Lecture notes: February 13, 2017

Topics:

1. Exam review
2. Hexadecimal arithmetic
3. Structure Layout

Exam topics

The exam will cover all of the material through February 10th's lecture, including the contents of the programming assignments. Topics include:

- structures (syntax, use, layout in memory)
- program memory layout (including the material from PA02)
- pointers (syntax, difference between a pointer and a regular value, etc)
- tools (gdb and make)
- files (how to use the file API, as in PA05)
- memory allocation (malloc and free, memory leaks)
- recursion basics (not how to write recursive functions, but how to reason about execution)

Hexadecimal arithmetic

You will need to be able to do arithmetic (addition and subtraction) on hexadecimal numbers. This may also include being able to convert between decimal and hex.

You may be tempted to convert everything to decimal, do the arithmetic, then convert back to hex, but it's actually easier to just do everything in hex and use the "normal" arithmetic algorithms.

```
   0x4a0
+  0x9c7
```

Rather than converting both numbers to decimal and doing the addition, do the addition in hex. (That way the only conversions you will have to do are of single hexadecimal digits). 0 + 7 is 7. a (10 in decimal) + c (12) is 16 (22), so you carry the 1. 1 + 4 + 9 is e (14). So the answer is:

```
   0x4a0
+  0x9c7
   0xe67
```

Structure layout

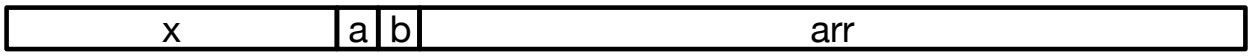
When you define a structure in C, the compiler is free to lay it out in different ways. However, the "standard" way to lay out a structure is what we will consider in class (you can more or less make sure you get this "standard" layout by using the `#pragma pack(1)` directive)

So if we have a structure like:

```
typedef struct {
  double x;
  char a;
  char b;
  int[5] arr;
} MyStruct;
```

We can tell how much space this will take up (i.e., what we will get if we ask for `sizeof(MyStruct)`): 30 bytes (8 bytes for `x`, 1 byte for `a`, 1 byte for `b`, 20 bytes for `arr`).

This also dictates how the structure is laid out in memory, with the fields placed one after another:



Moreover, if we have an array of MyStructs:

```
MyStruct many[10];
```

Those structures will be laid out back to back in memory.