OOPs: Programming technique:

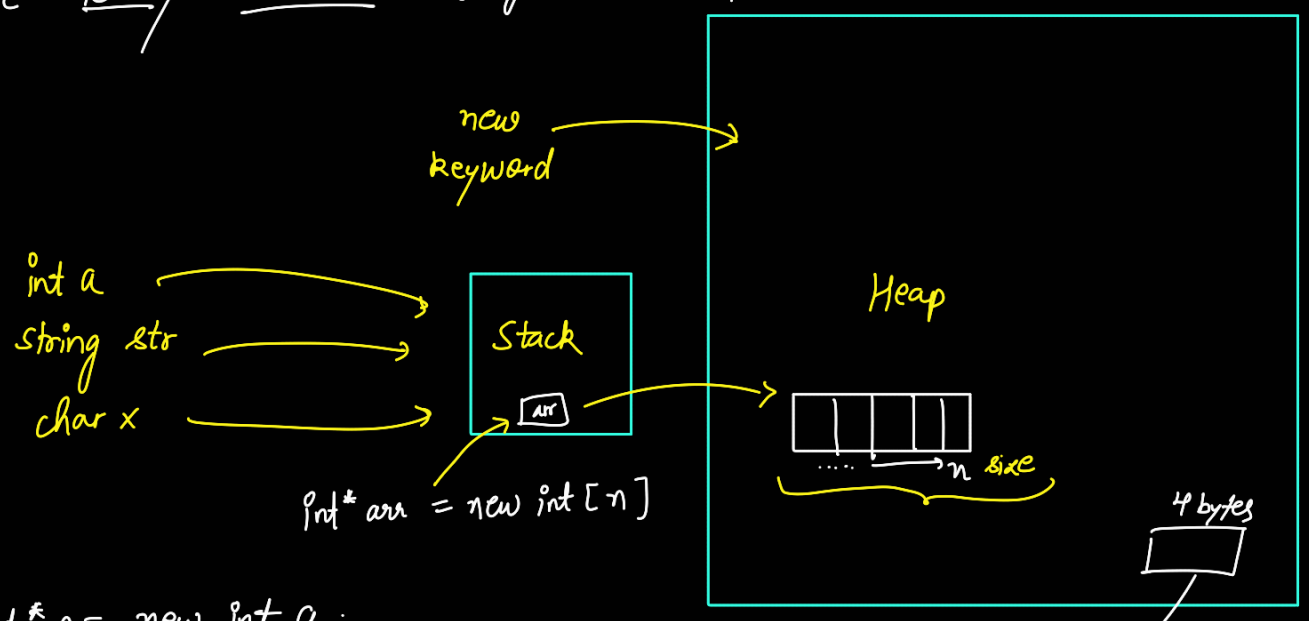
Object: → Entity → state/prop.
→ Behaviour/method.

Benefits → Readability ↑↑

→ 2D array last mein dekho

[Dynamic memory allocation using new keyword]:

1.



eg → int *p = new int a;

returns address of 4 byte memory in heap (allocates memory in heap).

dynamic memory allocation

2-D array

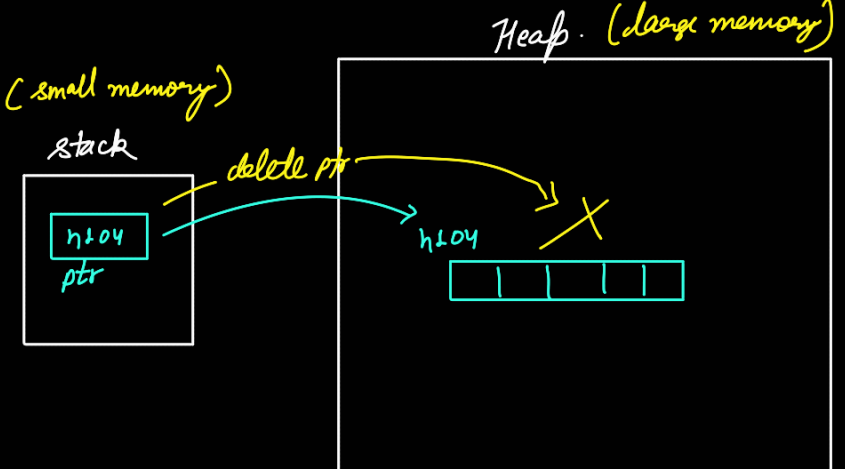
int** arr = new int *[n]

