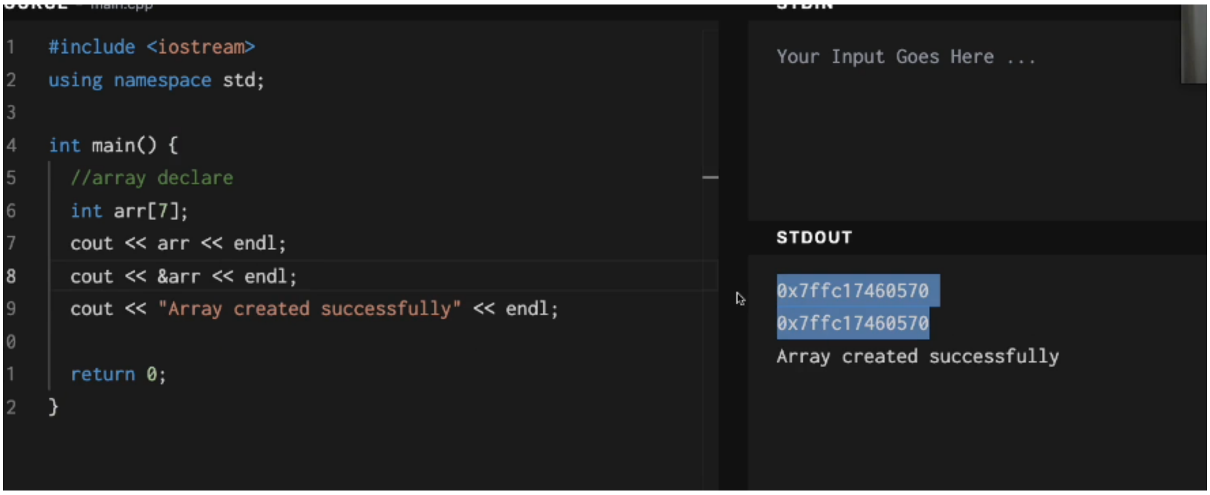
DSA by LOVE BABBAR

1.ARRAY

Simple we will create an array int a[2];

Int a[7];

Example of a base example



```
#include <iostream>
using namespace std;

int main() {
  //array declare
  int arr[7];
  cout << arr << endl;
  cout << &arr << endl;
  cout << "Array created successfully" << endl;

  return 0;
}
```

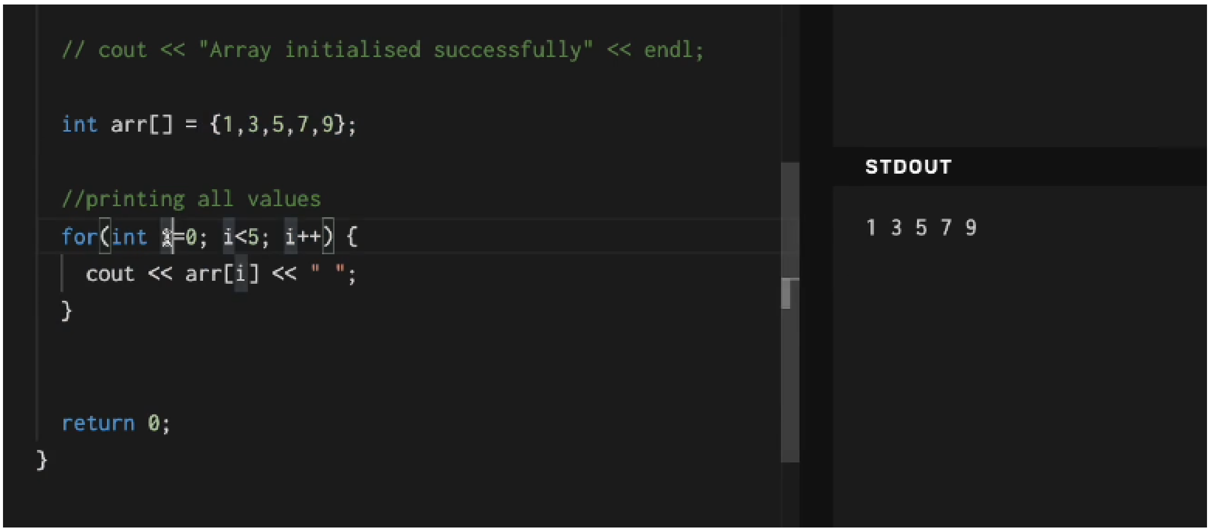
```
0x7ffc17460570
0x7ffc17460570
Array created successfully
```

Intialization

Static Array:

of an array int a[4]={1,2,3,4}

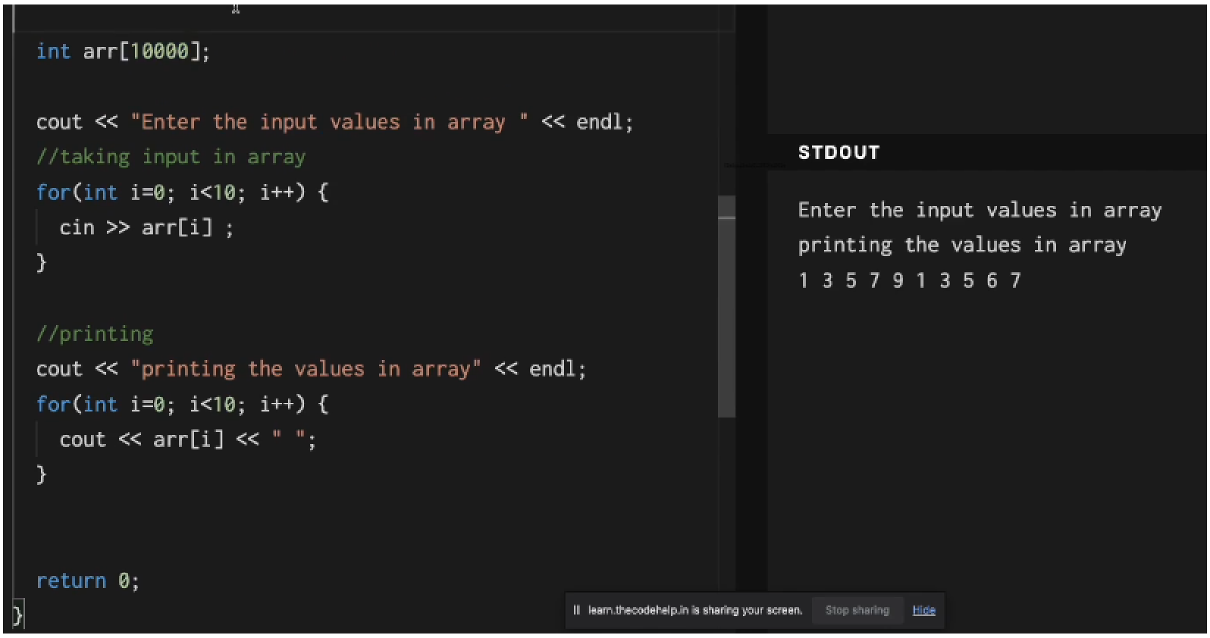
dynamic Array : int n; cin >> n ; , int a[n] === this is bad practice we do not use this



```
  // cout << "Array initialised successfully" << endl;

  int arr[] = {1,3,5,7,9};

  //printing all values
  for(int i=0; i<5; i++) {
    cout << arr[i] << " ";
  }


  return 0;
}
```

```
1 3 5 7 9
```

Taking as input fro the user in the array



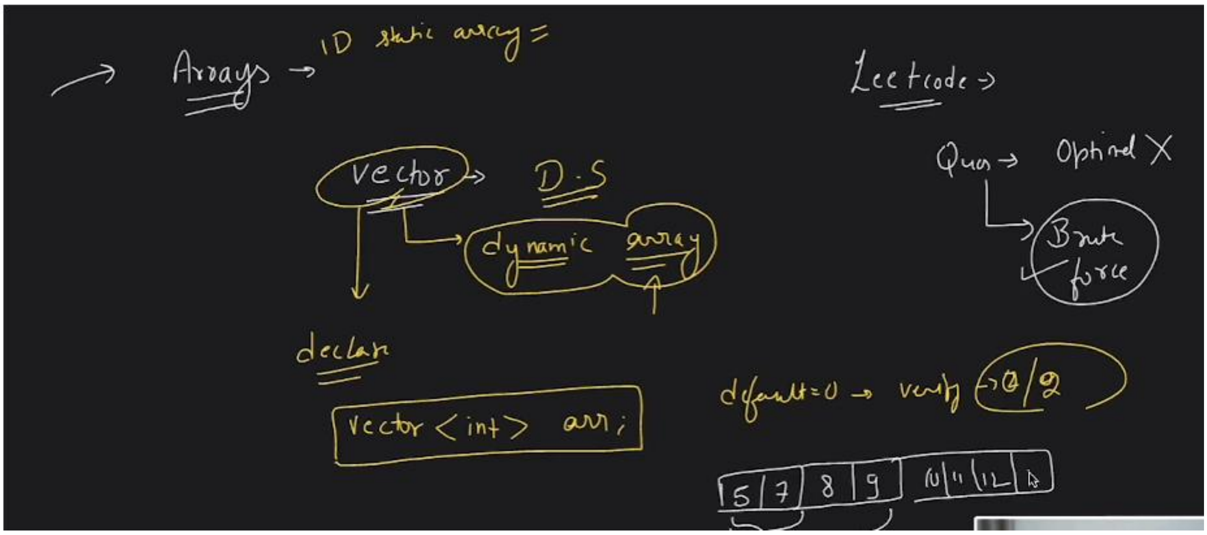
```
int arr[10000];

cout << "Enter the input values in array " << endl;
//taking input in array
for(int i=0; i<10; i++) {
  cin >> arr[i] ;
}

//printing
cout << "printing the values in array" << endl;
for(int i=0; i<10; i++) {
  cout << arr[i] << " ";
}


return 0;
}
```

STDOUT

```
Enter the input values in array
printing the values in array
1 3 5 7 9 1 3 5 6 7
```

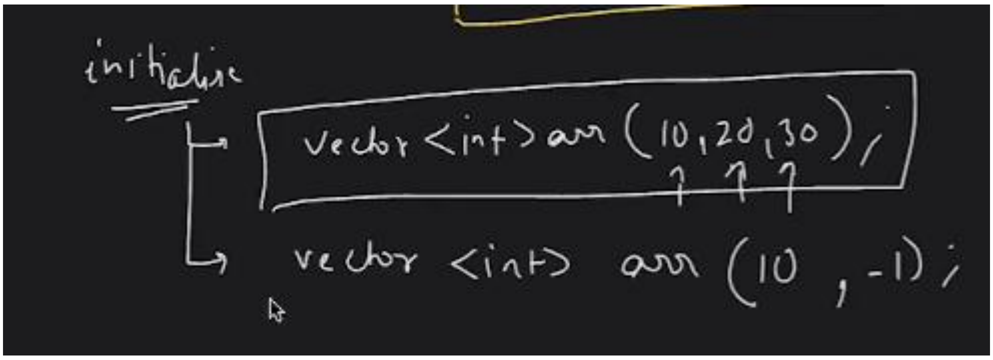
VECTOR

Vector double it is size as according tot the inputs as we have put on element that it double of one is 2 and if we put 2 elements than it doble to 4 this way it is working like this it is working oke



```
vector<int> arr;
```

Initialization of a vector



```
vector<int>arr(10,20,30);
vector <int> arr(10 , -1);
```

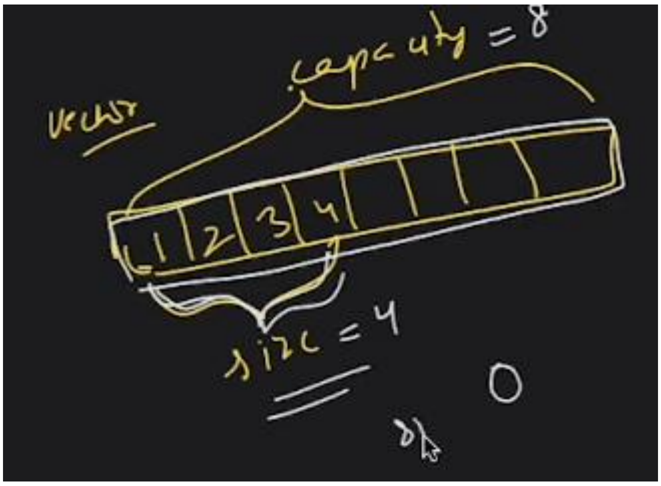
Dynamic vector :

Int n , cin>>n, vector<int>arr(n);

To push an element :: arr.push_back(5);

To pop back an elemenr: arr.pop_back(); == last lement will be remove from this

Size and capacity in vector



Vector<int>arr(10) ;

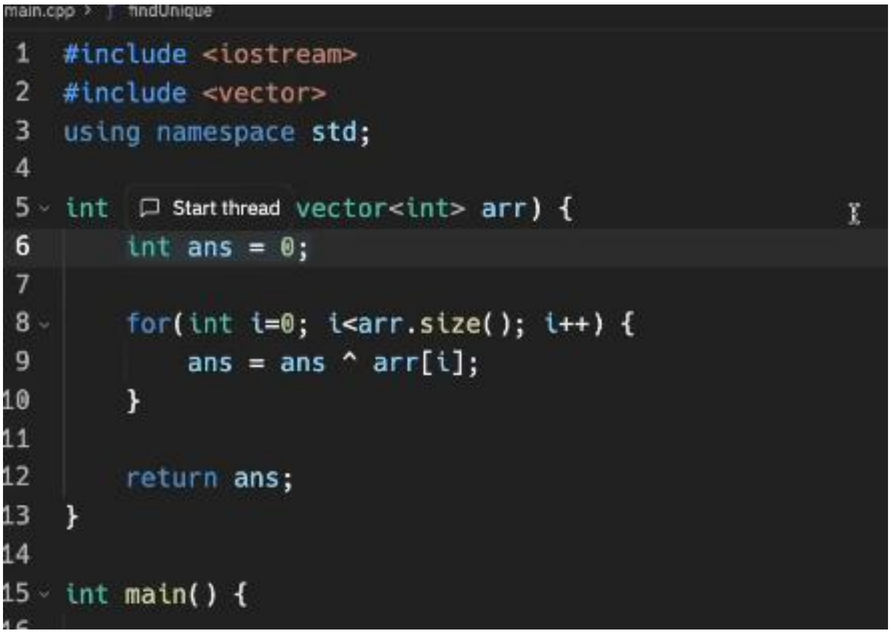
So we have give this size f we pint this all elements as we have not initialize anything so the 0 value will be there in the elements

To intailize with the element
vector<int> arr{1,2,3,4,5,6}

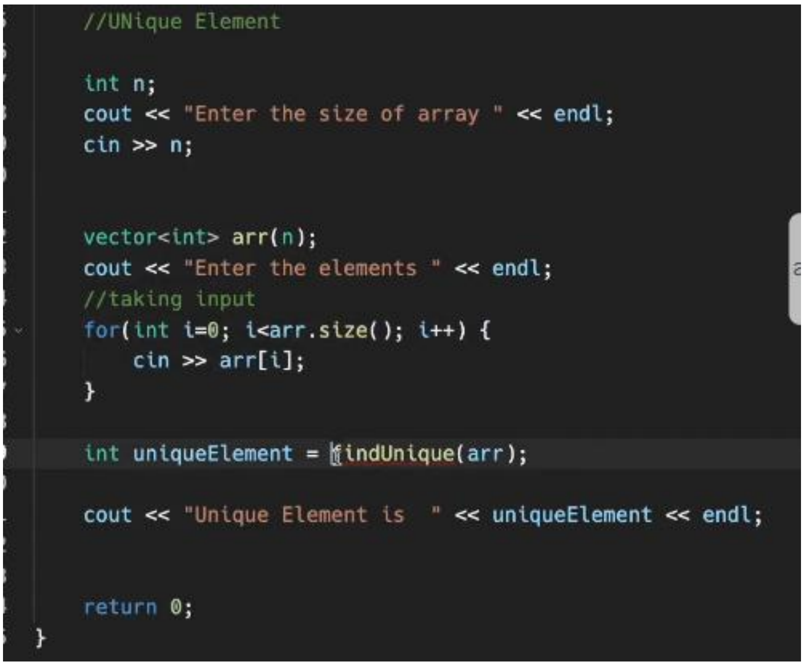
Find unique element throufh a vector

Wehen we xor o with any elemnt that element will be answer so we have takn help of xor like 0xor 1 = 1 or 0 xor 0 =0

So due to this in tsratunf we will initialize the a element =0



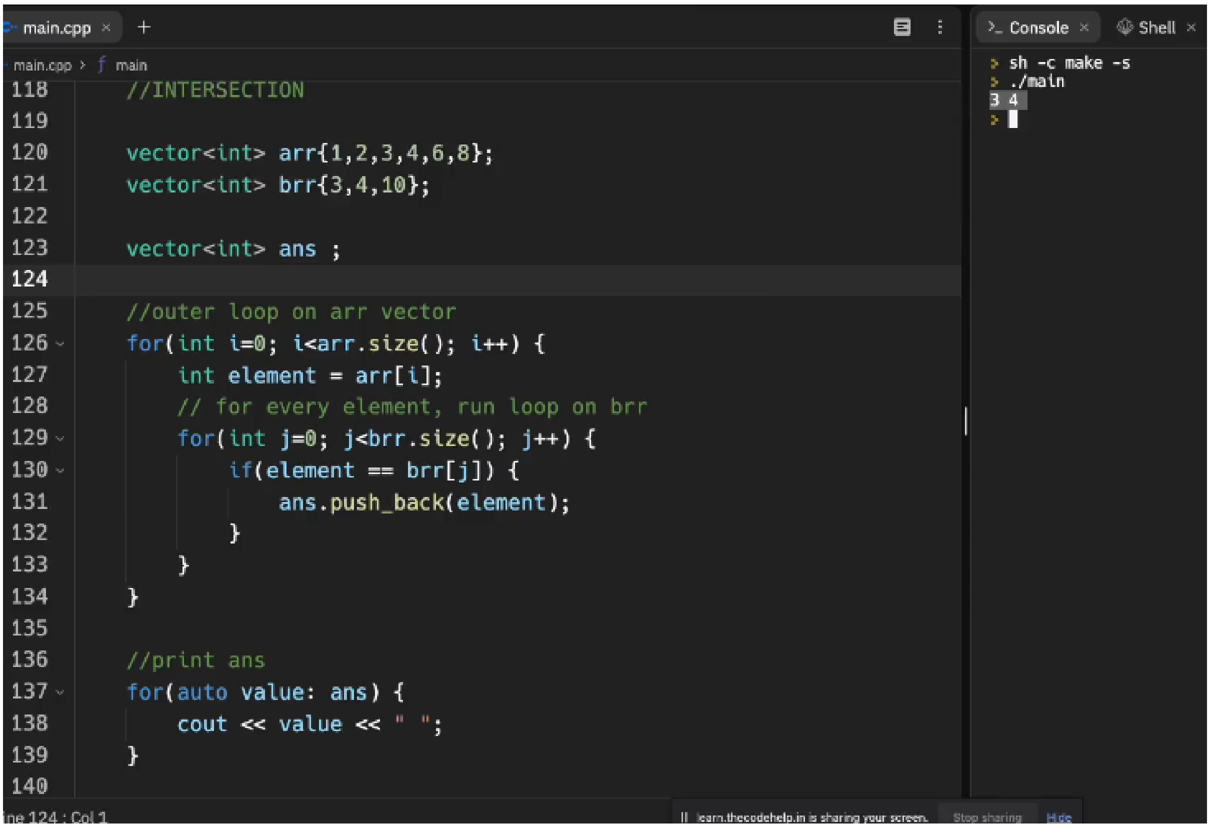
```
#include <iostream>
#include <vector>
using namespace std;

int vector<int> arr) {
    int ans = 0;

    for(int i=0; i<arr.size(); i++) {
        ans = ans ^ arr[i];
    }

    return ans;
}

int main() {
```



```
    //UNique Element

    int n;
    cout << "Enter the size of array " << endl;
    cin >> n;


    vector<int> arr(n);
    cout << "Enter the elements " << endl;
    //taking input
    for(int i=0; i<arr.size(); i++) {
        cin >> arr[i];
    }

    int uniqueElement = findUnique(arr);

    cout << "Unique Element is  " << uniqueElement << endl;


    return 0;
}
```

