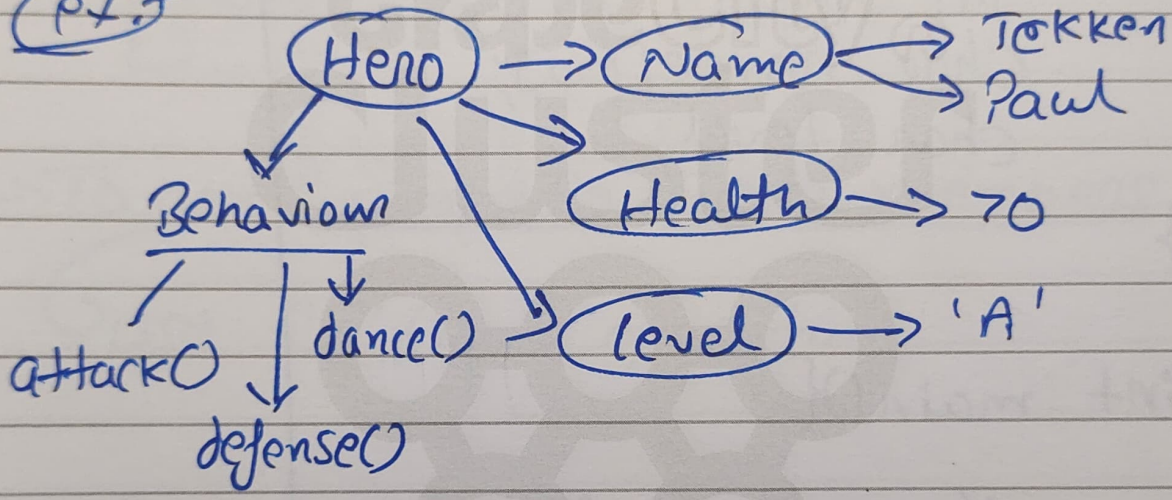
26/09/2023

[OOPs]

what → ? instance of class

⊙ object is a Entity → state/Properties, behaviour

ex:



⊙ [class] → user defined data type that act as a (cup) blueprint for individual objects, object

ex-

int (a);

String (str);

char (ch);

class → Hero → Ramesh

Hero → u.d. data type

Ramesh → variable

attributes & methods

example, sample, illustration, etc

allocate memory in Heap

Object → Instance of class

```
class Hero{

    //Properties
    int health;
    char name[100];
    char level;

}

int main(){

    //creation of object
    Hero h1;
    sizeof(h1);
}
```

→ output: 4

⇒ **Empty class** →

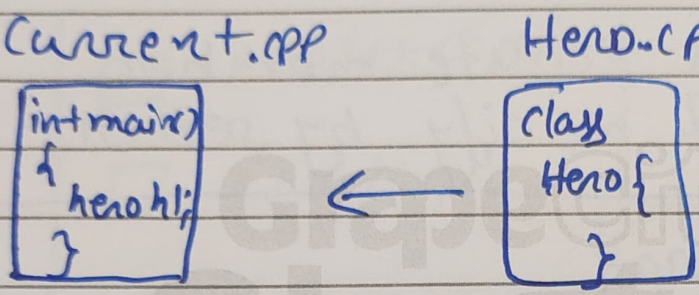
```
class Hero{

}
int main(){
    Hero h1;
    cout<<sizeof(h1);
}
```

← output: 1

#include "Hero.cpp"

This will include the content of Hero.cpp file in current file.



→ To access properties / Data members
↳ use "." (Dot) operator

⇒ Access Modifiers :
- → Public
- → Private → Default
- → Protected

⊙ Public: Available to everyone

⊙ Private: can only be accessed inside the class.
i.e. only Member f^n or friend f^n are allowed to access private members.

~~(2) Protected:~~

Getter / setter

↓ fetch → read

↓ cond^{n} Apply

⇒ Getter / setter:

we can access private members by getter & modify by setter.

