Lecture notes: January 30, 2017

Topics:

1. Pointer arithmetic
2. Arrays
3. Memory allocation

Pointer arithmetic

What does `*(p + 1)` mean?

If `p` is a `int *`, it means "access one integer past whatever `p` points to." Remember that because `p` is a pointer, it holds the address of a location in memory. Because `p` is an `int` pointer, that location in memory holds an integer. When we add 1 to the address, the compiler interprets this as "the address of the *next* integer." In other words, we will add 4 to the address in `p` (because integers take up 4 bytes). If `p` pointed to doubles, the compiler would interpret this as "the address of the *next* double" and add 8 to the address (because doubles take up 8 bytes).

We can put numbers other than 1 there. For example, `p + 3` will give the address of the integer that is three integers past whatever `p` currently points to. `p - 1` will give the address of the integer right *before* whatever `p` points to.

Arrays

Another data type!

Array data types

Arrays of ints, arrays of structs

Arrays of chars: the C way to represent strings

How do arrays work?

They're weird — they work a little bit like pointers:

```
int a[10] //a is an array of 10 integers
```

Can access the first integer with `a[0]`
Can access the second integer with `a[1]`
etc.

Note that these 10 integers are *guaranteed* to be next to each other in memory

So what's `a` itself? It actually refers to the first location of the `a` array. Can see this by printing `a`, printing the address of `a`, printing the address of `a[0]` — all the same!

`a[0]` really means: `*(a + 0)`
`a[1]` really means: `*(a + 1)`

And so on.

Can use pointers to represent arrays, too:

```
int * p = a
```

(aside: internally, your C compiler thinks that the type of a is `int *`! If you try `int * p = &a`, your code will work, but the compiler will complain)

Now we can use `p` the same way we would use `a`:

`p[0]` really means `*(p + 0)` which is the same as `*(a + 0)` which is `a[0]`
`p[1]` really means `*(p + 1)` which is the same as `*(a + 1)` which is `a[1]`

Dynamic memory allocation

What if we don't know how big we want an array to be? One way to do this is to use *variable length arrays*, but those are not always supported, and using them is potentially very dangerous (we don't allow you to use them, in fact: `-Wvla` gives a warning if you try to use them)

We can instead use the *heap* — that other space in memory where we can store data. The basic way we interact with the heap is to ask the program to either "give us X bytes of data from the heap that we can use to store data" or "take back this data so that it can be reused for something else."

```
void * malloc(size_t size) // give me size bytes of data from the
heap, return the address of the first byte of that chunk

free (void * ptr) //take back the chunk of memory where ptr points to
the beginning of that chunk
```

So, remembering that pointers can be used as arrays, here's how we could allocate an array of N integers when we don't know what N is at compile time:

```
int * arr = malloc(N * sizeof(int));
```

What's happening here?

1. find 40 bytes of memory on the heap.
2. reserve it for the program's use. Means that no other call to `malloc` will return any of that part of memory unless you call free
3. return the address of the beginning of the chunk

Note that the chunk is guaranteed to be 40 consecutive bytes in memory. `arr` points to the beginning of the chunk. So we can treat this chunk of memory just like an array:

`arr[0]`, `arr[5]`, etc.

When we're done with the memory, we can tell the program that we're done with it:

```
free(arr);
```

Now when we call `malloc()` again, we might get back the same memory. [Note: `arr` still points to that location in memory — this is potentially dangerous! Always a good idea to null-out a pointer when you free the data it points to)