Lecture notes: Week 3

In Week 2, we covered:

1. Data types
2. Arrays
3. Structs
4. Enums and unions
5. Pointer basics

Data Types

What is a data type?

It is a way of indicating *what a variable is*. A variable is, essentially, a box in memory that holds data of some sort.

When you declare a variable and give it a type:

```
int x;
```

You are creating a box in memory called `x`. But what is in that box? The data type tells you. Saying that `x` is an `int` tells your program several things about the box:

1. *What is the set of values this variable can take on?* An `int` in C can take on integer values from -2^{31} to $(2^{31} - 1)$. In contrast, an `unsigned int` takes on integer values between 0 and $(2^{32} - 1)$
2. *How much space does this variable take up?* An `int` in C typically occupies 32 bits. Indeed, there is a relationship (for integer types) between the answer to this question and the answer to question 1.
3. *How should operations on this variable be handled?* The interpretation of various arithmetic operations can change depending on the type of the variable. Performing division on `int`s is different than performing division on `float`s:
   ```
   3 / 2 = 1 //integer division
   3.0 / 2.0 = 1.5 //floating point division
   ```

Data types also help programmers understand what their code is doing.

C has relatively few built-in types, including `int` (integers), `long` (integers that take up 8 bytes), and `unsigned` versions of those two, `float` (floating point numbers that use 4 bytes), `double` (floating point numbers that use 8 bytes), `char` (characters, or integers that take up 1 byte).

Converting between data types

What happens when you try to use one data type in the place of another? In many cases, C will do an *implicit conversion* between the data types, turning the information into something compatible with the other type. For example, you can convert integers to floats:

```
int x = 4;
float y = x; //y is now 4.0
```

Or vice versa, in which case the conversion *truncates* the floating point value to preserve only the integer part:

```
float y = 2.7;
int x = y; //x is now 2
```

These implicit conversions also happen when you try to do operations that combine different types. C uses a complicated system of *type promotion* to figure out how to convert types to let you operate on them. So if you divide a float by an integer, C will convert the integer to a float and perform the division:

```
int x = 3;
float y = 2.7;
float z = y / x; //x gets treated as 3.0, so z = 0.9
```

And similarly if you divide an integer by a float:

```
int x = 3;
float y = 2.7;
float z = x / y; //x gets treated as 3.0, so z = 1.11111
```

C's implicit promotion rules are quite complicated, and beyond the scope of this course to cover in depth. You can read more about them here: https://www.geeksforgeeks.org/implicit-type-conversion-in-c-with-examples/ and in extreme depth here: https://en.cppreference.com/w/c/language/conversion

If you want to *explicitly* convert between types, you can use *explicit casts*, which force C to do the conversion. For example, the following code will have somewhat interesting behavior:

```
int x = 3;
int y = 2;
float z = x / y; //z = 1.0!
```

What's happening is that your program first computes `x / y`, which is done as integer division (where remainders are dropped) because both `x` and `y` are `int`s, resulting in `1`. Then, the integer `1` is stored in `z`, where it undergoes implicit conversion to a `float`!

If you want to get the result you expect, you need to tell C to *explicitly* convert `x` or `y` (or both) to `float`s first:

```
int x = 3;
int y = 2;
float z = (float) x / (float) y; //z = 1.5
```

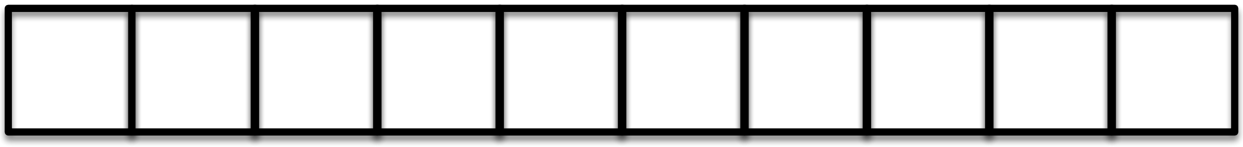
Note that these cast operations (a type name inside parentheses) obey order of operations rules, too, so you need to be careful to apply them in the right order.

```
int x = 3;
int y = 2;
float z = (float) (x / y); //int division happens before the cast, so z = 1.0
```

