CHAPTER 11

Dynamic Memory allocation

Embedded and other low-memory footprint applications need to be easy on the amount of memory they use when executing. In such scenarios, statically sized data types and data structures just are not going to solve your problem. The best way to achieve this is by allocation of memory for variables at runtime under your watchful eye. This way your program is not using more memory than it has to at any given time. However, it is important to note that the amount of memory that can be allocated by a call to any of these functions is different on every operating system .

Dynamic Memory Allocation

malloc, calloc, or realloc are the three functions used to manipulate memory. These commonly used functions are available through the stdlib library so you must include this library in order to use them.

```
#include stdlib.h
```

After including the stdlib library you can use the malloc, calloc, or realloc functions to manipulate chunks of memory for your variables.

Dynamic Memory Allocation Process

When a program executes, the operating system gives it a stack and a heap to work with. The stack is where global variables, static variables, and functions and their locally defined variables reside. The heap is a free section for the program to use for allocating memory at runtime.

Allocating a Block of Memory

Use the malloc function to allocate a block of memory for a variable. If there is not enough memory available, malloc will return NULL.

The prototype for malloc is:

Sample Code

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

Do not worry about the size of your variable, there is a nice and convenient function that will find it for you, called sizeof. Most calls to malloc will look like the following example:

Sample Code

```
1. ptr = (struct mystruct*)malloc(sizeof(struct mystruct));
```

This way you can get memory for your structure variable without having to know exacly how much to allocate for all its members as well.

Allocating Multiple Blocks of Memory

You can also ask for multiple blocks of memory with the calloc function:

Sample Code

```
1. void *calloc(size_t num, size_t size);
```

If you want to allocate a block for a 10 char array, you can do this:

Sample Code

```
char *ptr;
ptr = (char *)calloc(10, sizeof(char));
```

The above code will give you a chunk of memory the size of 10 chars, and the ptr variable would be pointing to the beginning of the memory chunk. If the call fails, ptr would be NULL.

Releasing the Used Space

All calls to the memory allocating functions discussed here, need to have the memory explicitly freed when no longer in use to prevent memory leaks. Just remember that for every call to an *alloc function you must have a corresponding call to free.

The function call to explicitly free the memory is very simple and is written as shown here below:

Sample Code

```
free(ptr);
```

Just pass this function the pointer to the variable you want to free and you are done.

To Alter the Size of Allocated Memory

Lets get to that third memory allocation function, realloc.

Sample Code

```
void *realloc(void *ptr, size_t size);
```

Pass this function the pointer to the memory you want to resize and the new size you want to resize the allocated memory for the variable you want to resize.

Here is a simple and trivial example to give you a quick idea of how you might see calloc and realloc in action. You will have many chances for malloc viewing as it is the most popular of the three by far.

allocation.c:

Sample Code

```
#include <stdio.h>
#include <stdlib.h>
void main() {
char *ptr, *retval;
ptr = (char *)calloc(10, sizeof(char));
if (ptr == NULL)
printf("calloc failed\n");
else
printf("calloc successful\n");
retval = realloc(ptr, 5);
```

```
11.if (retval == NULL)
12.printf("realloc failed\n");
13.else
14.printf("realloc successful\n");
15.free(ptr);
16.free(retval);
17.}
```

First we declared two pointers and allocated a block of memory the size of 10 chars for ptr using the calloc function. The second pointer retval is used for getting the return value from the call to realloc. Then we reallocate the size of ptr to 5 chars instead of 10. After we check whether all went well with that call, we free up both pointers.
You can play around with the values of size passed to either of the memory allocation functions to see how big a chunk you can ask for before it fails on you. Do not worry, your operating system has the ability to keep your program in check, you will not hurt it this way.
Here is the output:

Sample Code

```
1. ~/Projects/C_tutorials/dynamic memory allocation/samples $ ./all
2. ocation
3. calloc successful
4. realloc successful
5. ~/Projects/C_tutorials/dynamic memory allocation/samples $
```