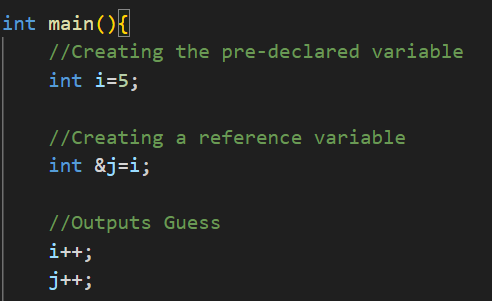
**Reference Variable**

Reference variable is a variable that points toward a preexisting memory unit that is already being pointed by preexisting variable. The code below shows how we create a reference variable and few operations associated to it.

See how we created the reference variables. Note that initially in the memory block to which I and J were pointing is storing 5. So when increment operator is run on both of them, the outputs are 6 and 7.

6

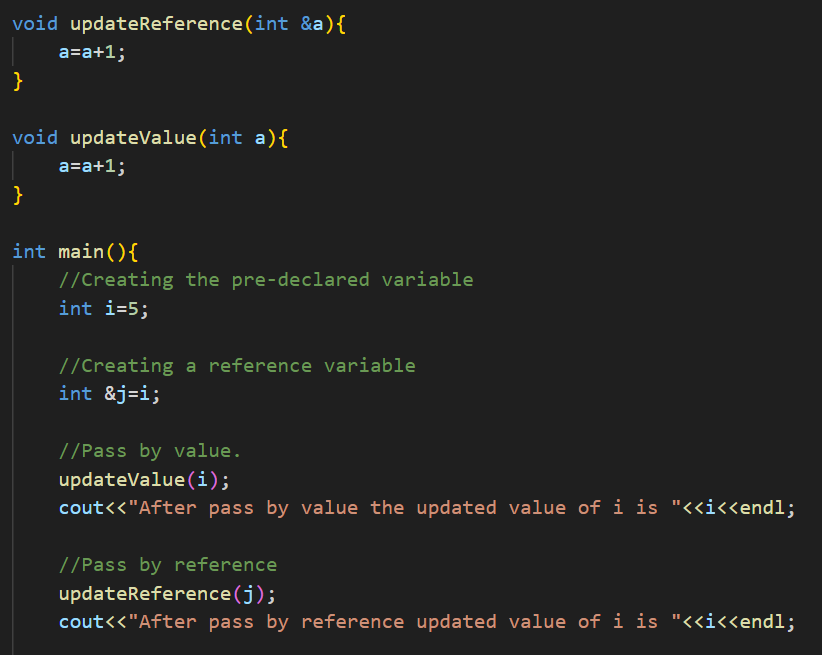
7

Next thing to ask is that why do we require this in the first place. Answer is, the main application is observed at the time of functional programming. Reference variable are used as arguments for function calling for supporting the pass by reference phenomenon. This has two advantages:-

1. Changes made in the function locally effect the memory and the variable globally.
2. Alternate phenomenon to pass by reference is pass by value. Under this we see that a copy of current argument is created and passed onto the function. Because of this,
3. New memory is created. This makes program take larger space.
4. Local changes are not visible globally.

The thing with pass by reference supported by reference variables is that since the memory address itself is being passed we see that no new memory is created as the work is conducted on the already present memory.

Lets see a set of programs to draw distinction

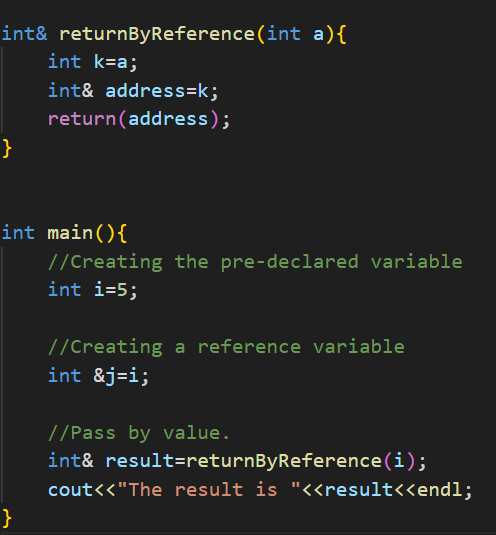
See the outputs are:-



As observable, our notations and observations made till this point are quite correct.