(Pt⇒)

```
class hero{
    Private:
     int value; //int health;
    Public:
     char level; // char level;
    int getvalue(){ return value;}
    char getlevel(){ return level;}
    void setvalue(int v){
                value = v;}
    void setlevel(char h){
                level = h;}
    };
int main(){
        hero Paul; //Hero Ramesh
        Paul.setvalue(80);
        [illegible]
        Paul.setlevel('A');
    cout<< Paul.getvalue() << endl;
    cout<< Paul.getlevel() << endl;
}
```

⇒ BTS (Behind the scene):

Hero Ramesh;

↓ BTS

- Health (int)
- Level (char)

Hero

Health
level

Sizeof(Ramesh) → (4+1) = 5 ✗

→ (4+4) = 8 ✓

(larger value set for both)

ex: double health;
char level; } → size = 8+8 = 16

working of Padding:

int a (4 bytes)

char b (1 byte) | Padding (3 bytes)

total size = 8 bytes

Static & Dynamic Allocation :-

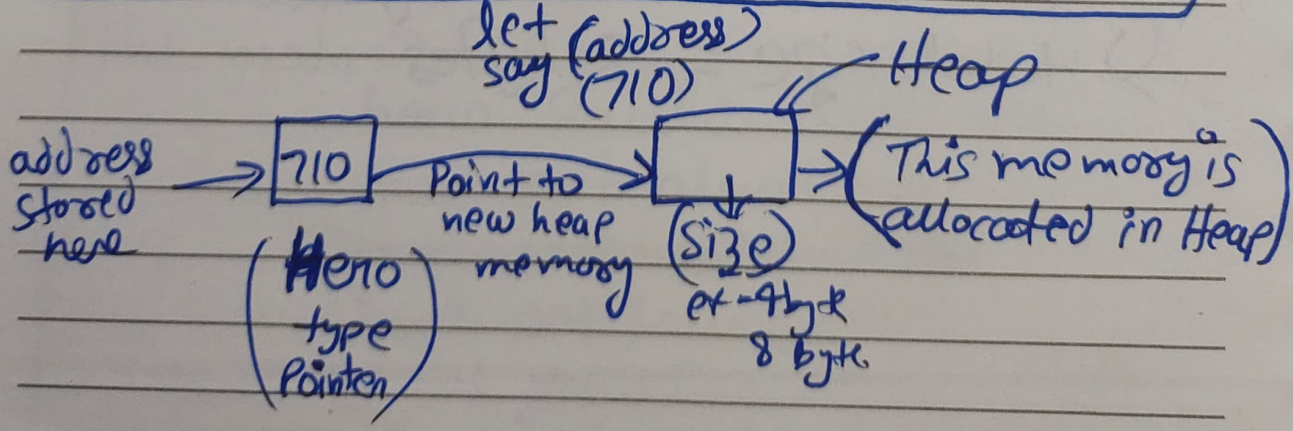
```
int main(){
    //Static allocation
    Hero a;
    a.setHealth(80);
    a.setLevel('B');
    cout<< a.level <<endl;
    cout<< a.getHealth() <<endl;

//Dynamically

Hero *b = new Hero;
  b->setLevel('A');
  b->setHealth(70);
cout<< (*b).level << endl;
cout<< (*b).getHealth()<<endl;
```

or

b->level & b->getHealth()



Constructor :- (A special member function)

Hero — class name

Ramesh — (object) (created → Constructor)

⊙ When any object is created, a constructor is called. (inbuilt)

Properties :-

- invoke at the time of object creation
- No return type
- No i/p parameter

⇒ Default Constructor :-

Hero Ramesh → Ramesh.Hero() is automatically called.

Constructor is used to initialize the data members of new objects & assign values to it.

code:

```
class Hero {
    Public:
        Hero() {
            cout<<"constructor called"<<endl;
        }
};
int main(){   cout<<"Hi"<<endl;
    // object created statically
      Hero ramesh;
    // Dynamically
      Hero *h = new Hero;
                (or) new Hero();
    cout<<"hello"<<endl;
```

output:

```
Hi
Constructor called
constructor called
Hello
```

→ By writing any constructor the default inbuilt constructor automatically vanished.

