Dynamic Memory Allocation in C

In this tutorial, you will learn to manage memory effectively. You will cover types of memory allocation in C and what are the significance of Dynamic memory allocation in C and the different ways to allocate memory dynamically such as using `malloc`, `calloc` `realloc` and `free`.

Memory management is an important aspect of C programming. When writing a program we must be assured that the allocated memory will be sufficient to store the concerned variable and space doesn't remain acquired by junk bytes as well. Basically, the C programming language manages memory in 3 different ways. They are

- Static Memory Allocation
- Automatic Memory Allocation
- Dynamic Memory Allocation

Till now we have learned about allocating memory space for variables in a static mode which leads to misuse of space. Suppose we want to store a name, we declared a string of 100 characters to store it. If we store the name 'Tim Cook,' just 8 characters are used. It will be great if there is a way to allocate space according to the data input by the user. Fortunately, C does support this type of space management, typically called dynamic memory allocation. "

Dynamic Memory Allocation

Dynamic Memory Allocation is the way of manually managing the memory either by allocating or releasing memory whenever we want while running the program. Whenever these functions are called they take memory from the heap which is the memory area and release the memory whenever not required, such that it can be reused. There are mainly two functions, performing memory allocation dynamically `calloc` and `malloc`.
Here is a short introduction to them.

C malloc()

The term "malloc' stands for 'memory allocation'. Malloc in C is used to allocate memory of a specified number of bytes at runtime.

Syntax :

The prototype of the malloc() function is as follows :

```
ptr = (int*) malloc(10 * sizeof(int));
```

Now all it depends on the property of the integer. If the integer contains two bytes, the allocated space will be 20 bytes. 40 bytes of space will be reserved if the integer has the size of 4 bytes. Now we will try to grasp the concept with a relevant example. Note that we have included a library file `stdlib.h`, as these functions are defined in this specific header file.

Example:

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

main()
{
 char name[50];
 char *define;
 strcpy(name, "Thomas Edison ");
 define = malloc(50* sizeof(char));

 if (define == NULL)
 {
  fprintf(stderr, "Error - Memory allocation failed. \n");
 }
 else
 {
  strcpy(define, " Thomas Edison invented incandescent lights.");
 }

 printf("Name is %s\n", name);
 printf("Invention : %s\n", define);
}
```

Output:

```
Name : Thomas Edison
Invention :  Thomas Edison invented incandescent lights.
```

So, the program gives total control of memory allocation to the program. He can change the value '50' to 20 or 100 as the demand of the situation. In this example, we have included a string header file to manipulate string operation which you will learn in detail in coming tutorials.

C calloc()

Another library function that is supported by the C language to allocate memory dynamically is the `calloc()` function that is considered better than `malloc`. The function "**calloc**" means "Contiguous Allocation" or serial and sequential allocation of adjacent cells.

Syntax:

The prototype declaration of this function is:

```
pointer = (data_type*) calloc( memory_units, size_of_element);
```

Here the `calloc()` accepts two arguments :

- memory units denote the number of elements
- size_of_element represent the size of the elements

Suppose we want to allocate space for 10 units of the float. We know the float variable occupies 4 bytes of space, hence 10 units of it implies 40 memory elements. So memory allocation for it will look like this:

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

Example:

```
#include <stdio.h>

int main()
{
 int n;
 int *p;
 int i;

 printf("Enter the number of elements in array :\n");
 scanf("%d", &n);

 p = (int*) calloc(n, sizeof(int));
 printf("\nEnter the elements in array :\n");
 if (p == NULL)
 {
  printf("\n Error : Memory allocation failed");
  exit(1);
 }
 else
 {
  for (i = 0; i < n; i++)
  {
   scanf("%d", p + i);
  }

  printf("The array is :\n");
  for (i = 0; i < n; i++)
  {
   printf("%d\t", *(p + i));
  }
 }

 printf("\n\nThe array in reversed order is:\n");
 for (i = n - 1; i >= 0; i--)
 {
  printf("%d\t", *(p + i));
 }

 return 0;
}
```

Output:

```
Enter the number of elements in array :
4

Enter the elements in array :
5
6
8
9
The array is :
5       6       8       9

The array in reversed order is:
9       8       6       5
```

The key difference between malloc and calloc function

The key difference between a `calloc` and the `malloc` is that the former allocates some multiple blocks of memory of the same size. Whereas the latter allocates a single block of the memory. Calloc function always initializes the allocated memory bits to zero while the malloc function does not and hence contains garbage values. Calloc function takes two arguments while the malloc needs only one to allocate the memory.

Resizing and releasing memory

When the execution of a program finishes, two situations may arise. One is you may no longer need the allocated space and the other is we have to redefine the size(i.e. increase or decrease). To serve these purposes during runtime, there are two separate functions in C, free and reallocate respectively.

C realloc()

As the name suggests, it reallocates the space for data storage. In other words, reallocate() function enables the feature of modifying the size of previously allocated memory without losing old data.

Syntax:

The prototype of `realloc()` function is :

```
pointer_name = realloc(pointer_name, new_size)
```

Like `calloc()`, `realloc()` also takes two arguments, one argument gives the first byte of memory allocated previously using either `malloc()` or `calloc()` and the second argument specifies the new size (either larger or smaller than the original size). Consider the below scenario

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

Suppose we want to increase the size of allocated memory by 5 more floats, what we have to do is to reallocate space for additional 5 units of the float. We know the float variable occupies 4 bytes of space, hence 10 units and additional 5 units imply (15*4 =60) memory elements. So memory reallocation for it will look like:

```
ptr = (float*) realloc(ptr, 15*sizeof(float));
```

Example:

Here is an example that demonstrates the working of `realloc()`:

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    int *p,i,n,i;
    printf("Enter the size of the array :");
    scanf("%d", &n);

    p = (int*) malloc(n * sizeof(int));
    for(i= 0; i < n; ++i)
    {
     printf("\nAddress of each element in array at index %d ==> %u ",i,p+i);
    }

    printf("\n\nEnter new size of the data: ");
    scanf("%d", &ns);
    p = realloc(p, ns);
    for(i = 0; i < ns; ++i)
          printf("\nAddress of each element in array at index %d ==> %u ",i,p+i);
}
```

Output:

```
Enter the size of the array :6

Address of each element in array at index 0 ==> 1731824
Address of each element in array at index 1 ==> 1731828
Address of each element in array at index 2 ==> 1731832
Address of each element in array at index 3 ==> 1731836
Address of each element in array at index 4 ==> 1731840
Address of each element in array at index 5 ==> 1731844

Enter the new size of the data: 2

Address of each element in array at index 0 ==> 1731824
Address of each element in array at index 1 ==> 1731828
```

C free()

The job of this function is to release the allocated memory. Actually, memory allocated by the functions like `calloc` or `malloc` does not get vacant automatically after the execution of the program. To explicitly free the space, we just have to call the function free() with the proper pointer holding the address to be released as an argument.

Syntax:

The prototype of the function is

```
free (pointer);
```

Example to demonstrate `free()`:

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

int main()

{
 int *ptr;
 //creating memory allocation dynamically

 ptr = (int*) malloc(sizeof(int));
 printf("\n Memory allocated...");
 *ptr = 100; //assigning value  printf("\n ptr = %d",*(ptr));
 free(ptr);
 printf("\n Memory is deallocated...\n ");  printf("ptr = %d",*(ptr));
 return 0;

}
```

Output:

```
Memory allocated... ptr c
Memory is deallocated... ptr c
```

In this example initially, the memory is dynamically allocated using `malloc()` function and it is assigned with an integer value. Later we have deallocated the memory using the `free()` function.