Dynamically Allocated Memory

In the heap, we can control the amount of memory allocated to a certain variable by using some built in C functions which we will discuss below.

We'll cover the following

Languages like MATLAB, Python, etc, allow you to work with data structures like arrays, lists, etc, that you can dynamically resize. That is to say, you can make them longer, shorter, etc, even after they are created. In C this is not so easy.

Once you have allocated a variable such as an array on the stack, it is fixed in its size. You cannot make it longer or shorter. In contrast, if you use `malloc()` or `calloc()` to allocate an array on the heap, you can use `realloc()` to resize it at some later time. In order to use these functions you will need to `#include <stdlib.h>` at the top of your C file.

The built-in functions `malloc()`, `calloc()`, `realloc()` `memcpy()` and `free()` are what you will use to manage dynamically allocated data structures on the heap, “by hand”. The life cycle of a heap variable involves three stages:

1. allocating the heap variable using `malloc()` or `calloc()`
2. (optionally) resizing the heap variable using `realloc()`
3. releasing the memory from the heap using `free()`

Allocating memory using `malloc()` or `calloc()`

These functions are used to allocate memory at runtime. The `malloc()` function takes as input the size of the memory block to be allocated. The `calloc()` function is like `malloc()` except that it also initializes all elements to zero. The `calloc()` function takes two input arguments, the number of elements and the size of each element.

Here’s an example of using `malloc()` to allocate memory to hold an array of 10 structs:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
  int year;
  int month;
  int day;
} date;

int main(void) {

  date *mylist = malloc(sizeof(date) * 10);

  mylist[0].year = 2012;
  mylist[0].month = 1;
  mylist[0].day = 15;

  int i;
  for (i=1; i<10; i++) {
    mylist[i].year = 2012-i;
    mylist[i].month = 1 + i;
    mylist[i].day = 15 + i;
  }

  for (i=0; i<10; i++) {
    printf("mylist[%d] = %d/%d/%d\n", i, mylist[i].day, mylist[i].month, mylist[i].year);
  }

  free(mylist);
  return 0;
}
```

Resizing a variable using `realloc()`

Let’s say you use `calloc()` to allocate an array of 3 floating-point values, and you later in the program want to increase the size of the array to hold 5 values. Here’s how you could do it using `realloc()`:

```
#include <stdio.h>
#include <stdlib.h>

void showVec(double vec[], int n) {
  int i;
  for (i=0; i<n; i++) {
    printf("vec[%d]=%.3f\n", i, vec[i]);
  }
  printf("\n");
}

int main(void) {

  double *vec = calloc(3, sizeof(double));

  vec[1] = 3.14;
  showVec(vec, 3);

  vec = realloc(vec, sizeof(double)*5);
  showVec(vec, 5);

  vec[3] = 7.77;
  showVec(vec, 5);

  free(vec);
  return 0;
}
```

Freeing a memory block using `free()`

You should always use `free()` to deallocate memory that has been allocated with `malloc()` or `calloc()`, as soon as you don't need it any more. Any memory allocated with `malloc()` or `calloc()` is **reserved**, in other words, it can't be used (for good reason) until it is deallocated with `free()`. If you fail to deallocate memory then you will have what's called a **memory leak**. If your program uses a lot of heap memory, that is not deallocated, and runs for a long time, then you might find that your computer (and your program) slows down, or suddenly freezes, or crashes, because there is no more memory to be allocated.

The rule is, anytime you use `malloc()` or `calloc()`, you **must** also use `free()`.

We're at the end of our discussion on pointers and dynamically allocated memory. Visit the links for further readings and challenge yourself with the exercises ahead.

After that, we'll dive into a very useful data type in C: Strings.