**Dynamic memory allocation:**

* In this we are going to see how to allocate memory dynamically.
* In short we call it as DMA.
* Whenever we declare a primitive variable the memory will be allocated during compile time.
* This is also called as static memory allocation.
* Example: int a; // where the variable “a” is allocated memory during runtime.
* Static memory allocation is nothing but fixed size allocation.
* Every primitive variable come under static memory variable.
* So it is similar in arrays too.
* Example: int array[5]; // even here the size is been fixed to 5 location.
* So let us consider the user defined data type structures.
* Example: struct employee e[5] // this also is a primitive allocation.

Dynamic memory allocation is increasing the size or decreasing the size of the array based on the number of elements storing or getting removed.

It is working is similar also in case of structure, where the memory is either decreased or increased based on the record we insert or remove.

* If we want to allocate memory dynamically we need to use the library/ header “stdlib.h”.
* This library will provide four function to allocate or deallocate the memory.
* The functions include:

1. malloc() 🡪 used to allocate memory for structures(user defined DT).
2. calloc() 🡪 used to allocate memory for arrays.
3. realloc() 🡪 used to resize the array.
4. free() 🡪 used to deallocate the memory allocated.

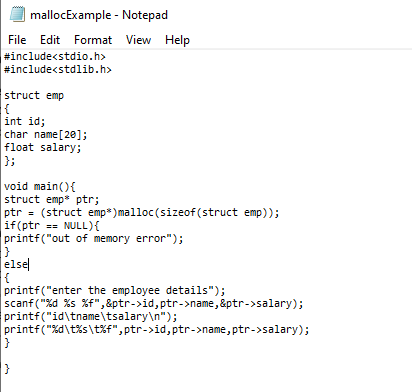
* In this DMA the pointers plays an important role.

**Allocating memory to structure variables using malloc():**

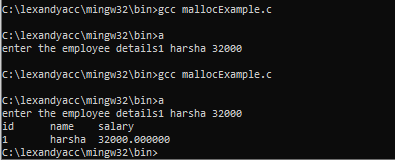
* malloc() is basically used to allocate the memory for user defined data types i.e structures.
* The prototype of this malloc() is given as:

Void\* malloc(size\_t size)

* The return type of malloc() is a void pointer.
* Size\_t represent unsigned, which is nothing but a positive integer value.
* So if the memory is **allocated successfully**, it **returns** **base address** of memory block.
* On **failure**, **it returns null pointer**.
* The return type is a void pointer because, it becomes easier to type caste it.
* Say we have 3 structures, emp, stu, and teach so we need to allocate memory for each them. We cannot allocate the same bytes of memory to all. So in prior we return the void pointer and it could be type casted to that respective structure type.
* Example:



**Output:**

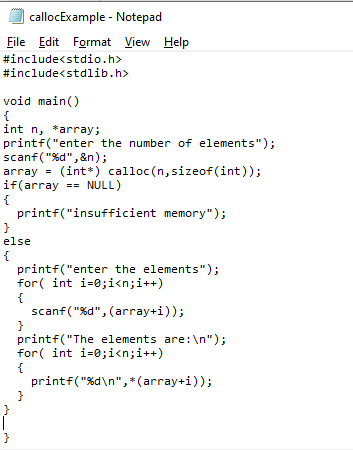


**Allocating memory dynamically to arrays using calloc():**

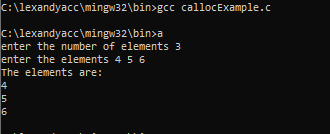
* Prototype of calloc():

Void\* calloc(size\_t n, size\_t size)

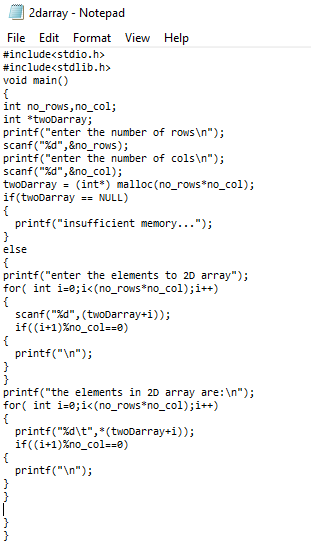
* Here “n” reperesents the size of the array. And size represents the size of elements in the array.
* So the calloc() will take two arguments, first one it is the size of the array(number of element to be inserted) and second one it is the size of each element in the array.
* If the **allocation is successful** it will **be allocated with n\*size** of memory block and **returns address** of memory block
* **On failure** it **returns** the **null pointer**.
* So as similar to malloc(), calloc() will also return the void pointer we need to type cast to the type of the pointer.
* Note that calloc() allocates the memory to the arrays, but we are again here fixing the size of the array.
* So now you may have the doubt that then how will calloc() contribute in dynamic memory allocation.
* Because of this there was a new concept introduced was realloc(), which we will see in below sections.
* Example:



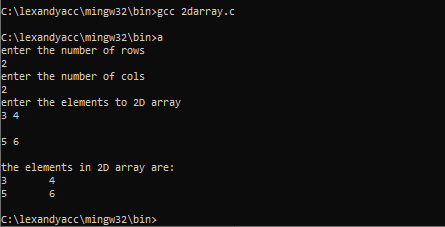
**Output:**



* So when you want to **declare or initialize 2D array** we have to **use malloc()** itself.
* Example:



**Output:**



**Realloc():**

* Realloc() is used to increase or decrease the size of array.
* Prototype:

Void\* realloc(void\* ptr, size\_t size)

* Where first argument is the array whose size needs to be increased or decreased.
* The second argument is the size of the new block.
* Say we are implementing the stack operation.
* In which say I have given some initial capacity of the stack.
* And then I want to increase the capacity we have to use the realloc().
* So how it is done:

**int \*stack;**

**int capacity=2; //note capacity is 2 so only two elements can inserted**

**stack = (int\*) calloc(capacity, sizeof(int));**

/\*now I need to increase the size of the stack because, assume I need to insert another element\*/

/\* now first I need to increment the capacity and then rellaoc() should be used\*/

**capacity++;**

**Stack = (int\*) realloc(stack, capacity\*sizeof(int));**

* Now we can resize the array however we want
* So now you may also have a question that instead of using realloc(), I may again use calloc() itself right?
* The answer for this is you will lose the data which you have already worked on, and the pointer will point to the new block not the extension of the same block.