M T W T F S S

Page No.:

Date:

YOUVA

Basic Programming Notes

- Ravinder Jetarwal

DATA STRUCTURES

- Ravinder Jetarwal

Live Class 8

Time and Space Complexity:

Time Complexity :- Amount of time taken by an algorithm to run as a function of length of input.

→ Not actual time but CPU operations

Need :- ① Resources are limited

② Measure algorithm to make efficient programs

③ Always asked by an interviewer after every question.

Space Complexity :- Amount of space taken by an algorithm as a function of length of input.

Eg ①
```
int a=1;
int b[5];
```
→ O(1) space complexity

Eg ②
```
int n;
cin>>n;
int *b= new int [n];
for (int i=0; i<n; i++){
        cout << b[i];
}
```
→ Case 1 :- n = 2
b[2]

case 2:- n = 2000
b [2000]

Thus, the amount of space required in Eg2 is depending on the size of input so it is O(n)

The amount of space required in Eg 1 is independant of i/p because size of array is fixed so O(1)

Live Class 9

Lets Learn Arrays

(1) An array is a datastructure used to store same type of data

Eg we need to store 10,000 integers, then we will not create 10,000 int variables instead, we will create an array of 1000 integers as follows

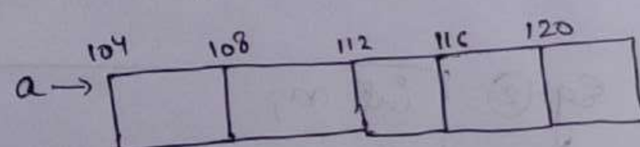
$\boxed{\text{int a[10,000];}}$

datatype = integer

space equivalent to 10,000 integers

(2) All the values inside an array are stored in a contigous memory order

Eg:- $\boxed{\text{int a[5];}}$ In memory,



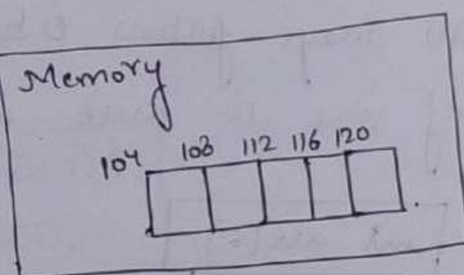
i.e 5 contigous integers of 4B each.
or an array of 5 integers.

(3) If enough equivalent space is not available for the entire array then it is the OS job to make it available. Unless ~~the~~ the contigous space is available, an array cannot be created.

Actual Memory storage of a variable.

Memory locations can only be identified using their addresses (mostly hexadecimal). A variable name can be used as an explanation, but on physical system a symbol table (also a datastructure) stores the variable name and the memory location/address of memory block assigned to that variable. This symbol table is used by the compiler to get the value of a variable or actuall the memory location that the variable is assigned to.

```
int arr[5];
```



When the compiler reads the line then it searches the symbol table for the variable arr and from the symbol table gets the address 104 and hence goes to the memory location 104 which is actually the address of the first element of the array

Symbol table

arr ⟶ 104.

NOTE:- The name of the array actually points to the first element of the ~~variad~~ array

⇒ An ~~oon~~ unitialised array contains garbage values in all the positions before we initialise the array.

① Initialisation of an array:-

① An unitialised array contains all garbage values

② Even if a single element is initialised, then the rest of the elements of the array are initialised to zero.

Eg:- $\boxed{\text{int arr[5] = \{10, 20\}}}$.

↳

10	20	0	0	0

$\boxed{\text{int arr[5];}}$

↳

gar	b	ag	e val	ue

all garbage values.

② Indexing in an Array:-

① All arrays follow 0 based indexing i.e for an array of size 10 there are elements from <u>0 to 9</u>

$\boxed{\text{int arr[6];}}$

↳ arr[0] arr[1] arr[2] arr[3] arr[4] arr[5]

i.e indexing starts from 0 and goes upto <u>n-1</u>

③ fill(starting address, ending address, element)

```
int main(){
    int arr[4];
    fill(arr, arr+4, 101);
    cout << arr[0]<<" "<< arr[1]<<" "<<arr[2]<<" "<<arr[3];
    return 0;
}
```

↳ Output:- 101 101 101 101

Thus fill(, ,) fills the entire elements of the array with the same number as specified in the fill function.

(4) Taking input in an array

For an array of n elements, we need to take an input for n elements i.e a repetitive task and hence we can use loops to take input in an array.

```
for (int i=0; i<n; i++){
        cin>>arr[i];
}
```

↳ i.e taking input for arr[0], arr[1], arr[2], arr[3] and arr[4] if we suppose n=5.