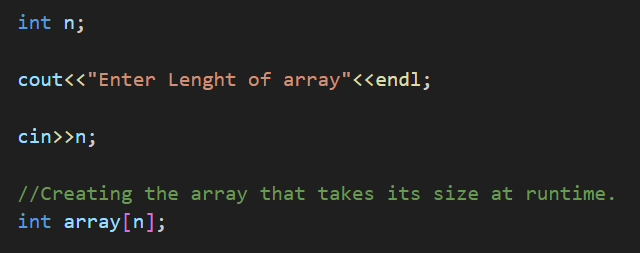
**Return By Reference**

Watch the following code and note down the observations.

This is wrong. This is because we are returning a address of variable of local scope into another function so the following issues may arise at the time of implementation.

1. The variable’s memory might not be accessible in the first place, as accessing local variables from other functions is kind of hard.
2. The data in the unit may be re-written externally which may lead error output.

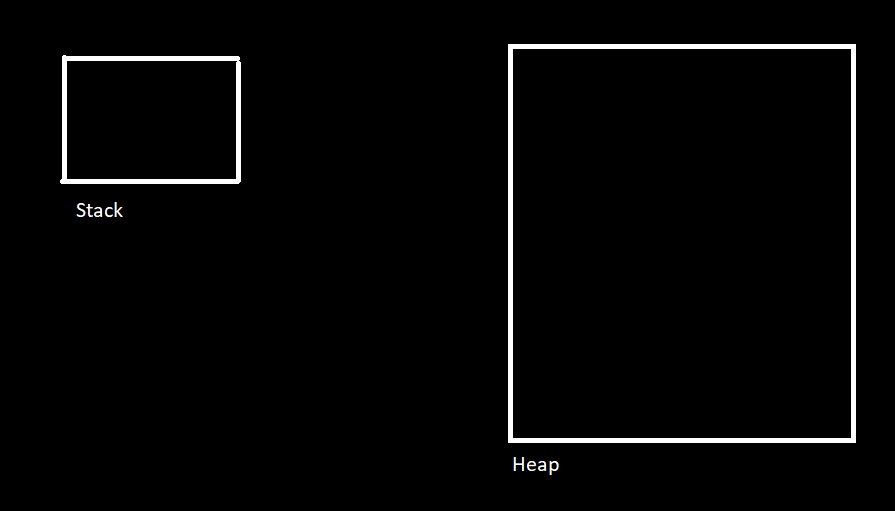
Lets talk about good and bad practices but this time in reference to arrays here. See the following code.

Generally when we create arrays we see that we pre-define its size. Here though we see that we are defining the size of the array in the run time. Now this is a bad practice, and here’s why?

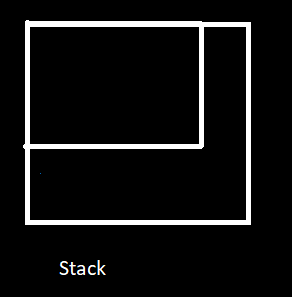
The memory allocation for creating an array can occur in 2 ways,

1. By stack
2. By Heaps

Though the normal methods that we have used till now to declare and work with arrays we see that, stack memory is allocated. Lets first visualize the stack and heap memories.



As observable the stack memory seems quite limited whereas the heap memory doesn’t. When the size of the array is declared at the time of code, the stack memory modifies its size. Something like this.

