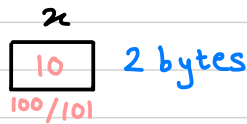




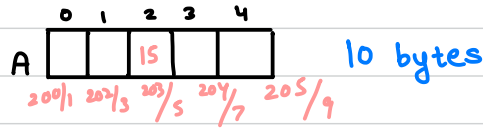
# ARRAYS

Scalar ↘

int x = 10;

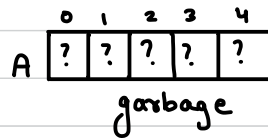


↗  
vector  
int A[5];  
A[2] = 15;

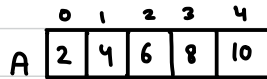


## DECLARATION OF ARRAYS

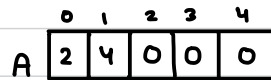
① int A[5];



② int A[5] = {2, 4, 6, 8, 10};

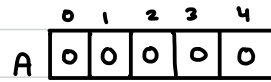


③ int A[5] = {2, 4};

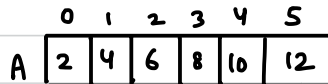


→ Rest of the elements get automatically initialized by 0.

④ int A[5] = {0};

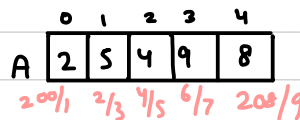


⑤ int A[] = {2, 4, 6, 8, 10, 12};



→ Depending upon the number of elements, size of the array is automatically allocated.

int A[5] = {2, 5, 4, 9, 8};



printf(" %d", A[2]);

printf(" %d", 2[A]);

printf(" %d", \*[A+2]);

→ OUTPUT  
4

→ Array addresses are contiguous.

for(i = 0; i < 5; i++)

printf(" %u", &A[i]);

↓ ↓  
for printing address.

## STATIC VS DYNAMIC ARRAY

Size of the array  
is static

Size of the array is dynamic

→ Once an array is created, its size cannot be modified.

In C

→ size of the array is decided at compile time

## In C++

→ The size of the array can be decided at run time also.

## ACCESSING HEAP

```
int n;  
cin >> n;  
int A[n];
```

```
void main()  
{
```

```
int A[5];  
int * p;
```

$$p[0] = 5$$

C++ `p = new int[5];`

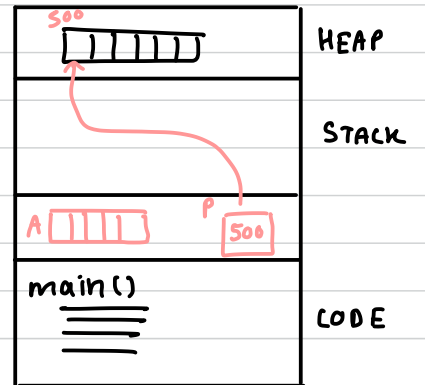
```
C
p = (int *) malloc(5 * sizeof(int));
<stdlib.h>
```

```
C++ delete [] p; } otherwise
```

c free(p);

## MEMORY LEAK

Shortage of memory.



## ONE WAY OF INCREASING SIZE OF ARRAY

```
int *p = new int[s];
```

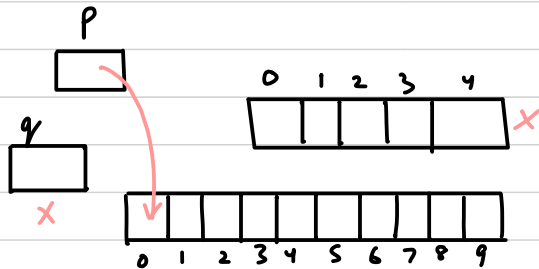
```
int *q = new int[10];
```

```
for (i = 0; i < 5; i++)
    q[i] = p[i]
```

```
delete [] p;
```

$$p = q;$$

```
q = NULL;
```



## 2D - ARRAY

① `int A[3][4] = { {1,2,3,4}, {2,4,6,8}, {3,5,7,9} };`

	0	1	2	3
0				
1			15	
2				

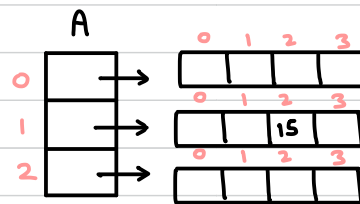
↓  
Array will be stored  
inside stack.

`A[1][2] = 15`

→ Array of pointers

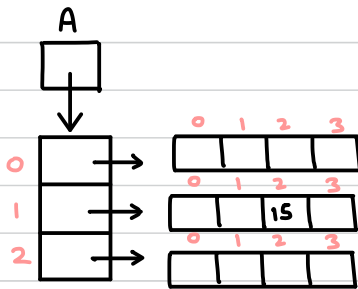
② `int *A[3];`

`A[0] = new int[4];` # Memory is  
`A[1] = new int[4];` created inside  
`A[2] = new int[4];` heap.



`A[1][2] = 15;` # We can access array  
of pointers in the same  
way.