Catch:

int* ptr = new int [5]

↓

pointer is allocated in stack but the memory it is pointing to is allocated in heap



↳ delete ptr;

↳ deletes heap memory to which the pointer ptr was pointing.

we have to manually de-allocate memory from heap using **delete** keyword

Animal * suresh = new Animal;

(*suresh).age = 45;

or

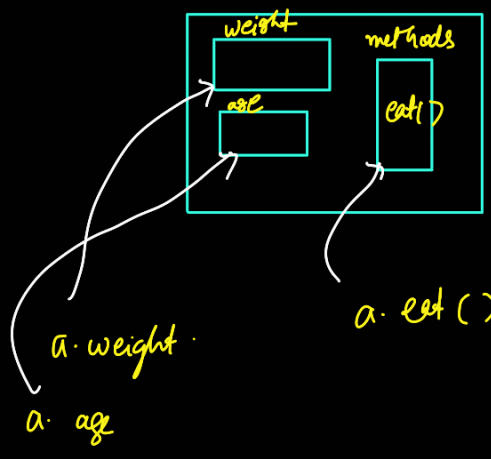
suresh → age = 45;

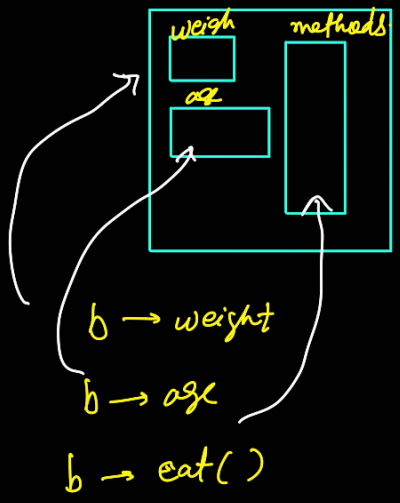
(→ operator to access values at pointers).

* keyword returns address of allocated memory in heap.

Animal * b = new Animal

Animal a





* this is a pointer to current object.

```
Class animal {
    int weight;
    void setweight ( int weight) {
        weight = weight ;
    }
};
```

↳ Can't diff. b/w weight (which one is parameter & which one is object's property.

this → weight means weight is

a class member.

points to current object

```
Class animal {
  int weight;
  void setweight ( int weight) {
  this → weight = weight ;}
};
```

→ means object ke weight ko weight (jo parameter me diya hai, set kardo)

Object Initialization

→ Constructor → object initialisation.

- created whenever an object is initialised
- → no return type
- → used for initialisation of objects of a class.
- → name is same as class.

copy constructor → `Animal ( Animal obj) {`

`} a, b;`

`Animal a = b;`

→ copy constructor called.

pass by value

`Animal (Animal obj)`

∞ loop

keep calling copy constructor infinitely.

Resolve

Pass by reference

destructor → used to free memory allocated during object initialisation by constructor.

(~ tilde sign)

Static obj creation ⇓ automatically called when the scope of that object gets ended.

Dynamic obj creation ⇓ delete manually.

[4 Pillars of OOPs]:

Polymorphism | Encapsulation | Inheritance | Abstraction.

1. Encapsulation → (Data-hiding) → reusability

eg → making of a class.

Perfect encapsulation: all data members marked private.

↳ security ↑ privacy ↑

2. Inheritance:

⇓ inherit properties → data members and member functions.

code: ?

child / sub / derived class.