This is the whole code of it

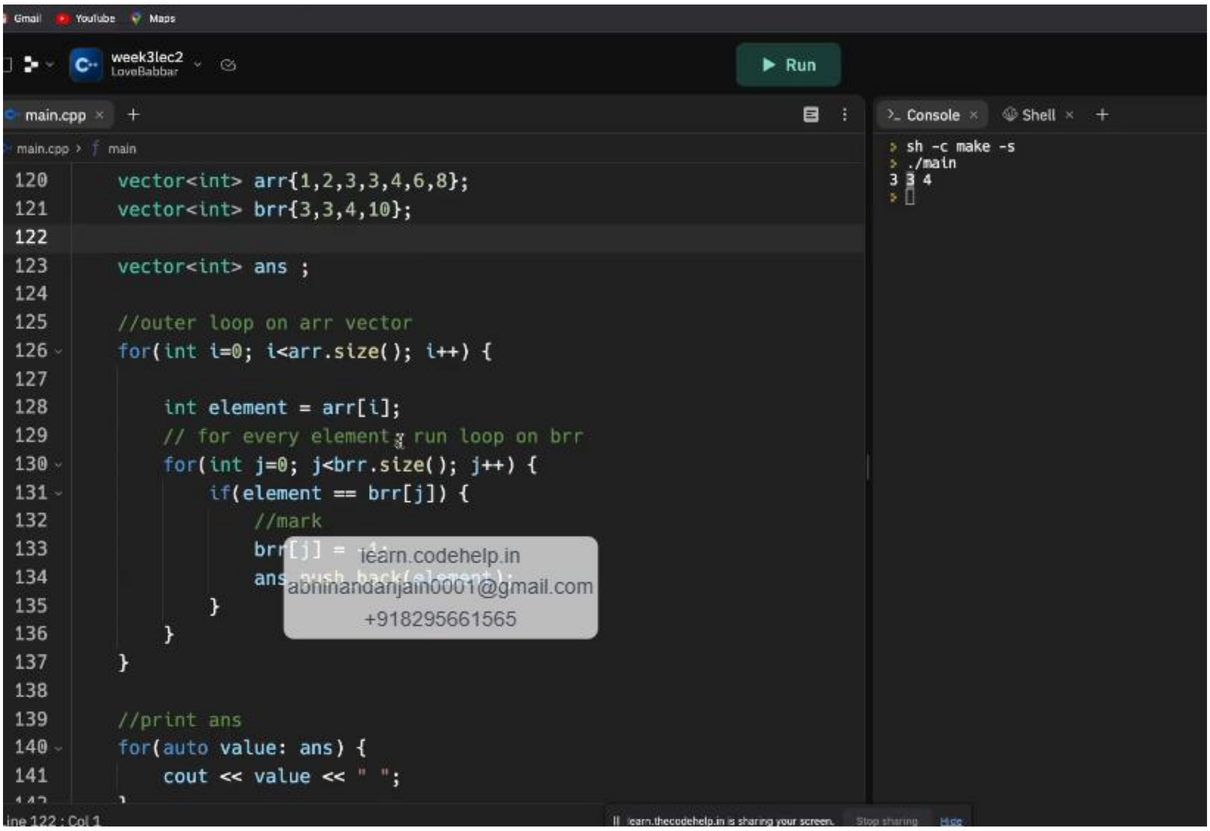
For intersection code is



```
    //INTERSECTION

    vector<int> arr{1,2,3,4,6,8};
    vector<int> brr{3,4,10};

    vector<int> ans ;

    //outer loop on arr vector
    for(int i=0; i<arr.size(); i++) {
        int element = arr[i];
        // for every element, run loop on brr
        for(int j=0; j<brr.size(); j++) {
            if(element == brr[j]) {
                ans.push_back(element);
            }
        }
    }

    //print ans
    for(auto value: ans) {
        cout << value << " ";
    }

```

```
> sh -c make -s
> ./main
3 4
> 
```

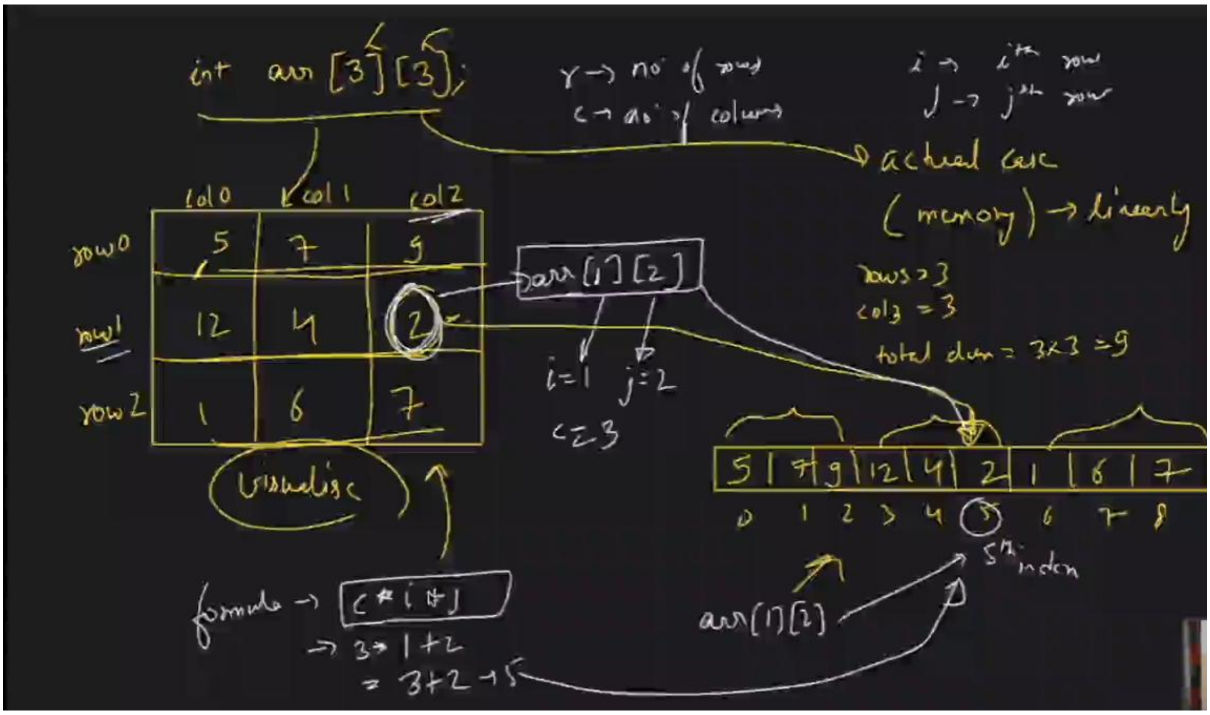
Here we have intersetction has been done if same element come more than once a time in a n array that will give na issue so we have mark that lemenet as a -1 or u can say int_min so we can use that so for example of that is



```
    vector<int> arr{1,2,3,3,4,6,8};
    vector<int> brr{3,3,4,10};

    vector<int> ans ;

    //outer loop on arr vector
    for(int i=0; i<arr.size(); i++) {

        int element = arr[i];
        // for every element run loop on brr
        for(int j=0; j<brr.size(); j++) {
            if(element == brr[j]) {
                //mark
                brr[j] = -1;
                ans.push_back(element);
            }
        }
    }

    //print ans
    for(auto value: ans) {
        cout << value << " ";
```

```
> sh -c make -s
> ./main
3 3 4
>
```

2d array

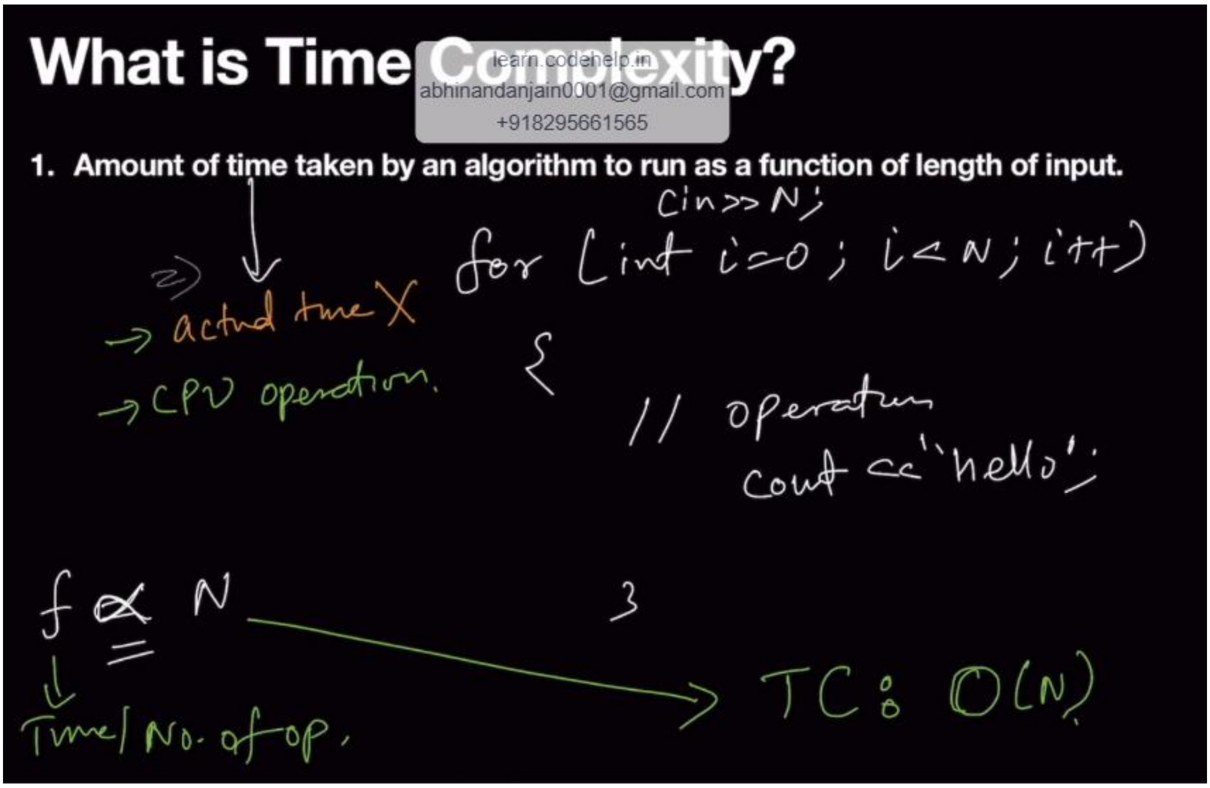


	col 0	col 1	col 2
row 0	5	7	9
row 1	12	4	2
row 2	1	6	7

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

As in the memory store in 1d array only of 2d array so to find the particular place we have to see this

Time complexity



What is Time Complexity?

1. Amount of time taken by an algorithm to run as a function of length of input.

→ actual time ✗

→ CPU operation.

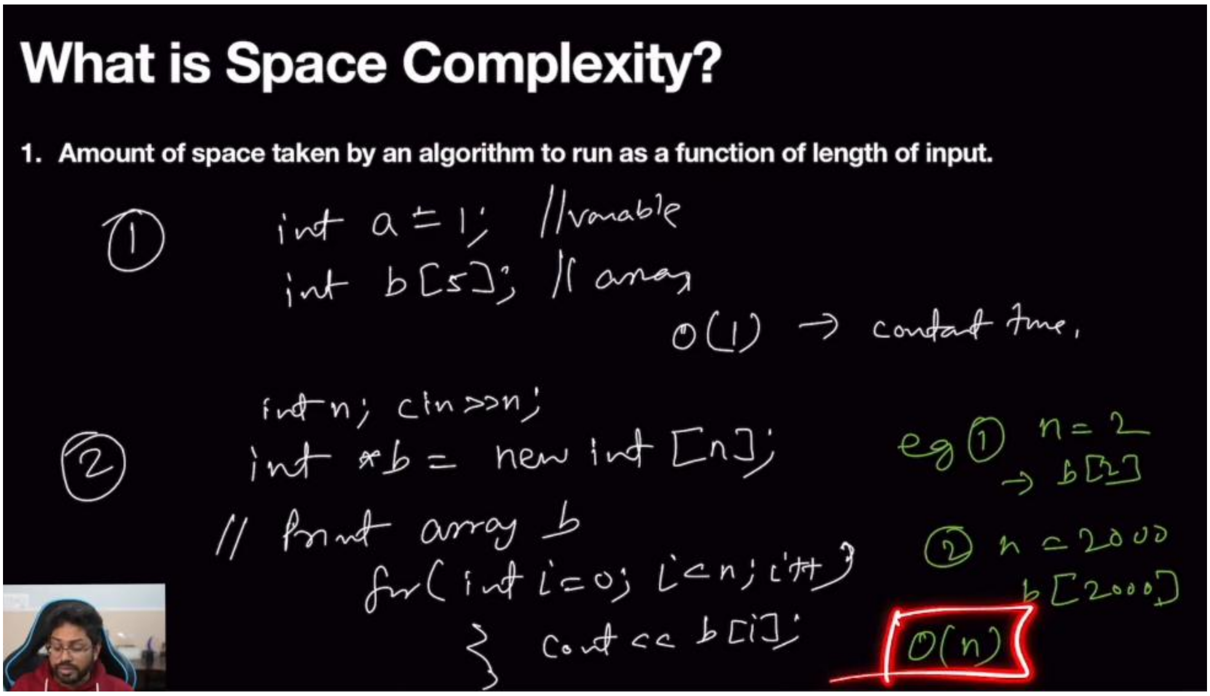
```
cin>>N;
for (int i=0; i<N; i++)
{
    // operation
    cout<<"hello";
}
```

$f \propto N$

$=$

Time/ No. of op.

$\longrightarrow$ TC: $O(N)$



What is Space Complexity?

1. Amount of space taken by an algorithm to run as a function of length of input.

①

```
int a = 1;   //variable
int b[5];    // array
```

$O(1)$ → constant time.

②

```
int n; cin>>n;
int *b = new int [n];
// Print array b
    for( int i=0; i<n; i++)
    {   cout<< b[i];
    }
```

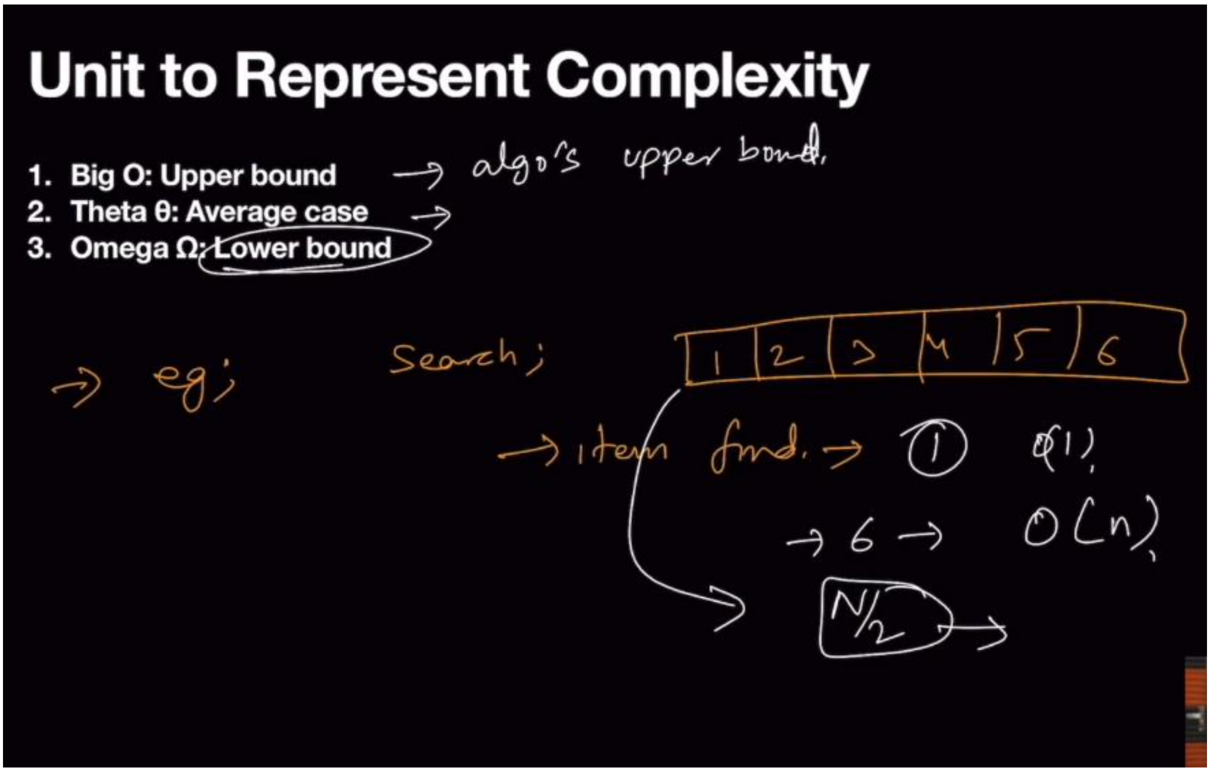
eg ① n = 2
→ b[2]

② n = 2000
b[2000]

$\boxed{O(n)}$

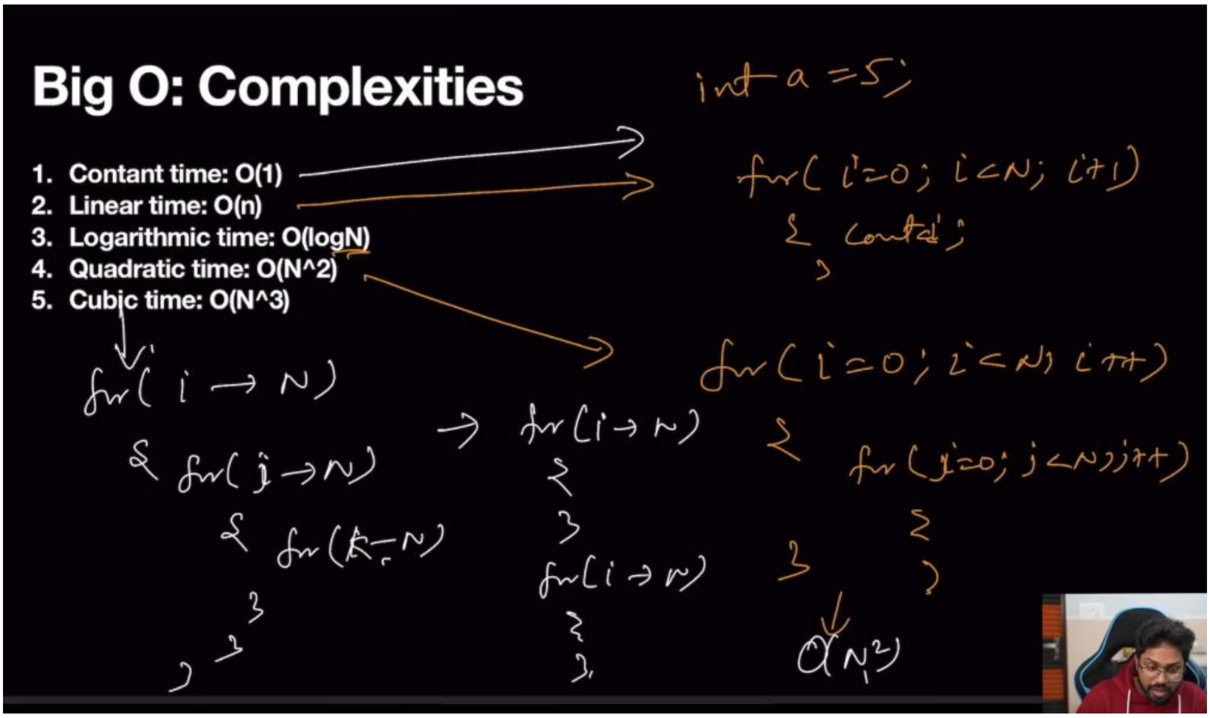
Here we are checking the complexity as in the coding we will always give the worst complexity Big O it is the worst case complexity so we have to find the always big o because what is the complexity in our bad case oke bro like suppose I have to found the 6 in the array as it is placed in the last iahve to read all the array then only I can move to it so it will give the complexity BigO(n) . oke
theta complecity is the middle one like suppose in middle how much time or space it will took

Omega complexity is the starting complecity means how fast we ca find our element like suppose we have to find the 1 in array and the 1 in starting so the complexity will the omega1



Unit to Represent Complexity

1. Big O: Upper bound
2. Theta θ: Average case
3. Omega Ω: Lower bound



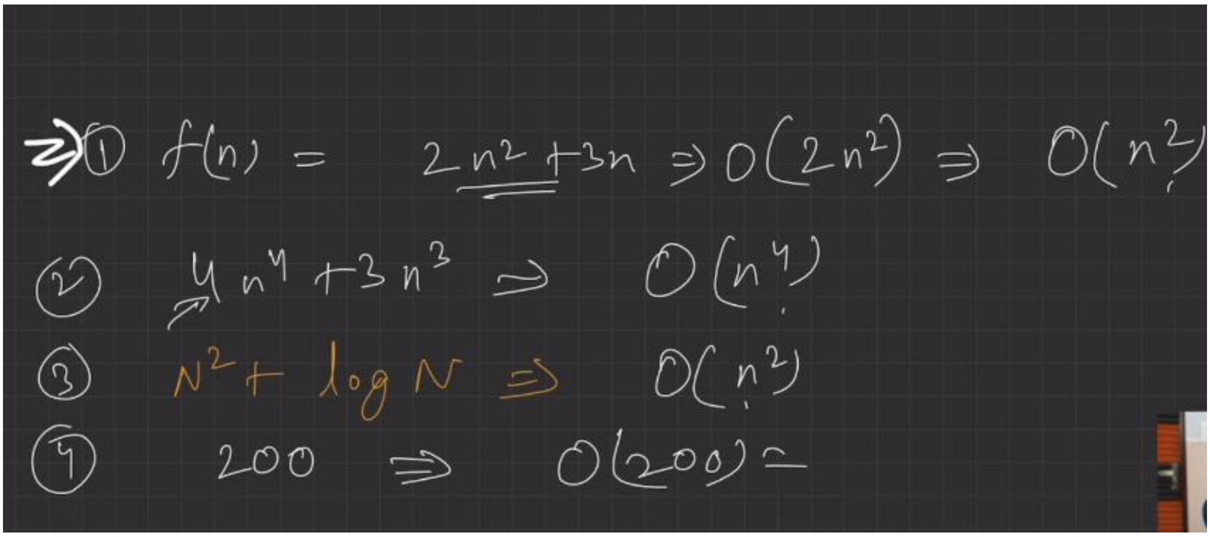
Big O: Complexities

1. Contant time: O(1)
2. Linear time: O(n)
3. Logarithmic time: O(logN)
4. Quadratic time: O(N^2)
5. Cubic time: O(N^3)

```
int a =5;

for( i=0; i<N; i+1)
{ contd;
}

for(i=0; i<N; i++)
{
    for(j=0; j<N; j++)
    {
    }
}
O(N^2)
```

```
for( i -> N)
{ for( j -> N)
    { for(k -> N)
      }
    }
}
```

```
-> for(i -> n)
   {
   }
   for(i -> n)
   {
   }
```

