Arrays - Representation

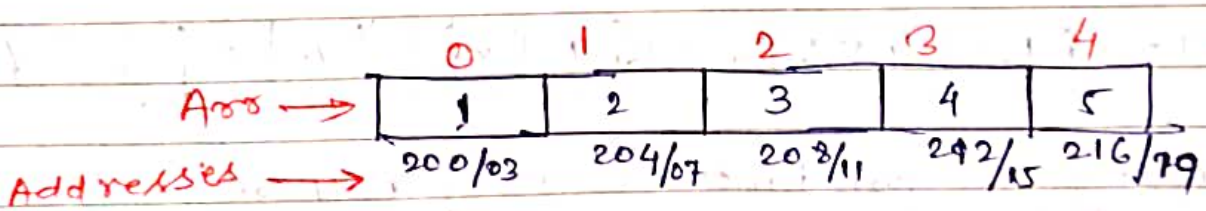
Array:- Array is a collection of elements. & all the elements are same data types, grouped under one name.

eg: `int arr[5] = {1,2,3,4,5}`

- `5` → Size of Array
- `0 1 2 3 4` → Index values.

* arrays are ~~vector~~ vector values

* If a single 'int' value take 4 bytes then the above array will take 4×5 (size) $= 20$ bytes.



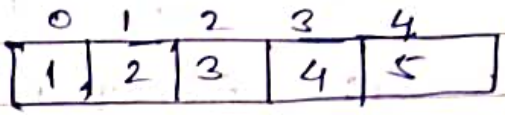
Addresses of the values inside array

* we can access or assign the element of array using index value..

eg: `cout << A[2];` // o/p = 3.

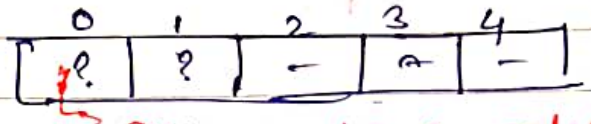
* Decleration of Array *

int arr[5] = {1,2,3,4,5};

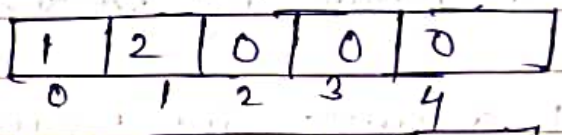
arr- 

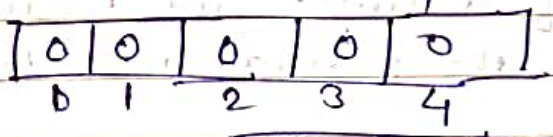
↘ Decleration + Initialization

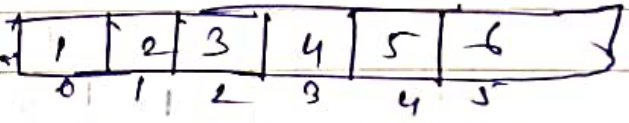
int arr[5]; /* Decleration */ with array size 5.

↳ arr → 

↳ any garbage value.

int arr[5] = {1,2}; arr→ 

int arr[5] = {0}; arr→ 

int arr_2[] = {1,2,3,4,5,6}; → 

* During the run time the array will be created inside the memory (stack/Heap) and assigning the values.

* using for loop we can traverse ~~att~~ inside the array and we can easily accessing the element.

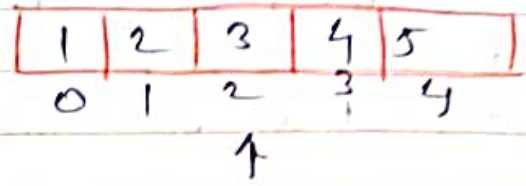
eg:

```
for(int i=0; i<6; i++){

    cout << A[i] << " ";

}
```

o/p: 1 2 3 4 5 6

* Multiple ways to accessing the elements.

```
cout << 2[A];
cout << *(A+2);
```



* Address of the element inside array is contigious.