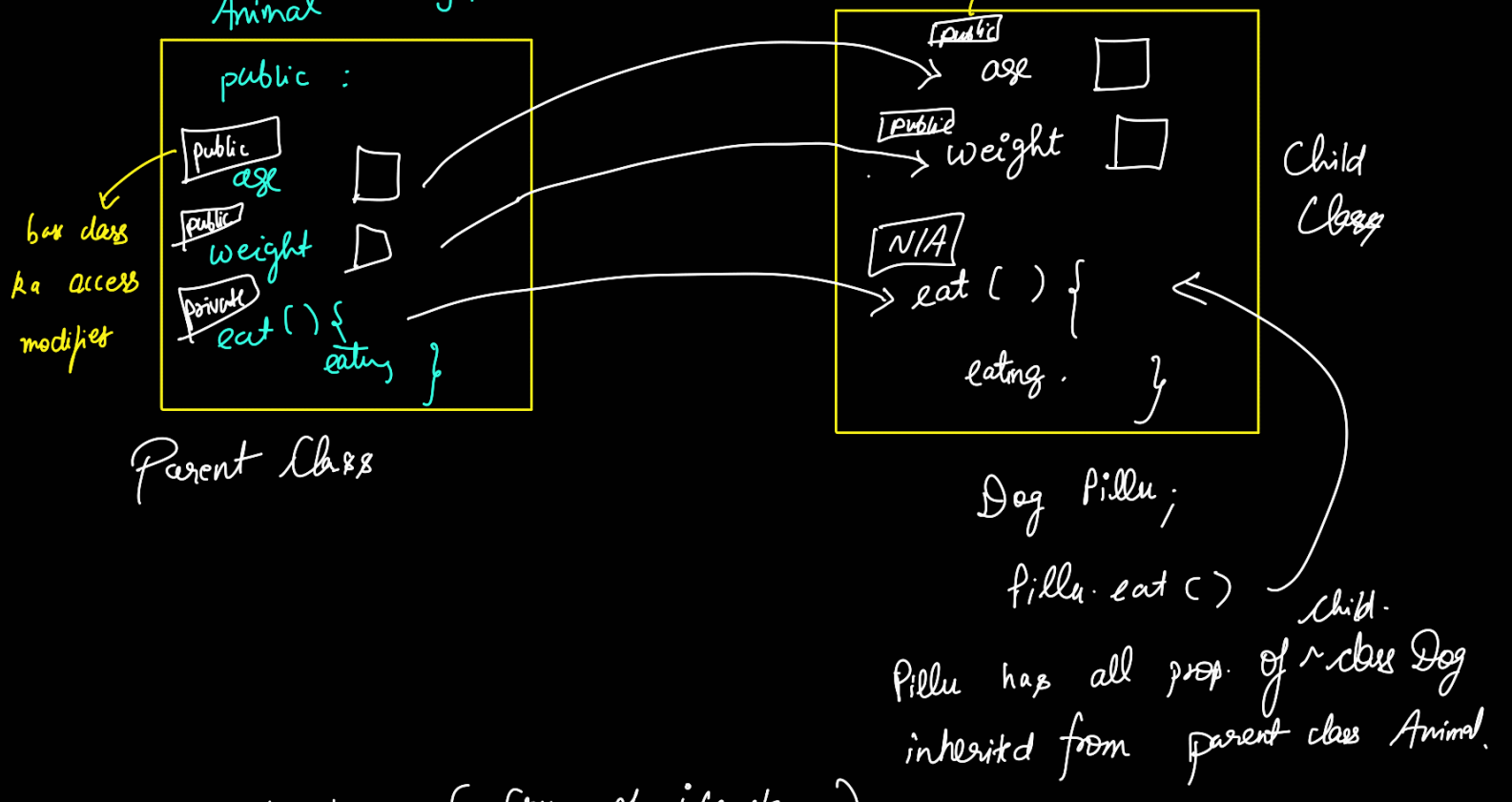
class child : parent

mode of inheritance
- public
- private
- protected.

```
class Animal {
public:
  int age;
  int weight;
void eat() { cout << "eating"; }
};
```

⇒ `class Dog : public Animal { };`

child class ka access modifier.



Pillu has all prop. of child class Dog inherited from parent class Animal.

Table of inheritance (Crux of inheritance)

Base Class ka Access modifier	Mode of inheritance		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private.

Private	N/A	N/A	N/A

Data member / fun. type	access		
	inside	outside	class
public :	✓	✓	✓
protected :	✓	✗ ⇒ also accessible inside derive class.	✗
private :	✓	✗	✗

inheritable

↳ non-inheritable.