Arrays

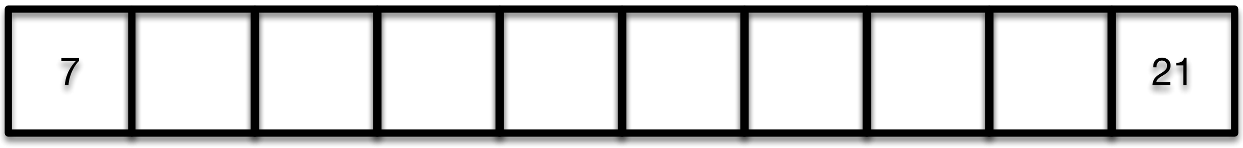
One complex data type that C has is *arrays*: a fixed-size sequence of another data type. When you declare an array, C will allocate (create space) for that many boxes of data, right next to each other:

```
int a[10]; //create 10 int boxes, right next to each other in memory
```



a is essentially a special variable that helps you get to the specific box you want:

```
a[0] = 7; //put 7 in the first box
a[9] = 21; //put 21 in the 10th box
```



If the address of `a[0]` is, say `0x4000` in memory (meaning the box containing the first element of a is at location `0x4000`) then the address of `a[1]` is right next to it (`0x4004` — remember, ints take up 4 bytes!), and a[i], more generally, will be a box in memory location `0x4000 + 4 * i`.

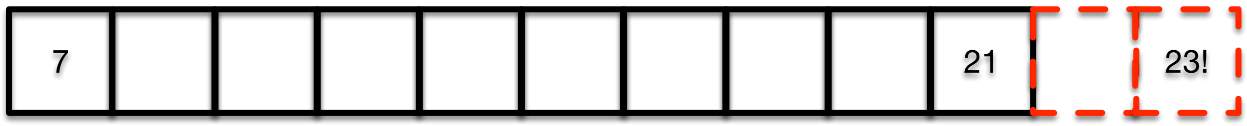
What's happening under the hood is that `a` really just tells you what the address of the first box is, and then C does some math to figure out what box `a[i]` is talking about. We'll return to this topic when we discuss the connection between pointers and arrays.

Out of bounds accesses

But C isn't smart: you can ask for a box that doesn't exist!

```
a[11] = 23; //put 21 in the 12th (?!) box
```

Your program will merrily assume that there *is* a 12th box in memory that it can assign to, and potentially disrupt the rest of the data in your program! It will try to put 23 in `0x402C` (`44` is `2C` in hex), even if that memory location is a different variable!



Many a security vulnerability comes from exactly these sorts of out of bounds accesses.

Initializing arrays

Often, you will initialize arrays by declaring the array (telling your program you want the boxes) and then writing a little loop to set up the values of the array:

```
int a[10];
for (int i = 0; i < 10; i++) {
   a[i] = i;
}
```

But you can also initialize arrays in other ways. You can give the array starting values:

```
int a[5] = {0, 2, 4, 6, 8};
```

You can also *partially* initialize an array, by only initializing some of the values. The rest will default to 0:

```
int a[5] = {1, 2}; //the other three elements will be 0
int b[5] = { }; //all of the elements will be 0;
```

You can even initialize an array by specifying the initial values without specifying the size:

```
int a[] = {2, 4, 6}; //a will be set up as a 3-element array
```

But what if you don't know how big you want the array to be? For example, if you want the user to tell you how many array elements to initialize (like in HW1)?

Unfortunately, "regular" arrays in C have to be a fixed, known size — your compiler needs to know how big the array is going to be before the program runs so it can set up enough boxes in memory to hold the array. If it doesn't know how many boxes to allocate, it cannot place them on the stack or in globals.

To handle dynamically-sized arrays, then, you need to *dynamically allocate* the array. This requires the use of *pointers*, which we will cover later.

