

OOPS :-

↳ class → A blueprint that defines the properties & functions of a particular object.
↳ object

example of a basic class

```
class student {  
    public  
    int roll number;  
    int age;  
};
```

static allocation



student s1;
Creating an object.

dynamic allocation.

student * s2 = new student;

dereferencing

(*s2).age = 1;

(*s2).roll no = 206;

s2 → age = 1;

both means the same.

Access Modifiers :- → public

↓
private

(By default)

Protected.

s1.display();
s2 → display(); } calling functions.

* Always create getters & setters for changing/displaying private attributes.

By default →

Constructor & Destructor :-

- Same name as class
- No return type
- No input arguments

eg- student () {

}

} we can override constructor too.

this → holds the address of current object!

Copy constructor :-

→ Done at the time of creation of the object.

static with static

Student s2(s1);

Student *s3 = new Student(20, 2001);

static with dynamic

Student *s5 = new Student(*s3);

Student *s6 = new Student(s1);

dynamic with static

both perform shallow copy.

Copy assignment operator :-

→ when both of the objects are already created & we want to copy 1st's value to 2nd.

Student s1(10, 1001);

Student s2(20, 2001);

s2 = s1;

This will copy all the data of s1 to s2.

dynamic

*s3 = s1;

s2 = *s3;

} no constructor will be called.

Destructors :-

- ↳ Same name as the class
- no return type
- no input arguments

called at the end / exiting from the function.

~Student() {

}

→ used to deallocate the memory.

Student s4 = s4; → Student s5(s4);
not 2 steps are 1.

* we should pass constant references to avoid illegal changes.

Shallow & deep copy :-

- ↳ Rather than copying the entire array we just copy the 0th index address.
- ↳ Copying the whole array.
- * Create a new array.
This \rightarrow name = new char [strlen(name) + 1];
- * strcpy (This \rightarrow name, name);
- * Inbuilt copy constructor is shallow copy.

int a = 5;
const int b = a;
↳ Allowed

int a;
a = 5;
int const b;
b = a; ~~X~~

int i = 5;
int &i = i;
5, i

int i;
i = i; ~~X~~
not allowed!

Initialization list :-

Student (int r) : rollnumber(r) {

}

rollnumber(r), age(a).

Initialization list.

- * We have to use initialization list for both constant data members & reference variable.

* For constant variables we can only call constant functions.

Static Properties :-

Static int total_students

↳ which belongs to a class not to each object.

→ to access static variables we need to

say student :: total_students ;
 ↓ ↓
 class_name scope resolution operator
 ↳ Variable.

* we cannot initialize static properties inside the class.

int student :: total_students = 0 ;

01. total_students

↓
 This will not give error but is logically incorrect.

- * Functions can also be static.
- * This means the property belongs to a class not to a single object.

eg:- static int getTotalStudents() {
 return total_students ;

* static ~~properties~~ functions doesn't have this key word.

OPERATOR OVERLOADING :-

```
return type operator ( ) ( inputs ) {  
    operator  
    (+, -, *, /)
```

}

eg: Fraction operator + (Fraction const &F1) {

} ;

For student S3 = ++S1 ;

Pre increment

✓ just pass by reference

```
Fraction & operator ++ ( ) {
```

```
    numerator = numerator + denominator ;
```

```
    simplify ( )
```

```
    return *this ;
```

```
}
```

* When our function returns some value our system stores it in a buffer. & if we are assigning it somewhere it gets assigned.

Post increment

```
Fraction & operator ++ (int) {
```

```
    Fraction Fnew ( num, deno ) ;  
    Fnew.simplify ( )
```

```
}
```

```
    return Fnew ;
```

nesting is not allowed as
post increment

Abstraction :-

Hiding the unnecessary details from the user.
↳ data hiding.

eg:- increasing volume of the TV using remote.

→ updation should not effect the enduser.

→ to avoid the errors.

Encapsulation :-

Combining the data & functions in a single entity called class.

eg:-

Student class	
name roll no.	...
address	
change address	

Inheritance.

Vehicle → Parent / Base

↓
car → child / Derived.

class Vehicle {

};

class car : access specifier Vehicle

}

↓
Public / private /
protected.