↓ otherwise behave same as private

```
class Animal {
    private:
        int age;
};

dog d1;
cout << d1.age &&
```

```
class Dog : public Animal {
    public:
        void print() {
            cout << this -> age;
        }
};
```

↓ age is inaccessible in the child class

diff b/w private & protected

```
class Animal {
    protected:
        int age;
};
```

```
class Dog : public Animal {
    public:
        void print() {
            cout << this -> age;
        }
};
```

↓ age is accessible inside child class

int main() { dog d1;

cout << d1.age

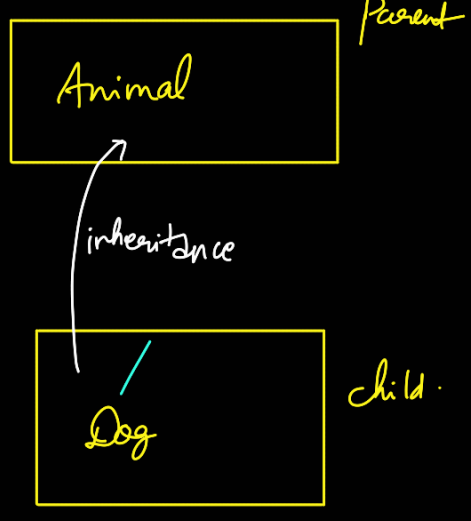
};

* Types of Inheritance :

- Single
- multiple
- multi-level
- Hierarchial
- Hybrid.

☆ I Single level inheritance :

↳ 1 parent 1 child Inheritance



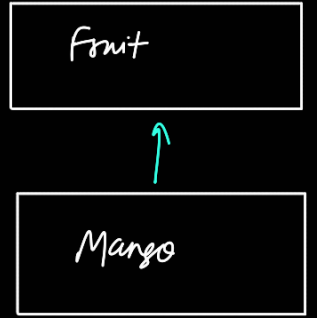
class dog : public Animal {

};

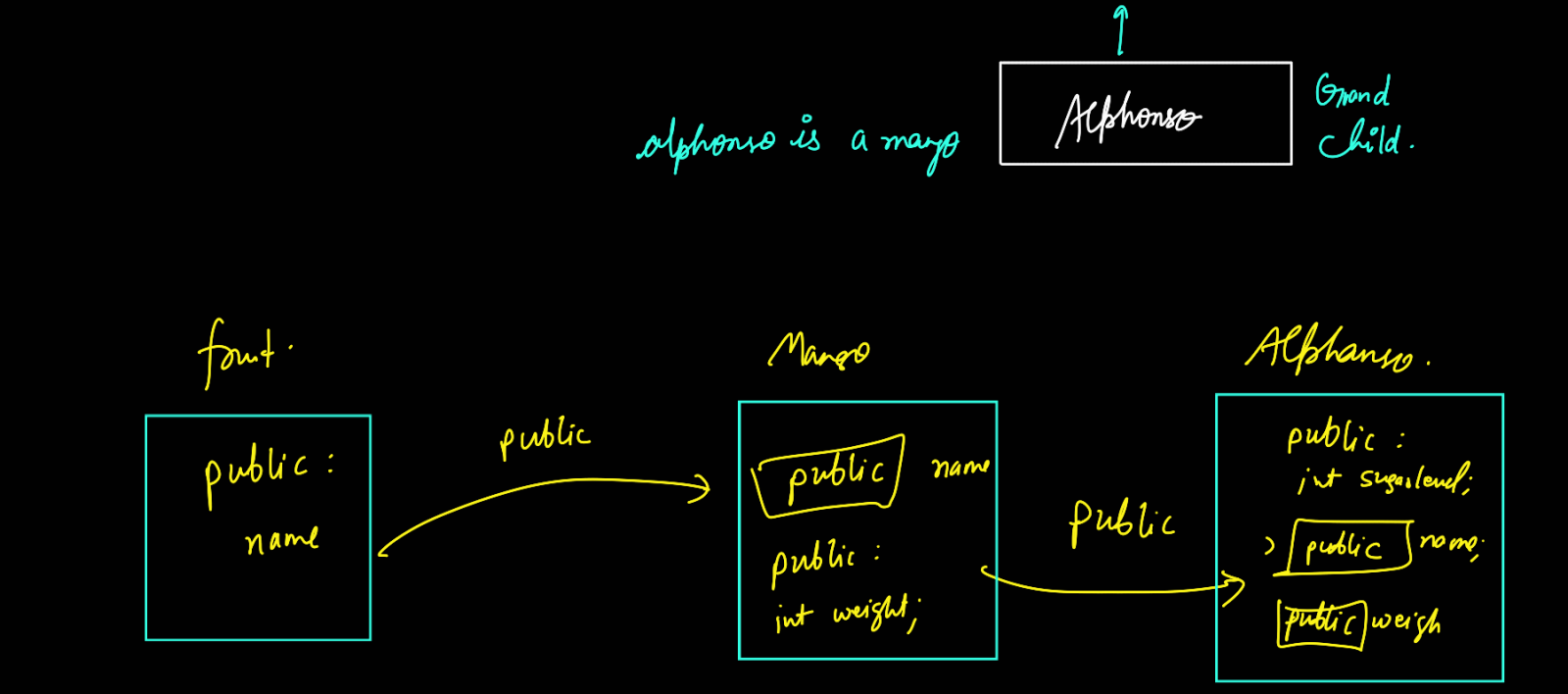
↳ eg → dog is a animal.