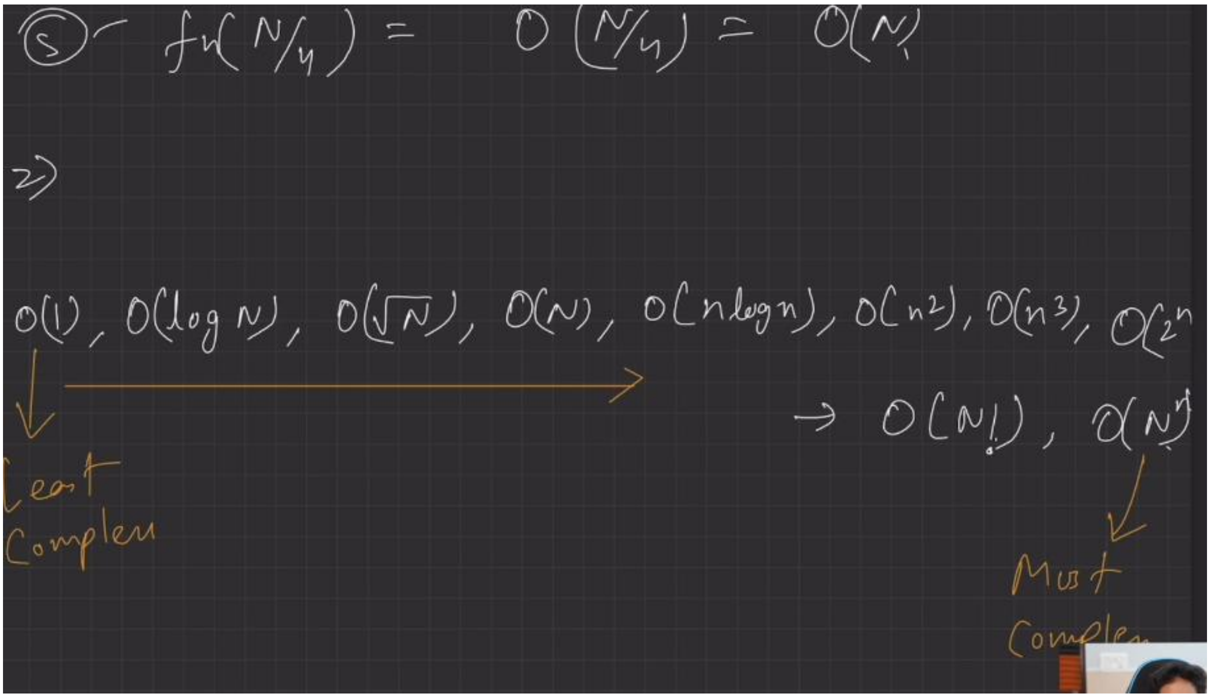


① $f(n) = 2n^2+3n \Rightarrow O(2n^2) \Rightarrow O(n^2)$

② $4n^4+3n^3 \Rightarrow O(n^4)$

③ $N^2+\log N \Rightarrow O(n^2)$

④ $200 \Rightarrow O(200) =$



⑤ - $fn(N/4) = O(N/4) = O(N)$

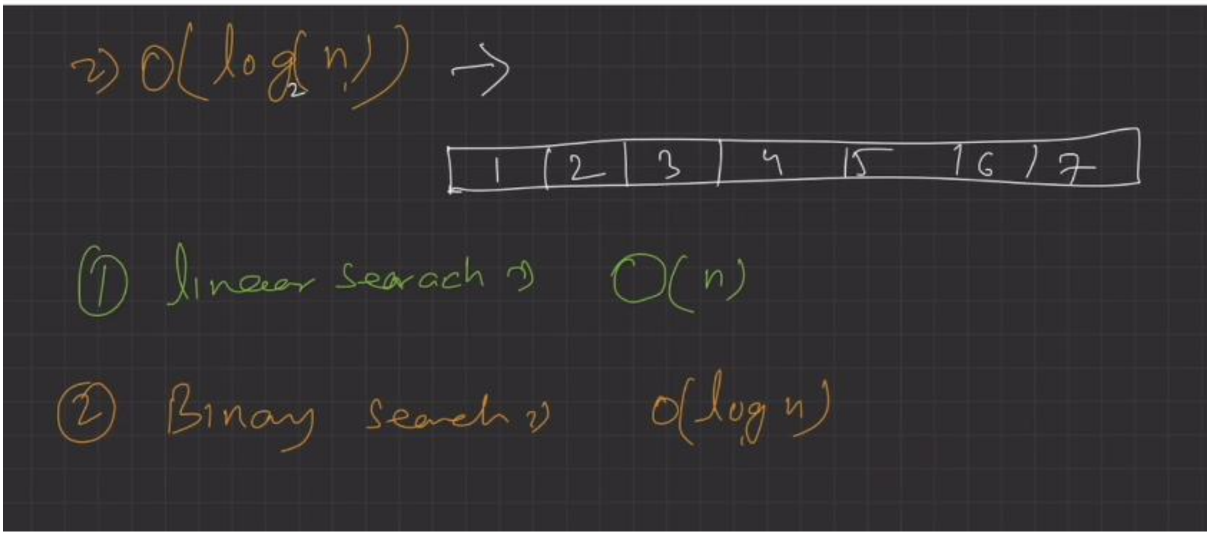
2)

$O(1), O(\log N), O(\sqrt{N}), O(N), O(n \log n), O(n^2), O(n^3), O(2^n$

$\rightarrow O(N!), O(N^N)$

Least Complex

Most Comple

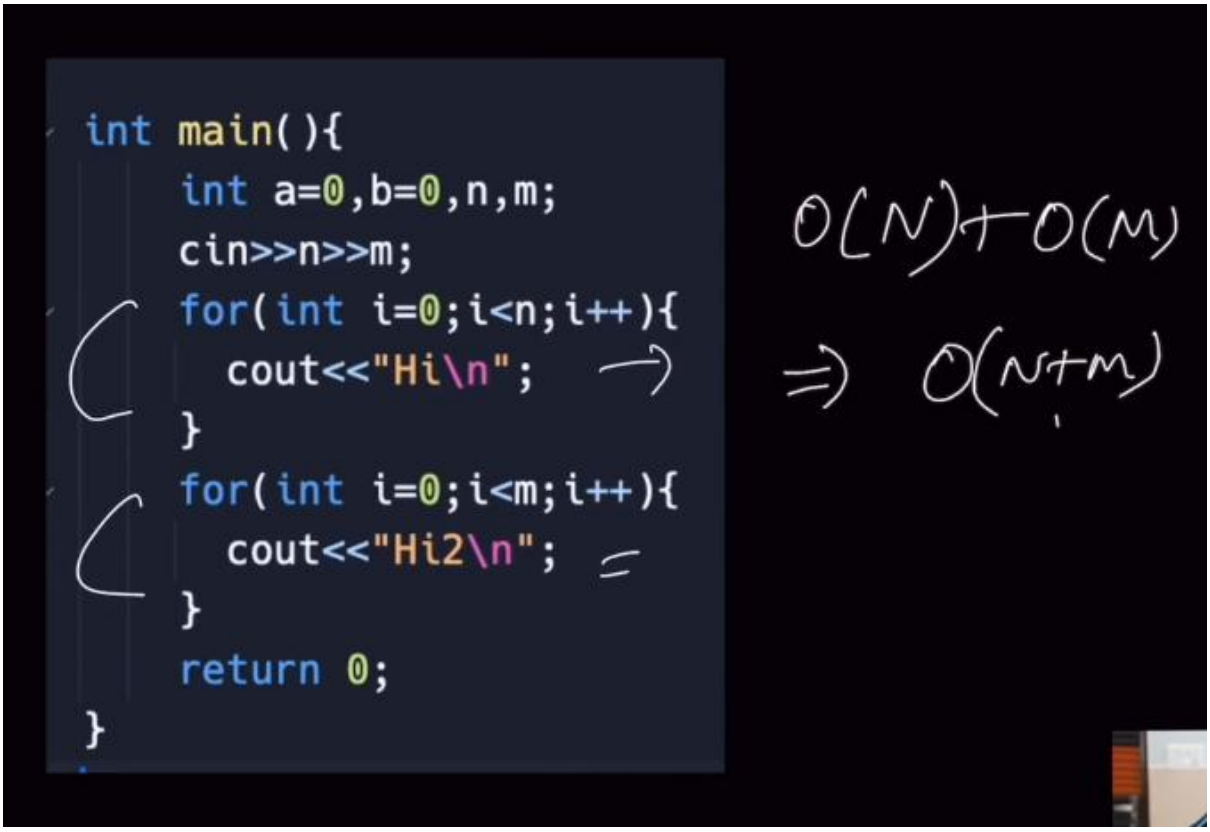


2) $O(\log_2(n)) \rightarrow$

1	2	3	4	5	6	7

① linear search ⇒ $O(n)$

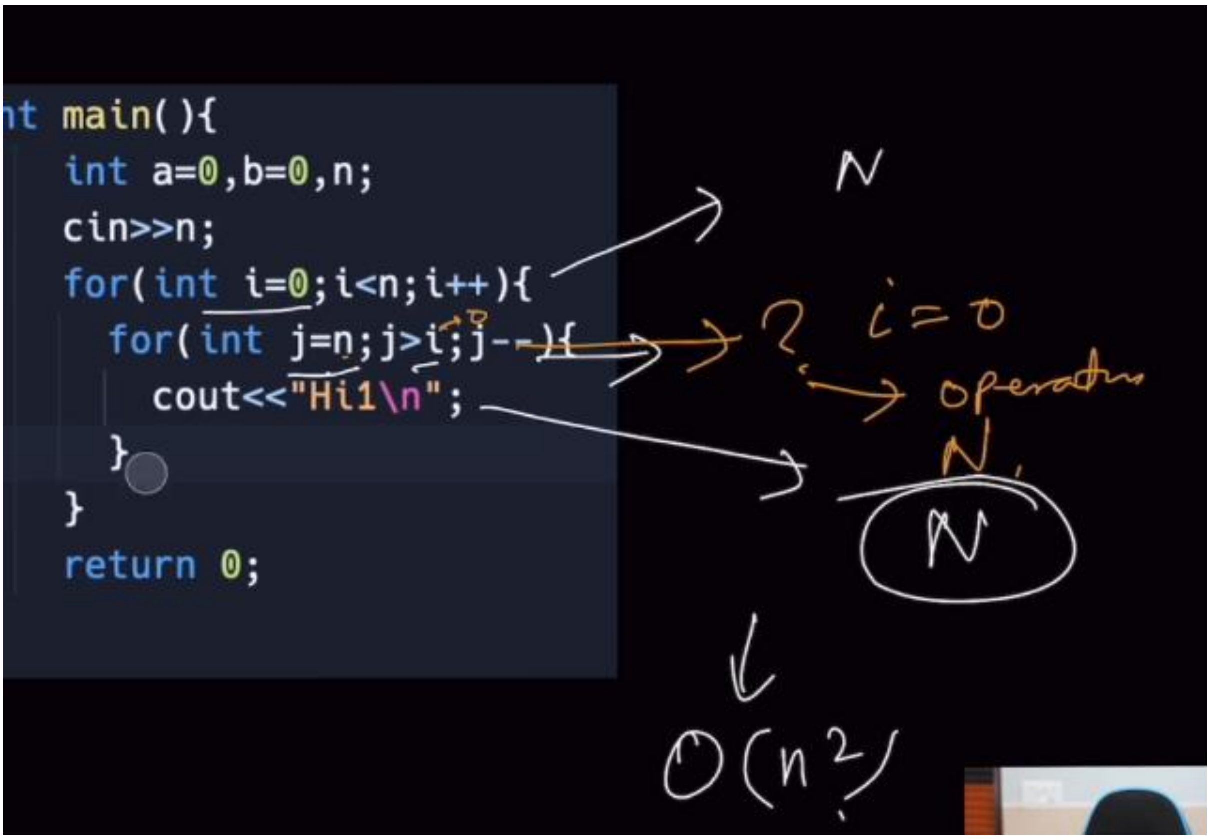
② Binay search ⇒ $O(\log n)$



```
int main(){
    int a=0,b=0,n,m;
    cin>>n>>m;
    for(int i=0;i<n;i++){
      cout<<"Hi\n";
    }
    for(int i=0;i<m;i++){
      cout<<"Hi2\n";
    }
    return 0;
}
```

$O(N)+O(M)$

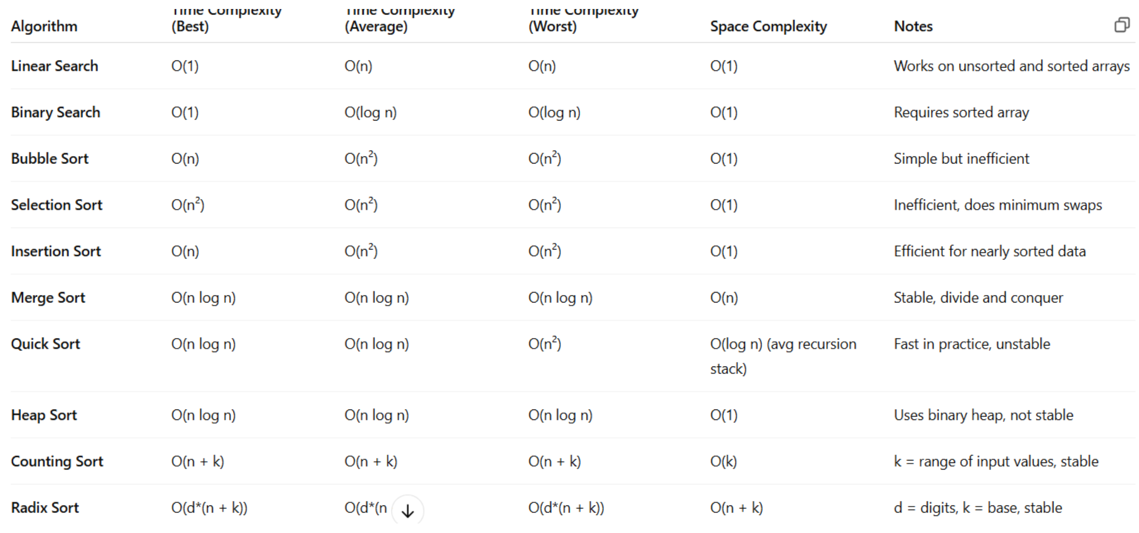
$\Rightarrow O(N+M)$



```
nt main(){
    int a=0,b=0,n;
    cin>>n;
    for(int i=0;i<n;i++){
      for(int j=n;j>i;j--){
        cout<<"Hi1\n";
      }
    }
    return 0;
```

$O(n^2)$

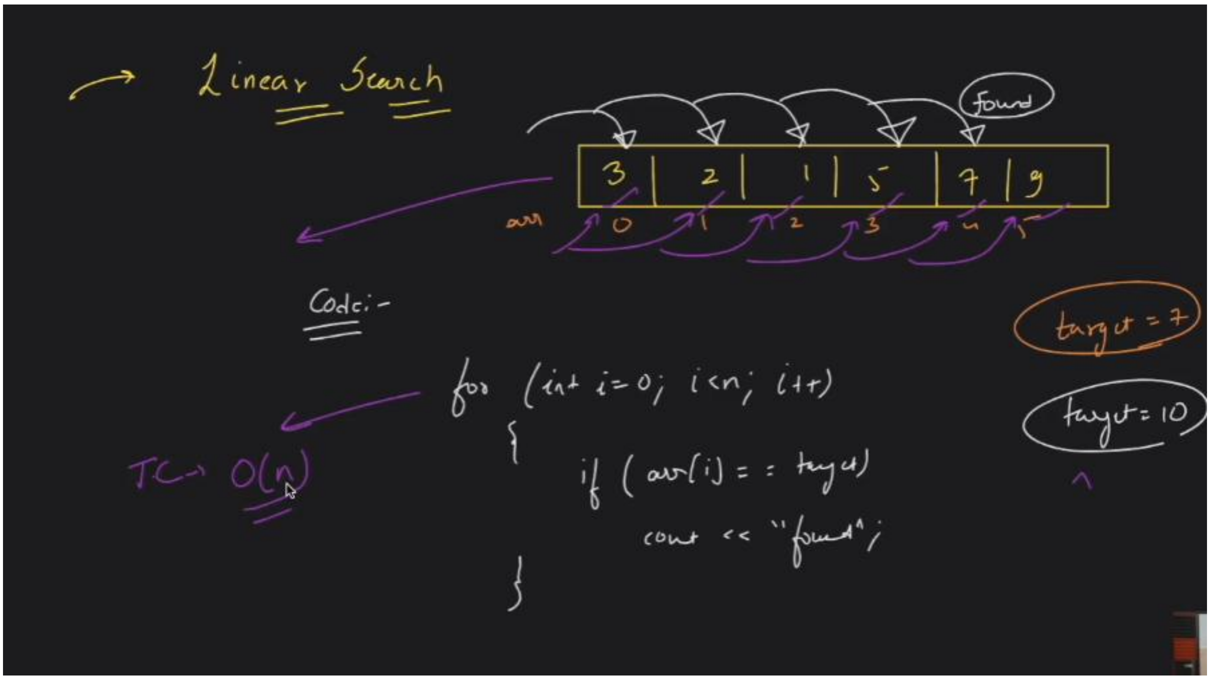
SEARCHING AND SORTING



Algorithm	Time Complexity (Best)	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity	Notes
Linear Search	O(1)	O(n)	O(n)	O(1)	Works on unsorted and sorted arrays
Binary Search	O(1)	O(log n)	O(log n)	O(1)	Requires sorted array
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)	Simple but inefficient
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	Inefficient, does minimum swaps
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)	Efficient for nearly sorted data
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Stable, divide and conquer
Quick Sort	O(n log n)	O(n log n)	O(n^2)	O(log n) (avg recursion stack)	Fast in practice, unstable
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)	Uses binary heap, not stable
Counting Sort	O(n + k)	O(n + k)	O(n + k)	O(k)	k = range of input values, stable
Radix Sort	O(d*(n + k))	O(d*(n	O(d*(n + k))	O(n + k)	d = digits, k = base, stable

1..linear search

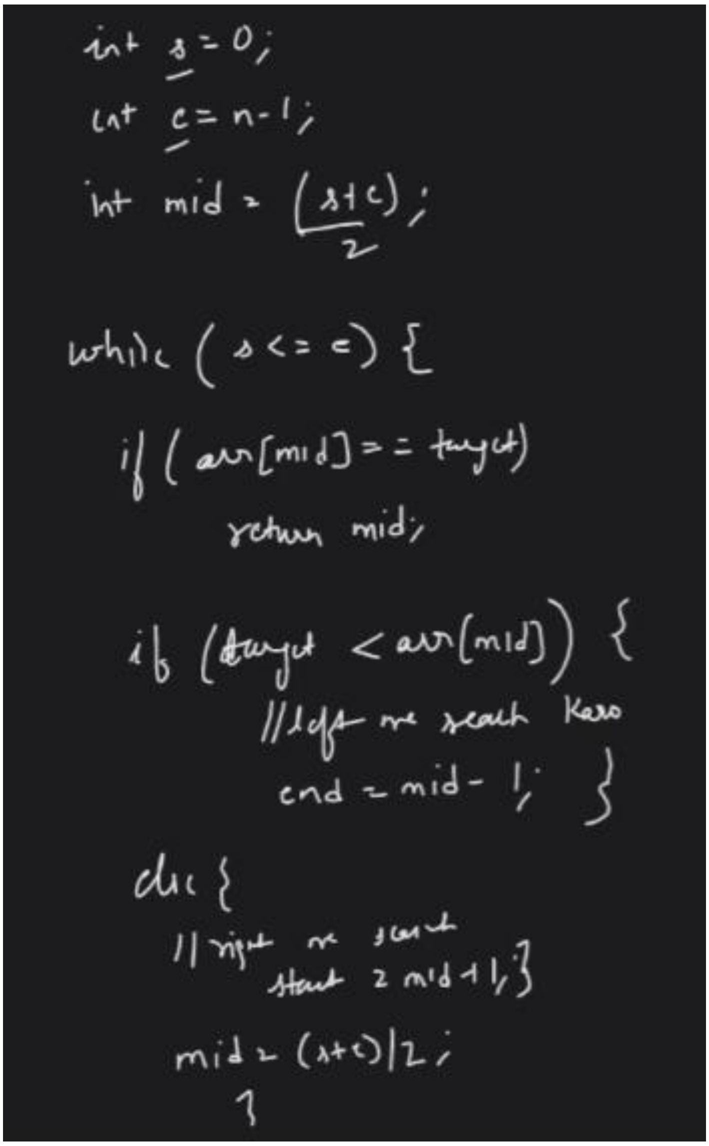
```
    #include <iostream>

using namespace std;


int main() {

  int arr[] = {10, 25, 30, 45, 60};

  int n = sizeof(arr) / sizeof(arr[0]);

  int key = 30;  // value to search

  int index = -1;


  for(int i = 0; i < n; i++) {

    if(arr[i] == key) {

      index = i;

      break;

    }

  }


  if(index != -1) {
        cout << "Element found at index: " << index << endl;
    } else {
        cout << "Element not found in array." << endl;
    }

    return 0;
}
```



