1 Dynamic Memory Allocation in C

Once again, we speedily move on to another very important topic in C: Dynamic memory allocation. This is one of the most important things in real life programming and it is also the root of many many bugs in programs.

What is dynamic memory allocation anyway? So far, all you have been able to do in order to create new variables was to declare them as "automatic" variables, which are created in a "scope" (i.e., pair of braces) and are destroyed as soon as execution leaves that scope. One exception was global variables which are declared outside all braces in the file body, and those existed until the program exits.

But, you can surely imagine that you might need to create storage during runtime. For example, you might need to create an array with a size that is not pre-determined.

In these cases, you can ask the system to allocate you a certain amount of memory. The system will in return give you a pointer to the start of that memory which you can use for whatever purpose you wish to. But, you should never lose that pointer, since you are supposed to free that memory when you are done with it, and free it only once. And, once the memory is freed, you should never ever try to play with it again. That might sound like too many rules, but that is exactly how things work in C.

There are four function in C which are used to dynamically allocate and free memory. There are `malloc`, `calloc`, `realloc`, and `free`. We shall see what each of them is. Note that these all reside in the header file `stdlib.h`.

2 `malloc()`

The `malloc()` function is your basic memory allocation routine. The prototype is as follows:

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

This function simply allocates `size` bytes of memory, and returns a `void *` to that memory. If the allocation fails (the system does not have enough memory to fulfill the request), it will return a `NULL` pointer. The memory area will be totally uninitialized. This is useful mainly when you want to create storage for a single piece of data (rather than an array) which would normally be a `struct`.

3 `calloc()`

This function is the one to use when you wish to create an array of something (`int`s, `struct`s, whatever). The prototype is as follows:

```
void *calloc(size_t nmemb, size_t size);
```

This call will create a memory of area suitable for an array of `nmemb` members, each of which needs `size` bytes of storage. Again, it returns a `void *` to the start of the allocated memory. Unlike `malloc()`, this function fills the allocated memory with zeroes. A `NULL` is returned if memory can not be allocated.

4 realloc()

This function is used to change the size of a memory block previously allocated by one of the two previous functions. The prototype is as below:

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

The `ptr` argument must be a pointer to previously allocated memory. The `size` argument specifies the new size of the memory. The function returns a pointer to the new block of memory of given size. It may or may not be the same as `ptr`, therefore you should not try to use `ptr` after a call to this function. The content of memory is unchanged up to the minimum of the old and new sizes. The rest of memory is uninitialized.

If you call this function with a `NULL ptr`, it becomes equivalent to `malloc()`. If `size` is zero, it is equivalent to `free()` (see below).

5 free()

This is the function that is used to return memory allocated by any of the functions above to the system. The prototype is below:

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

The function obviously returns no value at all. Note that if you try to free a piece of memory more than once, that means you are in deep trouble. All memory is automatically freed when the program exits.

6 Example

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

int main(void)
{
  int *array;
  int size = 1;
  int i;

  printf("Enter the number of values:");
  scanf("%d", &size);

  array = (int *)calloc(size, sizeof(int));

  for (i=0; i<size; i++) {
    printf("Please enter value #%d: ", i+1);
    scanf("%d", array+i);
  }

  for (i=0; i<size; i++) {
```

```
    printf("Value #%d is: %d\n", i+1, array[i]);
  }

  free(array);

  return 0;
}
```

7 Exercise (from last time)

Write a program that defines a struct for 3-dimensional real vectors and defines functions for the following operations: Addition, subtraction, multiplication with a scalar, dot product, cross product.