inheritance (:) can be replaced by "is a" relationship.

II Multilevel inheritance →



mango is a fruit



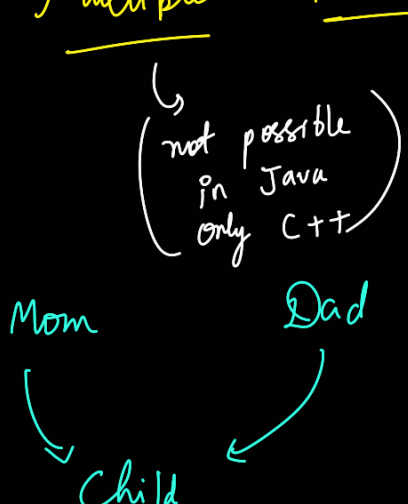
- Reusability ↑↑↑

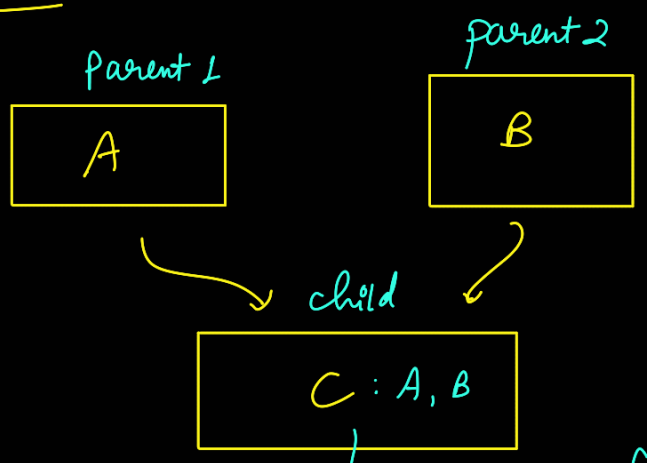
multi-level inheritance:

(III) Multiple inheritance →

(not possible in Java only C++)

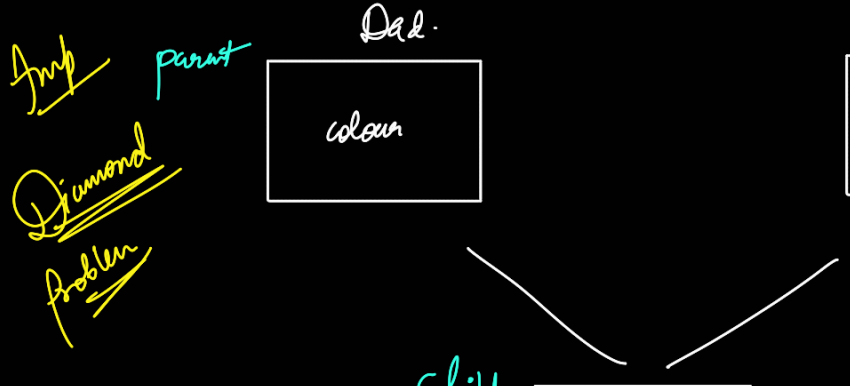
eg →





↳ inheritance from multiple parent classes

Syntax: class C : public A, public B ;



