# Dynamic Memory Allocation in C using malloc(), calloc(), free() and realloc()

**Dynamic Memory Allocation**.

**Dynamic Memory Allocation** can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.

C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:

1. malloc()
2. calloc()
3. free()
4. realloc()

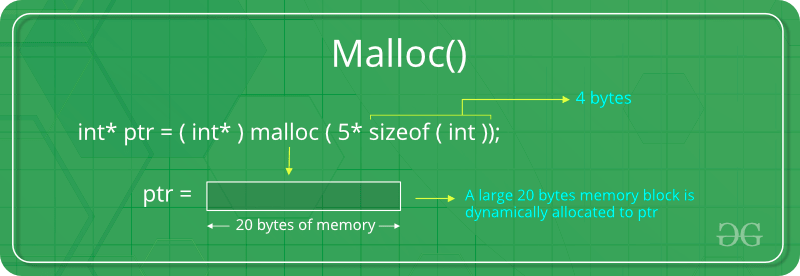
Lets see each of them in detail.

### malloc()

“**malloc”** or “**memory allocation”** method is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form.If the space is insufficient, allocation fails and returns a NULL pointer

**Syntax:**

**ptr = (cast-type\*) malloc(byte-size)**

For Example:

**ptr = (int\*) malloc(100 \* sizeof(int));**

### calloc()

“**calloc”** or “**contiguous allocation”** method is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value ‘0’.

**Syntax:**

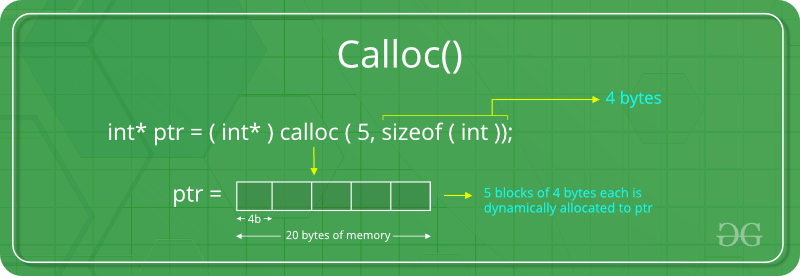
**ptr = (cast-type\*)calloc(n, element-size);**

For Example:

**ptr = (float\*) calloc(25, sizeof(float));**

This statement allocates contiguous space in memory

for 25 elements each with the size of float.



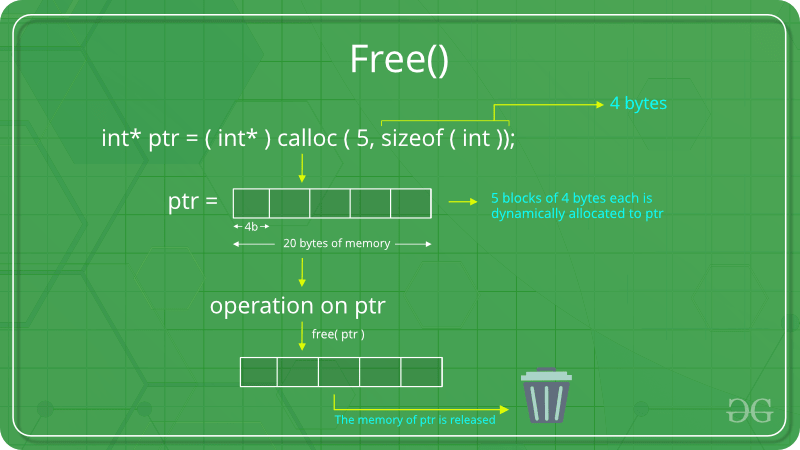
### If the space is insufficient, allocation fails and returns a NULL pointer

### free()

“**free”** method is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() are not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it. Free function will not return anything

**Syntax:**

**free(ptr);**



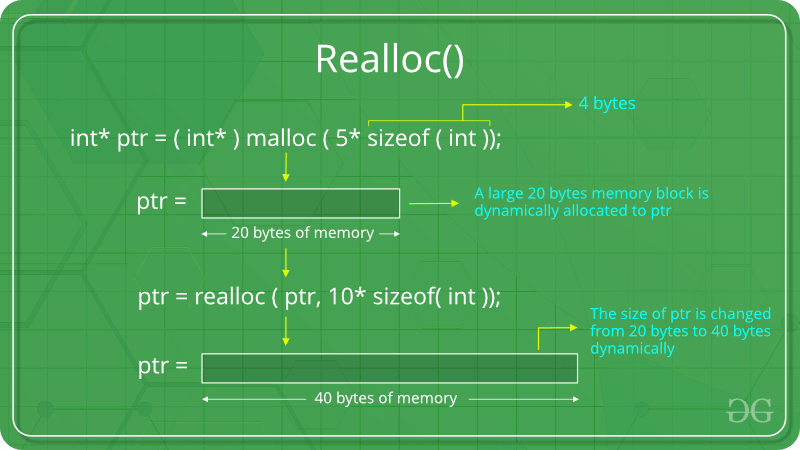
### realloc()

“**realloc”** or “**re-allocation”** method is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**.

**Syntax:**

**ptr = realloc(ptr, newSize);**

where ptr is reallocated with new size 'newSize'.



If the space is insufficient, allocation fails and returns a NULL pointer.

**Allocated Memory:**

* when we allocated memory by using malloc() or calloc() or realloc(). first it goes make pointer is created it has some adress point when we allocate it starts from there and go on to that how much the size of the memory it want. when we dellocate it come to the starting point of it (here make pointer plays a major part in free)**.**
* The extra space need not be just a word size - in fact, usually, it's more than that. There can be a header at the start of the allocated memory which has a size field. Simple case: Free takes the pointer passed in as argument, subtracts the size of the header struct, then reads the size field in the struct and frees the whole thing (header + allocated memory)**.**

**Most implementations of C memory allocation functions will store accounting information for each block, either in-line or separately.**

One typical way (in-line) is to actually allocate both a header and the memory you asked for, padded out to some minimum size. So for example, if you asked for 20 bytes, the system may allocate a 48-byte block:

* 16-byte header containing size, special marker, checksum, pointers to next/previous block and so on.
* 32 bytes data area (your 20 bytes padded out to a multiple of 16).

The address then given to you is the address of the data area. Then, when you free the block, free will simply take the address you give it and, assuming you haven't stuffed up that address or the memory around it, check the accounting information immediately before it. Graphically, that would be along the lines of:

\_\_\_\_ The allocated block \_\_\_\_

/ \

+--------+--------------------+

| Header | Your data area ... |

+--------+--------------------+

^

|

+-- The address you are given

Keep in mind the size of the header and the padding are totally implementation defined (actually, the entire thing is implementation-defined (a) but the in-line accounting option is a common one).

The checksums and special markers that exist in the accounting information are often the cause of errors like "Memory arena corrupted" or "Double free" if you overwrite them or free them twice.

The padding (to make allocation more efficient) is why you can sometimes write a little bit beyond the end of your requested space without causing problems (still, don't do that, it's undefined behaviour and, just because it works sometimes, doesn't mean it's okay to do it).