Memory Management

One of the most intimidating aspects of C is the fact that you have full control over memory. This gives you lots of room for mistakes in your code, but it can also offer a clearer picture of what your program is doing, as well as, more granular control.

Memory is divided into sections. A pointer holds the label (address) of one of these sections. Memory is further separated into larger segments by the operating system. Different programs have access to different segments. If a program tries to access a location outside of its segment the operating system will send it a segmentation fault signal and it will terminate the program.

Consider the following code where we try to dereference a null pointer.

```
int* i = NULL;

printf("What is NULL? %d\n", i);
printf("What is the value at NULL? %d\n", *i);
```

Output

```
What is NULL? 0
zsh: segmentation fault (core dumped)  ./memManagement
```

NULL is really just a macro for the number 0 so `i = NULL` is equivalent to `i = 0`. So what we've have just done is try to look at the data located at memory address 0. 0 is in fact the address of the very first byte of your memory. Locations that low are usually the property of the operating system itself. So essentially we tried to look in the operating systems sock drawer and it told us to go away.

Malloc

We need our pointers to hold addresses of memory that we actually have access to instead of assigning them random address. To do this well need to request an address from our operating system. This is what the C function `malloc` does. Consider its definition below and recall that `size_t` is really just an `unsigned long long`. Just another integer type.

```
void *malloc(size_t size);
```

This is a very simple function. Given an integer argument *n* it will return a pointer whose value is the first address in a sequence of *n* bytes that we have access to (no seg faults).

Recall that every type has a size. Then obviously if we don't request enough memory to store a given type thats kind of useless right?

```
int* i = malloc(1);
```

This is perfectly legal code, but it would be unwise to actually use `i` in this case. Imagine if you do not own the 3 bytes in front of `i` and something else requests that memory. You could easily step on your own variable. This is just one example of why behavior here is "undefined". Just make sure you always allocate enough space for the type your working with.

The proper way to do this is with the *sizeof()* function that we have seen before.

```
int* i = malloc(sizeof(int));
//or
int* j = malloc(sizeof(j)); //this makes it more type agnostic and some people prefer this
```

So `i` and `j` both have the required 4 bytes to play with, and we can do whatever we would like to do with 2 numbers.

Creating Arrays

We've already seen that arrays are just pointers. Now that we know how to request memory from the operating system you might be able to see how we can create our own arrays.

```
int* A = malloc(sizeof(int)*5); //array or 5 ints

//lets populate our array
for(int i=0; i < 5; i++)
//we can use index notation for arrays and pointers
    A[i]=i*2;


for(int i=0; i < 5; i++)
//Or we can use addition+dereference
    printf("%d ",*(A+i));

printf("\n");
```

Output

```
0 2 4 6 8
```

Free

Any memory you request with malloc remains in use until your program terminates. Unless you "free" that memory. Then it can be reused.

```
void free(void *ptr);
```

`free` says it takes a void pointer, but in C casts to void are implicit, just how casting between number types when doing math is implicit in any language.

Consider this python code

```
a = 0.5
b = 1

a + b = 1.5 # so b would be cast to a float before the sum is computed
```

If you fail to free something before your program terminates then it is called a memory leak. To see why this is bad just consider a program that regularly allocates memory to complete a process, but never lets go of that memory after it is done with it. The program will continuously accumulate memory until it either consumes all memory on the system (resulting in a crash) or is terminated by the operating system.