↳ Which colour would be

Child;

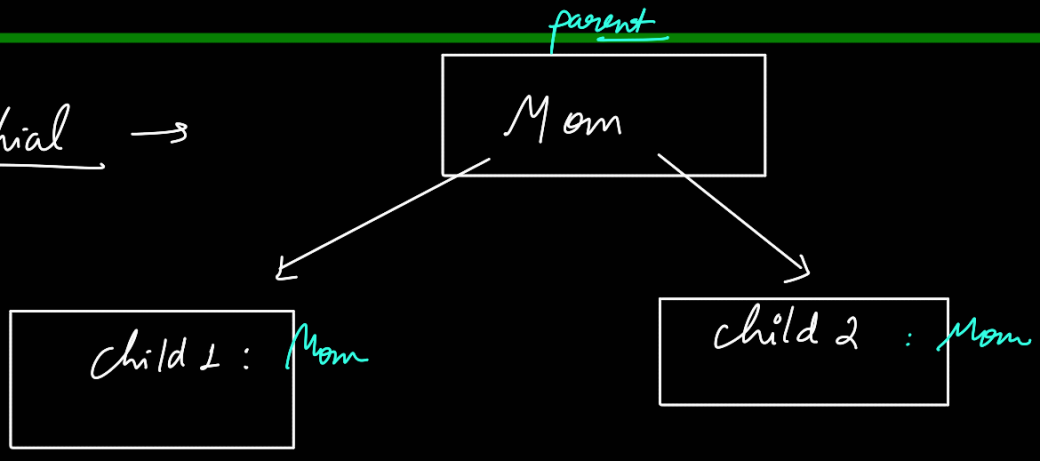
inherited?

use :: "scope res." operator

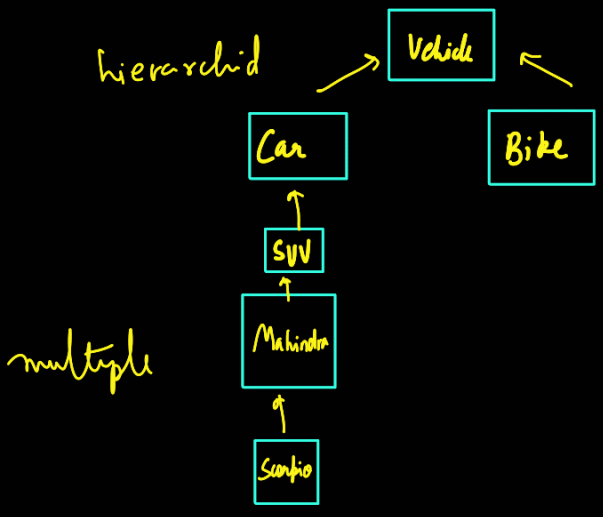
Child Pintu;

Pintu.Mom::colour or Pintu.Dad::colour

(IV) ⇒ Hierarchial →



(V) Hybrid Inheritance: combo of 2 or more types of inheritance.



(hierarchial + multiple)

POLYMORPHISM

↳ existing in many forms.

Runtime polymorphism.

compile time polymorphism

Compile time Polymorphism : 1) function overloading
2) Operator Overloading.

sum (int a, int b)

sum (int a, int b, int c)

operator overloading

↳ + se subtraction bhi kara sakte hain.