⇒ Parameterised Constructor :-

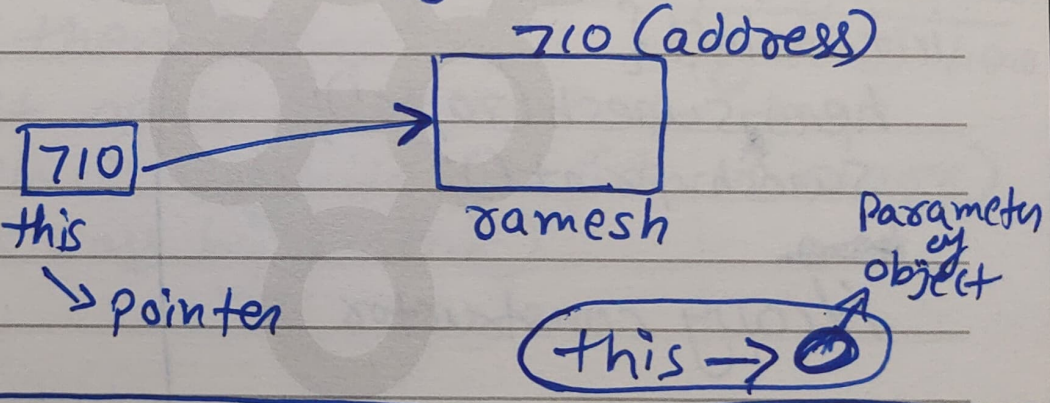
```
eg:=>   Hero (int health){
            health = health;
        }
```

→ inside the class hero

this keyword :-

The address of current obj. (ie. ramesh) is stored in "this"

this → It is a pointer, points at current obj.




```
hero(int op, char cc){
    cout<< "this -> " <<this <<endl;
    this->health = op;
    this->level = cc;
}
void Point(){
    cout<<health<<endl;
    cout<<level<<endl;
}
```

Inside the class hero

constructor

this → 0xf76ca0
16
S
0xf76ca0

(output)

```
int main(){
  hero *a = new hero(10, 's');
  a->print();
  cout << &(*a) << endl;
```

or

```
  hero ramesh (10, 's');
  ramesh.print();
  cout << &ramesh << endl;
}
```

⇒ Copy Constructor :

```
int main(){
  hero suresh(70, 'c');
  suresh.print();

  //copy constructor

  hero R(suresh);
  R.print();
}
```

We can make our own copy constructor

```
class hero{
    - -
    - - -
    - // copy constructor
    hero(hero &temp){
        cout << "copy constr. called" << endl;
        this->health = temp.health;
        this-> level = temp. health;
    }
}
```

(we did it from (.) because temp is statically allocated)

> we used (&) to use pass by reference.
if don't, then 'temp' call copy const. hero and it again comes back to temp and follows infinite loop (shows error) (it is pass by value)

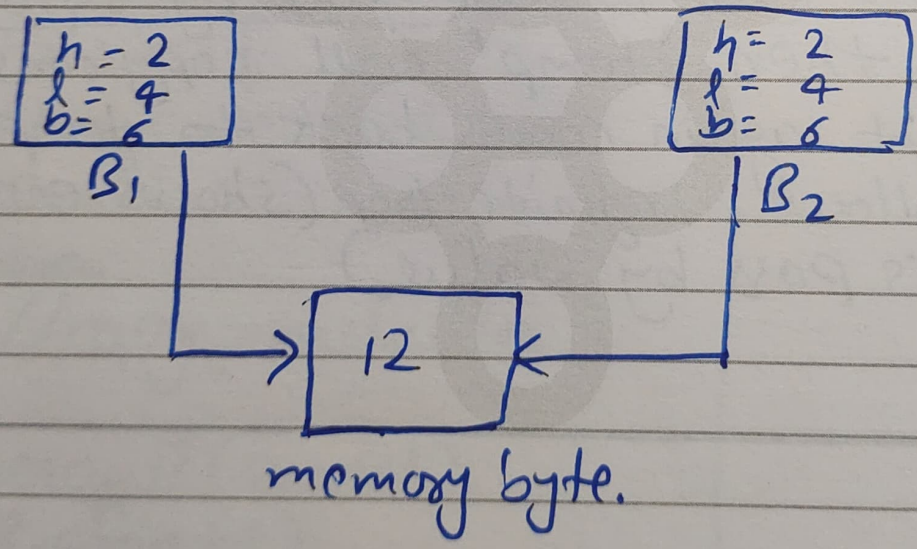
#Shallow & Deep Copy:

default copy constructor ⇒ Shallow copy

Shallow copy:

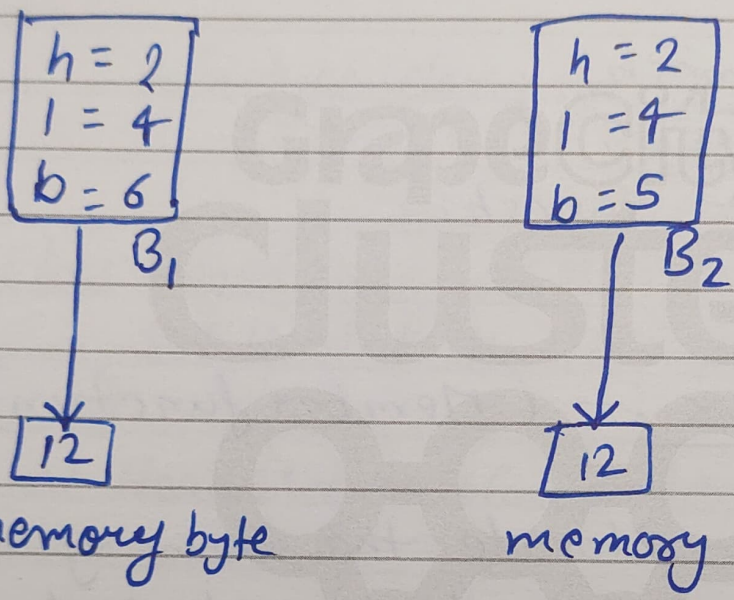
Object is created by copying data of all variables of original object. (for both)

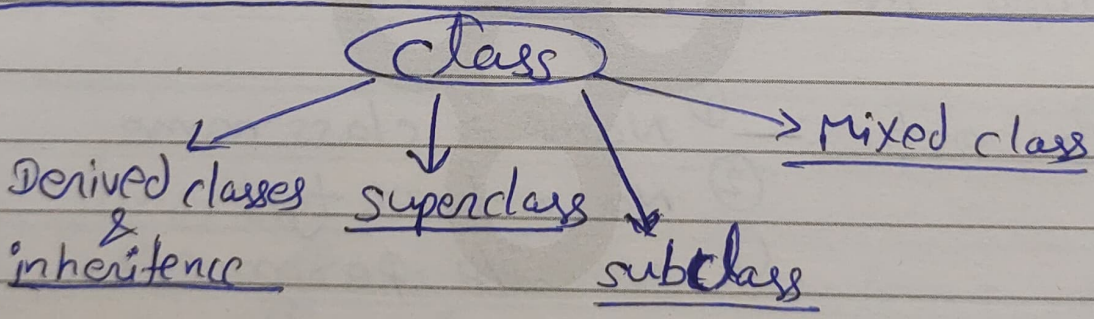
- Both object refers same memory location.
- Changes on one reflect on another.



Deep Copy:

- Allocate same memory to both object.
- we need to explicitly define copy const. & assign dynamic memory as well, if required.



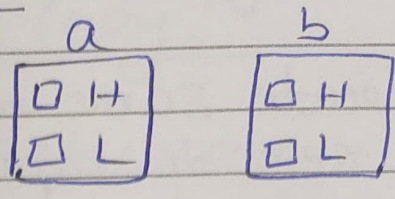


- Derived/inheritence — It is generated or derived from another class
- Subclass — A class that inherits its prop. from another class.
- Superclass — The class from which properties are inherited.