Static V/s Dynamic Array

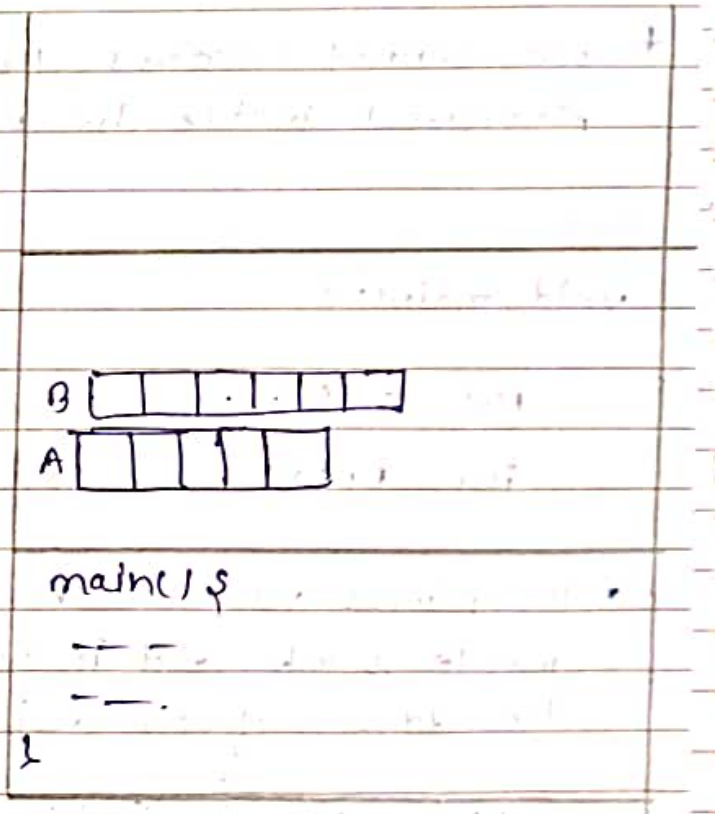
```
void main()
{
    int A[5];
```

→ This array will create inside 'stack'

* the size of the array was decided at the compile time.

* Memory is allocated at the runtime.

* we can create any size of array during runtime. in C++ and it will created inside stack only.



eg: int n;
~~cin >>~~ cin >> n; = input :- 6
int B[n];

* The array which is created inside heap memory, that's array size and memory, both are decided at runtime.

* we can use "new" keyword for creating an array inside heap memory.

* For accessing anything which is inside heap memory we required pointer variables or the pointers.

```
int *ptr;
```

* we cannot access the objects which are created inside heap without pointers.

Eg:

```
void main(){
    int ar[5];
    int *ptr;
```

* the pointer variable is created inside stack and it will store the address of the object/array.

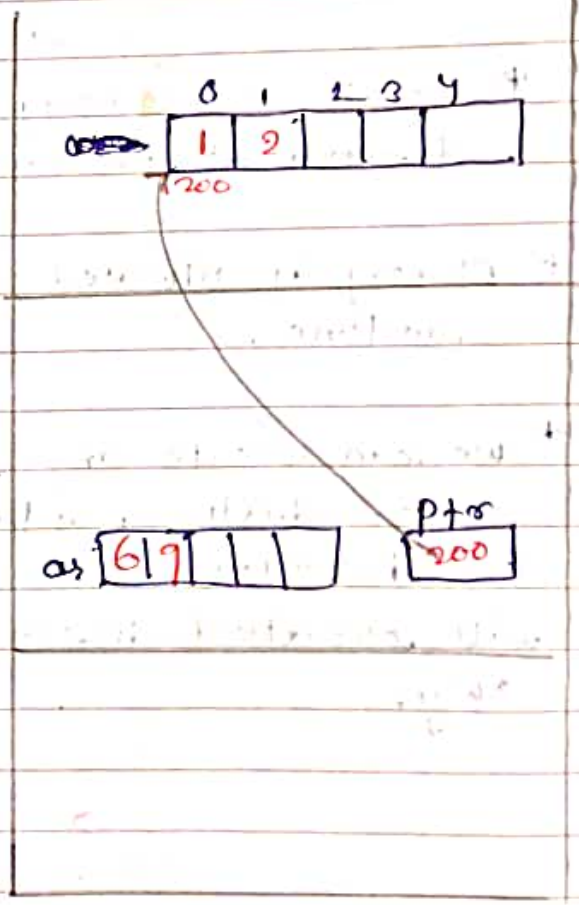
```
    ptr = new int[5];
```

* we can access the heap object using pointer.