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

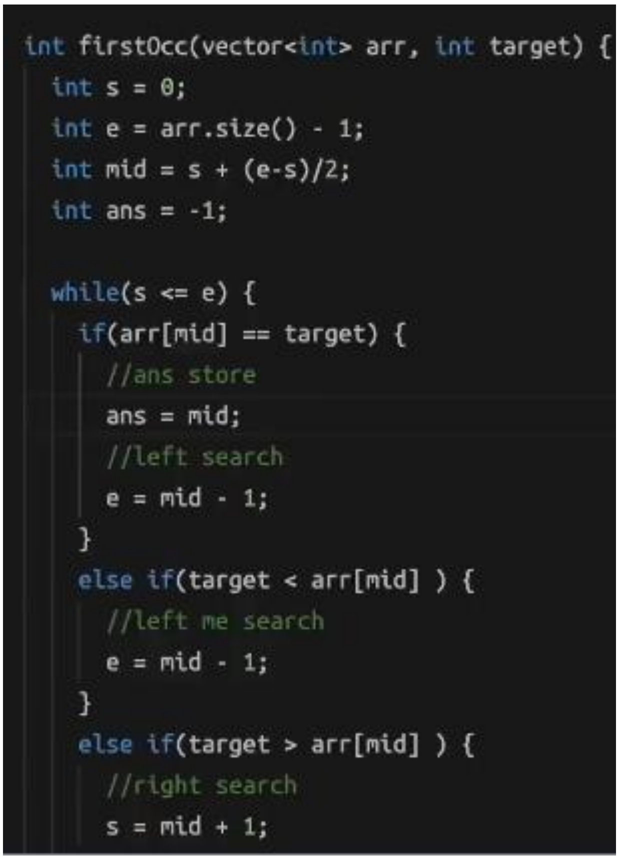
Bnary search cde

```
#include <iostream>
#include<algorithm>
#include<vector>
using namespace std;

int binarySearch(int arr[], int size, int target) {
  int start = 0;
  int end = size - 1;

  int mid = start + (end - start ) / 2;

  while(start <= end) {
    int element = arr[mid];

    if(element == target) {//element found, then return index
      return mid;
    }

    if(target < element) {
      //search in left
      end = mid - 1;
    }
    else {
      //search in right
      start = mid + 1;
    }

    mid = start + (end - start ) / 2;

  }

  //element not found
  return -1;

}

int main() {
  // int arr[] = {2,4,6,8,10,12,16};
  // int size = 7;
  // int target = 20;

  // int indexOftarget = binarySearch(arr, size, target);

  // if(indexOftarget == -1) {
  //   cout << "target not found" << endl;
  // }
  // else  {
  //   cout << "target found at " << indexOftarget <<" index " << endl;
  // }
  vector<int> v{1,2,3,4,5,6};
  int arr[] = {1,2,3,4,5,6,7 };
  int size = 7;

  if(binary_search(arr, arr + size, 7)) {
    cout << "Found" << endl;
  }
  else {
    cout << "Not found. " << endl;
  }


  return 0;
}
```

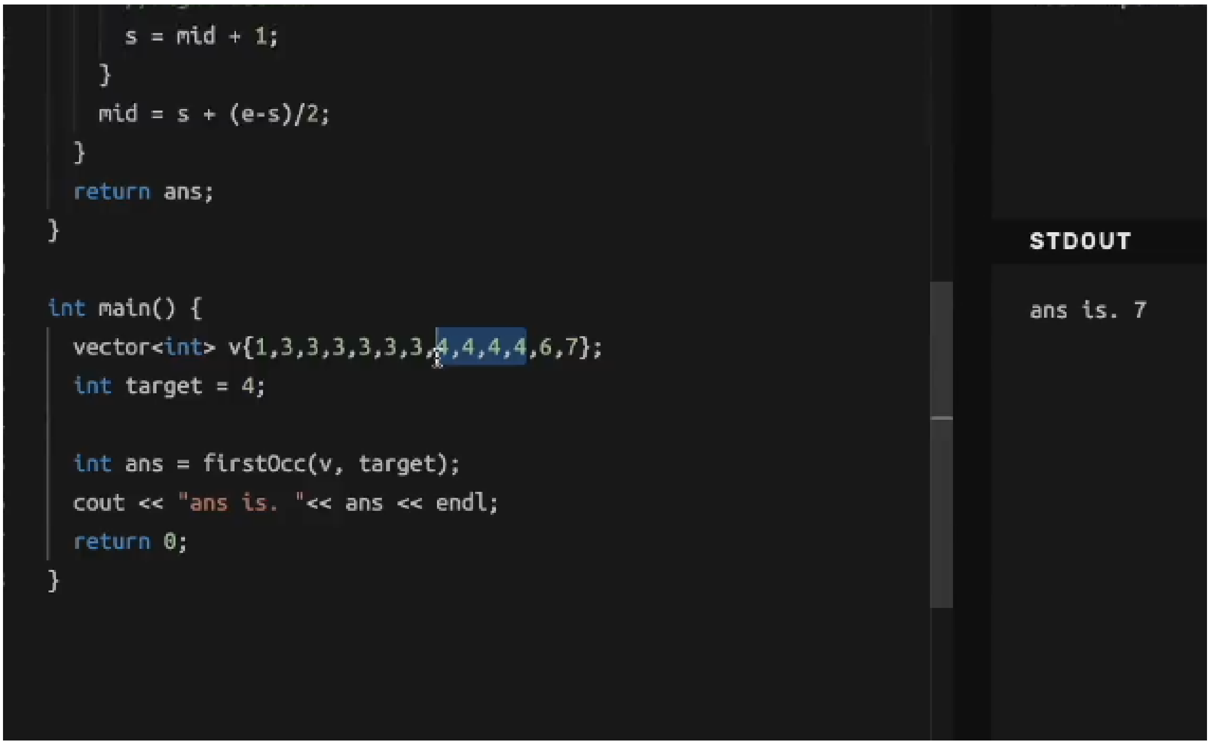


```
int s = 0;
int e = n-1;
int mid = (s+e)/2;

while (s <= e) {

  if (arr[mid] = = target)
      return mid;

  if (target < arr(mid]) {
      //left me search karo
      end = mid - 1; }

  else {
    //right me search
      start = mid + 1; }

  mid = (s+e)/2;
  }
```

First Ocuurence in binary search



```
int firstOcc(vector<int> arr, int target) {
  int s = 0;
  int e = arr.size() - 1;
  int mid = s + (e-s)/2;
  int ans = -1;

  while(s <= e) {
    if(arr[mid] == target) {
      //ans store
      ans = mid;
      //left search
      e = mid - 1;
    }
    else if(target < arr[mid] ) {
      //left me search
      e = mid - 1;
    }
    else if(target > arr[mid] ) {
      //right search
      s = mid + 1;
```

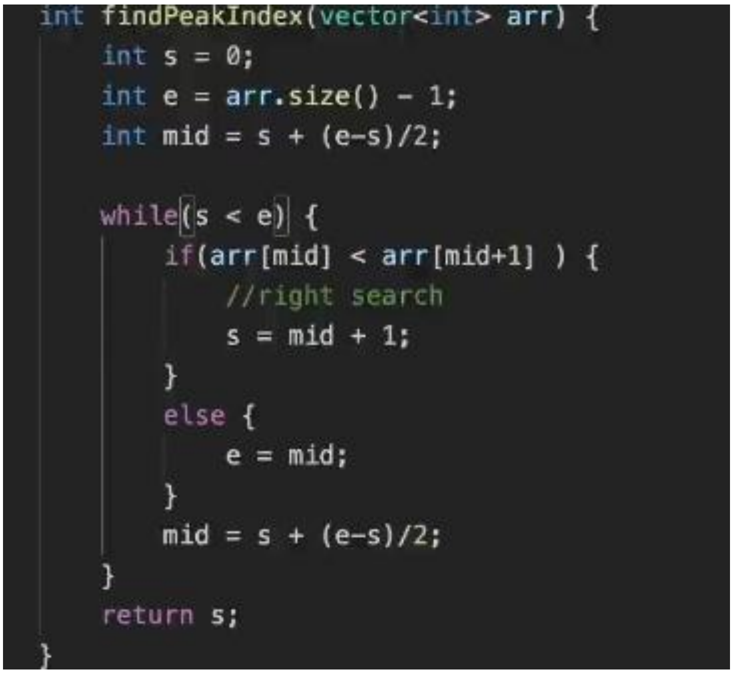


```
      s = mid + 1;
    }
    mid = s + (e-s)/2;
  }
  return ans;
}

int main() {
  vector<int> v{1,3,3,3,3,3,3,4,4,4,4,6,7};
  int target = 4;

  int ans = firstOcc(v, target);
  cout << "ans is. "<< ans << endl;
  return 0;
}
```

```
ans is. 7
```

In last accurence we will store in the if(arr(mid)==target) { s=m+1}’

Peak element code in binary search as the peak element in in middle and on the left side small element and on the irhght side also small but sorted and in middle only the peak element is kep so the code for that is



```
int findPeakIndex(vector<int> arr) {
    int s = 0;
    int e = arr.size() - 1;
    int mid = s + (e-s)/2;

    while(s < e) {
        if(arr[mid] < arr[mid+1] ) {
            //right search
            s = mid + 1;
        }
        else {
            e = mid;
        }
        mid = s + (e-s)/2;
    }
    return s;
}
```

Sorting

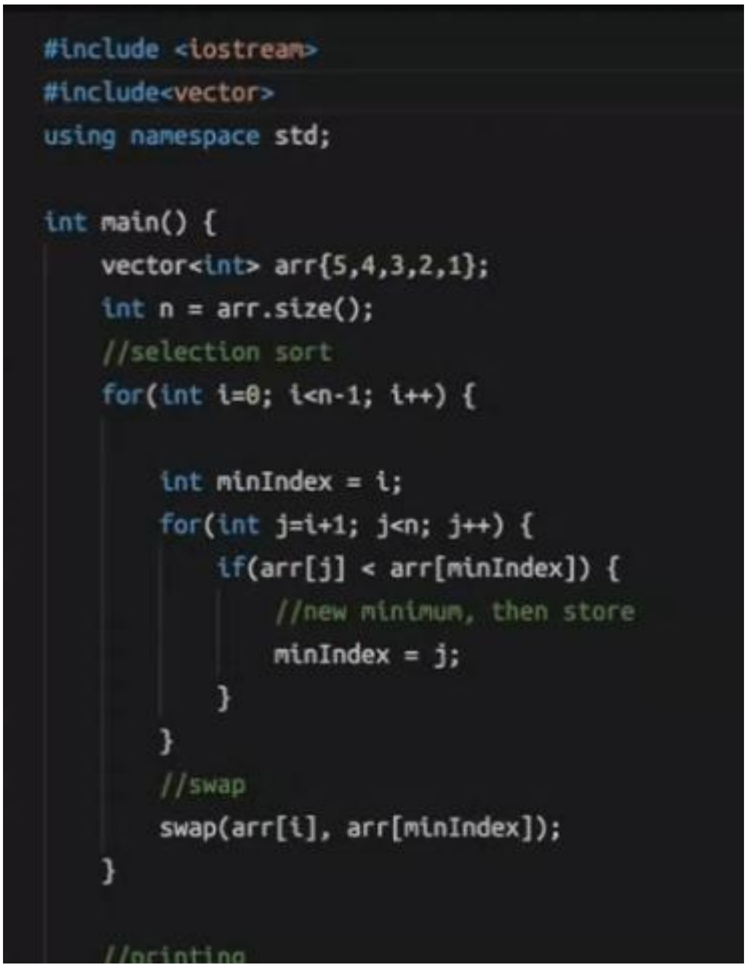
Sorting is technique there we have to arrange the element in ascending or descending order oke

Selection Sort

In here we will find the smallest element and place at the th index and in this way again find the second smallest element then place at the 1st index int this way it will work

Code

```
#include <iostream>
#include<vector>
using namespace std;

int main() {
    vector<int> arr{5,4,3,2,1};
// int arr[] = {10, 1, 7, 6, 14, 9};
// int n = sizeof(arr) / sizeof(arr[0]);

    int n = arr.size()
    for(int i=0; i<n-1; i++) {

        int minIndex = i;
        //inner Loop -> index of minimum element in range i->n
        for(int j=i+1; j<n; j++) {
            if(arr[j] < arr[minIndex]) {
                //new minimum, then store
                minIndex = j;
            }
        }
        //swap
        swap(arr[i], arr[minIndex]);
    }

    //printing
    for(int i=0; i<n; i++) {
        cout << arr[i] << " ";
    }cout << endl;

  return 0;
}
```



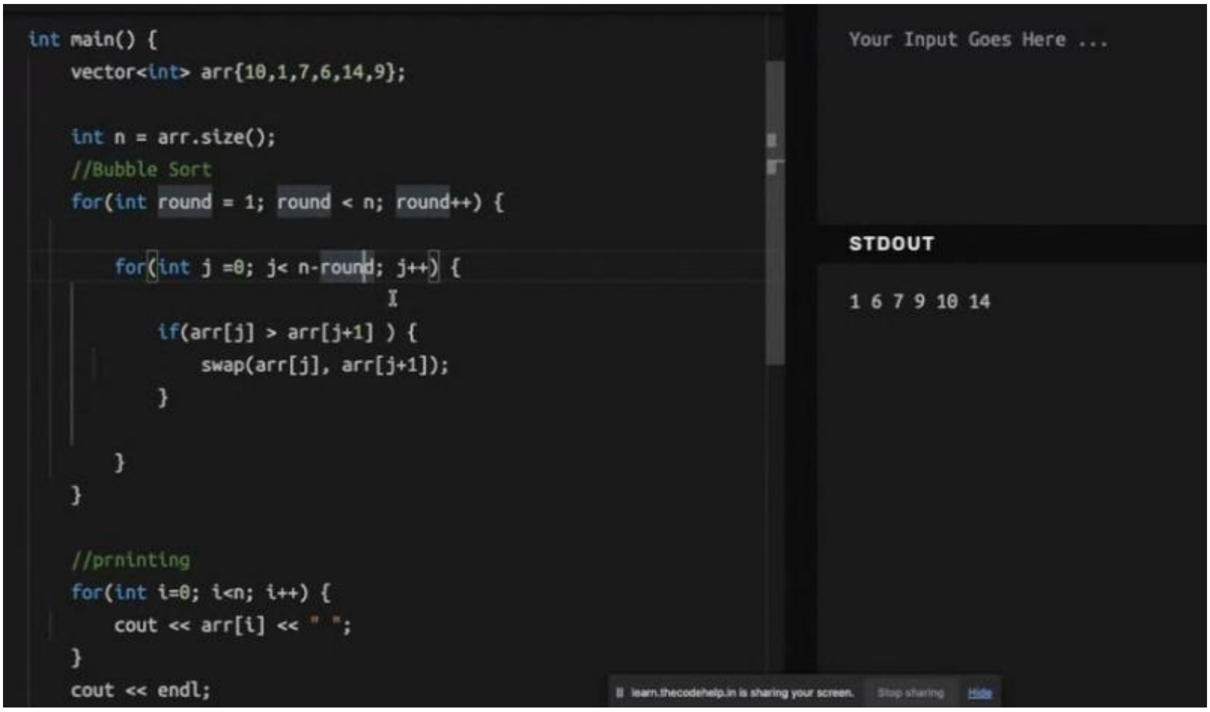
```
#include <iostream>
#include<vector>
using namespace std;

int main() {
    vector<int> arr{5,4,3,2,1};
    int n = arr.size();
    //selection sort
    for(int i=0; i<n-1; i++) {

        int minIndex = i;
        for(int j=i+1; j<n; j++) {
            if(arr[j] < arr[minIndex]) {
                //new minimum, then store
                minIndex = j;
            }
        }
        //swap
        swap(arr[i], arr[minIndex]);
    }

    //printing
```

BUBBLE SORT

In the bubble sort just we will swap the 1st and 2nd element and in this way we will sort all the numbers
in the first time 1st largest number will reach to its correct option

So how many element in the array we will do at that time bubble sort or u can say round and every time the it will sort from back only as largest element will sort first then its second largest element in this way it is goes on

```
#include <iostream>
#include<vector>
using namespace std;

int main() {
    vector<int> arr{10,1,7,6,14,9};
// int arr[] = {10, 1, 7, 6, 14, 9};
// int n = sizeof(arr) / sizeof(arr[0]);


    int n = arr.size();
    //Bubble Sort
    for(int round = 1; round < n; round++) {
        int swapCount = 0;
        for(int j =0; j< n-round; j++) {

            if(arr[j] > arr[j+1] ) {
                swap(arr[j], arr[j+1]);
                swapCount++;
            }

        }
        if(swapCount == 0) {
            //sort ho chuka hai, no need to check in further rounds
            break;
        }
    }

    //prninting
    for(int i=0; i<n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;



  return 0;
}
```



```
int main() {
    vector<int> arr{10,1,7,6,14,9};

    int n = arr.size();
    //Bubble Sort
    for(int round = 1; round < n; round++) {

        for(int j =0; j< n-round; j++) {

            if(arr[j] > arr[j+1] ) {
                swap(arr[j], arr[j+1]);
            }

        }
    }

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

```
1 6 7 9 10 14
```

INSERTION SORT

We have to sort the each element pace at it right place here we will do that we do not know sort the 0 elment we start form 1 st element and we will check at the back which is smaleest element from it if yes then place there and shift oke

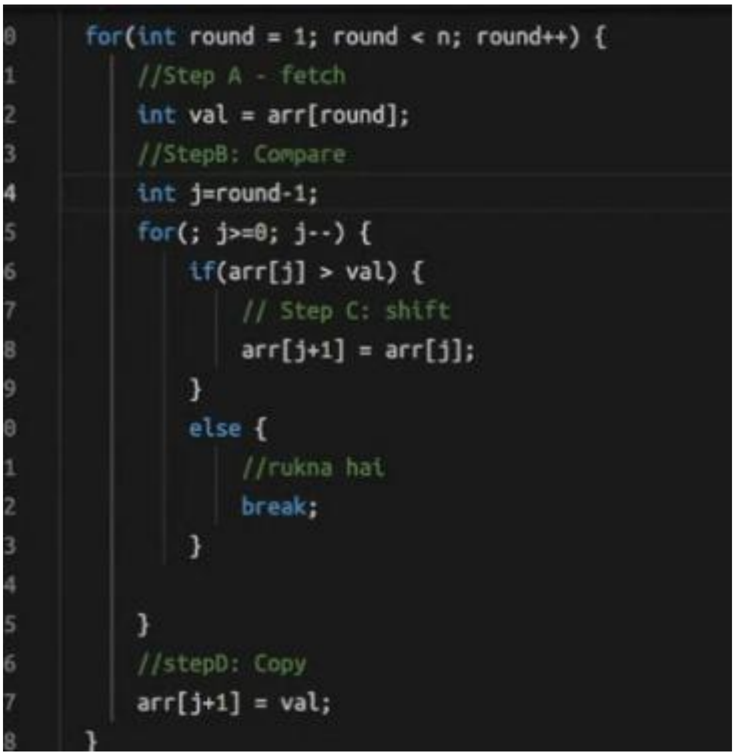
Like 10,1,1,7,

We will check the 1st elemnt = 1

1 compare with 10 small hai then we will shoft it

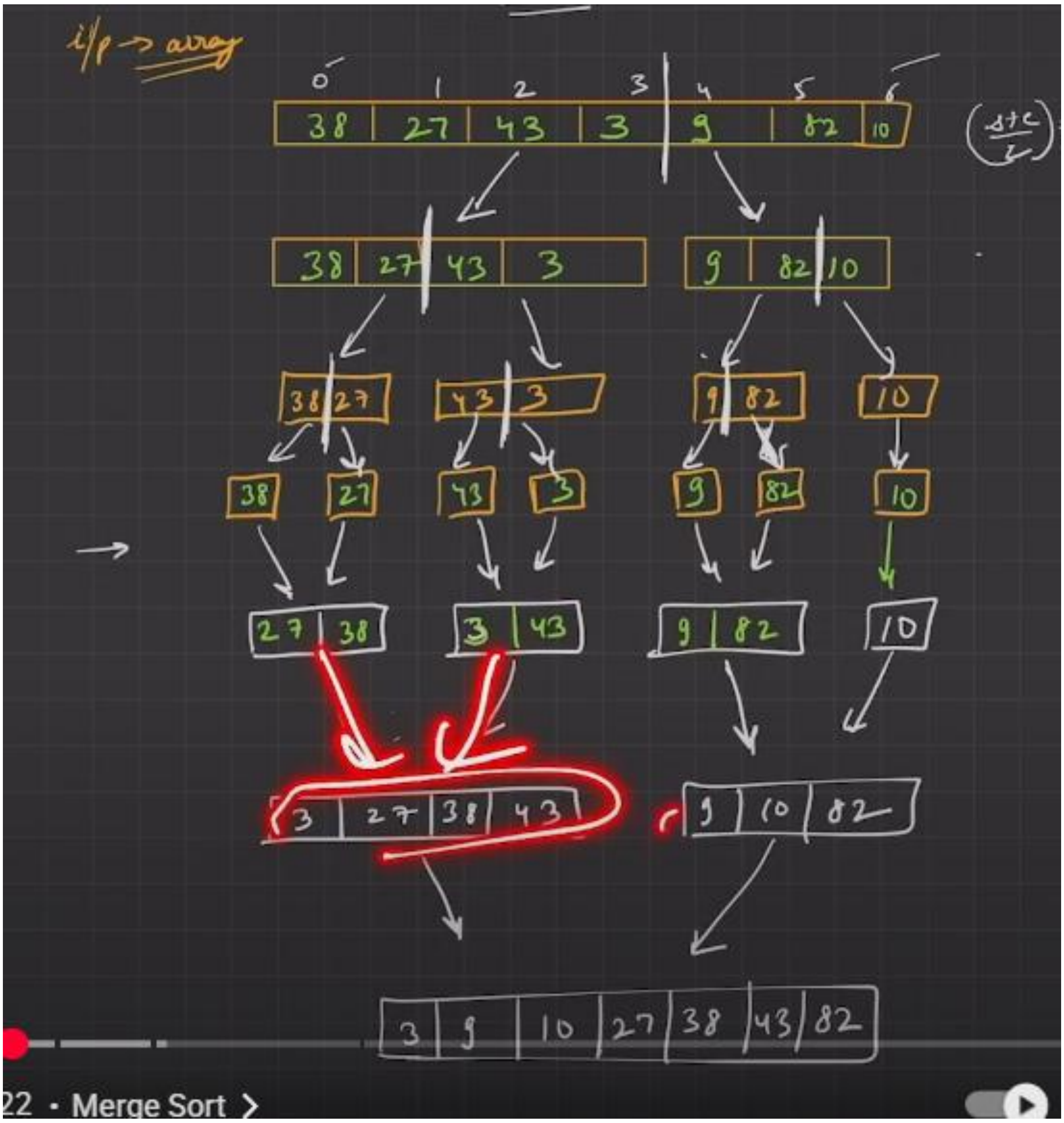
Time Complexity ==O(n2)