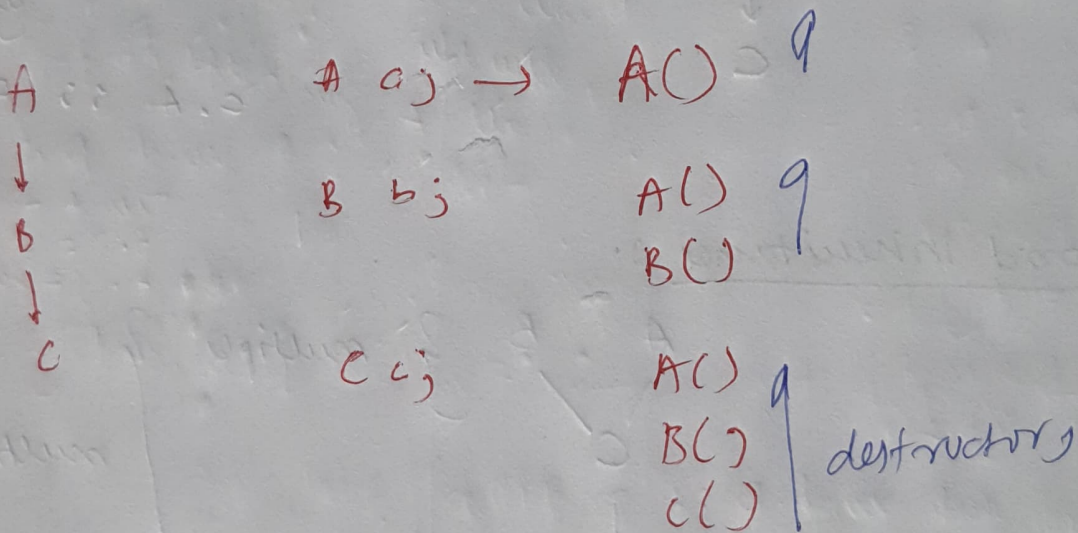
Access modifier of parent →

	Public	Private	Protected
Public	(Public) ✓	✗	(Protected only) ✓
Private Protected	(Protected) ✓	✗	(Protected) ✓
Protected Private	(Private) ✓	✗	(Private) ✓

Access modifier of child

→ By default in access modifier is private.

Order of constructor & destructor :-

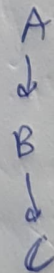


Types :-

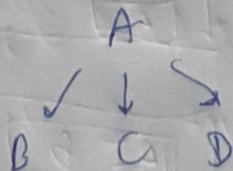
1) Single In.



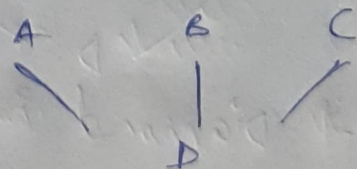
2) Multi level



3) Hierarchical :-



4) Multiple.

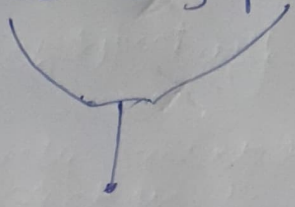


our
fun

4. multiple inheritance :-

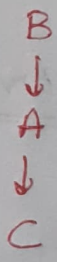
```
class C : public B, public A {
```

};



Access modifiers.

The constructor of the parent class which is written first will be called.
B in this case.



in C;

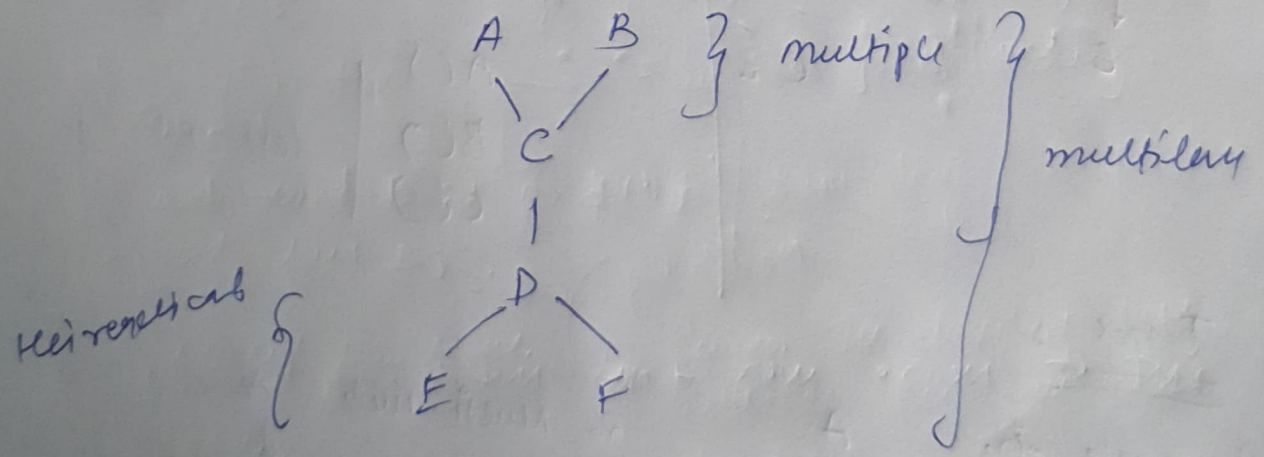
&

e. B ::

Functions of B C).

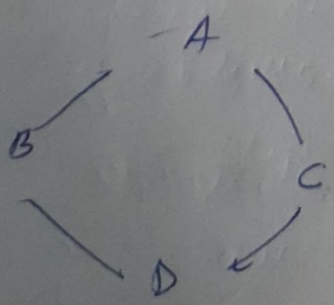
e. A :: Functions of A C).

5. Hybrid Inheritance :-

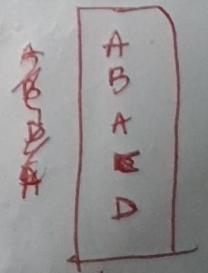
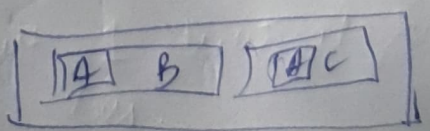
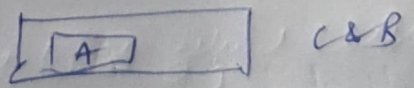
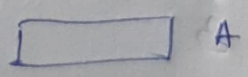


Hereditary

eg:



* Diamond Problem



order of const.

