Lecture notes: January 23, 2017

Topics:

1. Structures (cont)
2. Pointers (intro)

Structures (cont)

How to initialize structures?

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

//Simple style:
Point p;
p.x = 1.5;
p.y = 2.5;
```

Bad because it separates the declaration from the initialization

```
//All-at-once style:
Point p = {1.5, 2.5}
```

Bad because the order of fields isn't obvious, and you may mess it up

```
//Best style:
Point p = {.x = 1.5, .y = 2.5}
```

Makes clear which fields of the struct are initialized to what

Nesting structures?

Structures are data types, too, which means you can make a structure a field of another structure!

Pointers

What is a pointer?

We've seen a bunch of different *types* that C has: `int`, `float`, `short`, etc. We've even seen how to create our own types by creating structures. But there is a whole category of types that we have not looked at: *pointers*. A pointer type looks like `<typename> *`, and we read it as "pointer to <typename>". So, for example:

```
int * p;
```

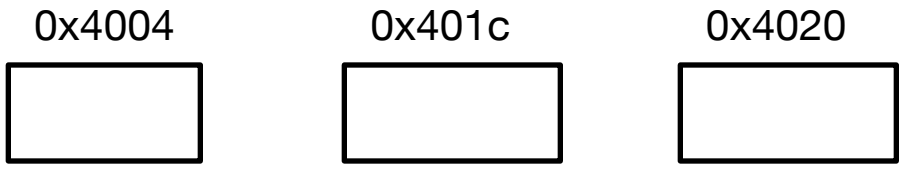
is a variable named `p` whose type is "pointer to `int`" (Important note: the variable we declared here is `p`, *not* `*p`)

So what does it mean to be a *pointer to an int*? To understand this, it's helpful to have a picture of what's going on in memory.

How should we think about memory?

We've already talked a bit about how programs are laid out in memory, with our discussion of the program stack. The important thing to understand about memory is that your program "thinks" in terms of *memory locations*, and every memory location has an *address*.

Think about memory as a bunch of boxes. Each box is a location where you can store some data. Each box has an *address* (sort of like how each house on a street has an address). A particular address refers to exactly one box (memory location). So for example, we could think of three memory locations, each with their own address:



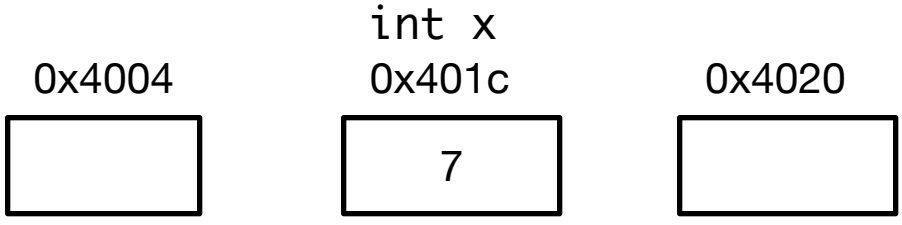
An important thing to note here is that *your computer only thinks about memory in terms of addresses*: it's like a GPS device that can only navigate to places based on their street addresses. The only thing the computer understands is memory addresses. If I want to store data in a memory location, I have to use an address. If I want to retrieve data from a memory location, I have to use its address.

What is a variable?

Humans are not so good at remembering addresses (quick: what's latitude and longitude of your hometown?) So in computer programs, we use *variables* as "handles" to let us talk about memory locations without talking about addresses. When we create a global variable:

```
int x = 7;
```

Your program chooses a particular memory location, and gives it an alternate name of `x`. So, for example, it might decide that memory location `0x401c` will be called `x`. Your program remembers this mapping, so that whenever you talk about `x`, the program generates code that talks about memory location `0x401c`. (Think of this mapping like an address book: it lets you talk about a particular street address not as, say "465 Northwestern Ave." but instead as "the EE building" — it's an alternate name for a particular location).



All variables in your program are just names given to memory locations (the details are a little trickier for variables that are local to a function, but the basic principle is the same). This means that *every variable in your program* also has an address that it is associated with.

What's interesting is that C provides a way to get at that address, using the *address of* operator, `&`. In our example:

```
&x //would return the address 0x401c
```

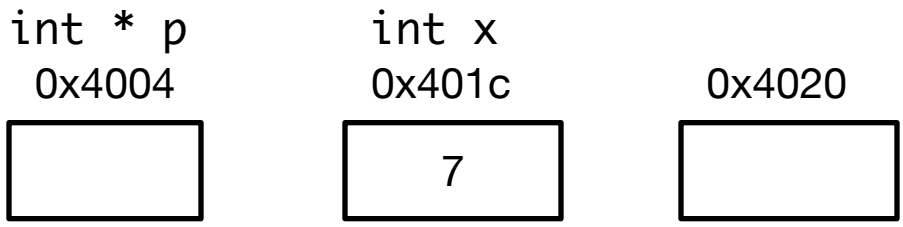
Now we're ready to answer the question: what is a pointer?

What is a pointer (take 2)?

A pointer is a data type that *holds an address.* The data stored for a pointer is always an address, and the type of that pointer (e.g., a pointer to an *int*) tells us *what kind of data is stored at that address*. So suppose we create our pointer again:

```
int * p;
```

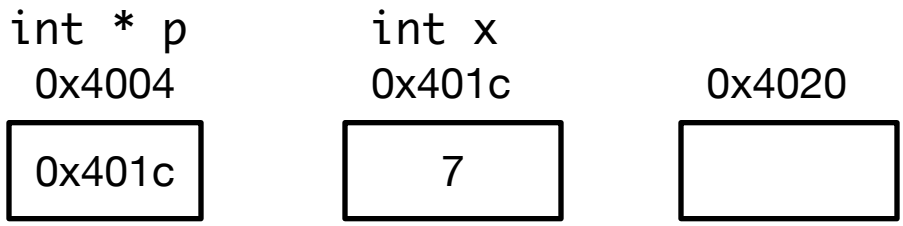
Remember, `p` is a variable, and all variables are just names given to addresses. Let's assume that our program has decided that `p` is the name of address `0x4004`.



`p` has type `int *`, which means that it holds the address of a memory location that stores an integer. Where might we get that sort of address from? Well, we can use the `&` operator!

```
p = &x;
```

Which stores the *address of* `x` in `p`:



Colloquially, we say that this means `p` *points to* `x`. Next, we'll see what we can do with this address stored in `p`.