Structs

C does not have many built in datatypes: `int`, `char`, `short`, `long`, `float`, `double`, and a few others, as well as arrays of each and pointers to each. But what if you want to talk about more complex pieces of data?

What if we want to represent a point on a graph? We cannot represent that point with just a single value, like a `float`. We need *two* values: an x coordinate and a y coordinate:

```
float point_x;
float point_y;
```

But what *is* a point? It's not a `float`. It's a `float` representing its x coordinate *and* a `float` representing its y coordinate. Can we define data types that let us say that a variable is <thing one> *and* <thing two> *and* <thing three>?

C *structures* let us do this. We can define a *new type* that lets us say that if a variable is a point, it is two `float`s!

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

And now when we declare a new variable, we can say that it *is a Point*:

```
Point p1;
Point p2;
```

To access the components of a structure, we use '.':

```
p1.x = 2.5;
p1.y = 3.7;

p2.x = p1.x - 3;
p2.y = p1.x * 2;
```

Initializing structures

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

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

```
p.y = 2.5;
```

This is bad because it separates the declaration from the initialization

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

This is better, but still not great because the order of fields isn't obvious, and you may mess it up

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

This approach makes clear which fields of the struct are initialized to what. You can even change the order:

```
Point q = {.y = 1.5, .x = 2.5}
```

Nested structures

What about more complicated structures? Well, a struct can have any type as one of its fields. Including another struct! What if we want to create a new type of point that in addition to an x coordinate and a y coordinate also stores a piece of integer data? Well, we could create a brand new struct:

```
typedef struct {
   int value;
   float x;
   float y;
} ValuePoint;
```

But that's a little annoying. It would be better to take advantage of the fact that we already *have* a Point data type:

```
typedef struct {
   int value;
   Point p;
} ValuePoint;
```

And now we can access the x and the y coordinates by referencing into the p field:

```
ValuePoint v = {.i = 2, .p = {.x = .5, .y = 1.5}};
v.p.x = 2.5; //update the x coordinate of the point
```

Enums and Unions

There are two other complex data types that C lets you define: enums and unions.

Enums

Think of enum as a data type that says: “this variable can only take on exactly these values” where each value is given a name:

```
typedef enum {
   FRESHMAN,
   SOPHOMORE,
   JUNIOR,
   SENIOR
} Class;

typedef struct {
   int id;
   Class class;
} StudentRecord;

StudentRecord me = {.id = 12345, .class = SENIOR};
```

This is useful when you have a small set of values that you want to keep straight easily. You *could* define the student’s class as an integer and just use the numbers 0, 1, 2, and 3 to distinguish their year, but it’s hard to read code written like that.

By convention, we write the enum values in all caps. But this isn’t strictly necessary.

It turns out, under the hood, C is actually treating `Class` as an `int`, and remembering that whenever you say `FRESHMAN`, you mean 0; whenever you say `SOPHOMORE`, you mean 1, etc. You can also override the default assignments of numbers to enum values:

```
typedef enum {
   FRESHMAN = 17,
   SOPHOMORE = 2,
   JUNIOR = 43,
   SENIOR = 3
} Class;
```

Unions

A `union` is a weird type. A `struct` says that a piece of data is a <thing 1> *and* a <thing 2> *and* a <thing 3>, etc. A `union`, on the other hand, says that a piece of data is a <thing 1> *or* a <thing 2> …

```
typedef union {
   int x;
   float f;
} IntOrFloat
```

And now if I create a variable of type IntOrFloat, I can treat it as an integer or a float, depending on which field I use:

```
IntOrFloat val;
val.x = 7; //store an integer
val.f = 2.5; //replace it with a float
```