```
#include <iostream>
#include<vector>
using namespace std;

int main() {
  vector<int> arr{10,1,7,6,14,9};
// int arr[] = {10, 1, 7, 6, 14, 9};
// int n = sizeof(arr) / sizeof(arr[0]);
  int n = arr.size();

  //insertion sort
  for(int round = 1; round < n; round++) {
      //Step A - fetch
      int val = arr[round];
      //StepB: Compare
      int j=round-1;
      for(; j>=0; j--) {
          if(arr[j] > val) {
              // Step C: shift
              arr[j+1] = arr[j];
          }
          else {
              //rukna hai
              break;
          }

      }
      //stepD: Copy
      arr[j+1] = val;
  }

  //printinhg
  for(int i=0; i<n; i++) {
      cout << arr[i] << " ";
  }
  cout << endl;



  return 0;
}
```



```
for(int round = 1; round < n; round++) {
    //Step A - fetch
    int val = arr[round];
    //StepB: Compare
    int j=round-1;
    for(; j>=0; j--) {
        if(arr[j] > val) {
            // Step C: shift
            arr[j+1] = arr[j];
        }
        else {
            //rukna hai
            break;
        }

    }
    //stepD: Copy
    arr[j+1] = val;
}
```

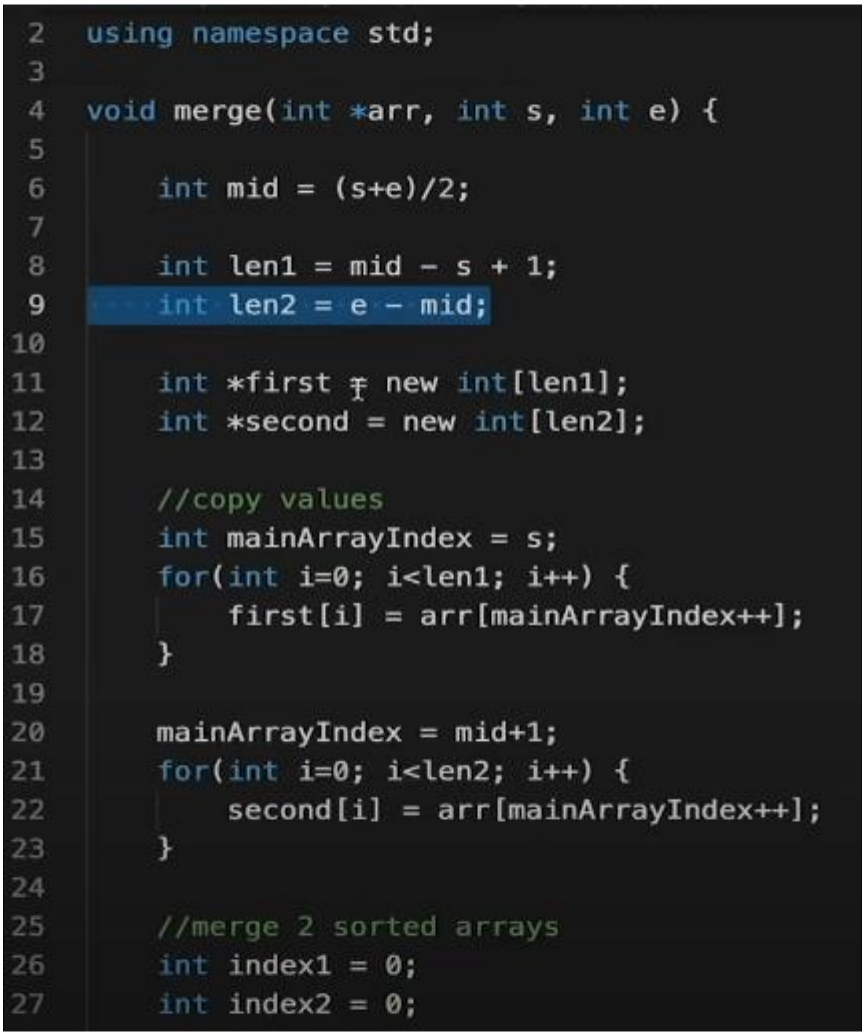
Merge SORT



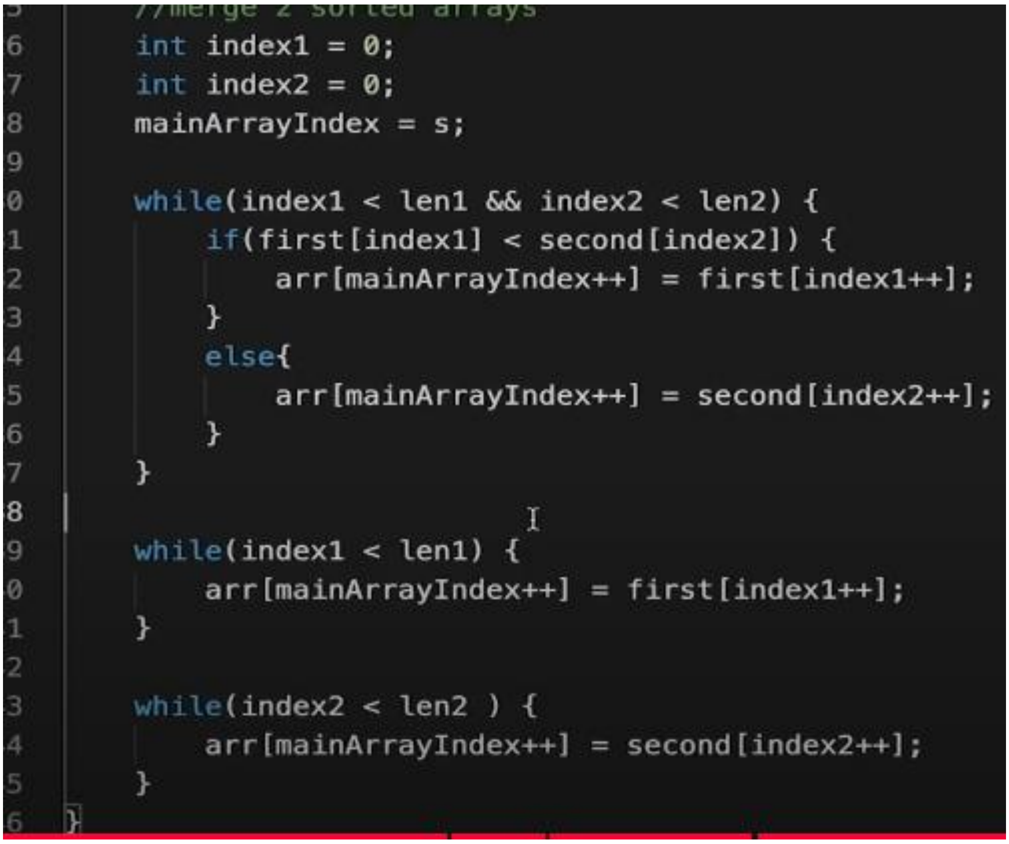
```
#include<iostream>
```

```
using namespace std;

void merge(int *arr, int s, int e) {

  int mid = (s+e)/2;

  int len1 = mid - s + 1;
  int len2 = e - mid;

  int *first = new int[len1];
  int *second = new int[len2];

  //copy values
  int mainArrayIndex = s;
  for(int i=0; i<len1; i++) {
    first[i] = arr[mainArrayIndex++];
  }

  mainArrayIndex = mid+1;
  for(int i=0; i<len2; i++) {
    second[i] = arr[mainArrayIndex++];
  }

  //merge 2 sorted arrays
  int index1 = 0;
  int index2 = 0;
  mainArrayIndex = s;
```

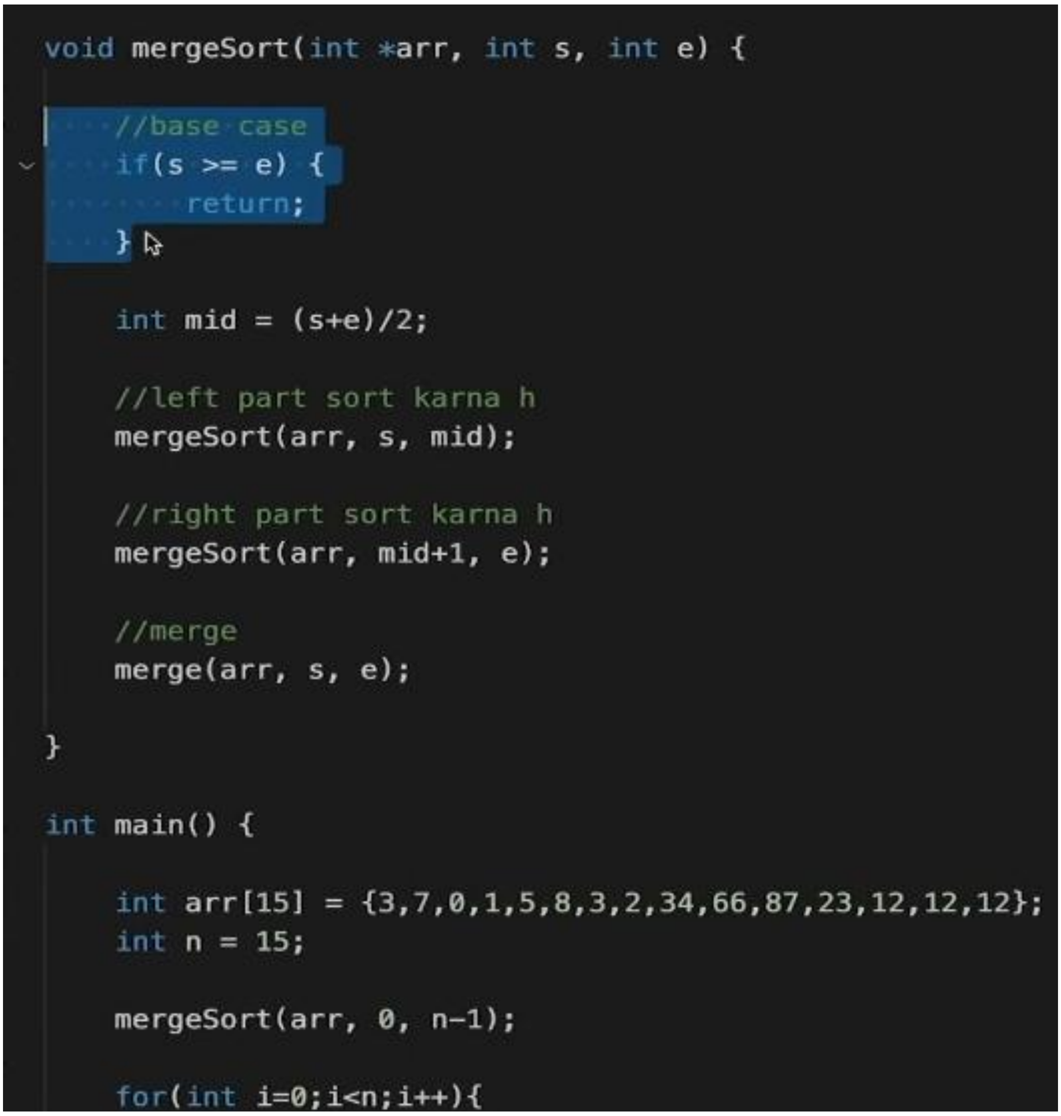
```
    while(index1 < len1 && index2 < len2) {
        if(first[index1] < second[index2]) {
            arr[mainArrayIndex++] = first[index1++];
        }
        else{
            arr[mainArrayIndex++] = second[index2++];
        }
    }

    while(index1 < len1) {
        arr[mainArrayIndex++] = first[index1++];
    }

    while(index2 < len2 ) {
        arr[mainArrayIndex++] = second[index2++];
    }

    delete []first;
    delete []second;

}

void mergeSort(int *arr, int s, int e) {

    //base case
    if(s >= e) {
        return;
    }
    int mid = (s+e)/2;

    //left part sort karna h
    mergeSort(arr, s, mid);

    //right part sort karna h
    mergeSort(arr, mid+1, e);

    //merge
    merge(arr, s, e);

}

int main() {

    int arr[15] = {3,7,0,1,5,8,3,2,34,66,87,23,12,12,12};
    int n = 15;

    mergeSort(arr, 0, n-1);

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

    return 0;
}
```



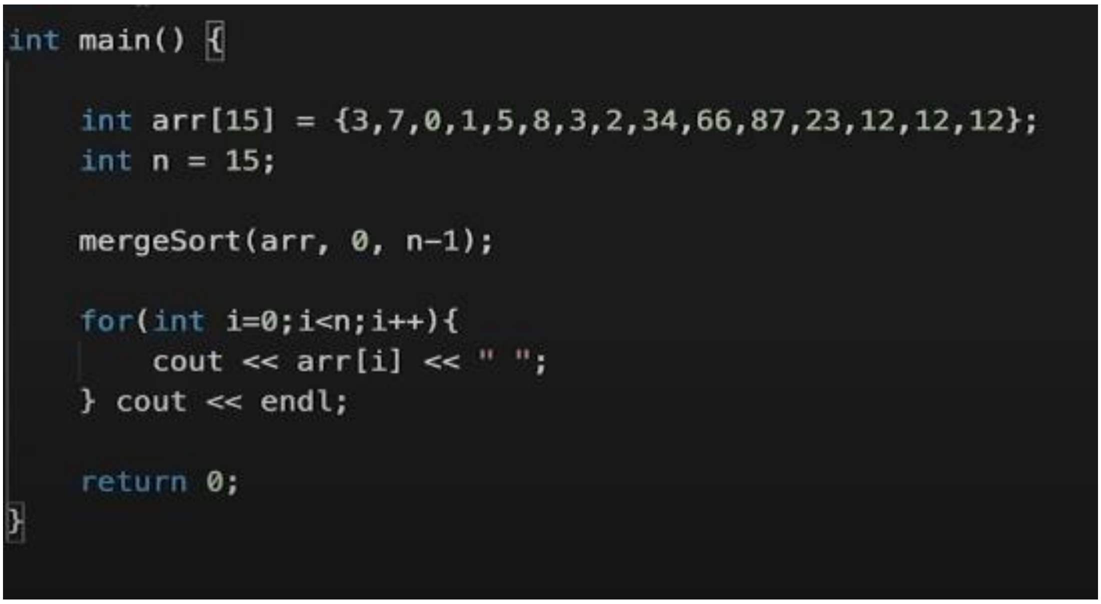
```cpp
using namespace std;

void merge(int *arr, int s, int e) {

    int mid = (s+e)/2;

    int len1 = mid - s + 1;
    int len2 = e - mid;

    int *first = new int[len1];
    int *second = new int[len2];

    //copy values
    int mainArrayIndex = s;
    for(int i=0; i<len1; i++) {
        first[i] = arr[mainArrayIndex++];
    }

    mainArrayIndex = mid+1;
    for(int i=0; i<len2; i++) {
        second[i] = arr[mainArrayIndex++];
    }

    //merge 2 sorted arrays
    int index1 = 0;
    int index2 = 0;
```



```
    //merge 2 sorted arrays
    int index1 = 0;
    int index2 = 0;
    mainArrayIndex = s;

    while(index1 < len1 && index2 < len2) {
        if(first[index1] < second[index2]) {
            arr[mainArrayIndex++] = first[index1++];
        }
        else{
            arr[mainArrayIndex++] = second[index2++];
        }
    }

    while(index1 < len1) {
        arr[mainArrayIndex++] = first[index1++];
    }

    while(index2 < len2 ) {
        arr[mainArrayIndex++] = second[index2++];
    }
}
```



```
void mergeSort(int *arr, int s, int e) {

    //base case
    if(s >= e) {
        return;
    }

    int mid = (s+e)/2;

    //left part sort karna h
    mergeSort(arr, s, mid);

    //right part sort karna h
    mergeSort(arr, mid+1, e);

    //merge
    merge(arr, s, e);

}

int main() {

    int arr[15] = {3,7,0,1,5,8,3,2,34,66,87,23,12,12,12};
    int n = 15;

    mergeSort(arr, 0, n-1);

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



```
int main() {

    int arr[15] = {3,7,0,1,5,8,3,2,34,66,87,23,12,12,12};
    int n = 15;

    mergeSort(arr, 0, n-1);

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

    return 0;
}
```

QUICK SORT

In the qiock sort we will take the 0 element and place in the middle as in the middle how ? pivot element = = 0th index eelment

We will count the element small from the our pivot element and then cout it and after we know how many small element it has so we will place our pivot element after that index . ex=3==pivot elemnt in my array from 3 small element is 2 I have count thata so in thatway only I will place my 3 after 2 index so all the small element can come here oke then I will sort the element from pivot as on the left siode there will be small element and on the right there will be greater element from the pivot so we wil;l sort the array by I and j taken and swap the element int this we will done a quick sort now we have come the right and left will call the quick sort then that part will sort and combine then our whole have things done

```
#include<iostream>

using namespace std;



int partition( int arr[], int s, int e) {


  int pivot = arr[s];


  int cnt = 0;

  for(int i = s+1; i<=e; i++) {

    if(arr[i] <=pivot) {

      cnt++;

    }

  }


  //place pivot at right position

  int pivotIndex = s + cnt;

  swap(arr[pivotIndex], arr[s]);
```

```
    //left and right wala part smbhal lete h
    int i = s, j = e;

    while(i < pivotIndex && j > pivotIndex) {

        while(arr[i] <= pivot)
        {
            i++;
        }

        while(arr[j] > pivot) {
            j--;
        }

        if(i < pivotIndex && j > pivotIndex) {
            swap(arr[i++], arr[j--]);
        }

    }

    return pivotIndex;

}

void quickSort(int arr[], int s, int e) {

    //base case
    if(s >= e)
        return ;

    //partitioon karenfe
    int p = partition(arr, s, e);

    //left part sort karo
    quickSort(arr, s, p-1);

    //right wala part sort karo
    quickSort(arr, p+1, e);

}

int main() {

    int arr[10] = {2,4,1,6,9 ,9,9,9,9,9};
    int n = 10;

    quickSort(arr, 0, n-1);

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


    return 0;
```

}

OOPS

OBJECT ORIENTED PROGRAM –BOTTOM up approach

Classes and object

POP

Top down approach

Functions

1. CLASSES==Blueprint of an object
   It reprsents a set of properties or methods common to all objects of a same time
   Example == Think of a **class like a car blueprint** — it defines what a car *should have* (like wheels, engine, etc.) but **does not create a car** by itself.

2. Object = An **object** is a **real-world instance** of a class. It has state and behaviour

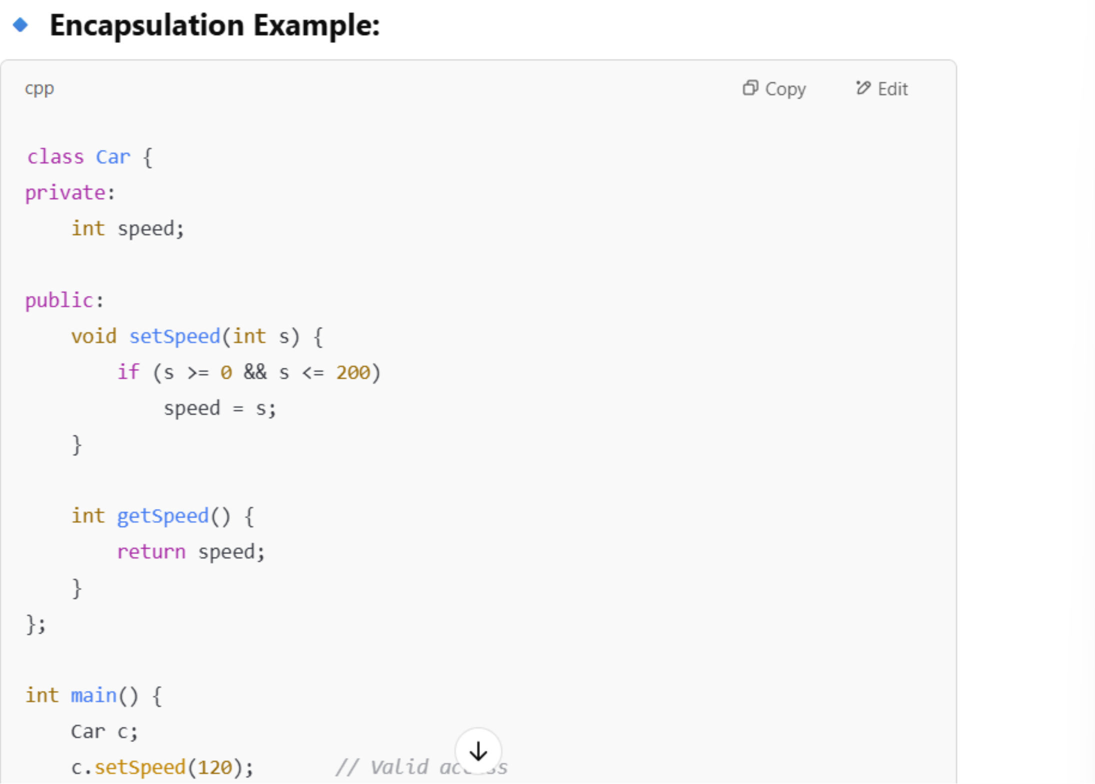
   It contains **actual values** for the properties and can use the methods defined in the class.

   Continuing the car example — the **real physical car** that you drive is an object based on the class (blueprint).