⇒ Assignment Operator :-

a = b;

i.e. { a.health = b.health;
a.level = b.level; }



Ex:

```
Hero Ramesh;
Hero Suresh;
Ramesh = Suresh;
```

Destructor :- (Member function)

⊙ Used to de-allocate the memory.
⊙ When object is about to get out of scope the destructor is called.

→ Properties :- ① Name → class name
② No return type
③ No i/p parameter

Ex:

```
//Destructor
~Hero(){
    cout<<"Destructor bhai called"<<endl;
}
```

⊙ Symbol for Destructor (~) (tilde)

⇒ Static Allocation → called Automatically
(Hero a)

⇒ Dynamic Allocation → Manually { Hero *b = new Hero
delete b; }
(Hero *b = new Hero)

#Static keyword : (Declare variables & fn at global scope)

- Create a data member that belongs to class.
- To access it, no need to create any object.
- Initialize (outside class)

Syntax:

```
data_type class_name :: name = value;
                  ↳ scope resolution operator
```

Ex⇒

```
Class Hero{
    Static int timeToComplete;
}
int Hero::timeToComplete = 5;
int main(){
    cout << Hero :: timeToComplete << endl;
}
```

Static functions :-

Properties → ① No need to create any object.
② No (this →) keyword
③ Can only access static member

Ex⇒

```
Static int random(){
        return timeToComplete;
    }
int main(){
    cout << Hero :: random() << endl;
}
```

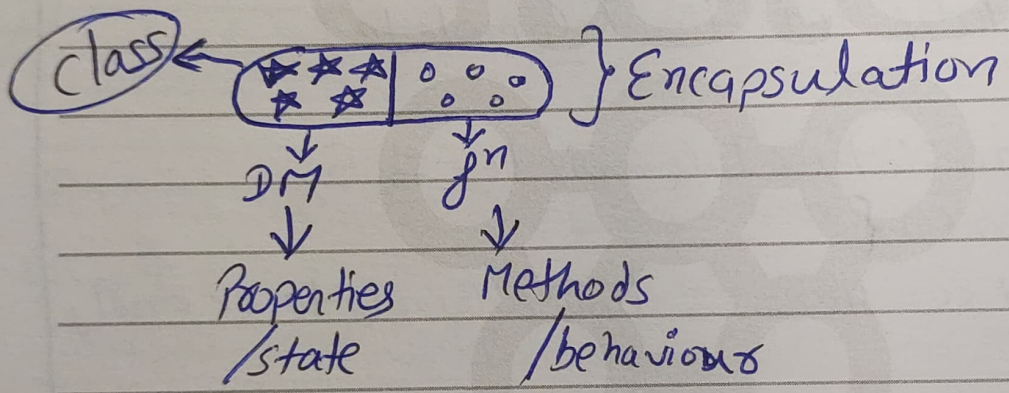
4 Pillars of OOPs Concept

① Encapsulation :- (Information Hiding or Data Hiding)

Wrapping up Data Member & functions.

Data Members → ★★★ ★

Functions → ∘∘∘ ∘∘



⇒ Fully encapsulated class :-

all → D.M → Private

```
Private:
  int num;
  string label;
```

→ can only access in same class

why?

Advantage:

① Data Hide → Security (↑es)

② If we want, we can make class "Read only"

