Advantage of overriding:

→ Reusability factor.

Dynamic Object creation:

```
class People {
    public:
        void speak() {
            cout << "speaking" << endl;
        }
};
class Dog : public People {
    public:
        void speak() {
            cout << "Barking" << endl;
        }
};
int main() {
```

① `People *p1 = new People`
`p1 -> speak()` // it will display speaking.

② `Dog *d1 = new Dog`
`d1 -> speak()` // it will display barking.

this we have discussed in dynamic object creation.

③ `People *p2 = new Dog`.
`p2 -> speak();`
Parent class Pointer — object is of Derived class

→ It is said to be upcasting when parent class pointer point child class object

This will print "speaking"

* How to print Banking as output:

→ This can be done by virtual functions.
We need mark base class function as virtual so that it will print Banking.

④ Dog *d2 = new People ; ✗
d2 → speak();
↳ parent class object
↳ Derived class pointer

It will give error bcz compiler to compiler behaviour change so we will explicitly typecaste.

i.e Dog *d2 = (Dog*) new People ; It is also said Downcasting.
d2 → speak();

output: banking.

* So we have understood the 4 different cases:

Dynamic object creation in Inheritance

① People *p1 = new People();
*pointer *object
② Dog *d1 = new Dog();
③ People *p2 = new Dog();
④ Dog *d2 = (Dog*) new People();

In all 4 cases the Data members and Member function will depend on pointer type i.e on the left hand side

→ virtual keyword i.e virtual func

→ If you want to access the DM & MF of type object then we need to use virtual function i.e use virtual keyword at parents class Member function

* Now we understand 4 cases when we define constructor of each class.

* We know for both static & Dynamic memory constructor call itself.

Case 1:

```
class People{
    public:
        People(){
            cout<<" I'm inside People constructor.";
        }
};
class Dog : public People{
    public:
        Dog(){
            cout<<"I'm inside Dog constructor.";
        }
};
```

Case 1: `People *p1 = new People;`
Constructor call itself

output: I'm inside People Constructor.

Case 2: `Dog *d1 = new Dog.`

output: I'm inside People constructor
i'm inside Dog
] → why?

Because Dog is Derived class of People. so while creating object the parent constructor calls then child constructor calls. Thats why both prints.

Case 3: ~~Animal~~ `People *p2 = new Dog();`

Here object is of Dog class. so the construction of parent class and derived class will be called.
Parent class: People, Derived class: Dog.

Casting: Dog *d2 = (Dog*) new Animal.

explicit typecasting — object

As object is of ~~Animal~~ People class so the parent class construction will be called

* we basically use Upcasting rather than using down casting.

→ consider Animal *a = new Dog.

we know it is an Animal, but what the name of an Animal we don't know

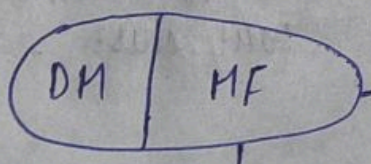
The actual animal is Dog. This is Implementation Hiding.

Ex: sort *a = new Quicksort();

we know we have sorting but what specific sorting may we have we have don't have idea.

We have the Quick sort implementation Hide.

* Abstraction: According to theoritical explanation. It is implementation hiding.

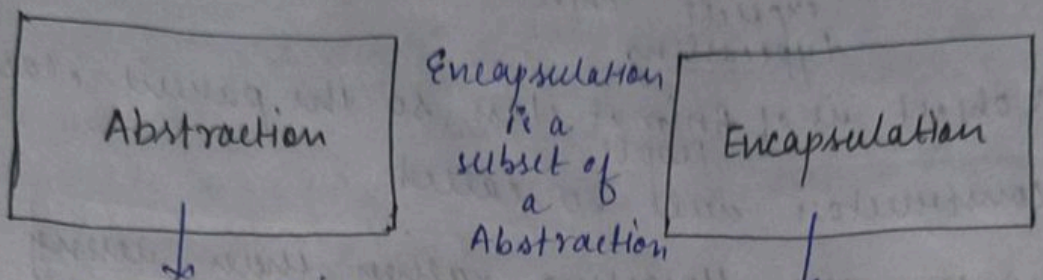


→ It was encapsulation wrap dm & mf.

we can also say that it is abstraction bez DM & MF are hidden inside an entity then implementation is hidden.

→ It is encapsulation as well as it is abstraction
→ In short we can say that Encapsulation is a subset of Abstraction.

*.

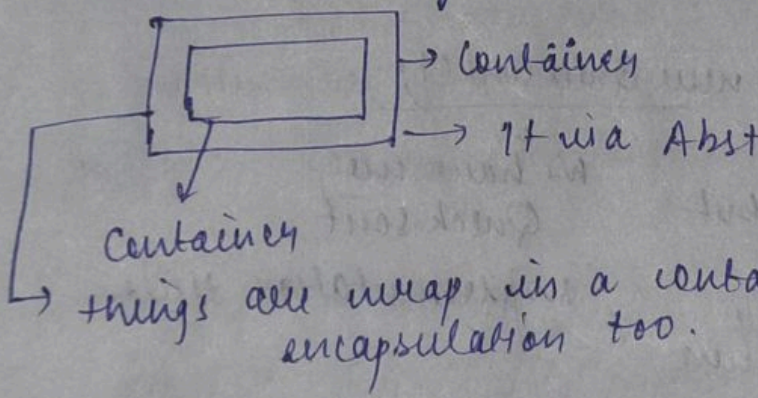


process which shows essential details. and hiding unnecessary things (i.e. Implementation Hiding)

wrap (DM/MF) in a class or Entity

perfect encapsulation. (marked all members under private)

* A container containing a container : It is a Abstraction



things are wrap in a container. so it is encapsulation too.

