**Memory Allocation in c :**

**1. Static Memory Allocation :**

In this memory allocation, memory is allocated at the compile time. The allocation and deallocation is done by the compiler itself. When a variable is declared inside the function without any dynamic allocation, it is assigned memory at compile time. It is also termed as a local variable as its scope is limited to its function only. Hence when the function executes completely, this memory is deallocated by the compiler. So allocation and deallocation of memory is done by the compiler itself.  
For Example :

int a = 10;

When a compiler encounters this statement, it assigns either 4 or 2 bytes(depending on architecture of the machine) to the variable a and also stores value 10 in it, at compile time.

This way of allocation of memory stores memory in the stack.

**2. Dynamic Memory Allocation :**

Dynamic memory allocation is done at the run time. The size of the memory allocated can be altered during the execution of the program. Dynamic memory is the memory allocated during runtime using different functions like malloc, calloc. This memory is accessed using a pointer. During execution of the program there might come a need to extend the current data structure. This extension can be done at runtime by using dynamic memory allocation.

The memory allocated at runtime also need to be freed. It's not deallocated on its own like static memory.

Syntax for allocating memory using malloc function :

datatype \* pointerName = (typecastDataType \*)malloc(sizeToBeAllocatedInBytes);

For example to dynamically allocate memory to of a single integer, we do

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

Similarly to allocated memory to an integer array of size n :

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

Malloc function returns the pointer of void type which is then typecasted in whatever form required. As it is typecasted in integer pointer using (int\*).

The memory allocated at run time needs to be deallocated manually by the programmer using “free”.

Calloc function is also similar to that of malloc besides it assigns default value as 0 which is not the same case in malloc.

And also the calloc function requires two arguments when it is called.

First is the no. of blocks of memory to be allocated and second as the size of each memory block allocated.

Above allocation which is done using malloc can be done using calloc in this way:

int \*ptr = (int\*)calloc(n,sizeof(int)); where n are no. of blocks of memory and sizeof(int) is size of each block.

int \*dynamicInteger = (int \*)malloc(sizeof(int));

free(dynamicInteger); //De-allocate the integer memory

The realloc() method is used to re-allocate the memory which already exist. It re-allocate the memory which is previously allocated using malloc or calloc function. It extend the previous memory and assign the previous memory in already existing blocks and garbage value in the newly created blocks.

int \*ptr = (int\*)malloc(4\*sizeof(int));

ptr = realloc(ptr,7);

now previous 7 blocks of memory are re-allocated with previous values and garbage values in newer blocks of the memory.

**Garbage Value :**

When a variable is not initialised with a specific value then it contains some value which might be already present at that specific memory location(which is unknown to us). Such values are termed as garbage values as they are of no use for us.

For Example :

Int a ;

Now, when this line executes, it has an unknown value stored at memory labelled as “a”. This value is of no use for us.

Garbage value can lead to undefined behaviour of program so its suitable to initialise a variable.  
So it's required to initialise a variable before using it in order to avoid any possible issue in the code.

When memory is allocated using malloc function(dynamically), values are initialised with garbage value. Where calloc initialise the memory with a default value (Zero for integer)

**Layout of the memory in c :**

**1 Stack Memory :**

When a function is called it is pushed into the call stack along with the local variable associated with it. Hence every time a function is called a frame is pushed into the stack, along with its local variables. This stack is termed a call stack and all of this memory is stack memory. This memory is stored by static allocation of the memory. A stack pointer is maintained in the call stack, when a function returns, its frame is popped out of the stack (along with the local variables). And hence the memory of the local variables is deallocated when the frame is popped out of the stack.

When the c program starts executing, the main function is pushed inside the call stack. And the main function is the last function to be popped out the call stack.   
When a function exits, control of the program comes back to the previous frame of the stack.

This memory does not need to be deallocated manually as it will be done by the compiler itself.

**2. Heap Memory :**

When a memory is allocated dynamically, it is stored in the heap memory.  
The memory is stored in the heap at run time and the location or address of this memory is maintained inside the stack in a pointer. And hence this memory is accessed using the pointer. The memory allocated in the heap part needs to be deallocated manually using the “free()”. Otherwise

it will lead to memory leak. Function like calloc, malloc, realloc are used to assign memory dynamically inside the heap. And “free()” is used to deallocate the memory and free it for other use.