③ Code Reusability

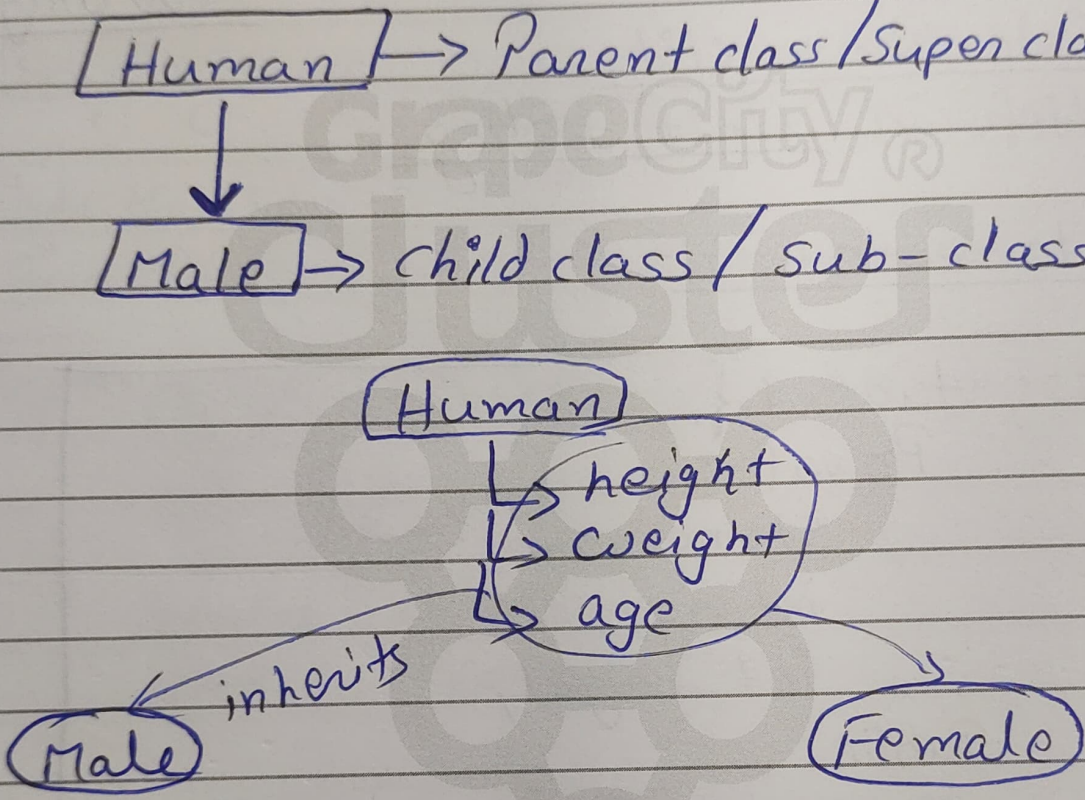
④ Helps in Unit testing

⇒ Implementation:-

```
class student{
        Private:
                string name;
                int age;
                int height;
        Public:
          int getAge() {
                return this->age;
            }
    };
int main(){
     Student first;
     cout << "All is well" << endl;
  }
```

(2) Inheritance :-

One class inherits the attributes & methods of another class.



Syntax:

```
class parent_class {
};

class child_class : access_modifier parent_class
{
};
```

```
class human{
    Public:
    int weight;
    int height;
    int age;
  int getage(){ return this->age;}
  void setweight(int w){ this->weight=w;}
};

class male:public human{
    Public:
    String colour;
    void sleep(){
        cout<<"male sleeping"<<end;
    }
};

int main(){
    male ram;        -> child class object
    cout<<ram.height<<endl;
    cout<<ram.age<<endl;
    ram.setweight(62);
    cout<<ram.weight<<endl;
    ram.sleep();
    cout<<ram.colour<<endl;
}
```

can access all D.M of parent class

⇒ Modes of Inheritance :

Base class member Access specifier	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible	NA	NA

∴ Private D.M. of any class can not be inherited.

⇒ protected → Similar private

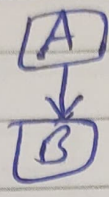
- accessible inside own class
- condition
  - child class / Derived class } can access

⇒ Types of Inheritance :—

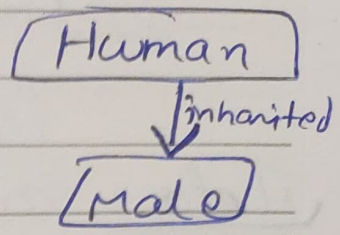
- Single
- Multi-level
- Multiple
- Hybrid
- Hierarchical

① Single Inheritance:

In this a single derived class, is inherited from a single base class.

