Note 4: C Basics P2

Kevin Moy

1 Review

Last note, we learned the most basic of basics of the C language. Here are some of the most important points to remember:

- Declare variables at the beginning of the function.
- Memory is a massive array with an address mapped to each cell.
- ALL data is stored in memory.
- Pointers contain addresses, presumably of other variables or objects. We can dereference this pointer to access and edit the value at stored at that address.
- Only 0 or NULL evaluates to false - everything else is true.

2 Dangers of Pointers

Spiderman's uncle said "with great power comes great responsibility". Nothing could be more true with pointers- with the power to access and edit pretty much anything in memory, we could be getting ourselves into some real trouble if we fuck around with them too much.

When we *declare* a pointer, e.g. `int *ptr`, all we're doing is **allocating space for the pointer**- NOT for the thing it's going to be pointing to (eventually). Remember that in C (and unlike Java), uninitialized variables contain garbage. So really the pointer might contain some random memory address when you declare it!

Consider this simple code snippet:

```c
void f() {
    int *ptr;
    *ptr = 5;
}
```

In `f`, we are *declaring* an integer pointer `ptr` and then setting the value AT its address (nondeterministic) to 5. Recall that in order to dereference `ptr` (access the value it's pointing to), we use the same notation `*ptr`.

3 Arrays

Arrays are just really just chunks of memory, with the additional enforcement that each cell must have the same type of value.

In C, arrays must be declared with the exact dimensions OR exact elements specified. Below are two sample array declarations:

```
int arr1[2];
int arr2[] = {1,2,3};
```

Note that `arr1`'s array contains garbage while `arr2`'s array contains initialized values.

Accessing elements by index is the same as in pretty much every other language: `arr[i]` returns the i-th element of `arr`, zero-indexed.

Really, **arrays are like a pointer to its first cell.** For example, if we have an array `arr`, accessing the element i through `arr[i]` is the same as `*(ar + i*sizeof(typeof(arr)))`. This latter method is known as **pointer arithmetic**, which is really just adding offsets to an address to jump it a certain number of cells forward.

NOTE: String `strng` can be represented by `char * strng` OR `char strng[]`. However, these two declarations differ slightly in incrementation and being able to initialize and declare in one line.

The allocated space for a pointer ONLY LASTS in local scope. So the below function `foo` is incorrect, because the space for the pointer would be de-allocated upon return:

```
char *foo() {
    char string[2];
    string[0] = 'x';
    string[1] = 't';
    return string;
}
```

Arrays in C don't actually know their length! This means that unlike Java, there's no `arr.length`, and no bounds checking. This means we can (accidentally or not) access the end of allocated space for an array. This is actually the heart of an infamous attack called the *buffer overflow* (take CS161 to learn more!).

So to prevent this, it is very good practice for functions with array parameters to **include the array's size as another parameter**. This is especially important for arrays without any terminator characters (int arrays). So a function `array_func(int[] arr)` should become `array_func(int[] arr, int len)`.

There are also array-related errors such as **segmentation faults** and **bus errors**. Segmentation faults, aka segfaults, crash a program because of an attempt to read or write an illegal memory location. Probably the most common segfault example occurs when trying to access an array out of bounds:

```
int arr[100];
arr[100] = 100; //Segfaults
```

Since `arr[100]` is an unallocated memory cell, it is illegal and the program throws a segfault and ends immediately.

4 Pointers in functions

Let's say we want to increment a varaible with function `inc(int x)` in the naive way: just incrementing it and returning.

```
void inc(int x) {
    x = x + 1;
}
int x = 99;
inc(x);
printf("x = %d\n", x);
```

Unfortunately, you'll find that running this will print `99`, which means that unfortunately `inc` did not do its intended job. This is because C is pass-by-value: the parameter x of `inc` is actually a *copy* of the argument- so the original argument is untouched throughout. How can we fix `inc` such that we actually increment the original variable `x` and print `100`?

We can use pointers! Pointers allow editing of data in memory directly. So if we just pass in a *pointer* to the variable we want to increment, we'll find we increment it permanently:

```
void inc(int *ptr) {
    *ptr = *ptr + 1;
}
int x = 99;
inc(&x); // pass in a pointer to x, i.e. x's memory address.
printf("x = %d\n", x);
```

Now this program prints 100. Notice that `inc` is still pass-by-value, but passing a copy of the *address* of x is still the same address- so we can still dereference and edit `x`'s value directly.

4.1 Changing Pointer Values

What if what we want to change the pointer value, i.e. address, itself? For example, what if we want to change the pointer to point to the next cell in the array:

```
void inc_ptr(int *ptr) {
    ptr = ptr + 1;
}
int arr[3] = {1,2,3}
int *q = arr;
inc_ptr(q);
printf("q = %d\n", *q);
```

This will unfortunately print `1` instead of `2` as desired. This is where pass-by-value now matters once again for pointers, since we're trying to edit the pointer (memory address) itself, and we've passed in a copy of it. Now what?