③



`int **A;`  
`A = new int*[3];`  
`A[0] = new int[4];`  
`A[1] = new int[4];`  
`A[2] = new int[4];`

# All the memory is  
allocated in heap.

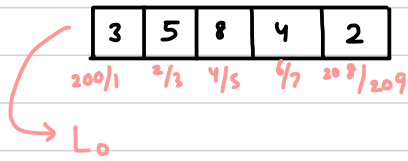
for column  
for (i=0; i<3; i++)  
{  
  for row  
  for (j=0; j<4; j++)  
  {  
    A[i][j] = \_\_;  
  }  
}

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

• Elements  
accessed in  
order

## HOW COMPILER GENERATES FORMULA FOR ADDRESS OF AN ARRAY

int A[5] = { 3, 5, 8, 4, 2 };



$$\text{Add}(A[3]) = 200 + 3 * 2 = 206$$

$$\text{Add}(A[3]) = L_0 + 3 * 2$$

$$\text{Add}(A[i]) = L_0 + i * w$$

Base Address  
Index  
Size of datatype.

FORMULA

No of operations = 2

$$\text{Add}(A[i]) = L_0 + (i-1) * w$$

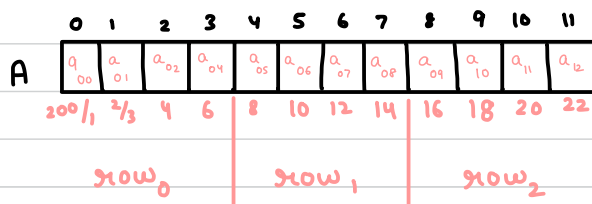
Formula, when indices are starting from one onwards

No of operations = 3

# More time consuming.

That's why C and C++ do not start array index with 1. Only for one extra operation, time taken by the program will increase and this will make the program slower.

## ROW MAJOR



int A[3][4];  
m n

$$\text{Add}(A[1][2]) = 200 + [1 * 4 + 2] * 2 = 212$$

$$\text{Add}(A[2][3]) = 200 + [2 * 4 + 3] * 2 = 222$$

$$\text{Add}(A[i][j]) = L_0 + [i * n + j] * w$$

4 operations

$$\text{Add}(A[i][j]) = L_0 + [(i-1) * n + (j-1)] * w$$

6 operations

## COLUMN MAJOR

	0	1	2	3	4	5	6	7	8	9	10	11
A	$a_{00}$	$a_{10}$	$a_{20}$	$a_{01}$	$a_{11}$	$a_{21}$	$a_{02}$	$a_{12}$	$a_{22}$	$a_{03}$	$a_{13}$	$a_{23}$
	col 0			col 1			col 2			col 3		

$$\text{ADD}(A[i][2]) = 200 + [2 * 3 + 1] * 2 = 214$$

$$\text{ADD}(A[i][3]) = 200 + [3 * 3 + 1] * 2 = 220$$

$$\text{ADD}(A[i][j]) = L_0 + (j * m + i) * w$$

## FORMULAS FOR nD ARRAYS

$$A[d_1][d_2][d_3][d_4]; \quad \text{4D ARRAY}$$

### Row Major

$$\text{ADD}(A[i_1][i_2][i_3][i_4]) = L_0 + [i_1 * d_2 * d_3 * d_4 + i_2 * d_3 * d_4 + i_3 * d_4 + i_4] * w$$

### Column Major

$$\text{ADD}(A[i_1][i_2][i_3][i_4]) = L_0 + [i_4 * d_3 * d_2 * d_1 + i_3 * d_2 * d_1 + i_2 * d_1 + i_1] * w$$

### Row MAJOR FOR nD

$$L_0 + \sum_{p=1}^n \left[ i_p * \prod_{q=p+1}^n d_q \right] * w$$

### COLUMN MAJOR for nD

$$L_0 + \sum_{p=n}^1 \left[ i_p * \prod_{q=p-1}^n d_q \right] * w$$

(self tried)  
(please check)

$A[d_1][d_2][d_3][d_4];$

4D ARRAY

HORNER'S RULE

Row Major

$$\text{Add}(A[i_1][i_2][i_3][i_4]) = L_0 + \underbrace{[i_1 * d_2 * d_3 * d_4]}_3 + \underbrace{[i_2 * d_3 * d_4]}_2 + \underbrace{[i_3 * d_4]}_1 + i_4 * w$$

$$4D \rightarrow 3 + 2 + 1$$

$$5D \rightarrow 4 + 3 + 2 + 1$$

$$nD \rightarrow n-1 + n-2 + n-3 \dots + 1 = \frac{n(n-1)}{2}$$

$$O(n^2)$$

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

$$i_4 + d_4 [i_3 + i_2 * d_3 + i_1 * d_2 * d_3]$$

$$i_4 + d_4 [i_3 + d_3 [i_2 + i_1 * d_2]]$$



$$O(n)$$

FORMULA FOR 3D ARRAYS

$\text{int } A[l][m][n];$

Row MAJOR

$$\text{Add}(A[i][j][k]) = L_0 + [i * m * n + j * n + k] * w$$

COLUMN MAJOR

$$\text{Add}(A[i][j][k]) = L_0 + [k * m * l + j * l + i] * w$$