Date

Memory Management.

→ Array in C++ are made in stacks.

→ If ~~int~~ I write array in CPP → ① It will have garbage value by default. ② ~~A loop of~~ it never gets out of Bo

→ Whenever we pass arr, we have to pass its starting ~~index~~ address.

eg void test (int * arr)

If we do this, we would also have to pass size of array as well.

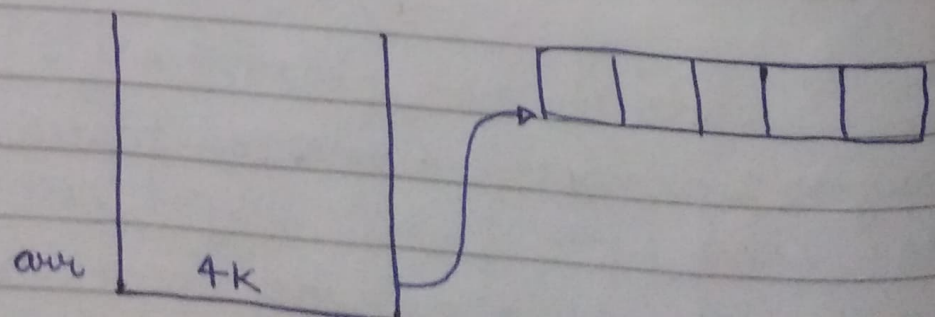
→ It is problematic because, we should make least use of stack (as it is imp.)

→ TO USE IN HEAP →

```
void test() {
    int n;
    cin >> n;
int* arr = new int[n];
}
```

[space allocation in heap.

* '*' - is used to store address. So int* means storing address of int type



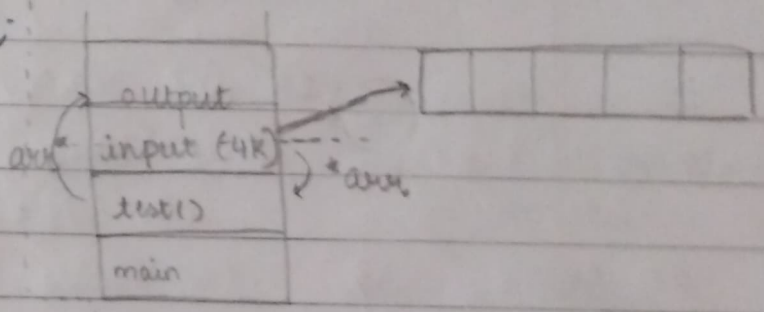
To take input of heap formed array

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

⇒

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

→ Khud ki input lega. Khud hi array banaega. bas call krdo func^ ko.



arr - acts a link.

IMP :- ARRAY ON STACK CANNOT BE RETURNED.

Now for stack :

→ won't work.

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

→ now this address cannot be returned because as soon as inputForStack is gone, array is go

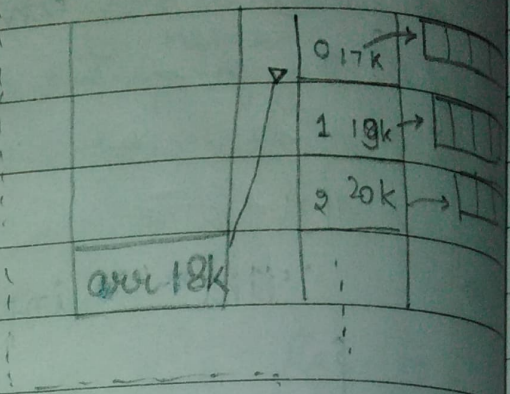
* There is col for every transaction from server. So its data (which is fetched from server). So it should be stored on heap.

Date

In 2-D.

we cannot use pointer method (we can, but its complicated) →

```
void test(){
    int n, m;
    cin >> n ;
    cin >> m ;
```




```
int ** arr = new int* [n]
```

adress

address of adress

array of address

```
for(int i=0 ; i<n ; i++)
    int* ar = new int[m];
    arr[i] = ar;
}
```

create array at every index of arr.

In 3-D :

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

- As the dimensions increase, this method of using heap becomes more and more complex and time consuming. So we were introduced with Vectors.

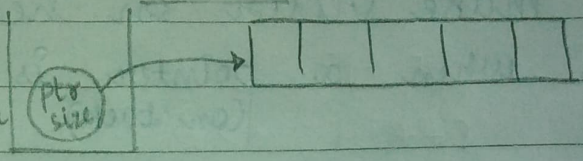
Page No.
Date

Vector: wrapper class which helps in doing these tasks in a more simplistic manner. So all this pointer work is done by itself in the background.

```
vector<int> arr(10,0);
for(int i=0; i<100; i++)   cout << arr[i] << endl;
```

⇒ THIS LOOP RUNS EVEN IT IS LARGER THAN SIZE OF VECTOR WITHOUT ANY ERROR. (To be saved, use .at())

⇒ It has both ptr and size variables.



copy constructor

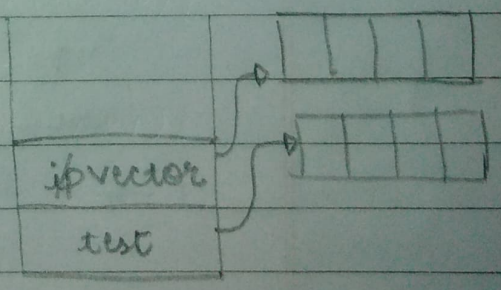
⇒ Copy constructor says, whenever anything is passed in form of '=' or return or as argument then copy constr-uctor is ~~passed~~ fired.

⇒ So when I write a = b, then data in a doesn't change rather it copies.