It's actually the exact same logic as standard variables: **to edit a pointer itself, we pass in a pointer to a pointer**. So now we'll have:

```
void inc_ptr(int **ptr) {
    *ptr = *ptr + 1; //increments the value of
                     //ptr (which is another pointer) by 1.
}
int arr[3] = {1,2,3}
int *q = arr;
inc_ptr(&q);
printf("q = %d\n", *q);
```

Now by passing in the address of the *pointer* in memory, we can edit it via `inc_ptr` in the same way.

Note that single and double pointers are really the only ones important for this class. However, for ”higher-order” pointers, careful tracing of what each level of pointer will allow you to understand and utilize them just fine.

5 Dynamic Memory Allocation

Dynamic memory allocation is the allocation of a chunk of memory for something AT RUNTIME. Note that `int arr[4];` is *static* memory allocation, i.e. memory allocated for the array at compile time.

C doesn't have a `.length` field for arrays or a `len()` function. However, there is a helpful `sizeof()` function that gives the size of a type OR variable in bytes. However, because sizes of objects and thus variables can be misleading, it is good practice to consistently use `sizeof(type)`.

An astute observer might notice that `sizeof` can help us immediately determine the size of some array `arr`, via `sizeof(arr)/sizeof(type)`.

5.1 Malloc

Now comes one of the fundamental functions for dynamic memory allocation: `malloc()`. This function allocates a specified number of bytes and returns a pointer to that allocated memory chunk. A standard usage for malloc'ing an integer pointer would look like this:

```
int *ptr;
ptr = (int *) malloc(sizeof(int));
```

`ptr` will point to a size-`sizeof(int)`, or size 4, memory chunk at some (again nondeterministic) starting location in memory. Note that `malloc`'s true return type is actually a void pointer- we **typecast** this pointer to an int pointer via `(int *)` in order to tell the compiler that we'll be placing an integer in the allocated memory space.

However, we never really use `malloc` for a single variable. **malloc is far more commonly used to dynamically allocate arrays.** For example, to allocate an array of 10 integers, we'd have:

```
int *ptr;
ptr = (int *) malloc(10*sizeof(int));
```

So now we allocate 10*4 = 40 bytes for 10 integers, i.e. allocating exactly enough space for our desired array.

5.2 Cleaning Up Malloc

Just like variables, uninitialized dynamically-allocated memory will contain garbage.

After we finish using this allocated memory, we must also manually **free** it. We do this with the `free(ptr)` function: this function frees the allocated space the pointer `ptr` is pointing to. Even though C is configured to automatically free all allocated memory when `main` returns, it's still good practice to know and free exactly what you've allocated! For example, if `main` somehow becomes a subroutine to a "higher-order" `main`, then you'll find yourself taking up memory pointlessly (literally) without ever freeing it.

However, be careful using `free`. Two main rules:

1. Do not `free` the same pointer (same memory) more than once.
2. Do not `free` a pointer that wasn't returned by `malloc`.

Not following these rules won't lead to exceptions/program termination during runtime, and can thus be very difficult to find. Why? Remember that the key part of C was optimizing performance, and a *huge* part of that is based on trust that memory allocation and deallocation is done correctly. So if C checks, one by one, that each `free` is done on the "right" memory chunks, the program slows considerably. So what happens instead is the memory allocator gets corrupted, and you won't find out until much later. When I took this course and did the RISC-V neural network project, I had this exact issue and spent days trying to debug this single problem. Follow the two freeing rules above religiously.

6 Pros and Cons of Pointers: Updated

So now that we have a deeper understanding of pointers, let's review a more comprehensive list of their advantages and disadvantages.

6.1 Pros

The main reason we use pointers is for memory efficiency- all pointers take up 4 bytes, which is usually much smaller than the arrays or structs it points to. So instead of stuffing all components of a struct or array into a function, we just have a single pointer to that object in memory- which means much cleaner and more concise code.

6.2 Cons

On the other hand, pointers are usually the source of countless bugs in code. Again, with the power to access and edit anything in memory comes with a lot of danger. In particular, be wary of **dangling references**, where `ptr` is used (dereferenced, incremented, etc.) before a corresponding `malloc` is called.

Another thing to look out for is **memory leaks**, which is the result of forgetting to `free` dynamically allocated memory. Such memory cannot be reused by the operating system because we **lose the pointer** to that chunk of memory. For example, consider the code below:

```c
char *ptr1 = malloc(100);
char *ptr2 = malloc(100);
ptr1 = ptr2;
```

We're never going to use the 100 bytes of memory originally pointed to by `ptr1` again, resulting in a memory leak of 100 bytes. To make matters worse, this 100-

byte chunk can now *never* be freed for reuse, because there isn't any reference (pointer) to that chunk anymore!

7 Conclusion

This note wraps up the basics of C needed for this class. Some key notes to remember:

- Pointers and arrays are basically the same thing. Pointers, however, have the additional feature of **pointer arithmetic**, where we can add an offset(s) to the memory address held directly.
- C is meant for efficiency; it's quite an "unprotected" language, which means it won't actually let you know when you execute bad practices such as memory leaks or going out of bounds for arrays. So this means there's a lot of overhead: we have to pay attention and know a lot more about our hardware constraints and requirements when programming in C.
- `malloc` dynamically allocates **heap memory**, and `free` deallocates this memory. Both must be done manually.