(5) Printing the array

```
for (int i=0; i<n; i++){
        cout << arr[i];
}
```

↳ prints the value of arr[0], arr[1], arr[2], arr[3], arr[4] if we suppose n=5

Bad Practise

```
int size;
cin >>size;
int arr [size];
```

→ Very bad practise } will study in memory allocation

(6) sizeof()

Consider an array arr → | 10 | 20 | 30 | 40 |

Now size of (arr) = 16B
size of (int) = 4B

$\therefore \frac{\text{sizeof(arr)}}{\text{size of (int)}} = 4$

i.e the array contains 4 elements/integers. (8

Consider another array,

arr →

10	20	.	.

sizeof (arr) = 16B

sizeof (int) = 4B

$\frac{\text{sizeof(arr)}}{\text{sizeof(int)}} = \frac{16}{4} = 4$

but the array contains only 2 elements

**

Thus, just by dividing sizeof(arr) by size of(datatype) we can never find the number of elements in the array, we will have to always maintain an explicit variable to keep a count on the ~~variables~~ elements in the array.

⑦ Formula of Array addressing

$\boxed{A[i] = \text{Value at } (\text{Base Address} + \text{size} * i)}$

Eg:- A[0] = value at (BA + 0) = Value at BA = 104 = 10
A[1] = value at (BA + 1) = Value at (BA + 4) = 108 = 20
A[2] = value at (BA + 2) = value at (BA + 8) = 112 = 30 (9

104	108	112	116
10	20	30	40
A[0]	A[1]	A[2]	A[3]

A[3] = value at (BA + 3) = Value at (BA + 12) = 116 = 40

104 + 12

(8) arr[i] and i[arr]

arr[i] = *(arr + i) ↑ value stored at address	i[arr] = *(i + arr) ↑ value stored at address

Eg:- arr[3] = *(arr + 3)

Now arr holds the address 104 and on adding 3, it becomes

$104 + 3 \times \underset{\uparrow}{4} = 104 + 12 = 116.$

(size of int)

This is called pointer arithmatic and hence on doing 104+3 we got 116 and not 107 because here we are dealing with address of an integer and not an integer itself.

∴ arr[3] = *(arr+3) = *(116)

i.e value at address 116 = 40

Similarly,

3[arr] = *(3 + arr)

= *(3 + BA)

= *(3×4 + 104)

= *(112 + 104)

= *(116)

i.e value stored at address 116 = 40.

Hence,
$\boxed{arr[i] = i[arr]}$

↳ One and the same thing.

(9) Functions with arrays

1. We can pass arrays to functions just like other variables.
2. Always remember to pass the size of the array along with the array in the function because as discussed earlier, we cannot find size of an array.

```
int main(){
    int arr[5]= {0};
    solve(arr, n);
    return 0;
}
```

n ← size of array

```
int solve(int arr [ ]){
```

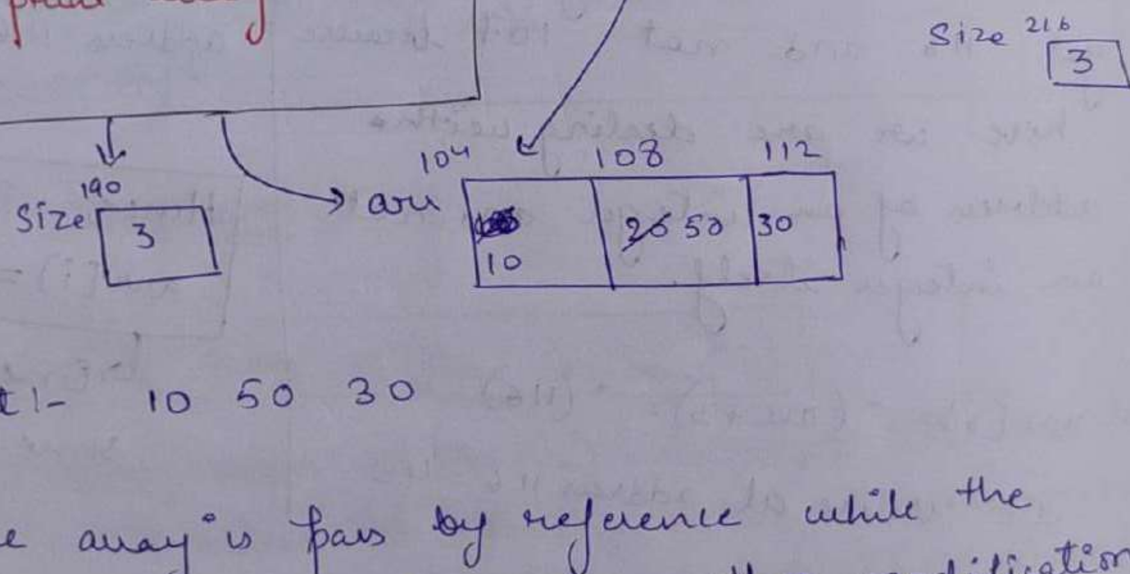
[ ] ← Size of array

Array is always pass by reference ← Note

Eg:-

```
int main(){
    int arr[]= {10,20,30};
    int size = 3;
    solve (arr, size);
    //print array
}
```

```
void solve(int arr[],
           int size){
    arr[1] = 50;
}
```



Output :- 10 50 30

Thus, the array is pass by reference while the variable size is pass by value. Thus, the modification on the array in the function solve() is carried out in the actual array.

Algorithm Linear Search

Ek ke baad ek saare elements check krke desired element nikal lenge.