**3. Code Segment :**

This part of the memory contains the instructions of the program. It stores the code of the program.

It has the compiled program code.

**4. Initialised and uninitialised Segment :**The initialised part contains the static and global variables which are initialised by the programmer. It contains the variables(mainly global and static variables) which are assigned the initial value at the compile time.

For example static int a = 10; this variable “a” will be stored inside the initialised part. Or any global variable which is not declared inside the function such as/

int b =90; where b is integer declared globally with initial value 90 will be stored in the initialised part.

The uninitialised part contains the static and global variables(mainly) which are not initialised by the compiler. Even though the static variable has default value as zero(0). Still it is not initialised by the programmer so it is considered in the uninitialised

Section.

int a ;

Or static int b ;

Here variable a is global variable and variable b is static variable and both of these variables are not assigned with any values hence they are part of uninitialised segment.

**Array :**Array is a linear data structure which stores data of similar type in contigous memory blocks.

So whenever we require to store many similar sort of data we can use array. For example let say there are 50 studetns who gave maths exam and now we want store their marks. Their marks can be stored using an array of integer type of size 50.

Declaration of Array : type variable\_name[size\_of\_array];

int arr[60];

This declaration will create an array of integer type, which means 60 contigous blocks of int type is allocated in memory.

Elements of an array can be accessed using an index, thus same variable name can be used to access many elements by just changing the index value of the array.

Any index "ind" of an array will be accessed using arr[ind] syntax.

In this when an array of 60 elements is created the name of the array act as the pointer to the first block of the continous memory allocated.

When we write arr[6] it means (arr(base\_address) + 6 \* (sizeof(datatype)) = address of seventh block. This is the way compiler resolve the address of the blocks.

In this specific example if the size of integer is 4 bytes (which depent on the architecture), if the initial address is 1000 then,  
Arr+6 = 1000 + 6\*4 = 1024 , this is address of the 6th integer block in the array.

2-Dimensional Arrays :

2-D array can be said as the array of array. It is like a matrix of many similar data blocks. The declaration of a 2d array can be done in these ways:

int arr[row][col];

Here row and col are simply the variable values indicating the no. of rows and columns in a 2-d array.

An 2-d array of rows 3 and columns 4 will be declared as int arr[3][4]; It can also be declared along with the initialization as

int arr[3][4] = {{1,2,3,5},{6,5,42,0},{5,2,9,1}};

Any element present at 2nd row and 2nd column can be accessed by syntax "arr[1][1]", 0-based indexing is used here.

**Pointers :**

A pointer is a variable which store the address of other variable. Pointer store address of location of other variable. Using pointer that memory location can be accessed directly and the data present at that location can be manipulated directly.

\* and & are operators which are used in pointers. & is used to get the address of that variable. and asterisk(\*) is used to get the data present at that location.

Syntax of declaration : datatype \* nameOfPointer;

for example int \*ptr;

This will simply declare an variable ptr which contains the address of other variable(int type).

Let say there is an integer variable val such as int val = 5; This value 5 of variable val is located somewhere in the memory and the name of this memory block is val. Address of this block can be gained using ampersand operator(&). and this address can be stored in an pointer of integer type. Such as :

int \*ptr = &val; Here ptr has address of variable va and the val or \*(&val) will print the same values.

int \*ptr1 will be used to point to the integer variable , char\*ptr2 will be used to point to the character type variable, float\*ptr3 will be used to point to float type variable. Thus the type of pointer decide what sort of value will the pointer point to.

Pointers are also very useful when calls are made to a function. It is used to make the call by reference.

Pointers can also point to other pointer. As pointer is also a variable and store the address of other variable in a memory block which contain certain address.

For example :

int val = 10;

int \*ptr1 = &val;

int \*\*ptr2 = &ptr1;

Here ptr2 contain the address of ptr1 and ptr1 contain address address of val. Here \*ptr2 = ptr1 = &val and \*\*ptr2 = \*ptr1 = val .