3. Features of Oops
   4 pillars of oops

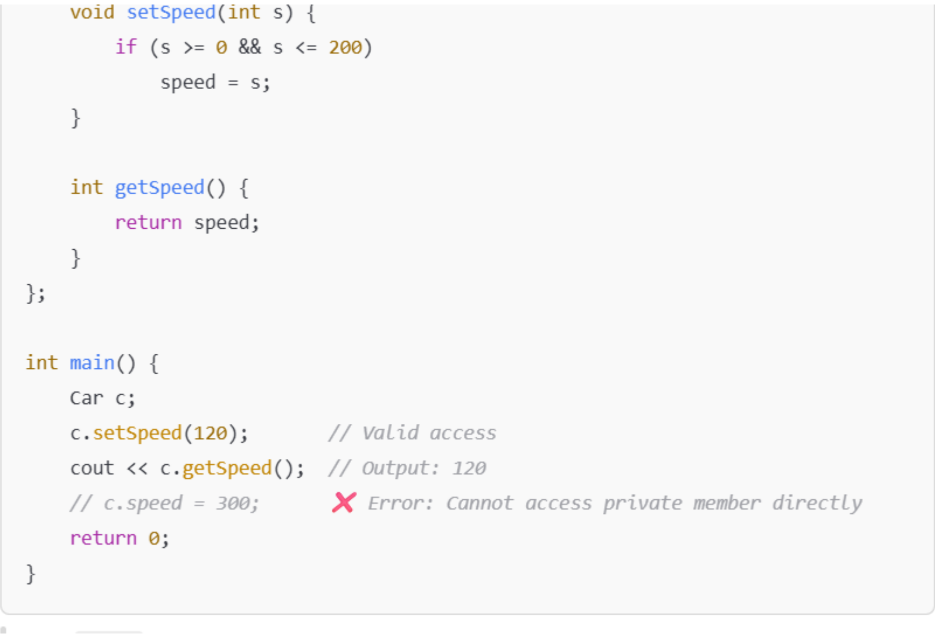
1.**Encapsulation**:wrapping of data member and data function in a singe unit.

In incapsulation just we can simply say that data is we can use the function but we do not no how it is working for

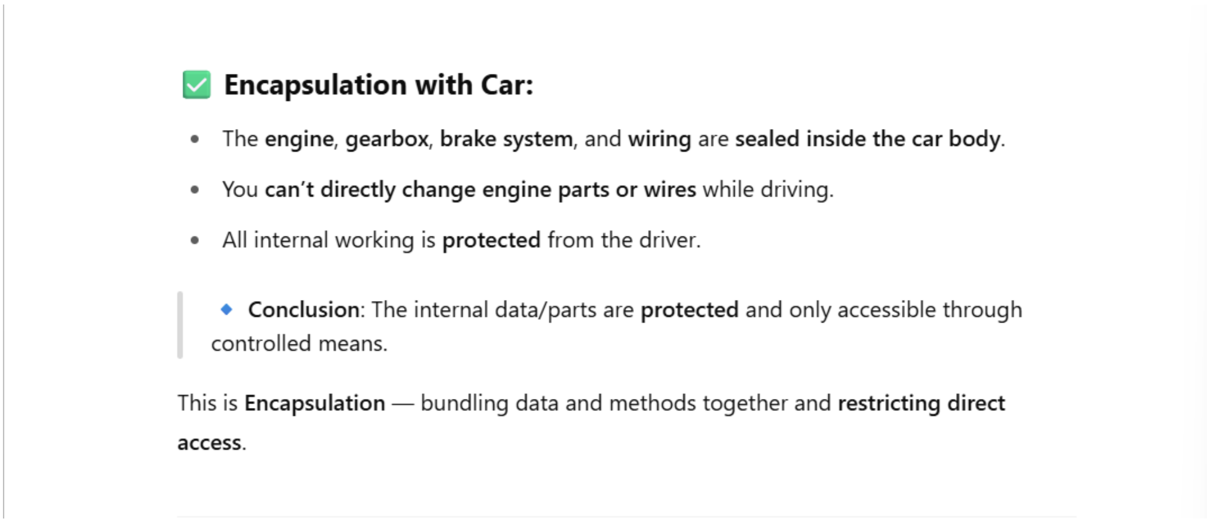


◆ **Encapsulation Example:**

```cpp
class Car {
private:
    int speed;

public:
    void setSpeed(int s) {
        if (s >= 0 && s <= 200)
            speed = s;
    }

    int getSpeed() {
        return speed;
    }
};

int main() {
    Car c;
    c.setSpeed(120);       // Valid ac  ss
```



```cpp
    void setSpeed(int s) {
        if (s >= 0 && s <= 200)
            speed = s;
    }

    int getSpeed() {
        return speed;
    }
};

int main() {
    Car c;
    c.setSpeed(120);       // Valid access
    cout << c.getSpeed();  // Output: 120
    // c.speed = 300;      ❌ Error: Cannot access private member directly
    return 0;
}
```



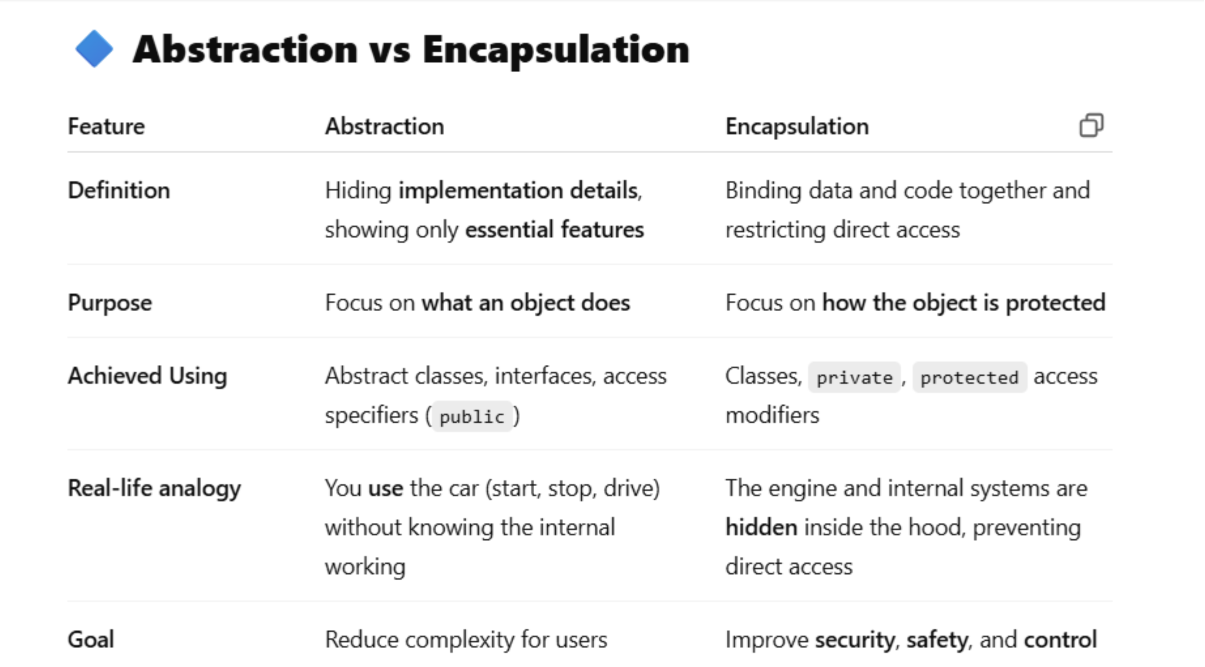
✅ Encapsulation with Car:

- The **engine**, **gearbox**, **brake system**, and **wiring** are **sealed inside the car body**.
- You **can't directly change engine parts or wires** while driving.
- All internal working is **protected** from the driver.

> 🔹 **Conclusion**: The internal data/parts are **protected** and only accessible through controlled means.

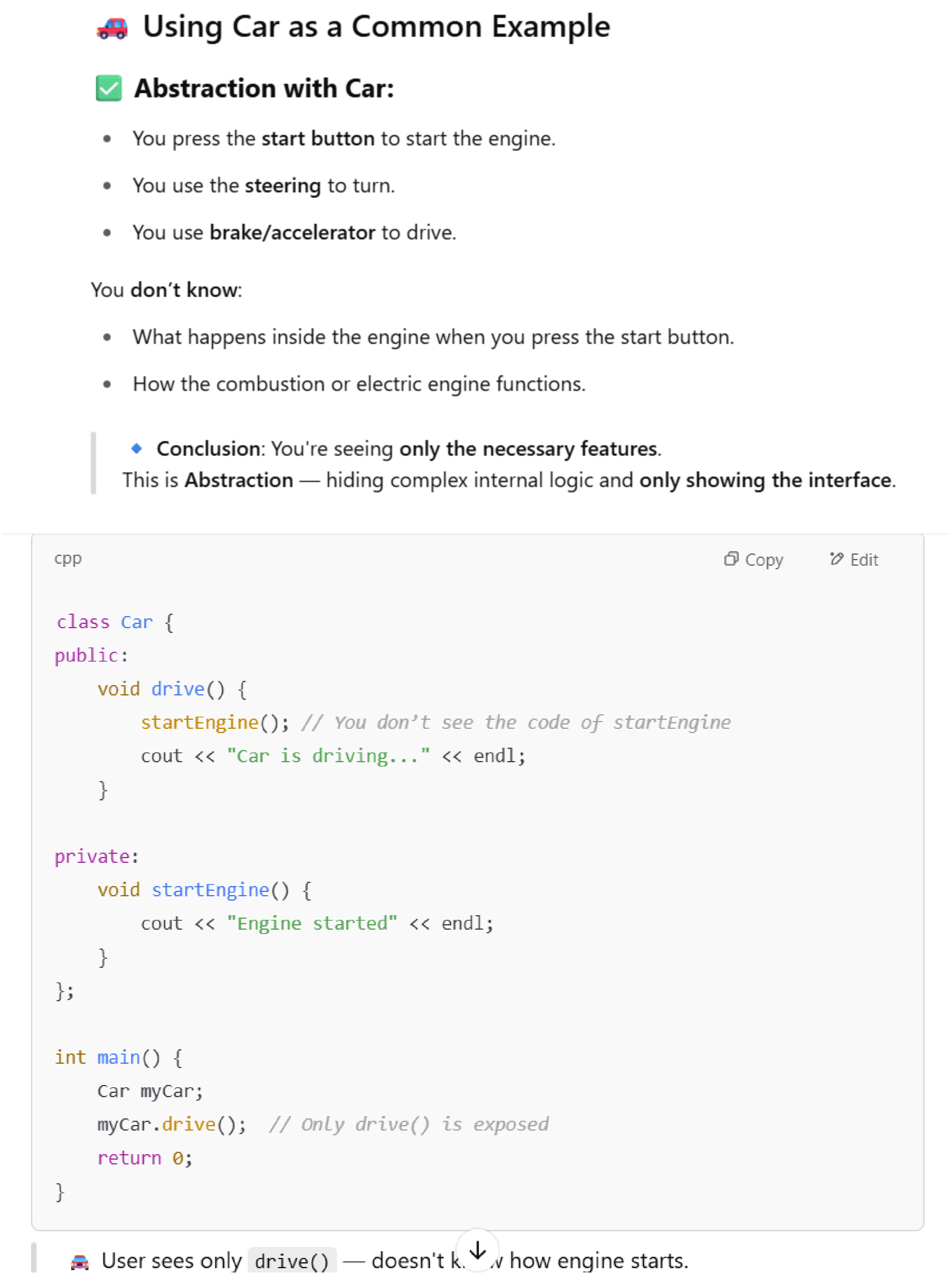
This is **Encapsulation** — bundling data and methods together and **restricting direct access**.

Abstarction : Hide implementation details and shoeing only essential features.



🔷 Abstraction vs Encapsulation

Feature	Abstraction	Encapsulation
Definition	Hiding **implementation details**, showing only **essential features**	Binding data and code together and restricting direct access
Purpose	Focus on **what an object does**	Focus on **how the object is protected**
Achieved Using	Abstract classes, interfaces, access specifiers (`public`)	Classes, `private` , `protected` access modifiers
Real-life analogy	You **use** the car (start, stop, drive) without knowing the internal working	The engine and internal systems are **hidden** inside the hood, preventing direct access
Goal	Reduce complexity for users	Improve **security**, **safety**, and **control**



🚗 Using Car as a Common Example

✅ Abstraction with Car:

- You press the **start button** to start the engine.
- You use the **steering** to turn.
- You use **brake/accelerator** to drive.

You **don't know**:

- What happens inside the engine when you press the start button.
- How the combustion or electric engine functions.

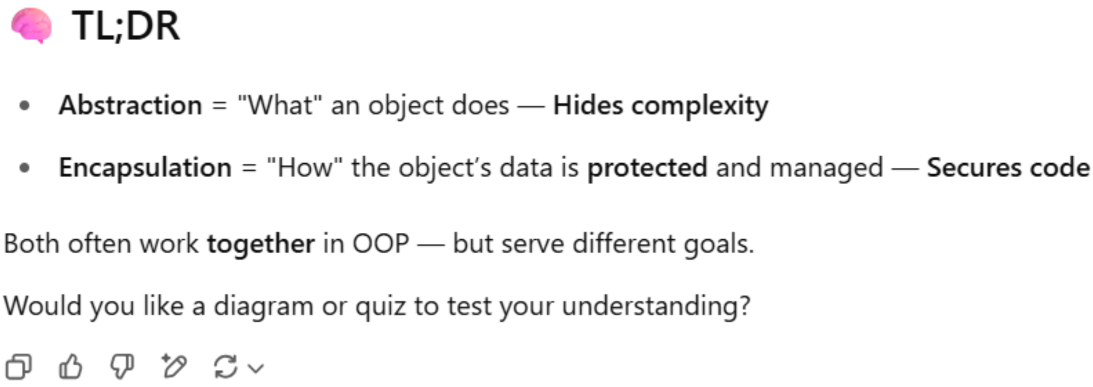
> 🔹 **Conclusion**: You're seeing **only the necessary features**.
> This is **Abstraction** — hiding complex internal logic and **only showing the interface**.

```cpp
class Car {
public:
    void drive() {
        startEngine(); // You don't see the code of startEngine
        cout << "Car is driving..." << endl;
    }

private:
    void startEngine() {
        cout << "Engine started" << endl;
    }
};

int main() {
    Car myCar;
    myCar.drive();  // Only drive() is exposed
    return 0;
}
```

> 🚘 User sees only `drive()` — doesn't k↓w how engine starts.

So this is called abstarctipon

Main difference



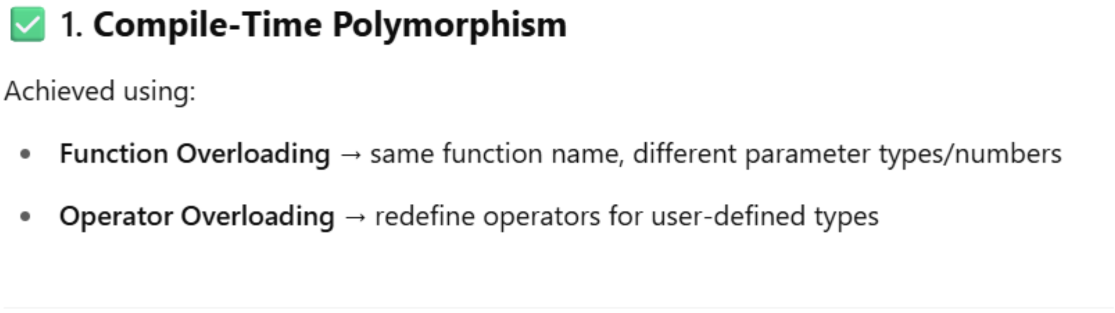
🧠 TL;DR

- **Abstraction** = "What" an object does — **Hides complexity**
- **Encapsulation** = "How" the object's data is **protected** and managed — **Secures code**

Both often work **together** in OOP — but serve different goals.

Would you like a diagram or quiz to test your understanding?

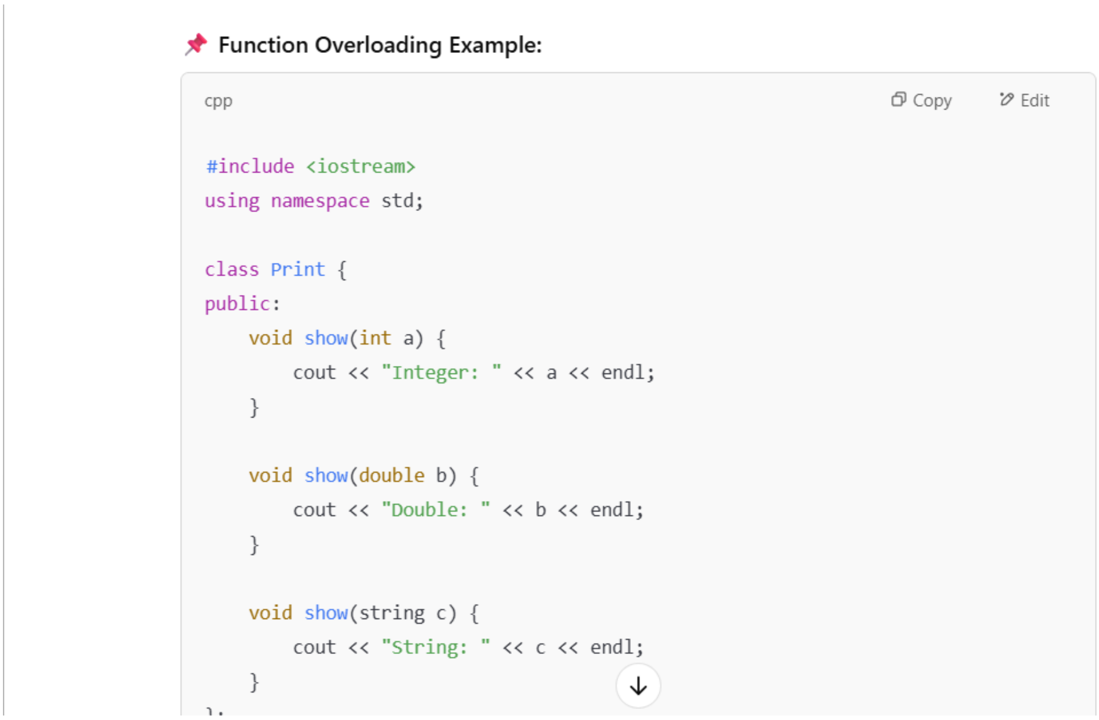
2. **Polymorphisim:** exist in Many forms



✅ 1. Compile-Time Polymorphism

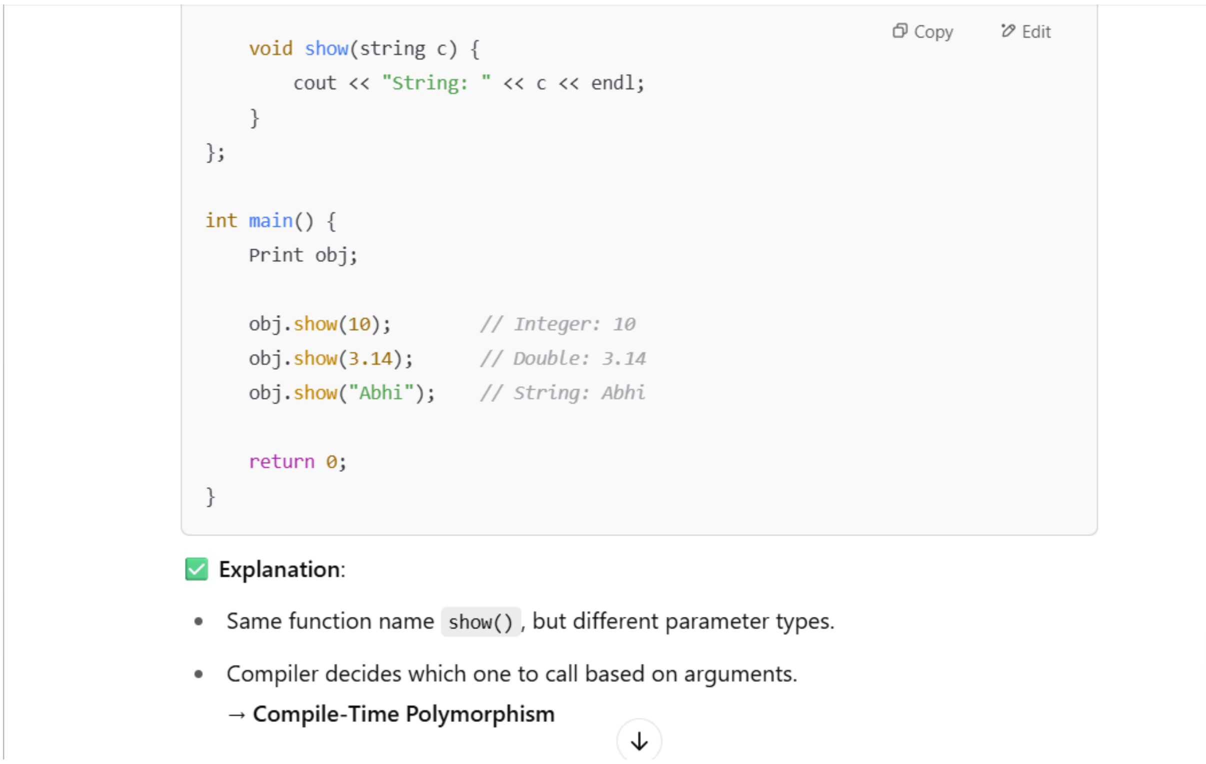
Achieved using:

- **Function Overloading** → same function name, different parameter types/numbers
- **Operator Overloading** → redefine operators for user-defined types