Eg:- arr | 10 | 20 | 30 | 40 | 50 |

<u>Case 1</u> :- target = 40

arr[0] = 10 ≠ 40
arr[1] = 20 ≠ 40
arr[2] = 30 ≠ 40
arr[3] = 40 = 40   i.e linear search is successful and the element 40 is present at the index arr[3].

<u>Case 2</u> :- target = 53

arr[0] = 10 ≠ 53
arr[1] = 20 ≠ 53
arr[2] = 30 ≠ 53
arr[3] = 40 ≠ 53
arr[4] = 50 ≠ 53
arr[5] = not exist

i.e linear search is unsuccessful and the element 53 is not present in the array arr.

```
bool findTarget(int arr[], int size, int target){
    for(int i=0; i<size; i++){
        if(arr[i]== target){
            return true;
        }
    }
    return false;
}
```

Find max element in an array

Range of int : $\underset{\text{INT\_MIN}}{\underset{\downarrow}{-2^{31}}}$ to $\underset{\text{INT\_MAX}}{\underset{\downarrow}{2^{31}-1}}$

To find max :- compare with INT_MIN
To find min :- compare with INT_MAX.

```
int findMax(int arr[], int size){
        int maxAns = INT_MIN;
        for (int i=0; i<size; i++){
                maxAns = max(maxAns, arr[i]);
        }
        return maxAns;
}
```

consider arr -> | 13 | 42 | 55 | 76 | 88 |

maxAns = -2^{31}

i=0 -> maxAns = 13

i=1 -> maxAns = 42

i=2 -> maxAns = 55

i=3 -> maxAns = 76

i=4 -> maxAns = 88

i=5 ≮ 5 so loop terminates and finally maxAns = 88

maxAns: $\cancel{-2^{31}}$ ~~13~~ ~~42~~ ~~55~~ ~~76~~ 88

Two Pointer Technique

i/p :-

10	20	30	40	50	60

o/p :- 10 60 20 50 30 40

Process :- Case ①

10	20	30	40	50	60

i i i j i j j j

Print a[i] and do i++
Print a[j] and do j--
Continue doing until i<j

when i>j means entire array is traversed.

10 60 20 50 30 40 and now i>j

①

10	20	30	40	50	60

→ 10 60

i j

②

10	20	30	40	50	60

→ 10 60 20 50

i j

③

10	20	30	40	50	60

→ 10 60 20 50 30 40

i j

④

10	20	30	40	50	60

→ i>j so nomore printing

j i

Output :- 10 60 20 50 30 40

Case ②

10	20	30	40	50	60	70

①

10	20	30	40	50	60	70

10 70

i j

(2)

10	20	30	40	50	60	70

i j

10 70 20 60

(3)

10	20	30	40	50	60	70

i j

10 70 20 60 30 50

(4)

10	20	30	40	50	60	70

i j

→ i.e we will also have to print for the condⁿ i==j to get the final output

Output:- 10 70 20 60 30 50 40

```
void extremePrint(int arr[], int n){
    int i=0;
    int j=n-1;
    while(i<=j){
        if(i==j){
            cout<<arr[i]<<" ";
            break;
        }
        else{
            cout<<arr[i]<<" ";
            i++;
            cout<<arr[j]<<" ";
            j--;
        }
    }
}
```

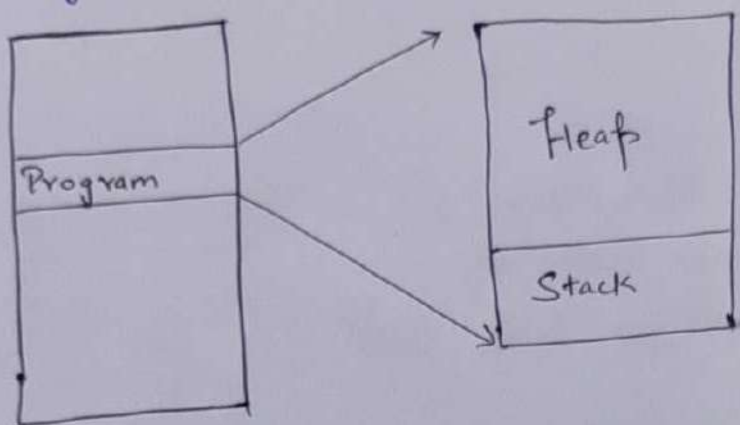
→ Application of 2 pointer technique

Live class 10

Doubt class with Lakshay Bhaiya.

Dynamic Memory Allocation

Consider a RAM which has a part of it allocated for a program as shown below



① → RAM itself is a part of the memory of the system.