Unions are pretty specialized types, and you won't often find yourself needing them. But some times they're the best way to solve a problem. For example, *tagged unions* are complex data types that let you both store different kinds of data in the same box, and also keep track of which type of data they store. To do this, we use `struct`s, `enum`s, and `union`s (oh my!):

```
typedef enum {
   INTEGER,
   FLOAT
} Type;

typedef union {
   int i;
   float f;
} IntOrFloat;

typedef struct {
   IntOrFloat value;
   Type t;
} TaggedData;

TaggedData d;

/* some code here */

//Print out the value in d depending on its type:
if (d.t == INT) {
   printf("%d\n", d.value.i);
} else {
   printf("%f\n", d.value.f);
}
```

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;
```

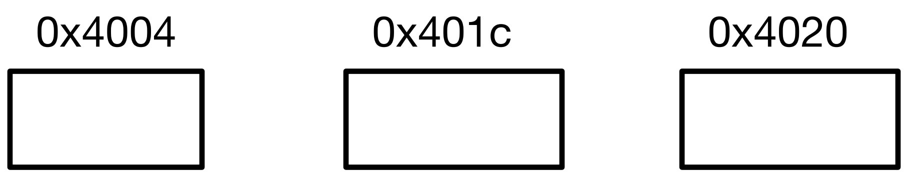
is a variable named `p` whose type is "pointer to `int`" (Important note: the variable we declared here is `p`, *not* `*p`)

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:



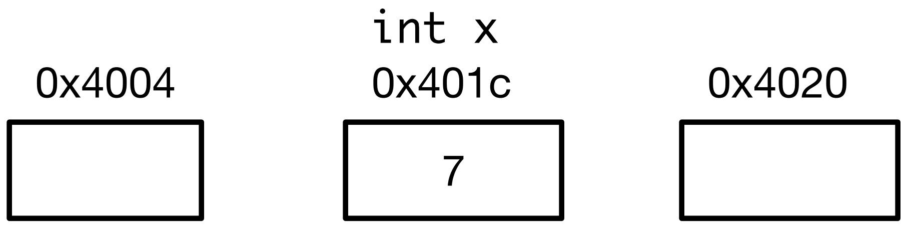
An important thing to note here is that *your computer only thinks about memory in terms of addresses*: it's like a GPS device that can only navigate to places based on their street addresses. The only thing the computer understands is memory addresses. If I want to store data in a memory location, I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

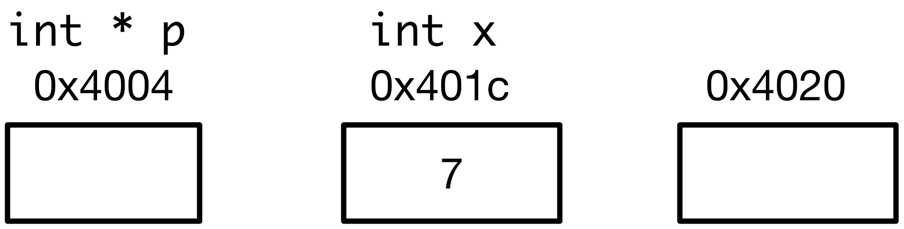
Now we're ready to answer the question: what is a pointer?

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;
```

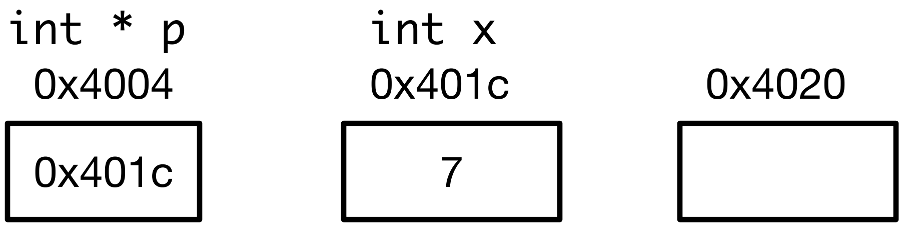
Remember, p is a variable, and all variables are just names given to addresses. Let's assume that our program has decided that `p` is the name of address `0x4004`.



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.