⇒ So, in vector

```
void inputVector(vector<int> arr){
  for(int i=0; i<arr.size(); i++){
    cin >> arr[i];
    for(int i=0; i<arr.size; i++){
    cout << arr[i]; }
```

→ THIS WILL PRINT INPUT.




```
void test(){
  vector<int> arr(10,0);
  inputVector(arr) // NOT O(1) but O(n)
  for(int i=0; i<arr.size(); i++)
  { cout << arr[i] << " "; }
```

→ This will print 0 0 0

(explaination on next page)

in test()
When the vector^ is passed into inputVector() the vector is copied (new vector created)

why? because vector is formed on stack but array within vector is made on heap.

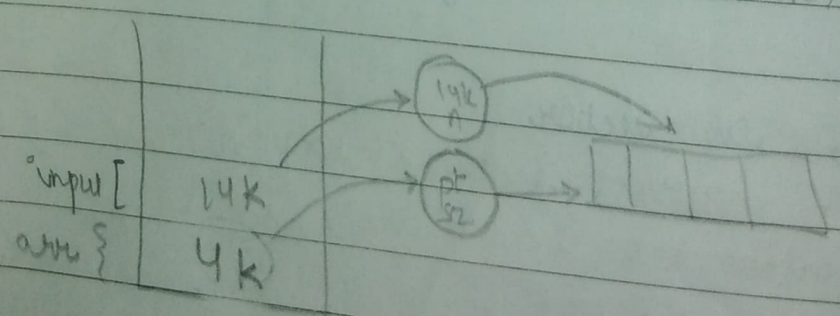
⇒ To make vector on heap:

NOTE: when a pointer (on stack) is copied then whatever the pointer points to will also be copied.

Making these changes to last program to form vector on heap.

void inputVector(vector<int>* arr)

In test(): vector<int>* arr = new vector<int>(10, 0);



Date

IMPORTANT.

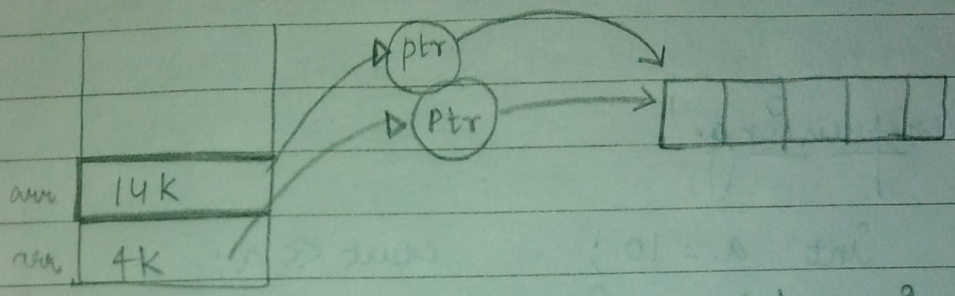
⇒ whenever using pointer use '->'. Dont use '.' operator

⇒ To access element, use arr->at(i); don't use []

In Java

Java created wrapper class to minimize use of pointers.

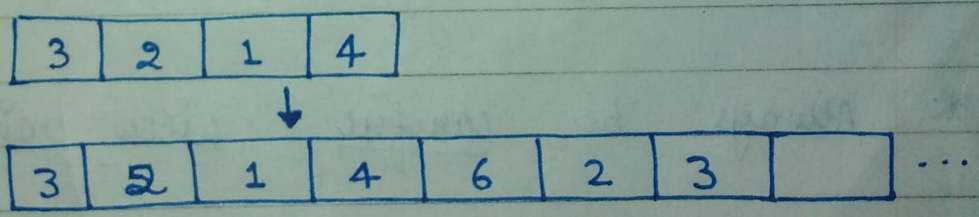
int[] arr = new int[10]



This seems as pass-by-reference but in reality copy constructor is fired.

Arraylist:

Arraylist doubles its size. copies older data and then fills the remaining list.



▷ &

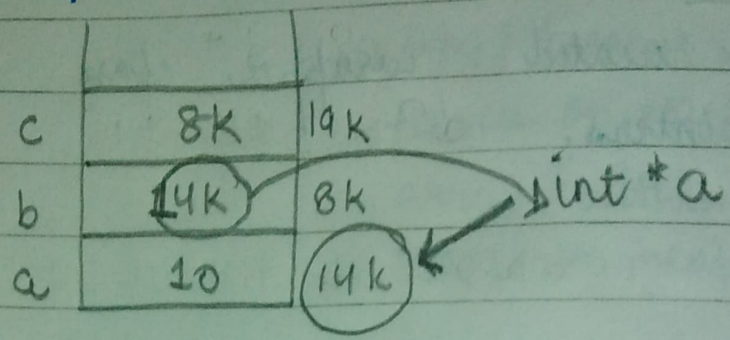
* & is used to retrieve address of any variable on stack.

```
int a = 10;
int *b = &a;
int **c = &b;
```

c	8k	19k
b	14k	8k
a	10	14k

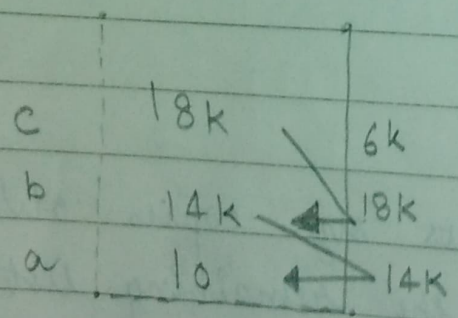
* '*' is used on heap.

⇒ & is a pointer itself that ~~copies~~ points or copies address of stack ! Then b copies ~~it~~ this pointer in itself .



⇒ Dereferencing.

```
int  a = 10;      cout << a;  // 10
int * b = &a;     cout << * b // 10
int **c = &b;     cout << **c; //10
```



* Always be careful with pointers .