OOPS :-

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

example of a basic class

```
class Student {
Public:
    int rollnumber ;
    int age ;
};
```

static allocation

```
Student S1 ;
```

creating an object.

Dynamic allocation.

```
Student * S2 = new Student ;
```

Dereferencing

```
(*S2).age = 1 ;
(*S2).rollno = 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 & construct too

this → holds the address of current object.

copy constructor :-

→ done at the time of creation of the object.

static → Student S2(S1);

with static

Student *S3 = new Student(20,2001);

static with dynamic → Student S4(*S3);

dynamic with dynamic → Student *S5 = new Student(*S3);

dynamic with dynamic → 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 is 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 our class
→ no return type
→ no input arguments

~Student(){

}

called at the end / exiting from the function

→ used to deallocate the memory.

Student S4 = S4; → Student S5(S4);
not 2 steps as I.

* we should pass constant references to avoid illegal changes.

shallow & deepcopy :-

↳ Rather than copying the entire array we just copy the 0th index address.

↳ copying the whole array.

* create a new array.
* this → name = new char [strlen(name) + 1];
* strcpy (this → 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;
```
✓ ✗

```
int i = 5;
int &j = i;
```
5
i, j

```
int i;
j = i
```
✗ not allowed!

Initialization list :-

```
Student (int r) : rollnumber(r){

}.
```

↓

```
roll number(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 ;

↳ Variable.

Class_name — Scope resolution operator

* We cannot initialize static properties inside the class.

```
int student :: totalstudents = 0 ;
```

01. totalstudents

↓

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() {
~~cout~~ return total_students ;
}

* static ~~properties~~ Functions doesn't have (this) key word.

OPERATOR OVERLOADING :-

return type operator(symbol) (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;
}

not the correct implementation

* 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 in post increment

Abstraction :-

Hiding the unnecessary details from the user.
(Data hiding.)

eg:- increasing volume of the TV using remote.

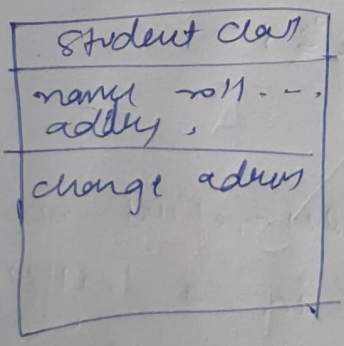
→ updation should not effect the endusers.

→ to avoid the errors.

Encapsulation :-

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

eg:-



Inheritance.

Vehicle → Parent / Base

↓

Car → child / Derived.

class Vehicle {

};

class Car : access specifier Vehicle

↓

Public / private / protected.

}

Inheritance of parent →	Public	Private	Protected.
Public	(Public) ✓	✗	(Protected only) ✓
~~Private~~ Protected	(Protected) ✓	✗	(Protected) ✓
~~Protected~~ private	(Private) ✓	✗	(Private) ✓

Access modifier of child.

→ By default the access modifier is private.

Order of constructor & destructor :-

A → B → C

A a; → A()

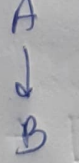
B b; → A() B()

C c; → A() B() C()

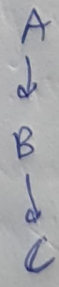
destructors

Types :-

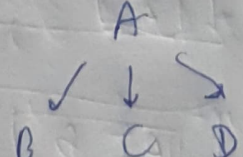
1.) Single in.



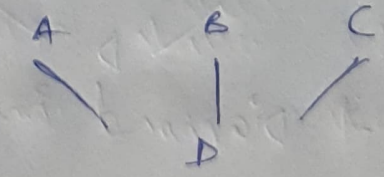
2. Multilevel



3.) Hierarchical :-



4. Multiple.



4. Multipl inheritance :-

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

Access modifiers.

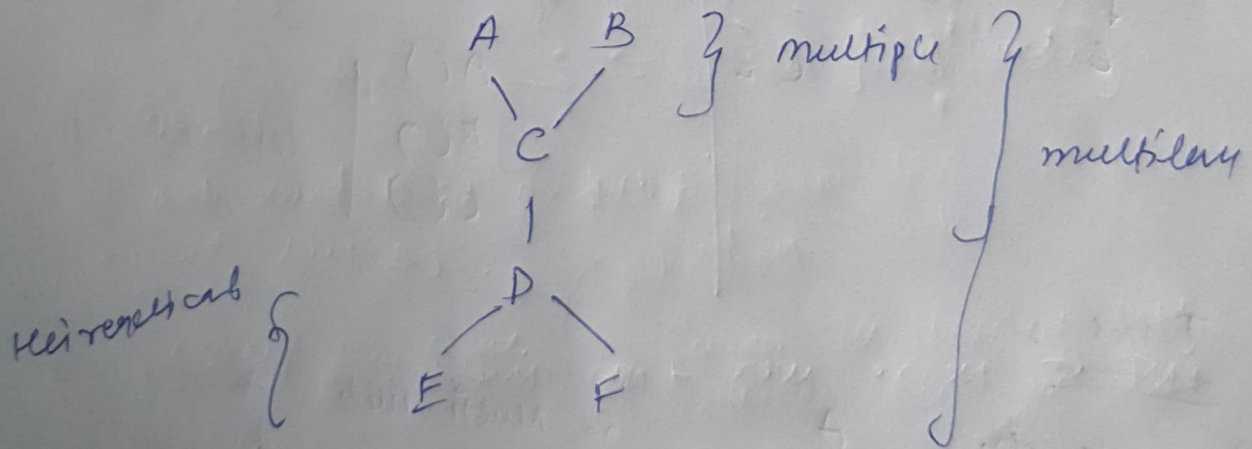
The constructor of the Parent class which is written first will be called.

B in this case.

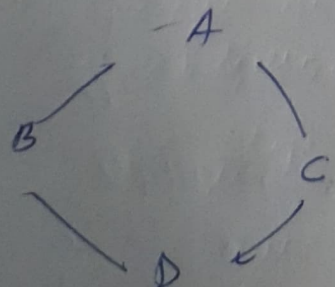
B
↓
A
↓
C