② The space allocated for the program is further divided into a stack and a heap as shown

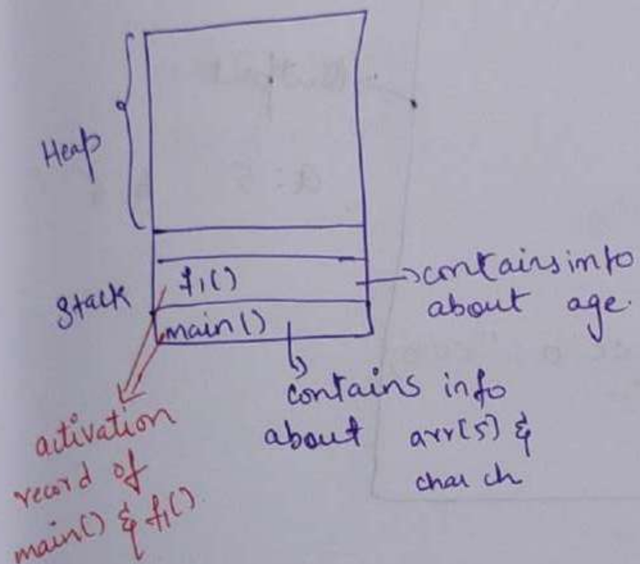
③ The heap is larger than the stack by default.

④ The stack contains the activation records of the functions i.e the function call stack. The activation record of any function contains information about all the variables of that function.

⑤ The heap is by default is larger than the stack and hence any element that requires more memory space is allocated to the heap and not to the stack. Memory allocation to the heap is called dynamic memory allocation.

Eg:-
```
int main(){
    int arr[5];
    char ch;
    f1();
}

void f1(){
    int age;
}
```



Now any variable with a large amount of memory requirement such as

```
int arr[100000000];
```

contains 10^8 integers.

1 integer = 4B.

10^8 integers = 4×10^8 B

$= 400 \times 10^6$ B

$= 400$ MB

i.e 10^8 integer's array would require 400MB of space and such a allocation will only be done in the heap by the programmer.

Hence, statements like

```
int size;
cin>> size;
int arr[size];
```

are considered as bad practise because depending on the size either the array arr[size] will be allocated space in the stack or in the heap but now the compiler doesn't know because it is user dependant. {Details will be discussed in the dynamic memory allocation chapter}

Different Methods of Swapping two numbers

① swap function :- C++ has an inbuilt swap function that can swap two numbers as shown below

```
int main(){
    int a = 6, b=5;
    swap(a,b);
    cout<<"a: "<<a<<" b: "<<b;
    return0;
}
```

→ Output

a: 5   b: 6

(2) Temp Variable Method

```
int main(){
        int temp=[illegible];
        int a=6, b=5;
         a=b;
         b=temp;
         cout<<"a: "<<a<<"b: "<<b;
         return 0;
}
```

→ Output

a: 5   b: 6

(3) Arithematic Method (interview Question)

```
int main(){
        int a=6, b=5;
        a= a+b; || a=11,, b=5
        b= a-b; || a=11, b=6
        a= a-b; || a=5, b=6
        cout<<"a: "<<a<<"b: "<<b;
        return 0;
}
```

→ output

a:5   b:6

④ Bitwise XOR method. (interview question method).

```
int main(){
    int a=6, b=5;
 i.  a= a^b; // a= 6^5 = 3
 ii. b= a^b; // b= 3^5 = 6
 iii. a = a^b; // a = 3^6 = 5
    cout << "a: " << a << " b: " << b;
    return 0;
```

→ Output

a: 5 b: 6

$a = 6 = (110)_2$

$b = 5 = (101)_2$

(i) a = a^b

a^b ⇒
110 (6)
101 (5)
011 = (3) ← a

(ii) b = a^b

a^b ⇒
011 (3)
101 (5)
110 = (6) ← b

(iii) a = a^b

a^b ⇒
011 (3)
110 (6)
101 = (5) ← a

① a=10, b=11

② a=3, b=17

③ a=77, b=13

① a=a^b
1010
1011
0001 = 1 (a)

b = a^b
0001
1011
1010 = 10 (b)

a = a^b
0001
1010
1011 = 11 (a)

i.e a = 11 & b = 10.

② a=a^b
00011
10001
10010 = 18 (a)

b = a^b =
10010
10001
00011 = 3(b)

a = a^b
10010
00011
10001 = 17(a)

a=17 b=3

③ a=a^b
1001101
0001101
1000000 = 64(a)

b = a^b
1000000
0001101
1001101
↳ 77(b)

a = a^b
1000000
1001101
0001101
↳ 13(a)

a=13 & b=77

Reverse an Array

We will use the swap() method along with the two pointer technique to swap the first and last element and then increment/decrement the pointers.

```
#include <iostream>
using namespace std;
int main(){
    int arr[]={1,2,3,4,5,6,7};
    int size= sizeof(arr)/sizeof(int);
    int low=0, high= size-1;
    cout<<"Before Reversing: ";
    for(int i=0; i<size-1; i++){
            cout<<arr[i]<<" ";
    }
    while(low<=high){
        swap(arr[low], arr[high]);
        low++;
        high--;
    }
    cout<<endl;
    cout<<"After Reversing: ";
    for(int i=0, i<size-1; i++)
            cout<<arr[i]<<" ";
    }
    return 0;
}
```

Output:-
Before Reversing: 1 2 3 4 5 6 7
After Reversing: 7 6 5 4 3 2 1

We could also use the inbuilt reverse

Live Class 11

Array Class 2

① Leetcode Question:- Single Number (Important for interview)

Already read that to find the unique number in an array, we can XOR all the elements in the array and this will cancel out all the duplicate elements. The only element that is left is the unique element of the array.