📌 **Function Overloading Example:**

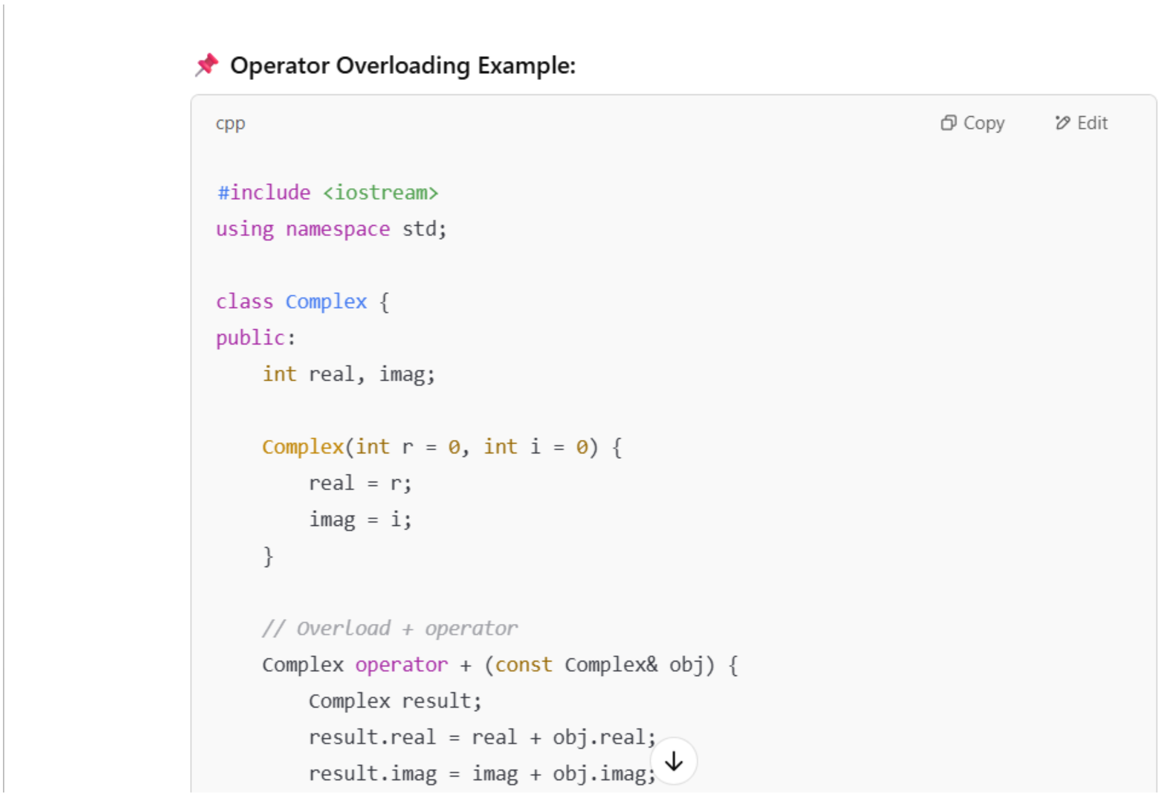
```cpp
#include <iostream>
using namespace std;

class Print {
public:
    void show(int a) {
        cout << "Integer: " << a << endl;
    }

    void show(double b) {
        cout << "Double: " << b << endl;
    }

    void show(string c) {
        cout << "String: " << c << endl;
    }
};
```



```cpp
    void show(string c) {
        cout << "String: " << c << endl;
    }
};

int main() {
    Print obj;

    obj.show(10);        // Integer: 10
    obj.show(3.14);      // Double: 3.14
    obj.show("Abhi");    // String: Abhi

    return 0;
}
```

✅ **Explanation:**

- Same function name `show()`, but different parameter types.
- Compiler decides which one to call based on arguments.
  → **Compile-Time Polymorphism**



📌 **Operator Overloading Example:**

```cpp
#include <iostream>
using namespace std;

class Complex {
public:
    int real, imag;

    Complex(int r = 0, int i = 0) {
        real = r;
        imag = i;
    }

    // Overload + operator
    Complex operator + (const Complex& obj) {
        Complex result;
        result.real = real + obj.real;
        result.imag = imag + obj.imag;
```

Copy Edit

```
        return result;
    }

    void display() {
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex c1(2, 3), c2(1, 7);
    Complex c3 = c1 + c2;  // Operator Overloading in action
    c3.display();          // Output: 3 + 10i

    return 0;
}
```

✅ **Explanation:**

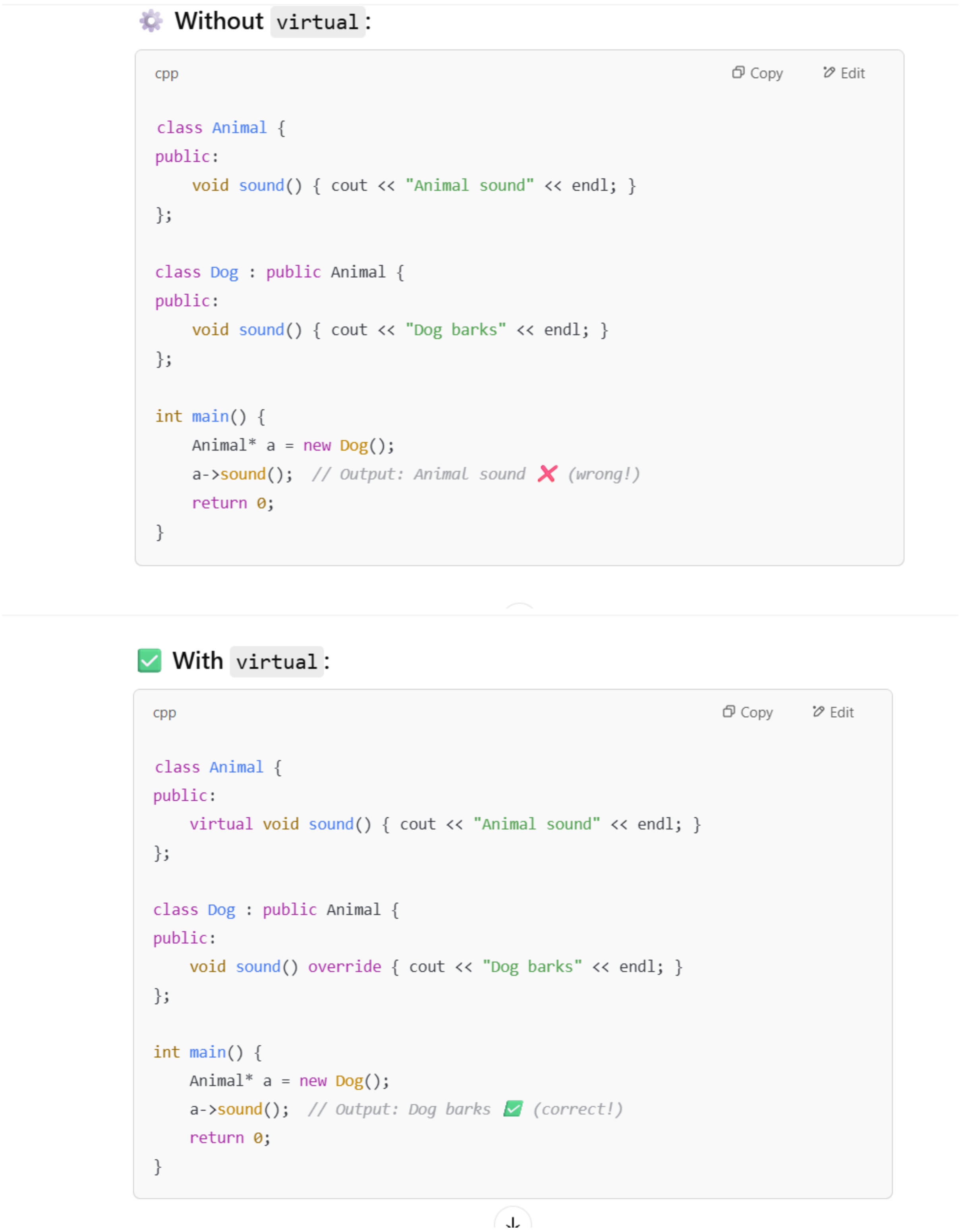
- We redefined how `+` works for user-defined `Complex` objects.
- This is **operator overloading**, a part of **compile-time polymorphism**.

✅ 2. **Run-Time Polymorphism**

Achieved using:

- **Function Overriding**
- **Virtual Functions + Pointers/References to base class**

📌 **Run-Time Polymorphism Example (Using Virtual Functions):**



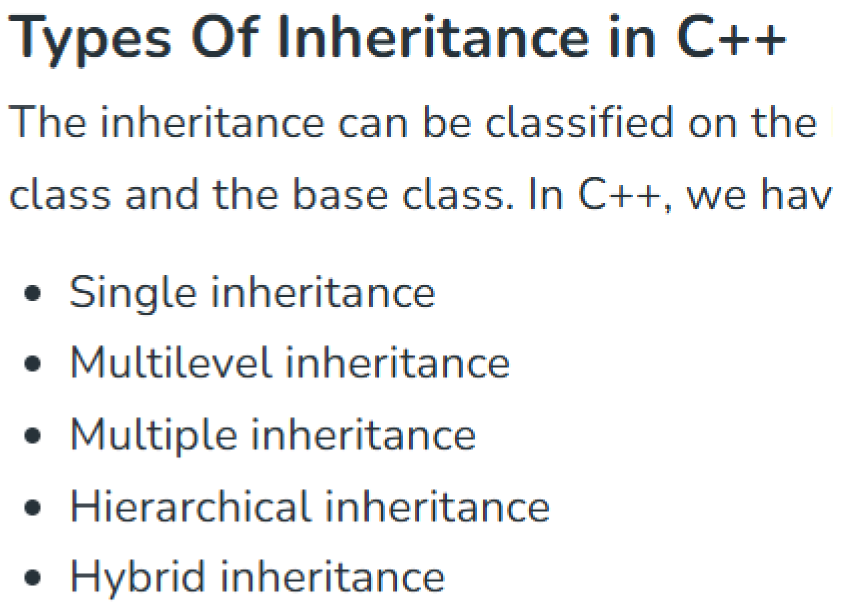
⚙️ Without `virtual`:

```cpp
class Animal {
public:
    void sound() { cout << "Animal sound" << endl; }
};

class Dog : public Animal {
public:
    void sound() { cout << "Dog barks" << endl; }
};

int main() {
    Animal* a = new Dog();
    a->sound();  // Output: Animal sound ❌ (wrong!)
    return 0;
}
```

✅ With `virtual`:

```cpp
class Animal {
public:
    virtual void sound() { cout << "Animal sound" << endl; }
};

class Dog : public Animal {
public:
    void sound() override { cout << "Dog barks" << endl; }
};

int main() {
    Animal* a = new Dog();
    a->sound();  // Output: Dog barks ✅ (correct!)
    return 0;
}
```

4. Inheritance : The capability of a class to access properties and characteristics from another class is called **Inheritance**.



Types Of Inheritance in C++

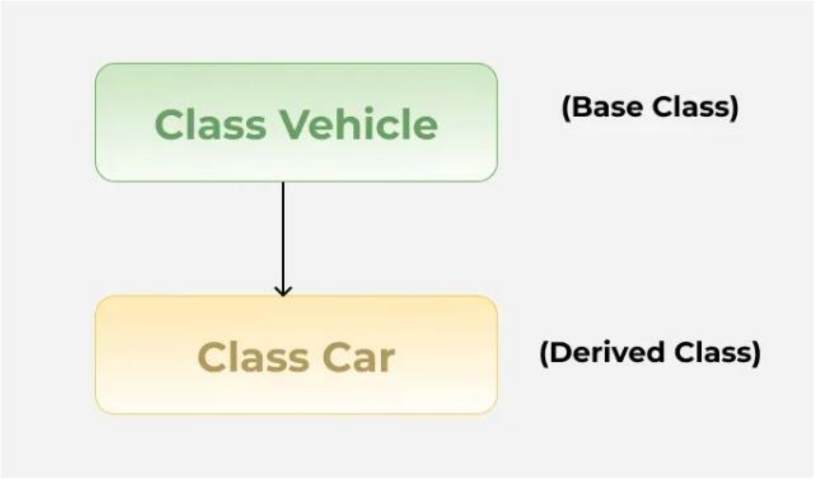
The inheritance can be classified on the

class and the base class. In C++, we hav

- Single inheritance
- Multilevel inheritance
- Multiple inheritance
- Hierarchical inheritance
- Hybrid inheritance

1. Single Inheritance

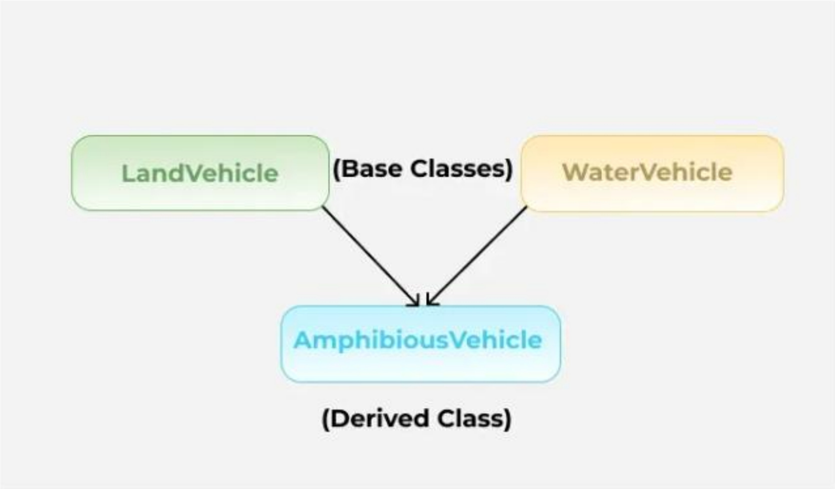
In single inheritance, a class is allowed to inherit from only one class. i.e. one base class is inherited by one derived class only.



Single Inheritance

2. Multiple Inheritance

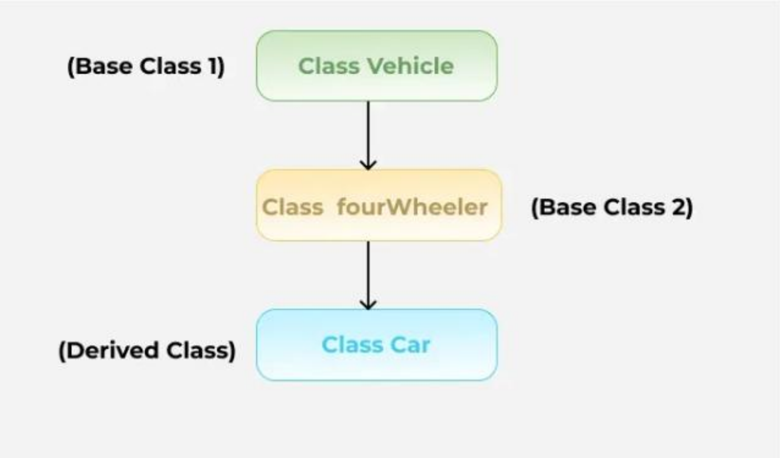
Multiple Inheritance is a feature of C++ where a class can inherit from more than one class. i.e one **subclass** is inherited from more than one **base class.**



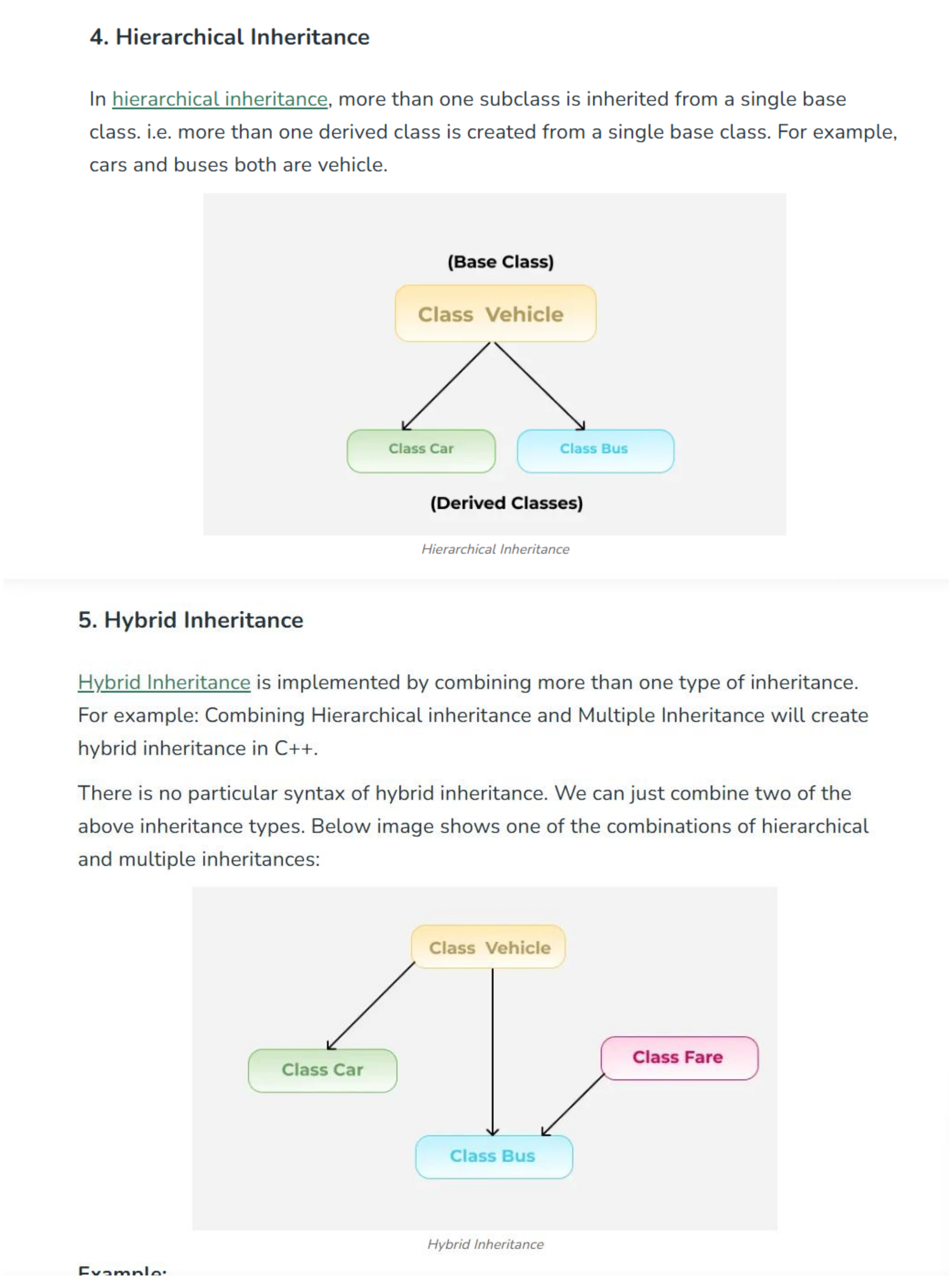
Multiple Inheritance

3. Multilevel Inheritance

In multilevel inheritance, a derived class is created from another derived class and that derived class can be derived from a base class or any other derived class. There can be any number of levels. For example, a vehicle can be a four-wheeler, and a four-wheeler vehicle can be a car.



Multilevel Inheritance



4. Hierarchical Inheritance

In hierarchical inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class. For example, cars and buses both are vehicle.

Hierarchical Inheritance

5. Hybrid Inheritance

Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance will create hybrid inheritance in C++.