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):

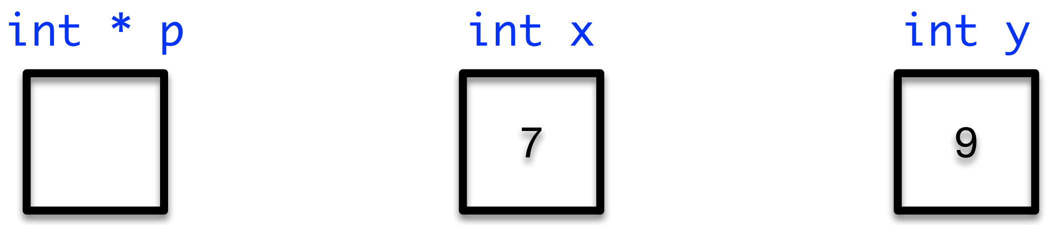
```
int x = 7;
int * p = &x; //p now points to x
*p = 10; //this is the same as x = 10
int y = *p; //this is the same as y = x
```

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!

Visualizing pointers

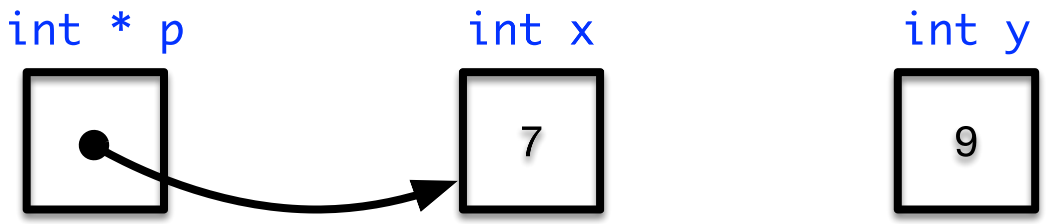
As we think more about how pointers work, it's going to be useful to visualize them in some way other than writing inscrutable hexadecimal addresses all the time. Instead, we'll capture the fact that a pointer points to something else with arrows:

```
int x = 7;
int y = 9;
int * p;
```



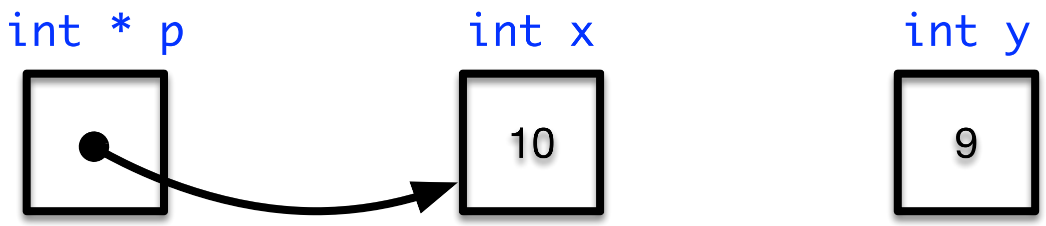
And now when we assign an address to p, we can use an arrow to represent the pointer:

```
p = &x;
```



And dereferencing p basically means "follow the arrow, and act as if you're talking about the box the arrow is pointing to:

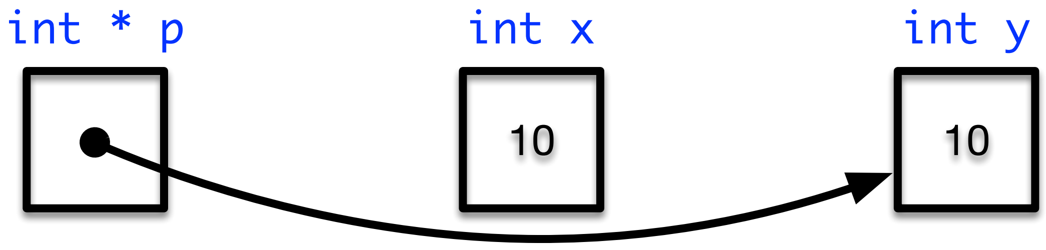
```
* p = 10;
```



And we can even change what the pointer points to:

```
p = &y;
```

```
* p = y + 1;
```



We'll see a lot more examples of manipulating pointers in next week's notes