```
int C c;
c.B :: Function of B C).
c.A :: Function of A C).
```

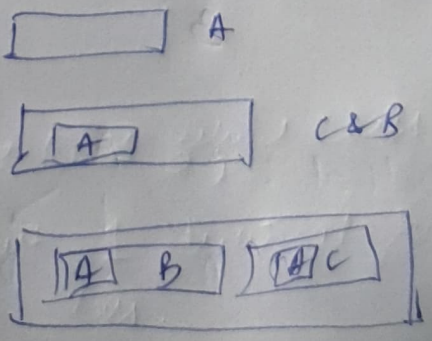
3. Hybrid Inheritance :-

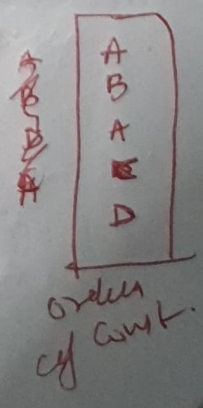


eg:



* Diamond Problem





order of const.

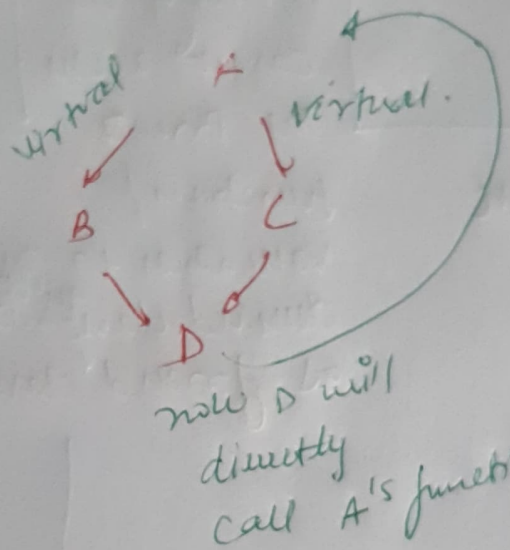
our option to tackle this problem is declaring that function again in child class.

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

B.B :: function().

Virtual :-

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



A
↓
B
↓
C
↓
D

After making parent virtual

Polymorphism :- many forms

↳ compile time → Function overloading.

↳ Run time → Method/overriding function.

↳ using virtual function.

↳ when the derived class overrides the base class function

Pure virtual function :-

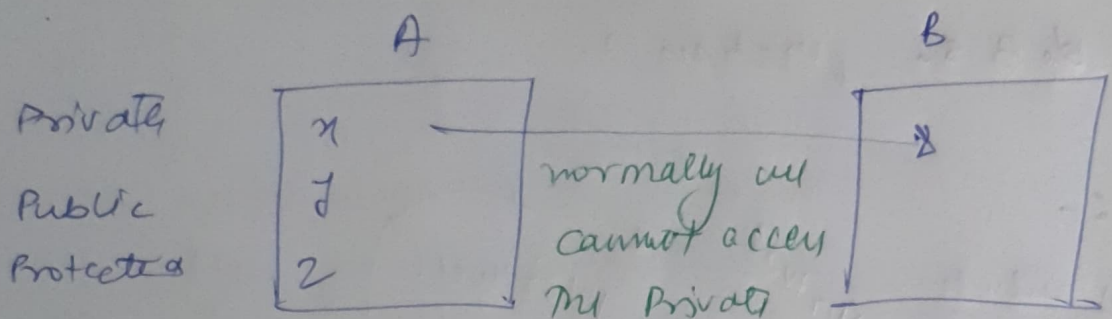
Virtual void print() = 0;
pure

classes which have atleast 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() {
        Twice t;
        t.x = 10
    } t.y = 20;
} cout - - - ;
};
```

* 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