Section Notes - Week 1 (9/17)

Why do we need to learn bits and bitwise arithmetic?
Since this class is about learning about how computers work. For most of the rest of the semester, you do not have to worry about bits because they are abstracted behind the arithmetic of your programming language. However, because you are working with C, you need an intimate knowledge of the what your code is actually doing at the bit-level, in order to prevent bugs and in order to code well.

Bit Arithmetic
Go through bitwise &, |, ^, <<, >> ,~ in whichever way you'd like

Some quizzical problems/examples:
1. x ^ (~x) = ?
    - Answer: 0xFFFFFFFF
    - (x & ~x → point out that this can be used by compilers to get a 0 in a register)
2. Bit masking!
    - This is a way to extract bits at any location you want.
3. Warning: the following are not the same!
   a) ~~x = ?
   b) !!x = ?

Answers:
   a) ~~x = x
   b) !!x = (x != 0)

LOGICAL OPERATIONS
4. What is (x > 0) - (x < 0)?
5. What is (x > y) - (x < y)?
6. What is (x || y) - (x && y)?
7. What is !x != !y?

2's Complement
- This is an easy way to preserve intuitive arithmetic at the bit-level while representing negative integers at the same time.

1) Give an example of sign/magnitude. And ask people to do the addition--> wrong answer!!
   For example: (using 4 bits for ease of discussion)
   1001 == -1
+ 0011 == 3
= 1100 == -4

This is bad!! Regular addition doesn't work in something like sign-magnitude.
Two's complement fixes this!

What is 2s complement?
- It is simply a bit **representation**. (this is important, I think it confused a lot of people).
- It is not a way to add numbers. It's not a particular base (like base 2 or something). It is a bit **interpretation**. Whenever we have a string of bits, we can interpret it differently.

- Binary (base 2) representation is also an **interpretation** of bits, because it assumes that everything is an **unsigned integer.** Binary tells you that to interpret a string of bits, you multiply the i-th digit (counting from the right) by 2^i (starting at i=0).
- 2s complement tells you that to interpret a string of bits, multiply the i-th digit (counting from the right) by 2^i (starting at i=0) EXCEPT for the most significant bit (the bit farthest to the left).
- The most significant bit is **interpreted** as the presence of -2^31 in the number we are representing. If the most significant bit is a 1, then we have -2^31. If the most significant bit is a 0, then we ignore it.

- Therefore, we see that 2s complement gives you a way to represent positive and **negative** numbers. 2s complement also preserves regular arithmetic.

For example:
  1001 == -7
+ 0101 == 5
= 1110 == -2
ARITHMETIC WORKS!

Why does arithmetic work? (Work this out in i*2^i form. Remember, everything is mod 2^31).

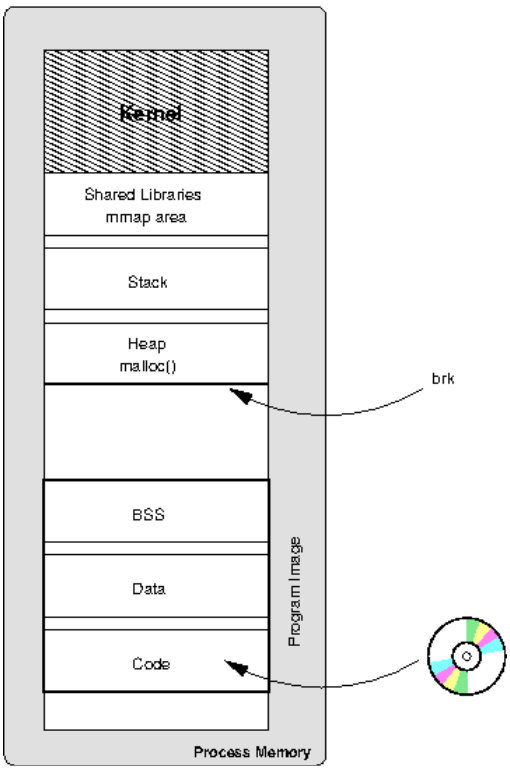
Therefore, for all these lovely reasons, 2s complement is the primary way bits are represented in a machine (for non-unsigned things that is).

Floating point arithmetic and the way it is represented (Sign, Exponent, Mantissa).
Caveats when doing comparisons with floating-point values (truncation etc.)

Memory and Datas

Whole Memory System Layout

- Go over memory layout (JUST A BUNCH OF BYTES really).
- Confusing: memory is indexed by bytes. But omg the data sitting at the memory location is also in bytes.
- In some programming languages, this is is abstracted from you. Unfortunately, in C, nothing is abstracted from you! So you have to be mindful whenever you code of whether you are working with data or with pointers (which is the memory location).
- Draw a picture of memory. Like this (maybe drawn to scale a bit better):



Talk about said picture. Relevant points: what the compiler does, what happens when you do an objdump (bss, text, data segments). Virtual memory, what the kernel is (very loosely), what the heap and stack are.