There is no particular syntax of hybrid inheritance. We can just combine two of the above inheritance types. Below image shows one of the combinations of hierarchical and multiple inheritances:

Hybrid Inheritance

Example:

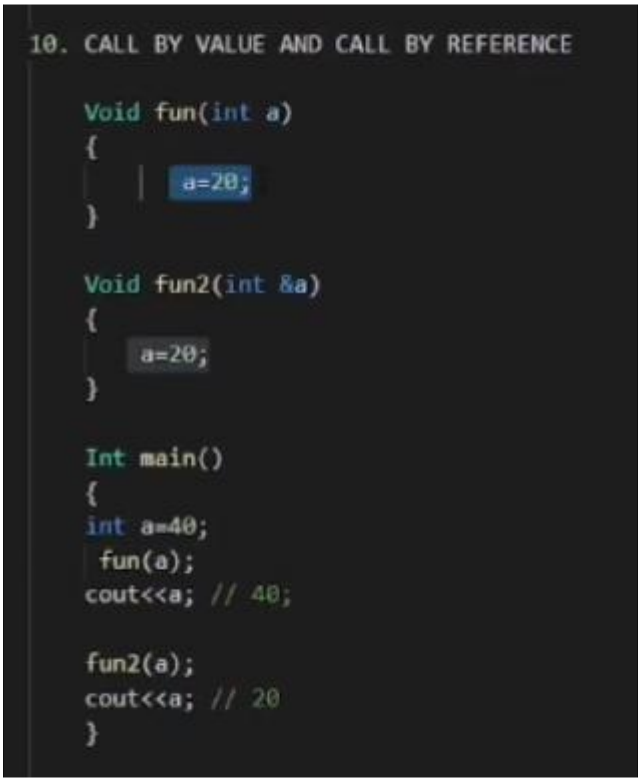
Modes of inheritabce , public , private , protected

Static Variable : as if we have made the static int a =7;

Then we call the object all the object will share this one menas ythere is no copy made by the ibjet they hasve to share this one only

Friend function: friend can access the private and protected it reeceives an ovject as an parameters

Call by value and call by reference :

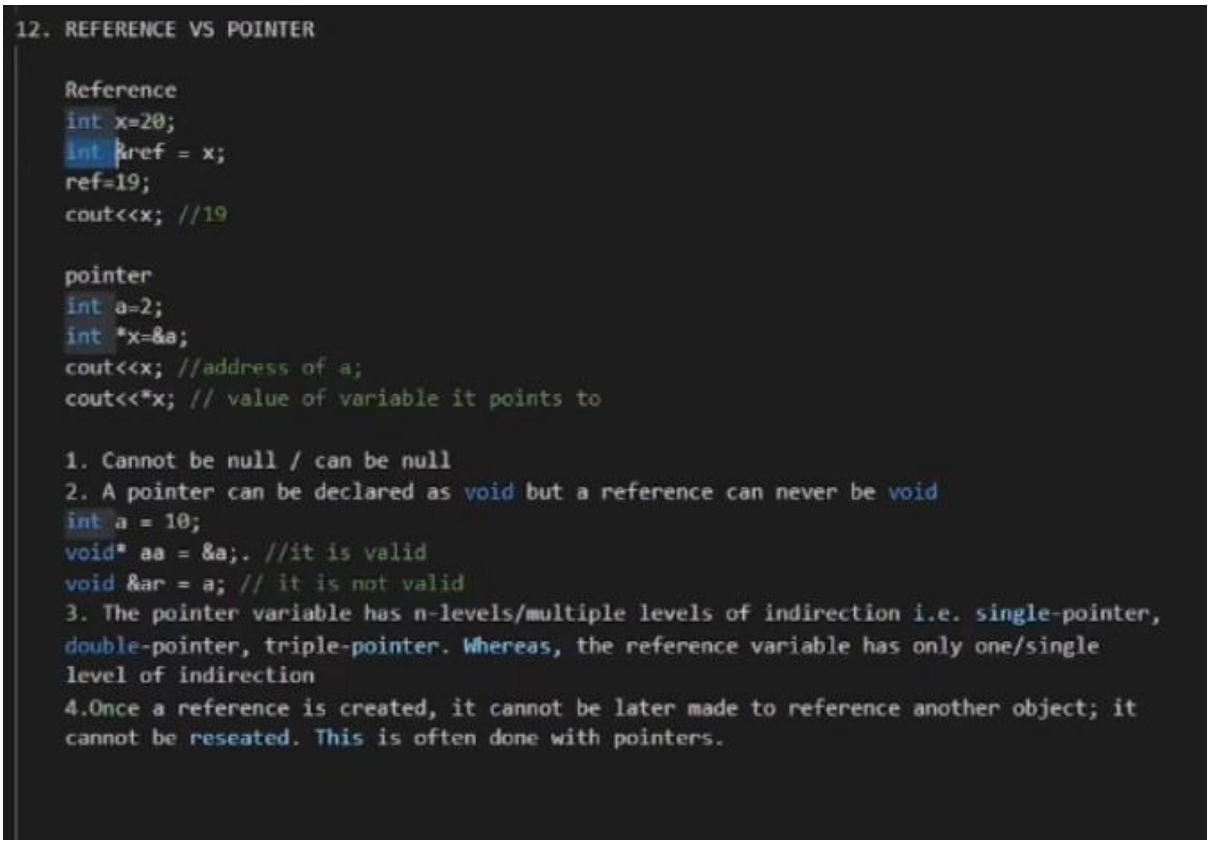


```
10. CALL BY VALUE AND CALL BY REFERENCE

    Void fun(int a)
    {
          a=20;
    }

    Void fun2(int &a)
    {
        a=20;
    }

    Int main()
    {
    int a=40;
     fun(a);
    cout<<a; // 40;

    fun2(a);
    cout<<a; // 20
    }
```

Reference and pointer==

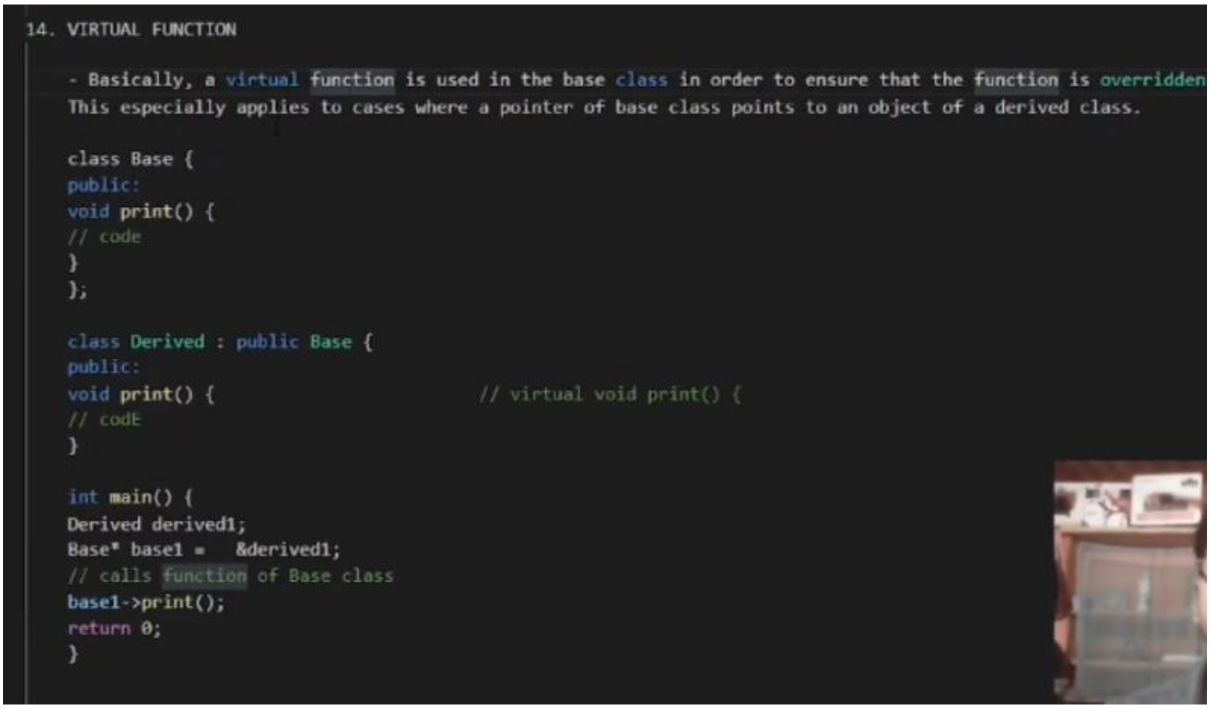
Reference is the another name of the variable ,can not be null,can not be void

Pointer store the address of he variable , can be null , an be void



```
12. REFERENCE VS POINTER

    Reference
    int x=20;
    int &ref = x;
    ref=19;
    cout<<x; //19

    pointer
    int a=2;
    int *x=&a;
    cout<<x; //address of a;
    cout<<*x; // value of variable it points to

    1. Cannot be null / can be null
    2. A pointer can be declared as void but a reference can never be void
    int a = 10;
    void* aa = &a;. //it is valid
    void &ar = a; // it is not valid
    3. The pointer variable has n-levels/multiple levels of indirection i.e. single-pointer,
    double-pointer, triple-pointer. Whereas, the reference variable has only one/single
    level of indirection
    4.Once a reference is created, it cannot be later made to reference another object; it
    cannot be reseated. This is often done with pointers.
```

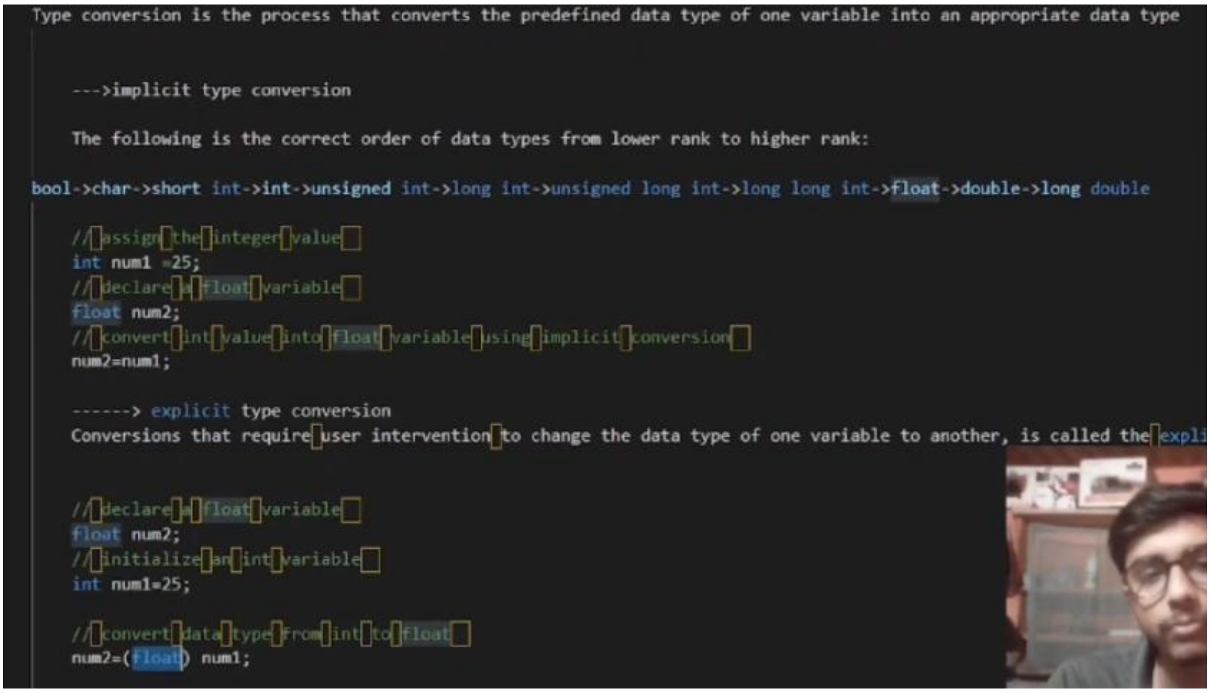
Virtual function



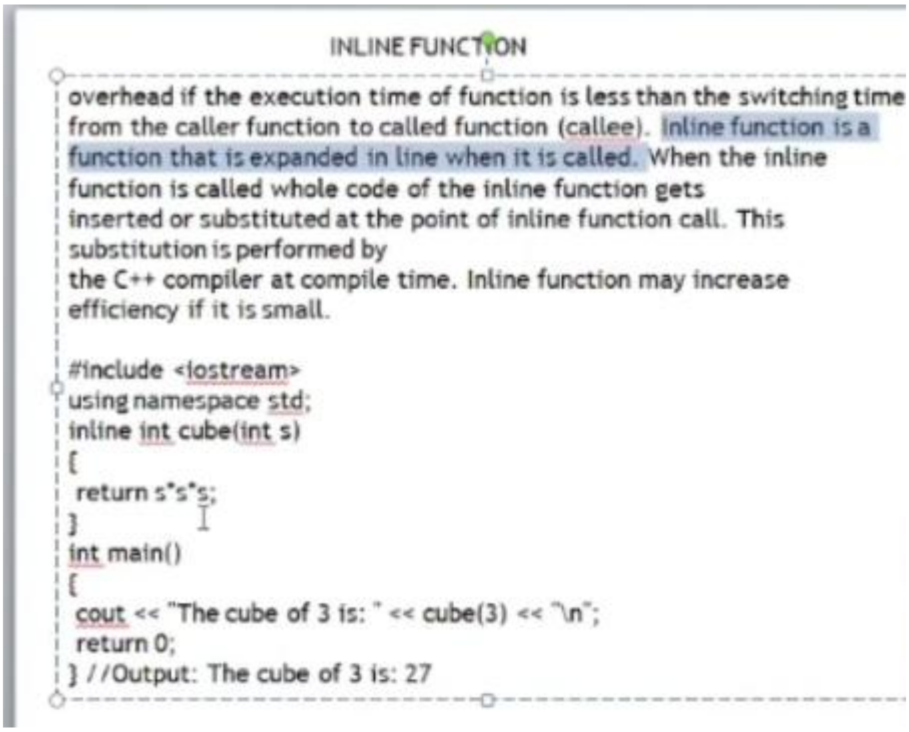
```
14. VIRTUAL FUNCTION

    - Basically, a virtual function is used in the base class in order to ensure that the function is overridden
    This especially applies to cases where a pointer of base class points to an object of a derived class.

    class Base {
    public:
    void print() {
    // code
    }
    };

    class Derived : public Base {
    public:
    void print() {                     // virtual void print() {
    // codE
    }

    int main() {
    Derived derived1;
    Base* base1 =   &derived1;
    // calls function of Base class
    base1->print();
    return 0;
    }
```

Type of conversion: implicit and explicit

Implicit: as the small data type can be put in higher



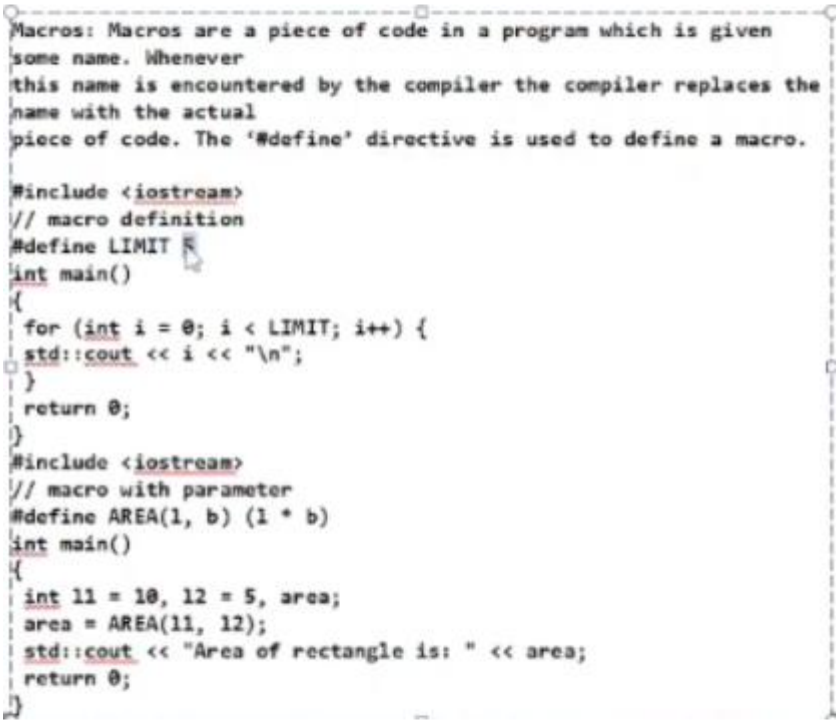
```
 Type conversion is the process that converts the predefined data type of one variable into an appropriate data type


    --->implicit type conversion

    The following is the correct order of data types from lower rank to higher rank:

bool->char->short int->int->unsigned int->long int->unsigned long int->long long int->float->double->long double

    // assign the integer value 
    int num1 =25;
    // declare a float variable 
    float num2;
    // convert int value into float variable using implicit conversion 
    num2=num1;

    ------> explicit type conversion
    Conversions that require user intervention to change the data type of one variable to another, is called the expli


    // declare a float variable 
    float num2;
    // initialize an int variable 
    int num1=25;

    // convert data type from int to float 
    num2=(float) num1;
```



INLINE FUNCTION

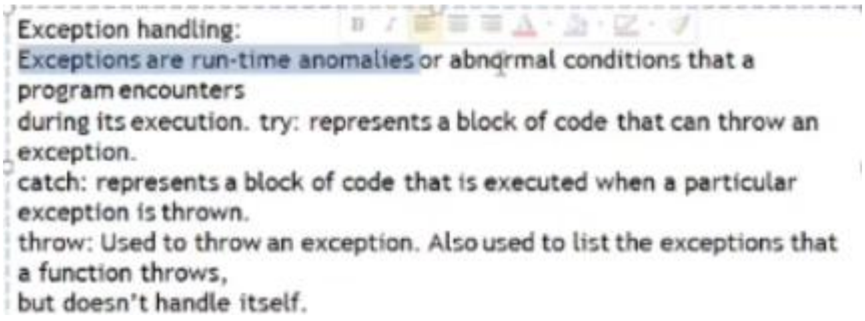
overhead if the execution time of function is less than the switching time from the caller function to called function (callee). Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time. Inline function may increase efficiency if it is small.

```
#include <iostream>
using namespace std;
inline int cube(int s)
{
 return s*s*s;
}
int main()
{
 cout << "The cube of 3 is: " << cube(3) << "\n";
 return 0;
} //Output: The cube of 3 is: 27
```



Macros: Macros are a piece of code in a program which is given some name. Whenever this name is encountered by the compiler the compiler replaces the name with the actual piece of code. The '#define' directive is used to define a macro.

```
#include <iostream>
// macro definition
#define LIMIT 5
int main()
{
 for (int i = 0; i < LIMIT; i++) {
 std::cout << i << "\n";
 }
 return 0;
}
#include <iostream>
// macro with parameter
#define AREA(l, b) (l * b)
int main()
{
 int l1 = 10, l2 = 5, area;
 area = AREA(l1, l2);
 std::cout << "Area of rectangle is: " << area;
 return 0;
}
```



Exception handling:

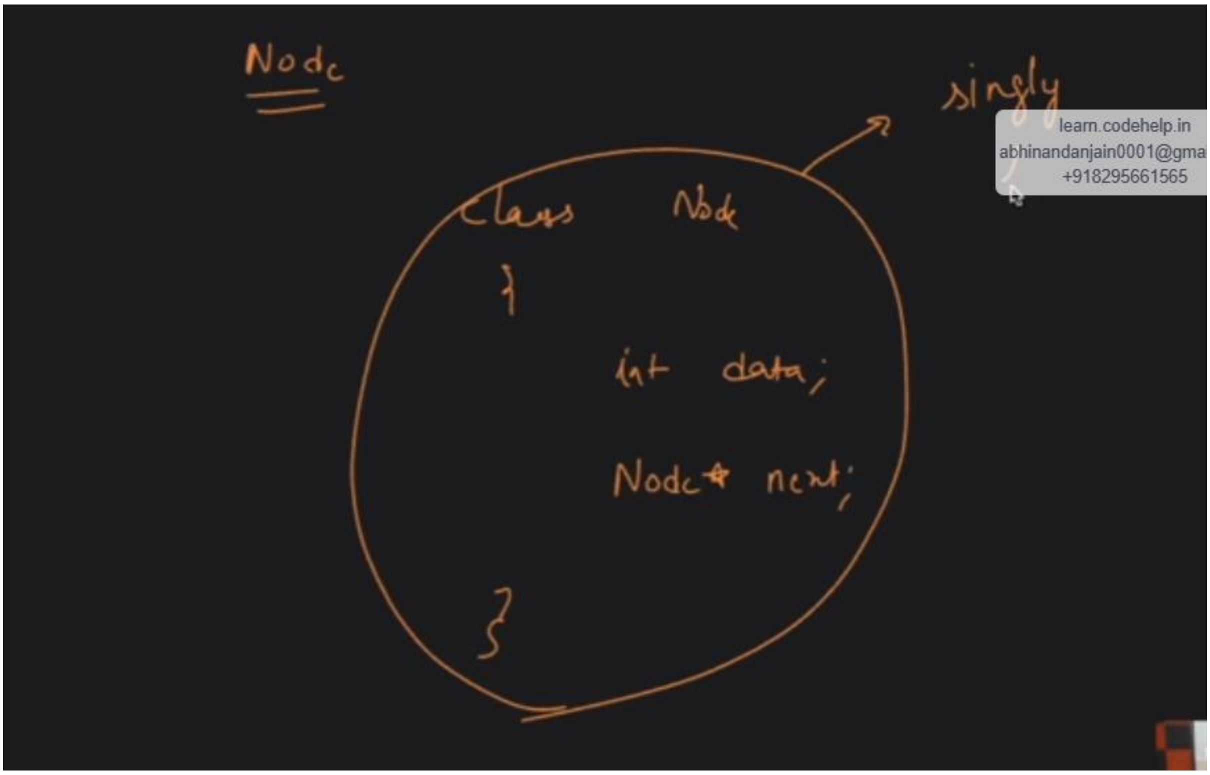
Exceptions are run-time anomalies or abnormal conditions that a program encounters during its execution. try: represents a block of code that can throw an exception.

catch: represents a block of code that is executed when a particular exception is thrown.

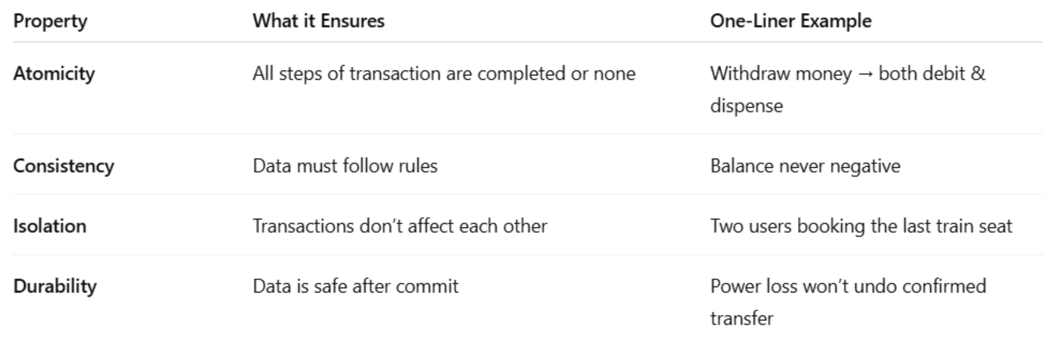
throw: Used to throw an exception. Also used to list the exceptions that a function throws, but doesn't handle itself.

LINKED LIST

Singly linked list has we can create the node in this side



```
Class    Node
{
      int  data;
      Node*  next;
}
```



Property	What it Ensures	One-Liner Example
Atomicity	All steps of transaction are completed or none	Withdraw money → both debit & dispense
Consistency	Data must follow rules	Balance never negative
Isolation	Transactions don’t affect each other	Two users booking the last train seat
Durability	Data is safe after commit	Power loss won’t undo confirmed transfer

Normalization is the process of **organizing data** in a database to **reduceduplicay** and **improve data accuracy.**

Joins in SQL

Join is used to combine rows from two or more tables, based on a related column between them.

Types of Joins

Inner Join Left Join Right Join Full Join