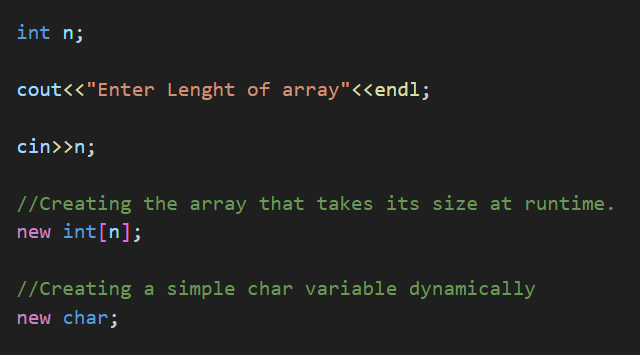
So, as observable that the stack memory increases its size. Though this adjustment is not possible when the size of the array is declared in runtime. Hence if a particular size is declared that is more than the pre-set size of stack memory we see that error is generated.

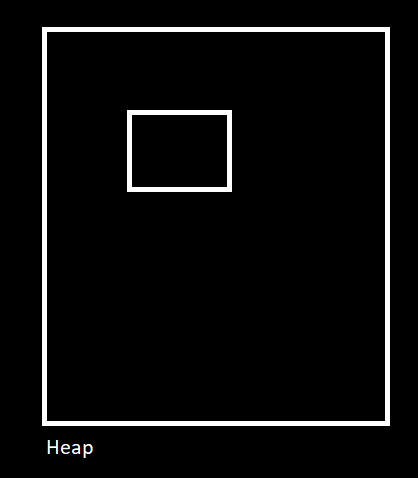
So what is the solution? We remember that the memory allocation of arrays occurs in two ways, stacks and heaps. Though the stack are limited in size, heaps are not hence we see that for avoiding the issues we just discussed we use heap allocation. But we just mentioned how, by default, the memory, allocated to the arrays is stack, then how do we access the heap memory?

This is done via the “new” keyword. Lets learn more about this.

**Dynamic Memory Allocation**

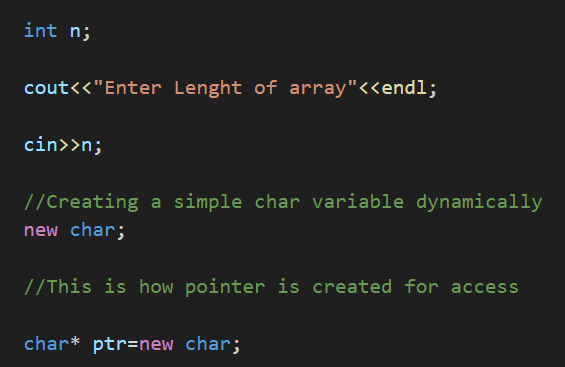
So in the last concept we have pretty laid the foundations of DMA. In short though I would like to summarize that, static memory allocation takes place when we work with **Stack Memory** and dynamic memory allocation takes place when we work with **Heap Memory.** Now we left the last discussion where we talked about how can we work with DMA. Well we do this, via the help of **new** keyword. Lets see the code of how we do it.



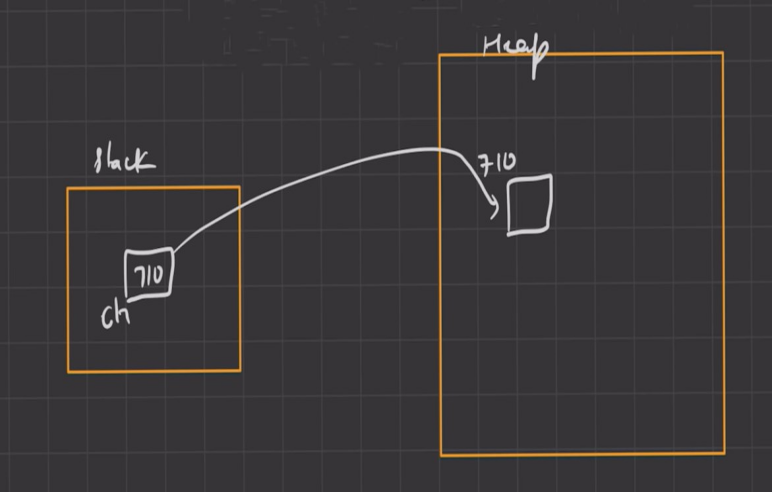


This is how a c char memory u unit is created I in heap.

