Programming in C

We’re beginning our switch from C₀ to C. It shouldn’t be too huge of a change - the syntax is very similar. Unfortunately, although C is quite powerful, it can be a very unsafe language to program in if you don’t know what you’re doing. The good news is that if you think through your program invariants the way we’ve been emphasizing throughout this whole class, you should have a rather straightforward time with it.

Also, if you’re ever planning on using C in the future, I highly recommend [*The C Programming Language*, (often referred to as K&R)](https://en.wikipedia.org/wiki/The_C_Programming_Language) by Brian Kernighan and Dennis Ritchie.

Function pointers

One type that C has that we haven’t seen in C₀ is the function pointer. When we talked about memory layout a while ago, we mentioned that under the heap is a section called code. That’s where all of our functions and their associated code are addressed. Function pointers allow us to pass around references to functions as if they were variables.

Suppose we have an array of numbers that we want to do various transformations on. Maybe we want to get the factorial of all of them. Maybe we want to square all of them. Maybe we want to set them all to zero. In C₀, we’d have to write several functions explicitly dictating that behavior, with a lot of the same code being used over and over. However, in C, we can do the following to make it much neater:

1. Write some functions that will perform transformations on one number

   ```c
   int factorial(int n) {
     if (n == 0) return 1;
     return n * factorial(n-1);
   }

   int square(int x) {
     return x * x;
   }
   ```

2. Write a function that will apply a generic transformation to each number in an array of numbers. Here’s our function prototype:

   ```c
   void map(int (*f)(int), int[] A, int n);
   ```

   Here’s where stuff gets messy. What kind of bizarre syntax is that? If we start at the *f and spiral out counterclockwise, we can interpret it as a **pointer** to a **function** (f) that takes an **int** as an argument and returns an **int**.

   We can typedef it to make things look less confusing: *typedef int (*int_int_trans)(int)*

   Now if you write `int_int_trans f`, it’s the same as writing `int (*f)(int)`

   So now we can write our function:

   ```c
   void map(int_int_trans f, int A[], int n) {
     for (int i = 0; i < n; i++)
       A[i] = f(A[i]);
   }
   ```
Strings... or are they?

Another difference between C\sb{0} and C is that strings are actually char arrays with the null terminator ‘\0’. So if you have a string that is \( n \) characters long, your array is going to need to have size \( n + 1 \). The null terminator is super important, because it tells C when we’ve reached the end of a string. Data is stored all clumped together in memory, so if there were no null terminator to tell C where a string stops, it would associate other data with the string that might not be appropriate to reference and manipulate.

Also, chars represented in memory as bytes, or 8-bit numbers. So while we see the ASCII value they encode, they’re actually 2’s complement numbers and can be compared as such.

Let’s write a function that concatenates two strings, much like string_join in C\sb{0}.

```c
//copies the characters in source to the end of dest
char* strcat(char* dest, char* source) {
    int i = 0;
    int j = 0;
    while(dest[i] != '\0')
        i++;
    for(j = 0; source[j] != '\0'; j++)
        dest[i+j] = source[j];
    return dest;
}
```

We’ll now test this function on ‘‘hello’’ and ‘‘world’’. What do we think is going to happen when we do each of the following?

- Allocating enough space for the entire string
- Allocating only enough for ‘‘hello’’
- Allocating no space

What actually happens?

- Allocating enough space for the entire string
- Allocating only enough for ‘‘hello’’
- Allocating no space