Syntax : return_type operator + ( )
{
}

HW : obj.print() ⟶ cout << obj ; → overloading of << and >> operators.

a + b ⟹ a.add(b)

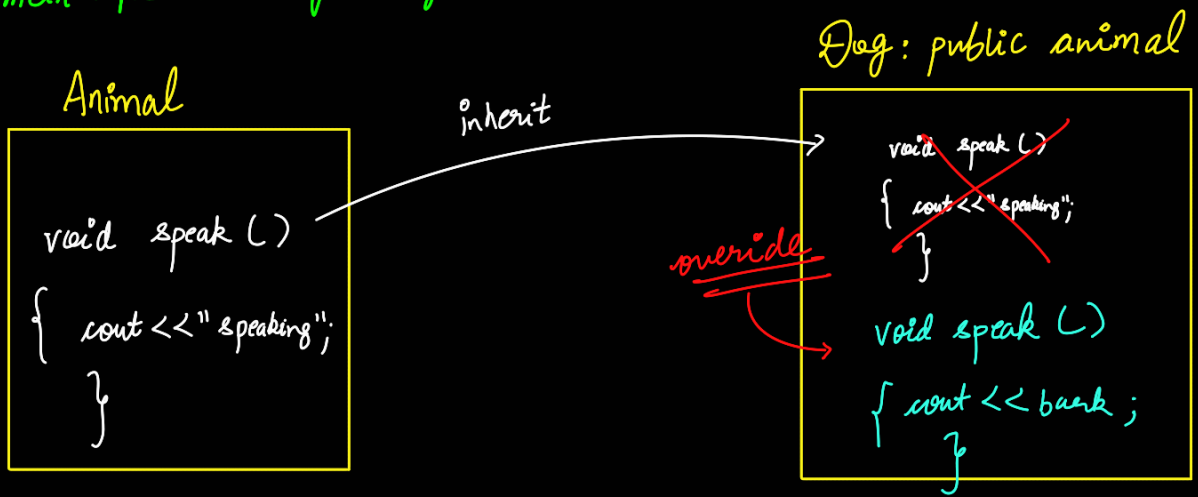
a → curr. obj.
+ → func^n call
b → i/p

see code for example :

Runtime Polymorphism : (Dynamic Binding)

↳ function overriding

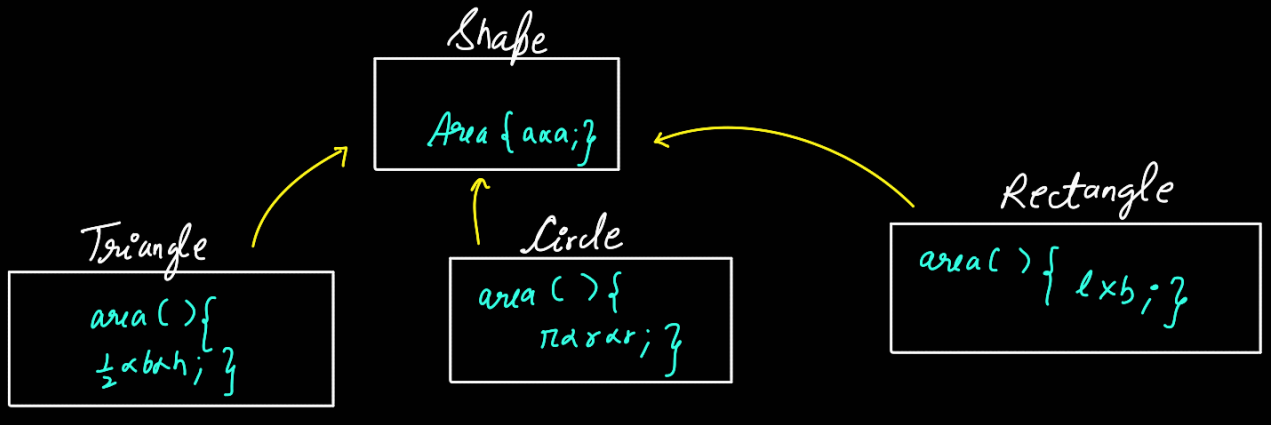
↳ parent class ke func" ko mein apne according change kar sakta hu.



dog d;

d.speak (); → o/p → bark ✓

eg →



☆ note in upcasting : Animal * a = new dog ();

a → speak.

o/p → speaking.

} When pointer of parent class and obj. is of child class then ↓ method/func" of parent class is called.