Eg:- arr[5] = {2, 4, 1, 4, 1}.

$2 \wedge 4 \wedge 1 \wedge 4 \wedge 1 = 2 \Rightarrow$ unique element

$\boxed{\text{NOTE:- } a \wedge 0 = a}$ →

$$\begin{array}{r} 110 \\ \wedge\ 000 \\ \hline 110 \end{array} \qquad \begin{array}{r} 111 \\ \wedge\ 000 \\ \hline 111 \end{array}$$

Homework :- ① Dutch National Problem or sort 0, 1 and 2 array

0	0	2	1	1	2	2	0	0	0

② Find 2's complement of the array

0	1	1	1	0	0	1	1	0

③ Alternate Solution to Single Number Question

(2) ~~Count~~ Sort 0's and 1's in an Array

(1) Approach :- Counting based

0	1	0	0	0	1	1

$\rightarrow O(n)$

(1) Count number of zeroes = 4

(2) Number of ones = size - 4 = 7 - 4 = 3

(3) Insert 4 zeroes first

(4) Then insert 3 ones to get

0	0	0	0	1	1	1

However, we need not maintain "numberOfOnes" variable because we can get it from size - number of zeroes.

2 Approach :- fill method

(1) count number of zeroes

(2) fill (arr, arr + numberOfZeroes, 0) $\rightarrow O(n)$

(3) fill (arr + numberOfZeroes, arr + size, 1)

(3) Approach :- sort() method

(1) sort (arr, arr + size)

↳ sorting by compiler takes O(nlogn) because compiler uses merge sort which has O(nlogn).

$\boxed{O(n) \text{ is better than } O(n\log n).}$

(3) Two Sum

i/p :- | 10 | 5 | 20 | 15 | 30 |
|---|---|---|---|---|

o/p :- Find ~~two~~ pairs of numbers whose sum is equal to 35. If yes, print the first pair found.

Eg :- 20+15 = 35 is a pair that satisfies the problem statement.

All the pairs are :- (10,10) (10,5) (10,20) (10,15) (10,30)
sum → 20 15 30 25 40
(5,10) (5,5) (5,20) (5,15) (5,30)
sum → 15 10 25 20 35
(20,10) (20,5) (20,20) (20,15) (20,30)
sum → 30 25 40 35 50
(15,10) (15,5) (15,20) (15,15) (15,30)
sum → 25 20 35 30 45
(30,10) (30,5) (30,20) (30,15) (30,30)
sum → 40 35 50 45 60

(learn all the patterns of finding pairs).

This is exactly how the compiler will form pairs of two elements from the array. Thus, the first pair should be (5,30) which is our desired output.

```
#include<iostream>
using namespace std;
pair<int, int> checkTwoSum(int arr[], int size, int target){
    pair <int, int> ans;
    ans.first = ans.second = -1;
    for(int i=0; i<size; i++){
        for(int j=0; j<size; j++){
            if(arr[i] + arr[j] == target){
                ans.first = arr[i];
                ans.second = arr[j];
                return ans;
            }
        }
    }
    return ans;
}
```

```
int main(){
    int arr[] = {10, 5, 20, 15, 30};
    int size = sizeof(arr)/sizeof(int);
    pair<int, int> ans = checkTwoSum(arr, size, 35);
    if (ans.first != -1){
        cout<<"Pair found"<<"("<<ans.first<<","<<ans.second
                    <<")"<<endl;
    }
}
```

Output:- Pair found (5,30).

this is what was expected.

NOTE:- If we wanted to print all the pairs then we can directly print in the function without returning anything

```
void printAllPairs(int arr[], int size, int target){
    for (int i=0; i<size; i++){
        for (int j=0; j<size; j++){
            if(arr[i] + arr[j] == target){
                cout<<"("<<arr[i]<<","<<arr[j]<<")";
            }
        }
    }
}
```

Output:- (5,30) (20,15) (15,20) (30,5)

as seen in the diagram earlier

(4) Print all triplets

10	20	30	40

```
for (int i=0; i<n; i++){
    for (int j=0; j<n; j++){
        for (int k=0; k<n; k++){
            cout<<arr[i]<<","<<arr[j]<<","<<arr[k]<<endl;
        }
    }
}
```

↳ prints all the 64 triplets in $O(n^3)$ time &
$O(1)$ space.

