Lecture notes: January 25, 2017

Topics:

1. Pointers (continued)

Using pointers

We saw that we can use pointers to store addresses of locations in memory. How can we *use* them?

The trick to pointers is that the operator * (the *dereference* operator) lets us access the memory location that the pointer points to (i.e., it lets us access the memory location at the address that is stored in the pointer):

```
int x = 7;
int * p = &x; //p now points to x
*p = 10; //this is the same as x = 10
int y = *p; //this is the same as y = x
```

The expression `*p` acts just like `x` wherever we use it. In fact, one way to think about pointers is they let you give alternate names to locations in memory. If a pointer `p` stores an address, `*p` is a name for that address in exactly the same way that a variable is a name for an address!

Pointers to things other than basic data types

Pointers don't have to point to ints or floats or doubles. They can also point to data types you create!

```
typedef struct {
   float x;
   float y;
} Point;

Point p = {.x = 1.5, .y = 2.5};
Point * q = &p; //now you can use * q anywhere you use p
```

Why is this useful?

With regular variables, once you create them, you name a memory location, but you can never change what memory location you're talking about. Pointers give you a way of creating a "dynamic" name — a name that you can use to talk about a memory location that can change depending on what you need to use it for:

```
int x = 7;
int * p = &x; //*p is now another name for x
int y = * p; //like saying y = x
p = &y; //*p is now another name for y
*p = 8; //like saying y = 8
```

One place this is especially useful is in writing functions. C functions are *pass by value*: when you pass an argument to a function, inside the function you are working on a *copy* of that argument. If you try to change the data inside the function, you're changing the copy, not the original data. The following implementation of a swap function doesn't work:

```
int a = 8;
int b = 10;

void swap(int x, int y) {
   int tmp = x;
   x = y;
   y = tmp;
}

void main() {
   swap(a, b); //a is still 8, b is still 10
}
```

Because inside swap, `x` and `y` are names for new memory locations that are holding *copies* of the data in `a` and `b`. When we swap them, `a` and `b` don't change. But we can use pointers to get around this problem. What if `x` and `y` held the *addresses* of `a` and `b`? Then `*x` and `*y` would be names for the same memory locations that `a` and `b` are. Changing them would change the values of `a` and `b`!

```
int a = 8;
int b = 10;

void swap(int * x, int * y) {
   int tmp = *x; //tmp = whatever is in the location x points to
   *x = *y;
   * y = tmp;
}

void main() {
   //remember, we have to pass in addresses now, not ints
   swap(&a, &b); //a is now 10, b is now 8
}
```

Chains of pointers

Pointers can point to any data type — even pointers!

```
int x = 7;
int * p = &x; //p points to x; *p is the same as x

int * * q; //q is a pointer to a pointer to an int
q = &p; //q points to p
```

In this example, `*q` is the same as `p`. `*(*q)` is the same as `*p` which is the same as `x`.