If you wanna call func" of child class; mark func" of parent class as virtual.

eg → Class animal {
virtual void speak () {
cout << "speaking";
};

Animal * a = new dog();

a → speak.

o/p → ~~speaking~~ barking

upcasting.
↓
ptr → parent class
obj → child class

downcasting
- ptr → child class
- obj → parent class.

Note: Whenever we upcast or downcast, then the method of class to which pointer belongs is called.

⇓ opposite

If you use virtual keyword in method of parent class.

4 types :

method called

parent * a = new parent();	→	Parent
child * a = new child();	→	child
Parent * a = new child();	→	Parent
Child * a = (child*) new parent();	→	child

pointer class

⇓

use virtual keyword

⇓

o/p reversed

4 types of constructor calling : → depends on object.

Constructor called

parent * a = new parent();	→	parent
child * a = new child();	→	parent then child.
Parent * a = new child();	→	parent then child.
		parent

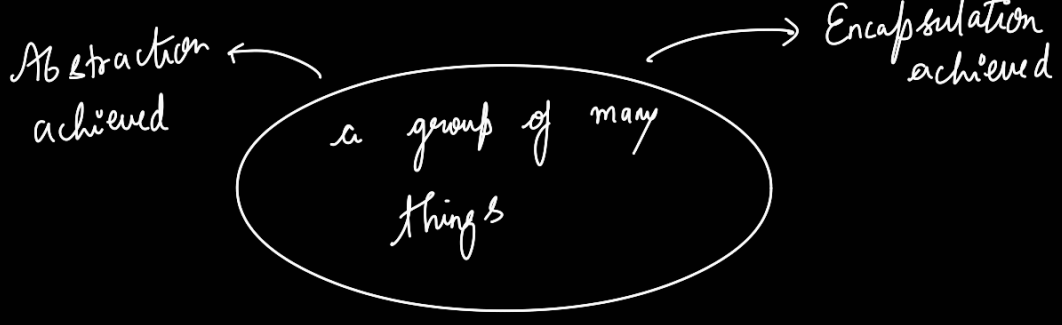
child* a = (child*) new parent( ); → [illegal]

bcuz child is inherited from parent, so we call both the constructors.

☆ Abstraction : [Implementation Hiding]
↳ only essential info is shown.

All pillars of OOPs are subsets of ABSTRACTION.

eg → a container containing multiple things. → Abstraction (+ encapsul.)
a container containing containers. → Abstraction (+ encapsul.)
↓
as encap ⊂ abstrac.



Ques Compile time polym. v/s Runtime polym.

[Dynamic Memory Allocation]:

2-D arrays

int** aar = new int*[n]

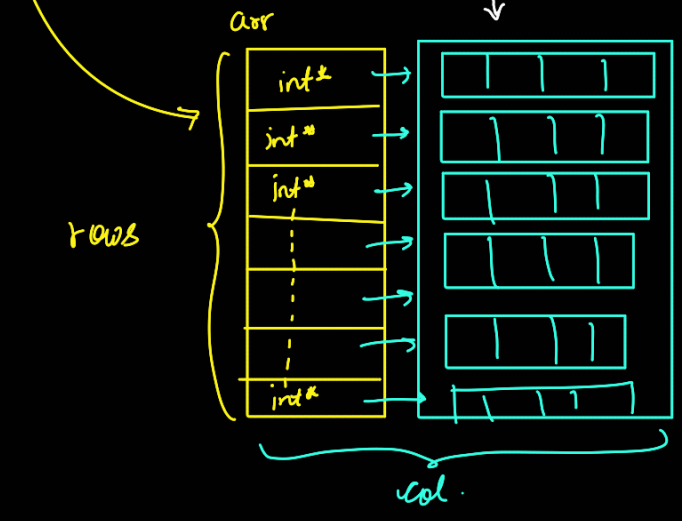
double pointer

array of int* type.

Humne ek array bnaya hai n size ka jiska andar data int* type ka hai and humne array ka naam arr diya hai.

☆ [Make a 2-D array in Heap memory]:

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



⇒ in C++ also:

```
vector<vector<int>> (row, vector<int>(col,0));
```

⇒ deallocate 1-D Array:

```
delete [] arr;
```

[de-allocate 2-D Array]:

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

⇓ then

```
delete [] arr;
```