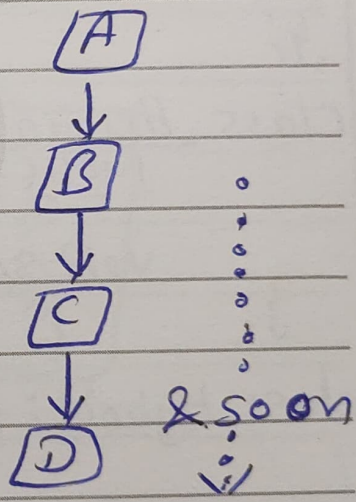


(or)



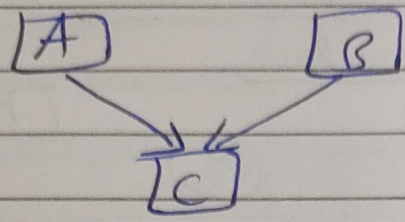
② Multilevel Inheritance:

when a class B is derived from a parent class A and another class C is derived from a class B.




```
Class animal {
        Public:
            void speak() { ... }
        };
class dog : public Animal {
        };
class Labra : public dog {
        };
int main() { labra one;
             one.speak();
        }
```

③ Multiple Inheritance:

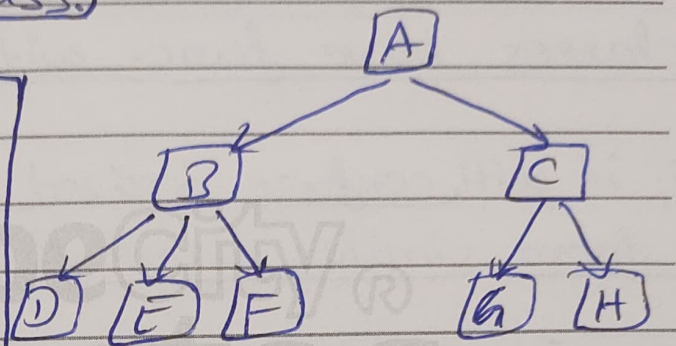


A class can inherit from more than one class.

```
class animal{
    Public:
    void bank(){....}
};
class Human{
    Public:
    void speak(){....}
};
class Hybrid: public animal, public Human{
};
int main(){
    Hybrid obj;
    obj.speak();
    obj.bank();
}
```

④ Hierarchical Inheritance:

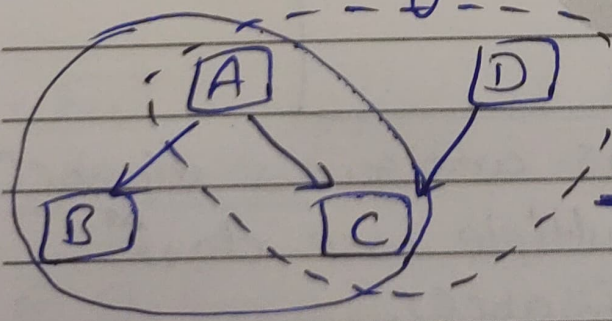
Several classes are derived from one common base class.




```
class A{ fn1()
};
class B: public A{ fn2()
};
class C: public A{ fn3()
};
main(){ A obj1;
        obj1.fn1();
B obj2;
obj2.fn1();
obj2.fn2();
C obj3;
obj3.fun1();
obj3.fn3();
}
```

⑤ Hybrid Inheritance:

Combination of more than 1 type of Inheritance.



'Multiple

Hierarchical

```
class A{        class D{
};              };
class B: public A{
};
class C: pubic A, public D
{ . . .
};
```

⇒ Inheritance Ambiguity :

when one class is derived from two or more base classes then there are chances that the classes have func. with same name.

So, it will confuse derived class to choose from similar name f^n.

```
class A{ public:
    void abc(){ I'm in A }
};
class B { Public
    void abc() { I'm in B }
};
class C: public A, public B { Public
            };
int main(){
        C obj;
        obj.abc();
    }
```

→ error: 'abc' is ambiguous obj.abc();

(or) 'abc' found in multiple base classes of different types obj.abc();

Sol^n :

obj.A::abc(); → I'm in A
obj.B::abc(); → I'm in B

Visibility	Private	Protected	Public
within the same class	Yes	Yes	Yes
In derived class	No	Yes	Yes
Outside the class	No	No	Yes

(3) Polymorphism :—

Single thing existing in multiple forms.