In particular, the heap

- The heap is a region of the address space reserved for dynamic/explicit memory allocations.
- This is contrasted with the stack, which is where static allocations are made and where the runtime environment lives.
- The heap is **dangerous** in C, because it is controlled explicitly by the user.

malloc() is the primary function by which the user allocates memory on the heap. It is up to the **user** to behave responsibly on the heap, which is the reason for Assignment 1. In particular, go over the following memory management errors that a programmer can make (these are stolen from the Asst 1 spec):
(For each problem, it might be a good exercise to first present the problem and then ask students why this might be bad, before going on to explain.)
1. Not freeing malloc-ed memory. This is bad because it might eat up all the memory in a system == crash.
2. Writing in freed/unallocated regions. Bad because this region can reallocated, in which case your data is no longer valid.
3. Writing outside your allocated regions!! This causes many problems: overwriting data in other regions, or thinking that your data is valid, but it's not because it might get reallocated and written over.

Data representations (yay) and how they are laid out in memory

- Primary data types: int, char, * (star anything is a pointer and is 4 bytes!).
- Secondary data types: arrays. They are laid out contiguously in memory. This also means that you can do pointer arithmetic on them!
    - Important fact: pointer arithmetic has as units, the size of the objects in the array. So, in the case where you increment a pointer by i, you actually increment the value of the pointer by i*(sizeof(object in array)).
    - So: char p[10];
    - Then p+1 actually increments by 1*(sizeof(char)) = 1
    - int p[10];
    - Then p+2 actually increments the value of p by 2*sizeof(int) = 8.
    - Examples you can do here are to ask them what the value of the pointer is after some arithmetic is done on it.
    - More on pointer arithmetic!!! (Examples and such)
- Structs: Structs are also laid out contiguously in memory, WITH A CAVEAT. PADDING. Padding is added to make sure that every element of the struct lies on a 4 byte boundary.
  Draw a picture of the following struct in memory:

```
struct play {
  int a;
  char b;
};
```

  Then you have padding at the end of the struct (TRY IT: SIZEOF(STRUCT). WRITE THE CODE. SHOW THEM.)

  Ask them what the size of the following struct is:

```
struct play{
  int a;
  char b[10];
  int *c;
  long d;
};
```

Question! Can you make the size smaller by rearranging? (**No**.)

Question! What about this struct?

```
struct inner {
  int a;
  int b;
};
struct play{
      char a[2];
      int *b;
      long c;
      struct inner d;
      char e;
};
```

Yes! Place the chars next to each other.

Unions- memory is like a structure except that its elements share the same memory.

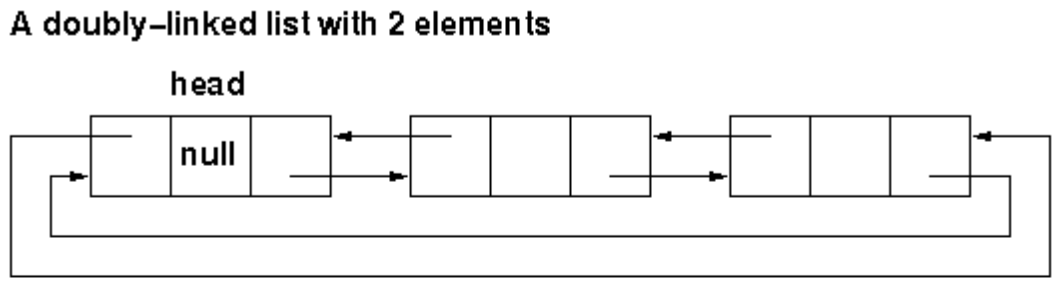
```
typedef union {
      int nums[2];
      char string[30]
} myUnion;

myUnion u[2];
for (int i = 0; i < 2; i++) {
      u[i].nums[0] = 12;
}
u[1].string[2] = 1; // changes memory used by u[1].nums[0]

for (int i = 0; i < 2; i++) {
      printf("%d\n", u[i].nums[0]);
}
```

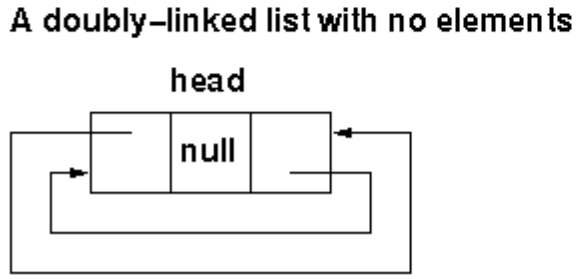
```
Output:
12
65337
memory layout (little endian) of nums[1]: 0C 00 01 00
answer should be 65548? 0x00 01 00 0C
```

Linked Lists (In prep for Malloc, for anyone who would like to stick around and review)

- Doubly linked lists: Traversal, insertion, deletion.

Draw a picture of a node/list, which looks like this:





Go through and try to write pseudocode that finds an element, deletes an element, and inserts an element. Use the struct outline:

```
struct node {
  struct node *prev;
  int value; //(or other things that the list will hold)
  struct node *next;
}
```