            00000000
```

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

Finding the bit in the *n*-th position

```
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.

Setting *n*-th bit on (car in use)

- 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)

```
void set_in_use(int car_num) {
    in_use = in_use | 1<<car_num;
}
```

- For example in case of setting the leftmost bit to 1: we have some number 0XXXXXXX | 10000000 - the result is 1XXXXXXX

Bitwise NOT ~

- The bitwise complement operator, the tilde, ~, flips every bit

- Trick: The largest possible value for an unsigned number:

unsigned int max = ~0;

- Zero is: 00000000 00000000
- Once we twiddle 0, we get all 1s: 11111111 11111111
- All 1s is the largest possible number

~ vs. !

- Note the big difference between ~ and ! : they cannot be used interchangeably
  - When you take the logical NOT (!) of a non-zero number, you get 0 (FALSE)
  - When you twiddle a non-zero number with ~, the only time you'll get 0 is when every bit was turned on

Turning the *n*-th bit off

- We need to leave 1s and 0s in non-target positions unaffected
- We need to set the *n*-th bit to 0

- To turn off a bit, we just need to AND it with 0: 1 AND 0 is 0
- If we want to indicate that car 2 is no longer in use, we want to take the bitwise AND of XXXXX**1**XX with 11111**0**11

- How can we get that number?

~(1<<position)

Set car state to unused

- The only bit we'll change is the one of the car_num we're interested in:

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

Bitwise Exclusive-Or (XOR) ^

- There is no Boolean operator counterpart to bitwise exclusive-or
- The exclusive-or (XOR) takes two inputs and returns a 1 only if both Boolean inputs are different
- Bitwise XOR performs the exclusive-or operation on each pair of bits

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

0	0	0
0	1	1
1	0	1
1	1	0

Thinking about XOR

- 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
- When you take A XOR 1, you flip A. If A is 0, you get 1; if A is 1, you get 0

0	0	0
0	1	1
1	0	1
1	1	0

Magic properties of double XOR

- If you apply XOR twice

C = A XOR B

D = 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

Magic trick: swapping numbers with XOR (no temp variable)

```
void swap (int *a, int *b) {
   *a = *a ^ *b;
   // Now, we can recover *a_orig by applying *a XOR *b_orig
   *b = *a ^ *b;
   // The value originally stored in *a, a_orig, is now in *b
   // and *a still stores a_orig ^ b_orig
   // This means that we can recover the value of b_orig by applying
   // the XOR operation to *a and a_orig.  Since *b stores a_orig...
   *a = *a ^ *b
}
```

A = 0101

B = 1001

A = A^B = 1100 Parity bits!

B = A^B = 0101 Which is A!

A = A^B = 1001 Which is B!

Very similar to the regular non-bitwise method

```
void swap (int *a, int *b) {
    *a = *a + *b;
    *b = *a  - *b; //now contains original *a
    *a = *a - *b; //now contains original *b
}
```

It is in essence the same:

XOR operator complements all bits so they become even

Flipping *n*-th bit

- XORing bit with 0 results in the same bit
- XORing bit with 1 flips it
- We can just flip the bit of the car we're interested in -- it doesn't matter if it's being turned on or turned off -- and leave the rest of the bits unchanged

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

Sample application 2: XOR encryption

- Very simple way of disguising a piece of text by XOR-ing each character with some value
- The same code that can encrypt text can also be used to decrypt it.

```
void encrypt(char *message) {
    char c;
    while (*message) {
        *message = *message ^ 31;
        message++;
    }
}
```

0	0	0
0	1	1
1	0	1
1	1	0

0	0	0
1	1	0
1	0	1
0	1	1

Sample application 3: WEXITSTATUS

```
#define __WEXITSTATUS(status) (((status) & 0xff00) >> 8)
```

- The unsigned int *status* passed to *waitpid*() encodes both the reason that the child process was terminated and the exit code
- The reason is stored in the least-significant byte (obtained by status & 0xff), and the exit code is stored in the next byte (masked by status & 0xff00 and extracted by WEXITSTATUS())

Sample application 4. sigset_t

- *sa_mask* field of *struct sigaction* is of type *sigset_t*
- Internally, it may be implemented as either an integer or structure type

How would you solve these puzzles?

- Convert DNA string of length 4 (which occupies 4 bytes) into a single unsigned incharteger (which occupies 1 byte only)
- If you encoded the answers to 32 categorizer questions as a single unsigned int:
  - How would you find the like-minded individuals with 1 operation?
  - How would you find best mismatches?

Tricky question: Find out if the number is a power of 2 in one operation

- Any power of 2 minus 1 is all ones: (2^N - 1 = 111....b) :
  - A power of two looks like this : 01000000 - a string of zeros, with a lone one
  - If you subtract 1 from a power of two, you'll get: 01000000 - 00000001 = 00111111 - a string of ones!

- If you take the bitwise AND of the two values, you get 0

Solution

```
int is_power_of2(int x) {
   return !((x-1) & x);
}
```

- Note that we have to use the logical NOT, !, instead of the bitwise complement since the bitwise complement will not negate non-zero values; it just flips bits.