Note that, when we do execute this code we see that, an address is returned. So the big problem that arises here, we have the address, how to we access the address to work with it? Well we use pointers. See the code.



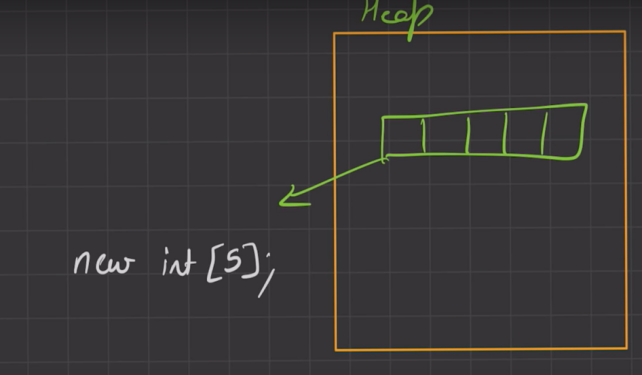
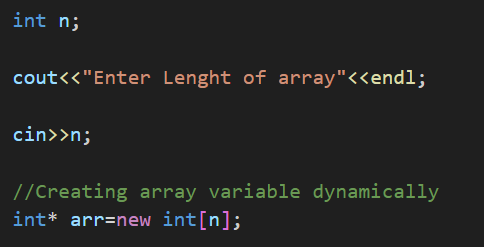
This is how we create pointers and access the heap memory units. Now lets discuss what happens behind the scenes in the memory unit.

Whatever we are discussing here we are discussing in reference to last code. The ptr variable created is created in stack memory while the address to which “new char” points is stored in heap. The stack pointer points to the heap memory unit. Next question is whats the size of the pointer? Surely 8 right? Well no, the general formulae for determining that is,

**(Size of pointer Variable+Size of datatype to which the pointer points)**

In this case it will be (8+1) bytes that is 9 bytes, not 8. If it would have been int, it will be (8+4) bytes, 12 bytes.

We have explained things in respect to basic datatypes, char, int and etc. Let talk in reference to arrays. See the code and the adjoining diagram.

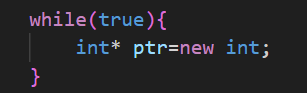
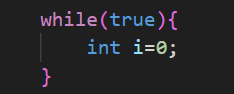


On left you see the code, whereas on the right you see the memory activity. Again? What the size of the memory unit?

Yes (5\*4+8) bytes or 28 bytes.

Lets focus on differences between the SMA and DMA. We have already discussed how one works on stack and the other works on heap. We have also discussed, how pointer sizes vary, 8 in static and **(Size of pointer Variable+Size of datatype to which the pointer points)** in dynamic. Lets see 2 sets of code and differentiate between them?

Code1 Code2



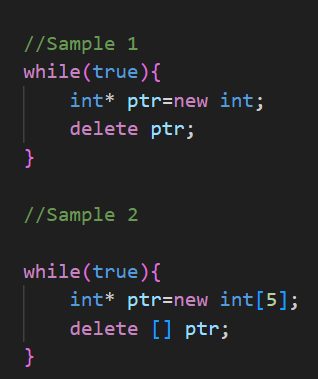
One thing that is clear from both of the code is that, both are running on infinite loop. But what we want to discuss here is that when either code will be run, what will happen in the memory unit?

Lets concentrate on **Code 1**. In code 1, we enter the loop, a int type number is stored in memory unit(Static/Stack). Size of that is 4. Then we exit the loop. When we exit it, the memory that was initially created, get deleted. Then we re-enter it, its get created again then deleted again. So we see that a total of 4 bytes is only consumed.