10	20	30	40

i=0 j=0 k=0 10,10,10 k=1 10,10,20 k=2 10,10,30 k=3 10,10,40	i=1 j=0 k=0 k=1 k=2 k=3	i=2 j=0 k=0 k=1 k=2 k=3	i=3 j=0 k=0 k=1 k=2 k=3
i=0 j=1 k=0 10,20,10 k=1 10,20,20 k=2 10,20,30 k=3 10,20,40	i=1 j=1 k=0 k=1 k=2 k=3	i=2 j=1 k=0 k=1 k=2 k=3	i=3 j=1 k=0 k=1 k=2 k=3
i=0 j=2 k=0 10,30,10 k=1 10,30,20 k=2 10,30,30 k=3 10,30,40	i=1 j=2 k=0 k=1 k=2 k=3	i=2 j=2 k=0 k=1 k=2 k=3	i=3 j=2 k=0 k=1 k=2 k=3
i=0 j=3 k=0 10,40,10 k=1 10,40,20 k=2 10,40,30 k=3 10,40,40	i=1 j=3 k=0 k=1 k=2 k=3	i=2 j=3 k=0 k=1 k=2 k=3	i=3 j=3 k=0 k=1 k=2 k=3

↳ all 64 combinations

If we want only unique entries i.e eliminate entries like 10,10,10 then the following loop can be used.

```
for(int i=0; i<n; i++){
    for(int j=i+1; j<n; j++){
        for(int k=j+1; k<n; k++){
            ≡
        }
    }
}
```

↳ prints triplets but eliminates duplicates such as 10, 10, 10

⑤ Rotate an Array:

i/p :-

10	20	30	40	50	60

i.e jump each element twice forward in a cycle.

Rotate the array by 2.

o/p :-

50	60	10	20	30	40

Approach:-

50	60	10	20	30	40

① Store last 2 elements in a temp array because we are shifting by 2.

② Shift the array by 2 elements starting from $(n-2)^{th}$ element

③ copy temp array in the begining of original array

①

10	20	30	40	50	60

temp

50	60

②

10	20	~~30~~	~~40~~	~~50~~	~~60~~
		10	20	30	40

③

50	60	10	20	30	40

↳ desired array.

Live Class 12

Vector STL in C++

1. The Standard Template Library (STL) provides a collection of templates classes and functions that offer common datastructures and algorithms to make programming more efficient and convenient.

2. A vector in C++ is a dynamic array that can grow or shrink in size, making it a versatile and efficient data structure for storing and manipulating sequences of elements.

Static array

```
int arr[5];
```

Dynamic array

```
cin>>n;
int *arr = new int [n];
```

↳ Dynamic memory allocation.

```
int n;
cin>>n;
int *arr = new int[n];
for(int i=0; i<n; i++){
        int data;
        cin >> data;
        arr[i] = data;
}
for (int i=0; i<10; i++){
        arr[n+i]= 80;
}
```

→ Now suppose the user inputs n=5.

In this case, an array of size 5 is created. but when the second loop starts execution, it produces malloc() error. Thus, this is a problem with array datastructure

Solution :- ① We need something that can grow or shrink dynamically.

② Vector exactly serves this purpose and does not require its size to be initialised beforehand because it can shrink and grow dynamically.

③ Internal Working :-

There is no space allocated in the memory when a vector is first declared.

Memory starts getting allocated when an (integer is) element is pushed in the array (vector).

When the vector is full its capacity is doubled (i.e dynamically grown). However, the size of the vector is equal to the amount of space required by the elements of the vector.

	Capacity	Size	Vector
vector<int> v;			
v.push_back(1);	1	1	[1]
v.push_back(2);	2	2	[1|2]
v.push_back(3);	4	3	[1|2|3|]
	4	4	[1|2|3|4]
v.push_back(4);	4		
v.push_back(5);	8	5	[1|2|3|4|5| | |]

However, our main concern is size of the vector and not the capacity.

Deletion from a vector →

v.pop_back();
v.pop_back();

[1|2|3|4]

i.e pop is always from the end of array.

[1|2|3]

⇒ Thus, in case of vectors, there is no concern of the size beforehand as in the case of arrays. Thus, the problem is solved.

⇒ Now the code will work if we use vectors instead of an array

Clear the vector :-

```
V.clear();
```

clears the vector v i.e empties the vector.

Declaration & initialisation

i)
```
vector<int> arr; //default with no data, 0 size
```

ii).
```
vector<int> arr(5, -1); //vector of size = 5 with all -1 entries
```

iii)
```
vector <int> arr{1, 2, 3, 4, 5};
```

↳ does not work for all compilers because was introduced in 2011

Copy a vector

```
vector<int> arr = {1,2,3,4,5};
vector<int> arr2 = (arr);
```

First and last Element

v[0]; or v.front(); gives the first element

v[v.size()-1] or v.back(); gives the last element

Print a vector

```
for (auto it : v) {
        cout << it << " ";
}
```

→ Works like a foreach loop and prints each elements of the vector v.

Live class 13

Array class 3

2 Dimensional Array

1D array :-

10	20	30	40

2D array :-
(logical view).

	col0	col1	col2	col3
row 0 →	(0,0)	(0,1)	(0,2)	(0,3)
row 1 →	(1,0)	(1,1)	(1,2)	(1,3)
row 2 →	(2,0)	(2,1)	(2,2)	(2,3)
row 3 →	(3,0)	(3,1)	(3,2)	(3,3)

i.e each element has the index as $\boxed{\text{(row, column)}}$
↳ indexing of a 2D array

This is just a logical view of the 2D array but actually, the 2D array is also stored in a linear manner in the memory just like a 1D array.