eg:

```
ptr[0] = 1;
ptr[1] = 2;
ar[0] = 6;
ar[1] = 9;
```

}



In 'C' language we can create array inside heap such way.

```
int arr[5];
int *ptr;
ptr = (int *) malloc(5 * sizeof(int));
```

* After the execution of program or if the memory of heap not required then we must deallocate the memory from heap.

* If we cannot do it, then it may cause memory ~~leack~~ Leak.

* In c++ we can delete the unused memory using delete keyword.

eg: delete []p;

? In 'C' language we use free:

eg:

free(p);

* Once the array of some size is created (heap/stack) then it cannot be resize.

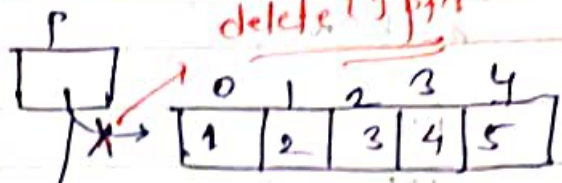
* If we want to increase the size of the array then it is possible only inside heap, but the same array's size cannot be increased.

↳ There is some alternative ways to increase.

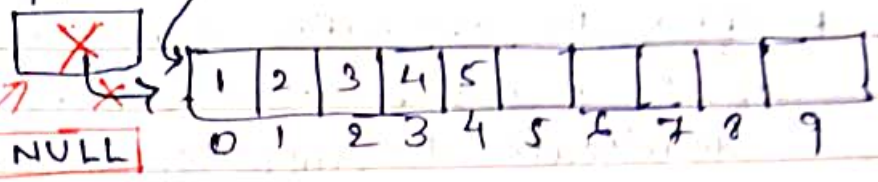
* Inside stack the size of the array cannot increase.

* How to Increase Array Size *

```
int *ptr = new int[5];
```




```
int *ptr2 = new int [newsize];
eg: newsize = 10;
for(int i=0; i<5; ++i)
    ptr2[i] = ptr[i];
ptr = ptr2;
ptr2 = NULL
```

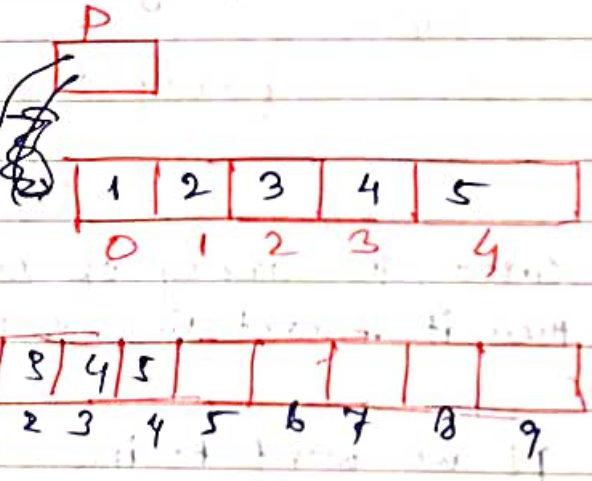


* If we want to increase the size then create an another array with bigger size and copy all the elements from previous to new array.

* then, delete [] ptr, pointer, and assing "ptr = ptr2" so, first pointer also points to the new array and make second pointer NULL, so, it will remove and the previous array deleted from the memory.

Eg:

```
int *p = new int [5];
int *q = new int [10];
```




```
for (int i=0; i<5; ++i)
    q[i] = p[i]; // all the element copied to q
delete [] p
p = q;
q = NULL;
```

Now the p pointing to the new array.