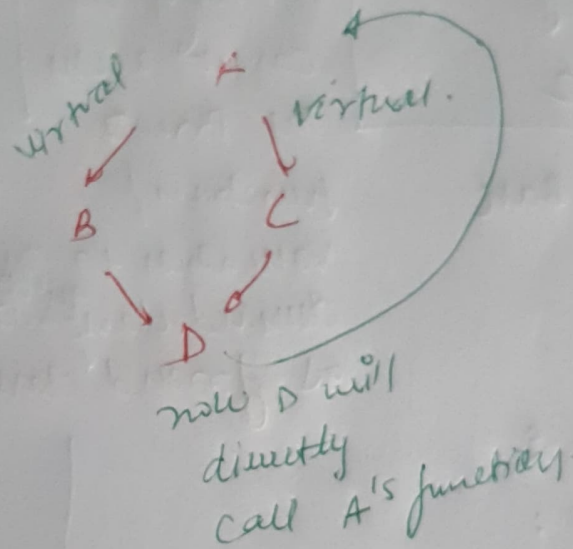
one option to tackle this problem is declaring that function again in child class.

→ We have to explicitly mention which class function we want to call.

Ex B is function C.

Virtual :-

↳ Functions which are present in the base class & are overridden in the derived class.



A
↓
B
↓
C
↓
D

after making Parent Virtual

Poly morphisms :-

many forms

↳ compile time

↳ Run time

↳ using

virtual functions.

→ Function overloading.

→ Method/overriding

Function -> when the derived class overrides the base class function.

Pure virtual function :-

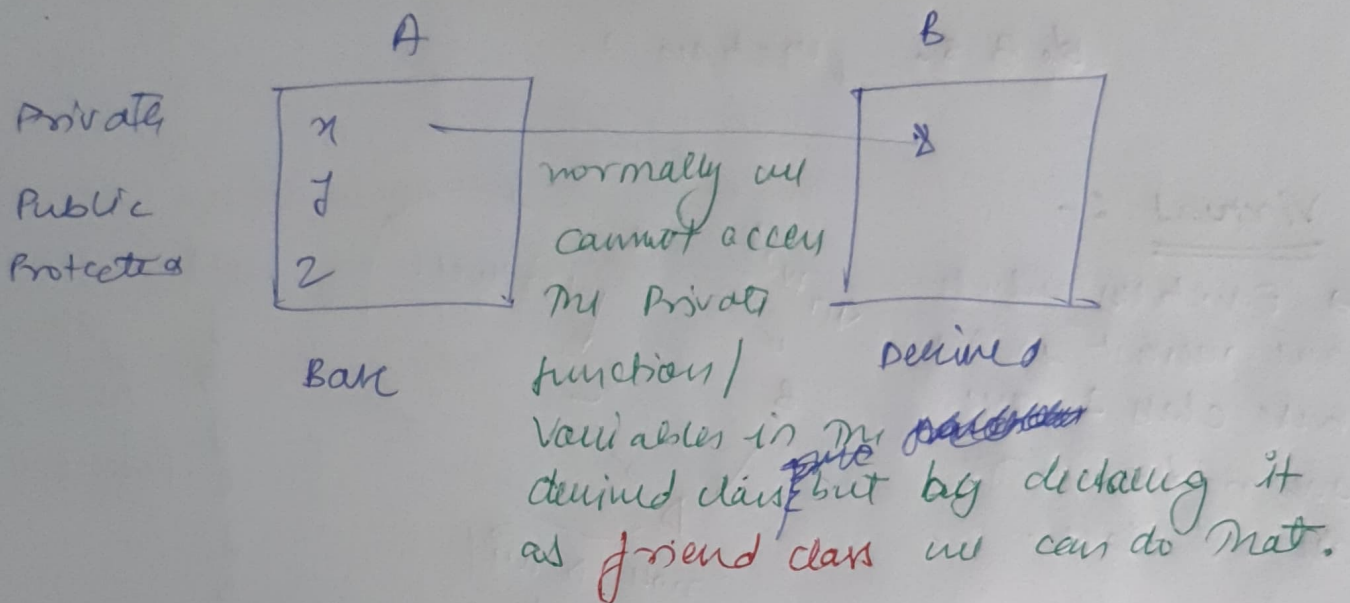
Virtual void print() = 0;

Classes which have at least 1 virtual function is known as abstract class.

Abstract means incomplete.

↳ we cannot make object of abstract classes.

Friend functions & classes :-



class A {

private :

int x;

protected :

int y;

public :

int z;

friend void B :: print();

friend class B;

}

class B {

void print() {

cout << x;

t.x = 10;

t.y = 20;

cout << endl;

}

};

* Friend functions don't have access to this pointer.

* we can put friend functions under any access modifier.

* we can also make the whole class a friend class.