Consider a 2D array arr[4][4]

arr →

	col0	col1	col2	col3
row 0	0,0	0,1	0,2	0,3
row 1	1,0	1,1	1,2	1,3
row 2	2,0	2,1	2,2	2,3
row 3	3,0	3,1	3,2	3,3

The inmemory representation of the 2D array will be as

arr[4][4] →

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
(0,0)	(0,1)	(0,2)	(0,3)	(1,0)	(1,1)	(1,2)	(1,3)	(2,0)	(2,1)	(2,2)	(2,3)	(3,0)	(3,1)	(3,2)	(3,3)

Formula:- $\boxed{c*i+j}$ c = total no. of columns.

Now the element arr[2][3] will be stored in the memory on the index given by the above formula.

$i=2 \quad j=3$ then
$$c*i+j$$
$$= 4*2+3$$
$$= 8+3$$
$$= \underline{\underline{11}}$$

i.e arr[2][3] will be stored in index 11 which can also be verified through the diagram.

Creation:-

int arr[4][3]; →

	col0	col1	col2
row 0			
row 1			
row 2			
row 3			

NOTE:- Only in case of static array, by default the values are set to garbage values while in case of dynamic arrays, default value is 0.

Initialisation:-

```
int arr[4][3] = {{10,20,30}
                 {11, 12, 13}
                 {15, 16, 17}
                 {20, 21, 22}
}
```

Just like 1D array, even if 1 element is iniatilised and not all are initialised then rest elements = 0.

Accessing the elements

To access elements we use arr[i][j] where i represents the row and j represents the column.

```
int main(){
    int arr[3][2] = {{10,20},
                     {30,40},
                     {50,60}
                    };
    int rowSize = 3;
    int colSize = 2;
    for (int row = 0; row<rowSize; row++) {
        for (int col=0; col<colSize; col++) {
            cout << arr[row][col]<<" ";
        }
        cout <<endl;
    }
}
```

Row wise → Traversing an array.

Eg for initialising & accessing a 2D array.

↳ Output :-

```
10 20
30 40
50 60
```

Column wise traversal

```
arr[3][3] =
{{10, 20, 100},
 {30, 40, 200},
 {50, 60, 300}
};
```

```
for (int row = 0 ; row < rowSize; row ++) {
    for (int col = 0, col < colSize; col++) {
        cout << arr[col][row]<<" ";
    }
    cout <<endl;
}
```

→ (valid only for square matrix)

Square matrix is a matrix where rows = cols

↳ Output :-

```
10  30  50
20  40  60
100 200 300
```

Diagonal traversal of a 2D array

```
for (int row=0; row<rowSize; row++){
    for (int col=0; col<colSize; col++){
        if (row == col){
            cout << arr[row][col];
        }
    }
}
```

Consider the arr[3][3] = { {10,20,100},
{30,40,200},
{50, 60, 300}
};

the output => 10, 40 300

Optimised code :-

```
for (int i=0; i<rowSize; i++){
    cout << arr[i][i] << " ";
}
```

→ Output :- 10 40 300

NC General column wise traversal

arr [3][2] = { {10,20},
{30,40},
{50, 60}
};

```
for (int col=0; col<colSize; col++){
    for (int row=0; row<rowSize; row++){
        cout << arr[row][col];
    }
    cout << endl;
}
```

→ Output :- 10 30 50
20 40 60

Secondary diagonal Traversal of a 2D array:-

```
arr[3][3] = { {10, 20, 100},
              {30, 40, 200},
              {50, 60, 300}
            };
```

```
for(int row=0; row<rowSize; row++){
    for(int col=0; col<colSize; col++){
        if(row + col == rowSize-1){
            cout << arr[row][col] << " ";
        }
    }
    cout << endl;
}
```

Output:-
100
40
60

Taking input of a 2D array:-

```
int main(){
    int arr[3][2];
    for(int i=0; i<3; i++){
        for(int j=0; j<2; j++){
            cin>> arr[i][j];
        }
    }
    cout <<"Array is : "<<endl;
    for(int i=0; i<3; i++){
        for(int j=0; j<2; j++){
            cout << arr[i][j]<<" ";
        }
        cout <<endl;
    }
}
```

→ taking i/p row wise.

↳ Output 10 20 30 40 50 60 (user input)
Array is:
10 20
30 40
50 60

Whenever we pass ~~array~~ as a 2D array to a function, we will have to pass its rowSize as well as colSize. The compiler converts it into a 1D array as seen earlier by the formula $(c*i+j)$. Thus, we need to pass the column as well while passing a 2D array to a function.

2D array and functions → (array is always pass by reference but vector may be pass by reference or pass by value).