* sort * a = new Quicksort(); → sort * a is also an encapsulation bez sorting technique must to written inside sort class.

↓ generic sort which an user have. left part doesn't know whether we have use Quick, Heap, Insertion sort

↓ Implementation In right side we have use Quick sort.

* Once we make a class or an object we can say that encapsulation take place.

Point to remember

* Important things that can be ask from OOPs:

1. Inheritance : Diamond Problem
2. Polymorphism : Diff b/w Compile Time and Run Time (Most Important)
3. Encapsulation : } Need to difference b/w encapsulation & Abstraction.
4. Abstraction :

(Encapsulation is a subset of an Abstraction. They may overlap.)

Real life ex of Abstraction. (generic sort func)

* Dynamic Memory Allocation:

Program start

or

Stack Memory
- Local variable & func parameters are stored in it
- Small memory (By default)

Heap Memory
- By default (Large memory)

Q Why do we need Heap Memory:

⇒ We know that if we want create an array of size n

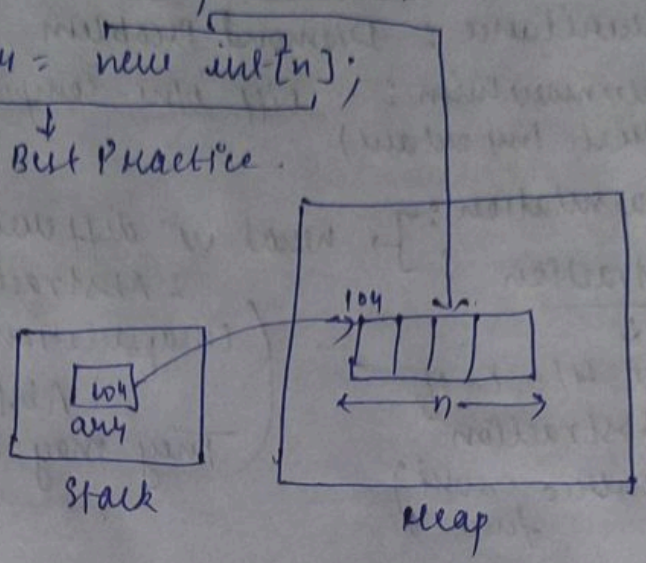
∴ cin >> n;
arr[n]; } It is considered to be bad practice.

Because suppose we want to an array of length 10 lakh but we know stack memory is limited we can't allocate the stack memory for array of length 10 lakh so that's why we need heap memory.

i.e `int * arr = new int[n];`

allocate heap memory to array of int integer type and length n.

But Practice.



* Creating different type of Memory in Heap Memory:

(1) ~~create~~ Allocate heap memory of int type.

`int * a = new int .`

(2) Create Heap Memory of char type

`char * a = new char .`

(3) create float Heap memory of float type

`float * a = new float .`

(4) Allocate Heap Memory to 1-D Array

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

(5) Allocate Heap Memory to 2-D Array ✓ Most Most Important

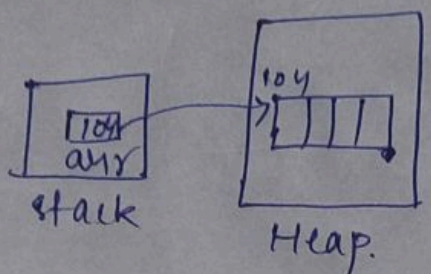
Q How to allocate Heap memory to 2-D Array?

for 1-D Array

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

single pointer — integer — array of size n

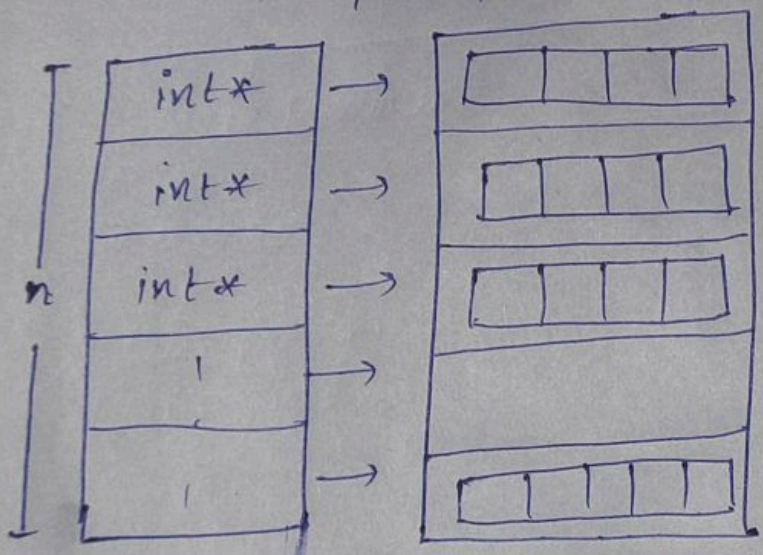
The integer array of size n created in Heap memory is hold by single * pointer.



for 2-D array

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

double pointer — array of int*




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

arr[i] — pointer

new int[m] — integer array of size m

Heap memory to

* Dynamically creating 2-D Array:

```
int ** arr = new int*[row];
for( int i=0 ; i<row ; i++){
    arr[i] = new int[col];
}
```

→ 2D Array Creation

arr[row][col]