"Cheat Sheet" - Week 4

CS50 — Fall 2012
Prepared by: Doug Lloyd '09

October 8, 2012

Pointers

Pointers have an awful reputation among beginning programming students because they give you the ability to make some pretty dramatic mistakes. As long as you are careful with them, though, you'll be fine. Just remember the following things!

1. **Pointers are Addresses** A pointer is merely an address in memory. When you dereference a pointer with the dereferencing operator (`*`) you are simply examining (and perhaps manipulating) the piece of data that is at that location. For example, in the following code:

   ```
   int *p;
   p = 0x6562AD3E;
   printf("%d\n", *p);
   ```

   All that is happening is that we are looking at the data contained in memory location 0x6562AD3E, interpreting it as an integer, and printing it to the screen. Likewise we could do the following:

   ```
   int *q;
   q = 0x44A3F320;
   *q = 17;
   ```

   And that would change whatever is in memory location 0x44A3F320 to be "17". Of course, most people don't know off the top of their heads where everything in memory is, so using memory addresses like the ones I've just used is somewhat impractical. That's why C provides the addressing operator (`&`). Here's an example of it.

   ```
   int i = 18;
   int *m;
   m = &i;
   (*m)++;
   ```

   After the execution of this code, `i` would have the value 19, instead.

2. **Arrays are secretly pointers.** It's true. After declaring:

   ```
   double averages[40];
   ```

   Any reference to the array name `averages` is really a pointer to its first element, `averages[0]`. Likewise, `averages`' other elements can be accessed by saying `*(averages + n)`, which is equivalent to `averages[n]`.

3. **Pointers give us the power to allocate memory dynamically.** Sometimes, you may not know how much memory you'll actually need when you are programming, and it will be dependent on other factors. Heretofore, we've been using only static memory–memory whose size is known by the program at compile time. C gives us the power to allocate memory on the fly (dynamically) as we need it, but we need pointers to do it. To do so, we need to use `malloc()` and `sizeof()`, two tools provided by C to make this happen. Assume that we wait for an input from the user to figure out how many elements an array needs. True, in C99 and making use of the CS50 library you can simply do:

   ```
   int arrsize = GetInt();
   int arr[arrsize];
   ```

But it wasn't always possible to do that. You used to have to do this (for simplicity's sake we'll use the CS50 library again):

```
int arrsize = GetInt();
int *arr = (int *) malloc(sizeof(int) * arrsize);
```

(And you'll still need to do this in other situations, we promise!). But notice what `malloc()` does. I am asking `malloc()` to give me `arrsize` blocks of contiguous memory, with each block the size of an `int` (4 bytes apiece). Take care though, to always make sure that you actually get that memory back! If `malloc()` is unable to allocate memory for you, because of an error or because no memory is left, it will return a `NULL` pointer. Dereferencing a `NULL` pointer will crash your program. So always do a `NULL` check:

```
int *arr = (int *) malloc(sizeof(int) * arrsize);
if(arr == NULL) {
   printf("Error -- no memory left!\n");
   return 1;
}
```

If I passed the `NULL` check, I could then use this memory in the same way that I did the first example in this section, with one really important exception...

4. **All `malloc()`'d memory must subsequently be `free()`'d.** Failure to do so results in what's called a *memory leak*, and it sounds just as bad as it is! When we have finished using a dynamically-allocated piece of memory, we need to give it back to C, in case it needs to use that memory again for something else. If we "leak" memory by failing to do so, we run the risk of running out of memory and causing our program to crash. It's super easy to `free()`:

```
int x = GetInt();
char *word = (char *) malloc(sizeof(char) * x);

// do stuff

free(word);
```

Just be careful not to double `free()`, and also be careful only to `free()` memory that was previously allocated with `malloc()`, otherwise your risk of a memory leak is still present!

Structures

A structure is basically a super-variable. We use structures to logically group together a collection of elements (which we will call *fields*) that together comprise a larger element. For example, here's the potential definition of a structure (`struct`) for a student:

```
struct student_t {
   string name;
   int year;
   float gpa;
};
```

Once we've defined the structure, we can then use it like any other type. We access fields of the structure with the dot operator (`.`). We can also have pointers to the structure like we can have pointers to any other type. If we have a pointer to a structure, we access its fields with the arrow operator (`->`).

```
struct student_t john;
(struct student_t *) mary = (struct student_t *) malloc(sizeof(struct student_t));
john.name = "John";
mary->name = "Mary";
john.year = 2013;
mary->year = 2014;
john.gpa = 3.48;
mary->gpa = 3.55;
```