2D Array

2-Dimention Array

* Declaration of 2D Array:

"Method 1:"

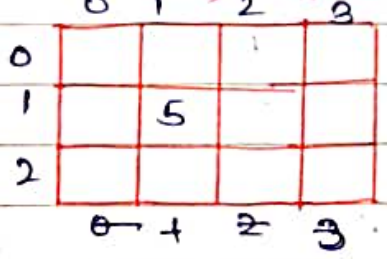
① `int A[3][4];`

`int B[3][4] = { {1,2,3,4}, {2,4,6,8}, {3,6,9,12} };`

//Initialization & Declaration.

Representation of 2D Array:

`int A[3][4]` (Row → 3, column → 4)

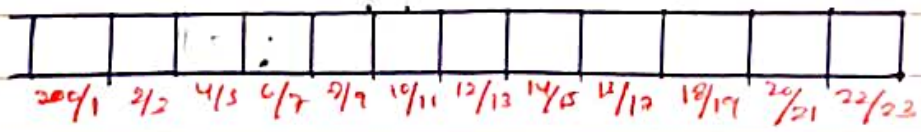


we can access element inside a 2D array using both ↑ row and column.

Eg:

```
A[1][1] = 5;
```

* 2D Array Represent inside memory:



suppose int = 2 byte

→ Addresses.

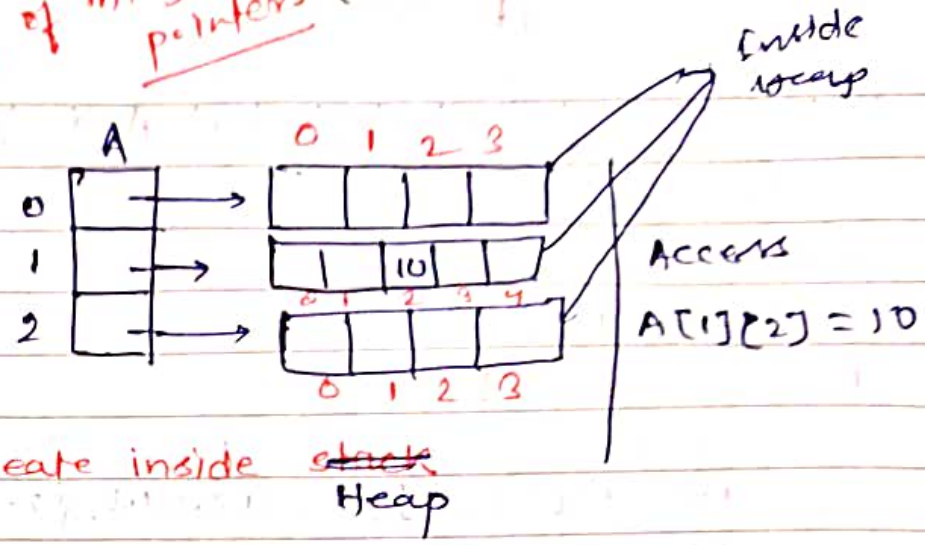
* this 2D array is created inside stack.

* Method 2 *

Array of integer pointers (stack)

int *A[3];

* using pointers *



* this array is also create inside ~~stack~~ Heap

Eg: int *A[3];

A[0] = new int[4];
A[1] = new int[4];
A[2] = new int[4];

* Method 3 *

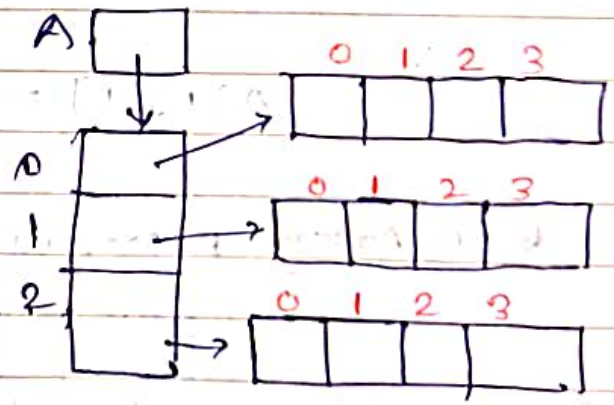
* using double pointer.

int **A; → stack Declaration.