(Poly → many
morph → forms)

Types
① → Compile time polymorphism (static poly. / early binding)
② → Run time polymorphism (Dynamic poly.)

① Compile Time Polymorphism :

Fun is called at the time of program compilation.

- Achieved by fn overloading or operator overloading.

Types → (a) Function overloading
→ (b) operator overloading

(a) Function Overloading :

Same f^n name but different input argument

int fun(int n) **or** int fun(string S)
int fun(int m, int p) int fun(char C)

or

int fun(int x, int y, int w=0, int z=0)

Default argument (da)

if value is passed for these ↑, then da is ignored.

⊙ only change in return type doesn't mean overloading, arguments must be different.

(b) Operator Overloading :

It is an idea of giving special meaning to an existing operator, in C++ without changing its original meaning.

Ex:→ + → 2 integer → Subtract

⊙ '+' is a binary operator (a+b)
⊙ '++' is a unary operator (++i or i++)
'--' " "
'!', '&', "sizeof" ,,

Syntax:

```
return_type operator+ ( ){
}
```

i/p argument ↓

a + b
↓ curr. obj.
↓ i/p argum.

ex:

```
class B{ public: int a; int b;
  int add (){ return a+b; }
  void operator+(B &obj){
      int value1 = this->a;
      int value2 = obj.a;
      cout << value2 - value1 << endl;
  }
};
int main (){
    B obj1, obj2;
    obj1.a = 4;
    obj2.a = 7;
    obj1 + obj2;        // output => 3
}
```

operator overloading

or

```
cout << "Hello world";
output => Helloworld
```

also

```
void operator() (){
    cout << " Mai bracket hu" << this->a << endl;
}
main(){
    obj1();
}
```

output => mai bracket hu 4

② Run Time Polymorphism :

⊙ Achieved by f^n overriding

(a) Method Overriding / f^n overriding :

Occurs when a derived class has a definition for one of the member f^n of the base class.

```
class Animal {
      Public:
      void speak()
     { cout << "speaking" ;}
  };
class Dog : public Animal { Public:
            void speak(){
              cout<< "Barking"; }
          };
```

Rules: ① F^n name must be same
② F^n argument / parameter must be same
③ Possible through Inheritance only.

```
int main() { Dog obj;
             obj.speak(); // Barking (output)
    }
```

④ Abstraction : (Implementation Hiding)

- Most essential & important features of OOPs.
- Display only essential information & hiding the details.

Advantages →

1. only you can make changes to your data or f^n.
2. (↑) reusability of the code.
3. Avoid duplication of your code.

⇒ Interfaces:

It is a way to describe the behaviour of a class without ~~taking~~ committing the ~~permi~~ implementation of the class.

- C++ interface is a pure virtual f^n

In order to create interface, we need to create abstract class which is having only pure virtual methods.

Syntax: virtual datatype f^nname(Param1, Parm2...)=0

Ex: virtual void sound() = 0;

⇒ Friend class :- / Friend Function

They can access private, and protected members of other classes in which it is declared as a friend.

ex:- (friend class)

```
class GFG{
Private:
  int a;
Protected:
  int b;
Public:
  fun(){
    a=2;
    b=3;}

  friend class F;
};

class F{
  Public:
  void display(GFG &t){
  cout<<t.a<<" "<<t.b;}
};
int main(){ GFG g;
            F fi;
            fi.display(g);
            return 0;
}
```

(friend fn)

```
class GFG{

  _ _
  _ _
  _ _
  _ _
  _

⇒ friend void func(GFG &obj);
};

void fun(GFG &obj)
  { cout<<obj.a<<" "<<obj.b;
  }

int main(){
  GFG obj;
  fun(obj);
```