Lets see **Code 2**. In code 2, we enter the loop, a int type number is stored in memory unit(Dynamic/Heap). Size of that is 4. Then we exit the loop. When we exit it, the memory that was initially created, doesn’t gets deleted like the previous case. It stays there. Note that till now we have only consumed 4bytes of data. Now we re-enter the loop, another memory unit of 4 bytes is created in the heap, and when we exit it doesn’t get deleted, making the total size of the heap, 8 bytes. This goes on and on, until, heaps memory limit is crossed and the program crashes.

Now we discussed this problem in heap. What the solution? We use the delete keyword.

See the code.

 The two samples beside show application of delete for predefined and complex datatypes. In either case, the created heap memory is deleted, and memory doesn’t over sizes.

**Creating A 2D Array With DMA**

Now lets see the issue here. The issue of creating a 2D array while runtime mode is on is similar to that of arrays. Stack limitations. Hence here also we need DMA to create 2D array in runtime. Lets see the actual procedure of mixing up DMA and 2D arrays.

**Step 1**: We will create a pointer, where the pointer points to array we are going to create next.



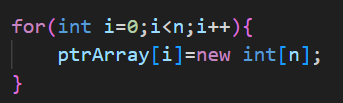
**Step 2**: Now next step is, link each of this pointer to an array of pointers. The code is,



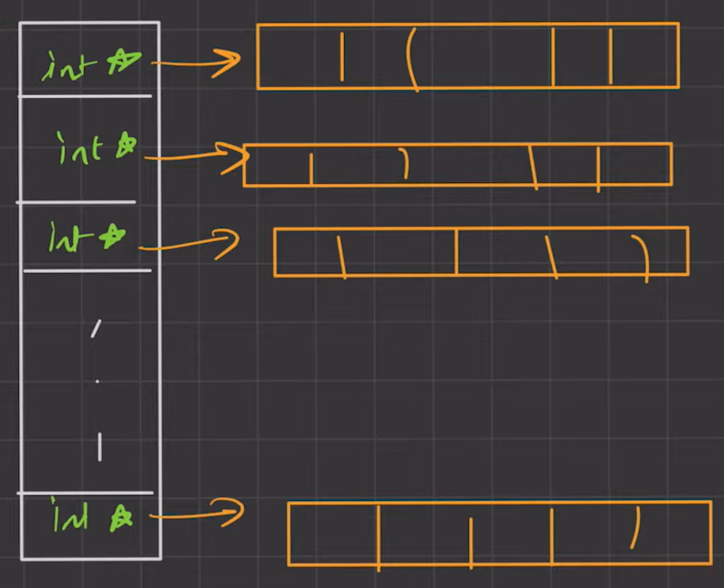
Lets see the diagrammatic representation of this process.

The “int\*\* ptr” expression creates the pointer to array of pointers which we created by the expression “new int\*[n]”, represented by the white boxes.

**Step 3**:Now the next step is, link all the pointers of the pointer array to arrays separately. The code for doing it is as follows.

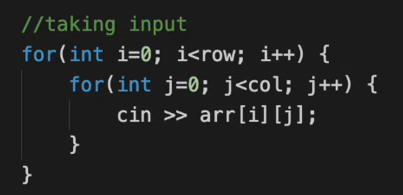


Now the memory representation can be as follows.

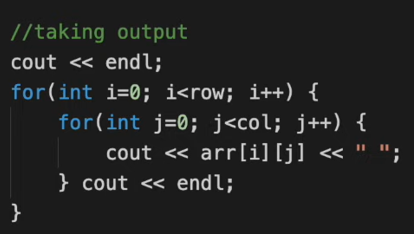


As observable, each and every pointer of the pointer array it pointing a external array, create a superficial 2D array.

**Step 4**: Now that we have created our next step, enter values into it. The code for doing that is,

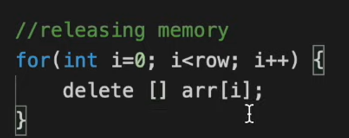


And we print the output like this,



Though the last thing continuously clearing the heap memory since it can’t do It by itself. The codes for it is as follows.

1. For clearing the rows.



1. For clearing the pointer array.