A = new int *[3] → inside Heap

A[0] = new int [4];
A[1] = new int [4];
A[2] = new int [4];

} inside Heap

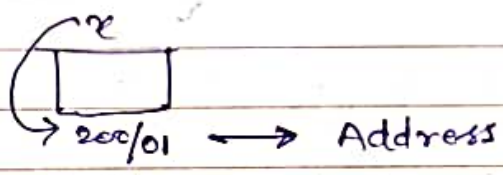


* Accessing any 2D Array, we can use nested for loops.

```
eg:  for(int i=0; i<3; ++i) {
         for(int j=0; j<4; j++) {
             /*
                 code of program
             */
         }
     }
```

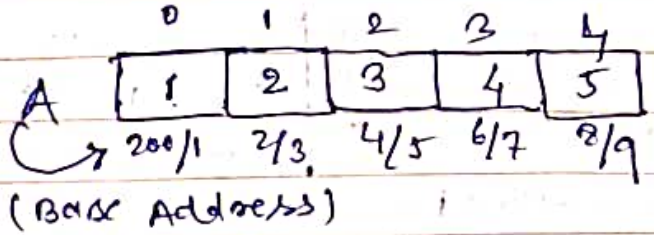
Array In Compilers

int x = 10:



* During execution the memory is allocate for the variables ~~and~~ and its store its crossponding address.

int A[] = {1, 2, 3, 4, 5}



A[3] = 4 . . . . .

formula

add(A[3]) = L0 + 3 * sizeof(int) (2, bytes)

= 200 + 3 * 2

add[A[3]) = 206

* Add(A[i]) = Lo + index * sizeof(int);

Base Address — 0 based Indexing — size of Data Type — index value

* 1 based Indexing *

Add(A[i]) = Lo + (index-1) * sizeof(int)

* Row Major formula * 2D *

int A[3][4];

Eg: here int = 2 byte.

	0	1	2	3
0	a_{00}	a_{01}	a_{02}	a_{03}
1	a_{10}	a_{11}	a_{12}	a_{13}
2	a_{20}	a_{21}	a_{22}	a_{23}

2D Array in paper

	0	1	2	3	4	5	6	7	8	9	10	11
A												
	200/1	2/3	4/5	6/7	8/9	10/11	12/13	14/15	6/17	8/19	20/11	22/23

This is how during execution 2D array look like in the compailer.

It will store the element row by row.

Eg:

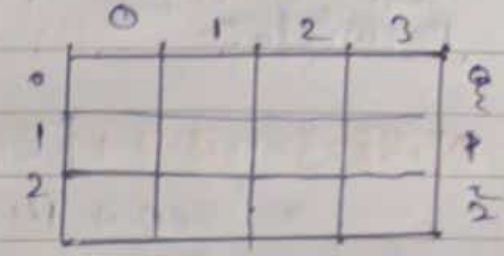
	0	1	2	3	4	5	6	7	8	9	10	11
	a_{00}	a_{01}	a_{02}	a_{03}	a_{10}	a_{11}	a_{12}	a_{13}	a_{20}	a_{21}	a_{22}	a_{23}
	200/1	2/3	4/5	6/7	8/9	10/11	12/13	14/15	16/17	18/19	20/21	22/23
	Row-0				Row 1				Row 2			

	0	1	2	3
0	a_{00}	a_{01}	a_{02}	a_{03}
1	a_{10}	a_{11}	a_{12}	a_{13}
2	a_{20}	a_{21}	a_{12}	a_{23}

$Add[A[i][j]) = 200 + [4 + 2] * 2$

$= 200 + 6 * 2$

$= 212$



A[2][1]

$Add[A[2][1]) = 200 + [2 * 4 + 1] * 2$

$= 200 + [9] * 2$

$= 218$

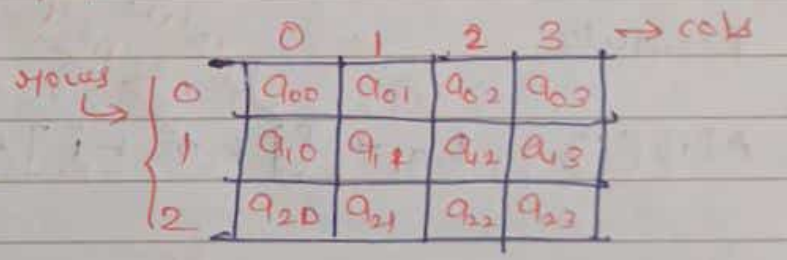
Formula :