Searching in a 2D array

Approach :- Search each element one by one and return true if the target element is found just like in linear search.

```
bool SearchElement(int arr[][4], int row, int col, int target){
    for(int i=0; i<row; i++){
        for(int j=0; j<col; j++){
            if(arr[i][j]== target) return true;
        }
    }
    return false;
}
int main(){
    int arr[3][4]= {{10,20,30,40},
                    {21,22,23,24},
                    {32,12,34,36}
                   };
    int rowSize=3, colSize=4, target=36;
    bool ans= searchElement(arr, rowSize, colSize, target);
    cout<<ans;
    return 0;
```

Output:- 1

Vector for 2D array

`int arr[4][3];` <=> `vector<vector<int>> arr (4, vector<int>(3,0));`

↳ 2D array

↳ 2D vector

arr → name of vector

4 → #(rows)

3 → #(cols)

0 → initialised value

arr →

0	0	0
0	0	0
0	0	0
0	0	0

```
rowSize = arr.size();
colSize = arr[0].size();
```

Declaration & Initialisation

```
int main (){
    vector <vector<int>> arr(4, vector<int>(3,23));
    int rowSize = arr.size();
    int colSize = arr[0]. size();
    for (int i=0; i<rowSize; i++) {
        for (int j=0; j<colSize; j++){
                cout << arr[i][j]<<" ";
        }
        cout<<endl;
    }
}
```

↳ Output :-

```
23 23 23
23 23 23
23 23 23
23 23 23
```

Find minimum element of 2D array

Approach:- We will again use linear search to find the smallest element

```
int findMinimum(int arr[][4], int rowSize, int colSize){
    int minValue= INT_MAX;
    for(int i=0; i<rowsize; i++){
        for(int j=0; j<colsize; j++){
            minValue= min(arr[i][j], minValue);
        }
    }
    return minValue;
}
```

→ $O(n^2)$ or $O(\text{rowSize} \times \text{colSize})$

& Space complexity = $O(1)$.

Row wise Sum

We will traverse to each row and for each row we will add up all the column values

i.e row 0 →
col0 + col1 + col2 → print sum

row 1 →
col0 + col1 + col2 → print sum

row 2 →
col0 + col1 + col2 → print sum

```
void printRowSum(int arr[][4], int rowSize, int colSize){
    for(int i=0; i<rowSize; i++){
        int sum=0;
        for(int j=0; j<colSize; j++){
            sum= sum+arr[i][j];
        }
        cout << sum<<endl;
    }
}
```

Column Wise Sum

Approach:- col 0 ↓
row0 + row1 + row2 → print

col 1 ↓
row0 + row1 + row2 → print

col 2 ↓
row0 + row1 + row2 → print.

```
void printColSum(int arr[][4], int rowSize, int colSize){
    for(int col=0; col< colSize; col++){
        int sum = 0;
        for(int row=0; row < rowSize; row++){
            sum= sum+ arr[row][col];
        }
        cout<< sum<<endl;
    }
}
```

Diagonal Sum

Definitely diagonal sum is only possible for only a square matrix.

Approach:-

```
for(row: 0->n)
        sum=sum + arr[i][i]
```

```
void printDiagonalSum(int arr[][4], int rowSize, int colSize){
        int sum=0;
      for(int i=0; i<rowSize; i++){
            for(int j=0; j<colSize; j++){
                  if(i==j){
                        sum=sum+arr[i][j];
                  }
            }
      }
}
```

→ $O(n^2)$

R Optimised solution ↓

```
for(int i=0; i<rowSize; i++){
        sum=sum + arr[i][i];
}
cout<<sum;
```

→ $O(n)$

Transpose of a Matrix

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 10 & 11 & 12 \\ 1 & 20 & 21 & 22 \\ 2 & 30 & 31 & 32 \end{array} \xrightarrow{\text{transpose}} \begin{array}{|c|c|c|} \hline 10 & 20 & 30 \\ \hline 11 & 21 & 31 \\ \hline 12 & 22 & 32 \\ \hline \end{array}$$

Approach:- ① We are basically changing rows into columns and columns into rows.

OR

we are changing i to j and j to i

② We can create an array and do

$\boxed{\text{newarr}[i][j] = \text{oldarray}[j][i]}$

for all the rows & columns.

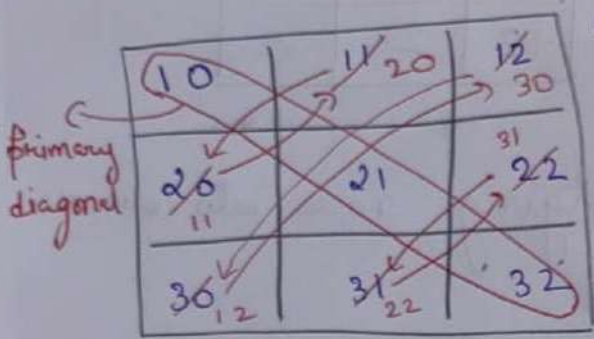
```
void transpose(int arr[3][3], int rowSize, int colSize){
    int ans[100][100]={0};
    for(int i=0; i<rowSize; i++){
        for(int j=0; j<colSize; j++){
            ans[i][j]= arr[j][i];
        }
    }
}
```

↳ uses additional array.

Approach 2 :- swap arr[i][j] with arr[j][i] for only upper matrix. (otherwise 2 times swap occurs and same matrix is obtained).

~~Str~~ No need to swap primary diagonal or swap them with themself to form upper triangle.

i.e loop works only for the upper triangle.




```
for(int i=0; i<rowSize; i++){
    for(int j=i; j<colSize; j++){
        swap(arr[i][j], arr[j][i]);
    }
}
```

↳ without use of external array.