calloc() versus malloc()

The name **malloc** and calloc() are library functions that allocate memory dynamically. It means that memory is allocated during runtime(execution of the program) from heap segment.

- **Initialization:** malloc() allocates memory block of given size (in bytes) and returns a pointer to the beginning of the block. malloc() doesn't initialize the allocated memory. If we try to acess the content of memory block then we'll get garbage values.

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

  calloc() allocates the memory and also initializes the allocates memory block to zero. If we try to access the content of these blocks then we'll get 0.

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

- **Number of arguments:** Unlike malloc(), calloc() takes two arguments:
  1) Number of blocks to be allocated.
  2) Size of each block.
- **Return Value:** After successfull allocation in malloc() and calloc(), a pointer to the block of memory is returned otherwise **NULL** value is returned which indicates the failure of allocation.

For instance, If we want to allocate memory for array of 5 integers, see the following program:-

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

int main()
{
    int *arr;

    // malloc() allocate the memory for 5 integers
    // containing garbage values
    arr = (int *)malloc(5 * sizeof(int)); // 5*4bytes = 20 bytes

    // Deallocates memory previously allocated by malloc() function
    free( arr );

    // calloc() allocate the memory for 5 integers and
    // set 0 to all of them
    arr = (int *)calloc(5, sizeof(int));

    // Deallocates memory previously allocated by calloc() function
    free(arr);

    return(0);
}
```

We can achieve same functionality as calloc() by using malloc() followed by memset(),

```
ptr = malloc(size);
memset(ptr, 0, size);
```

> ***Note: It would be better to use malloc over calloc, unless we want the zero-initialization because malloc is faster than calloc. So if we just want to copy some stuff or do something that doesn't require filling of the blocks with zeros, then malloc would be a better choice.***

Use of realloc()

Size of dynamically allocated memory can be changed by using realloc().

As per the C99 standard:

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

realloc deallocates the old object pointed to by ptr and returns a pointer to a new object that has the size specified by size. The contents of the new object is identical to that of the old object prior to deallocation, up to the lesser of the new and old sizes. Any bytes in the new object beyond the size of the old object have indeterminate values.

The point to note is that **realloc() should only be used for dynamically allocated memory**.
If the memory is not dynamically allocated, then behavior is undefined.
For example, program 1 demonstrates incorrect use of realloc() and program 2 demonstrates correct use of realloc().

Program 1:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int arr[2], i;
    int *ptr = arr;
    int *ptr_new;
```

```
   arr[0] = 10;
   arr[1] = 20;

   // incorrect use of new_ptr: undefined behaviour
   ptr_new = (int *)realloc(ptr, sizeof(int)*3);
   *(ptr_new + 2) = 30;

   for(i = 0; i < 3; i++)
     printf("%d ", *(ptr_new + i));

   getchar();
   return 0;
}
```

Output:

Undefined Behavior

Program 2:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int *ptr = (int *)malloc(sizeof(int)*2);
   int i;
   int *ptr_new;

   *ptr = 10;
   *(ptr + 1) = 20;

   ptr_new = (int *)realloc(ptr, sizeof(int)*3);
   *(ptr_new + 2) = 30;
   for(i = 0; i < 3; i++)
       printf("%d ", *(ptr_new + i));

   getchar();
   return 0;
}
```

Output:

10 20 30