$$Add[A[i][j]) = L_0 + [i * column\ not\ j] * sizeof(int)$$

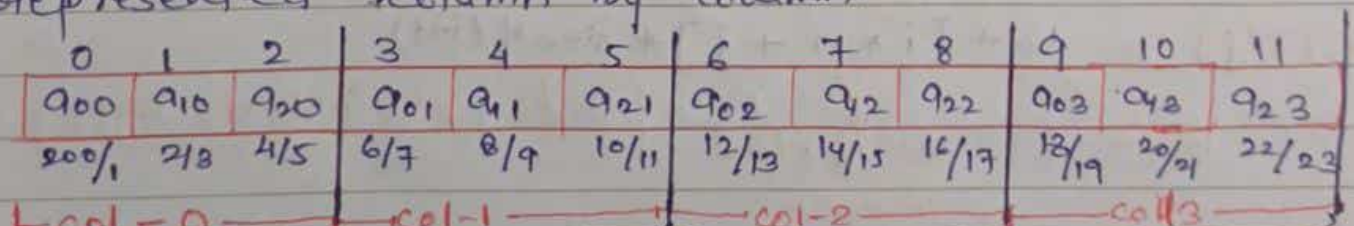
Base Address

* Column - Major Representation :

int A[3][4];



* In column - major form all the elements are represented column by column.



	0	1	2	3
0	a_{00}	a_{01}	a_{02}	a_{03}
1	a_{10}	a_{11}	a_{12}	a_{13}
2	a_{20}	a_{21}	a_{22}	a_{23}

* Assume that int = 2 byte

col 0			col 1			col 2			col 3		
a_{00}	a_{10}	a_{20}	a_{01}	a_{11}	a_{21}	a_{02}	a_{12}	a_{13}	a_{03}	a_{13}	a_{23}
200/1	9/3	4/5	4/7	4/9	10/0	10/13	14/15	14/17	18/19	8/11	22/23

Formula:-

	1	2	3
0	a_{00}	a_{01}	a_{02}	a_{03}
1	a_{10}	a_{11}	a_{12}	a_{13}
2	a_{20}	a_{21}	a_{22}	a_{23}

$A[1][2] = 200 + [2*3+1]*2$

$= 200 + 14$

$= 214$

$A[2][3] = 200 + [3*3+1]*2$

$= 200 + 20$

$= 220$

int [3] [4]

↓ ↓

m n

$A[i][j] = L_0 + [j*m + i]*$ sizeof (int) ⟷ column major

$A[i][j] = L_0 + [i*n + j]*$ sizeof (int) ⟶ row major.

L_0 → Base Address

Formula:

$A[i][j] = L_0 + [j*m + i]*$ sizeof(int); → Column Major.

(L_0: Base, j: colm, m: no of element in each colm, i: row no, sizeof(int): size of Data Type.)

$A[i][j] = L_0 + [i*n + j]*$ sizeof(int)

(L_0: Base, i: row, n: no. of elem in each row, j: colms, sizeof(int): size of Data Type.)

Reverse of row major

* column Major for n Dimension *

$$L_0 + i_4 + i_3 * d_4 + i_2 * d_3 * d_4 + i_1 * d_2 * d_3 * d_4$$

~~$L_0 + i_4 + i_3$~~

$$L_0 + i_4 + d_4 (i_3 + i_2 * d_3 + i_1 * d_2 * d_3)$$

$$\boxed{L_0 + i_4 + d_4 [i_3 + d_3 * (i_2 + i_1 * d_2)]}$$

↳ Time Taken by this formula = $(n-1) = \theta(n)$

* Formula for 3D array *

eg: int A[l][m][n];

row - Major

→

addr(A[i][j][k]) =

$$\boxed{L_0 + [i * m * n + j * n + k] * w}$$

Column-major mapping:

←

$$addr(A[i][j][k]) = L_0 + [k * l * m + j * l + i] * w$$