

Agenda

- Hierarchy and its type.
- Association
- Inheritance
- Type and Mode of Inheritance
- Diamond problem
- Virtual base class
- RunTime Polymorphism

Hierarchy

- It is a major pillar of oops.
- Level / order / ranking of abstraction is called hierarchy.
- Its main purpose is to achieve reusability.
- Advantages of reusability:
 1. To reduce developers efforts.
 2. To reduce development time and development cost.

Types of Hierarchy:

1. Has-a/Part-of Association/Containment
2. Is-a/Kind-of Inheritance/Generalization
3. Use-a Dependency
 - This hierarchy represents how classes depend on each other.
 - Dependencies occur when one class relies on another class but does not own or control its lifetime.
 - For example, if Class A uses Class B as a method parameter or local variable, there's a dependency between A and B.
4. Creates-a Instantiation
 - This can often be seen in factory design patterns or in scenarios where one class encapsulates the creation logic of another class.

Association

- If has-a relationship exist between two types then we should use association.
- Example:
 1. Room has-a wall
 2. Room has-a chair
 3. Car has-a engine
 4. Car has-a music player
 5. Department has-a faculty
 6. Human has-a heart
- If object is part-of / component of another object then it is called association.
- Composition and aggregation are specialized form of association.
- If we declare object of a class as a data member inside another class then it represents association.

```
class Engine{
};
class Car{
    private:
        Engine e; //Association
};
int main( void ){
    Car car;
    return 0;
}

//Dependant Object : Car Object
//Dependency Object : Engine Object
```

1. Composition

- If dependency object do not exist without Dependant object then it represents composition.
- Composition represents tight coupling.
- If we create object of dependency class as data member inside the dependent class it represents composition.

2. Aggegration

- If dependency object exist without Dependant object then it represents Aggregation.
- Aggregation represents loose coupling.

Inheritance

- If "is-a" relationship exist between two types then we should use inheritance.
- Inheritance is also called as "Generalization".
- Consider example:
 1. Employee is-a Person
 2. Book is-a product
 3. Car is-a Vehicle
 4. Rectangle is-a Shape
 5. Loan Account is-a Account

```
//Parent class
class Person//Base class
{
};

//Child class
class Employee:public Person//Derived class
{
};
```

- During inheritance, members of base class inherit into derived class.
- If we create object of derived class then non static data members declared in base class get space inside it. In other words non static data members of base class inherit into derived class.
- Using derived class name, we can access static data member(if public) of base class. In other words, static data member inherit into derived class.
- All the data members(private/protected/public, static/non static) of base class inherit into derived class but only non static data members get space inside object.
- Size of object = sum of size of non static data members declared in base class and derived class.
- We can call non static member function of base class on object of derived class. In other words, using derived class object we can call non static member function of base class. It means that, non static member function inherit into derived class.
- We can call static member function of base class on derived class. In other words, using derived class name, we can access static member function of base class. It means that, static member function inherit into derived class.
- Following function's do not inherit into derived class:
 1. Constructor
 2. Destructor
 3. Copy constructor
 4. Assignment operator function
 5. Friend Function
- Except above five function's, all the member's of base class(data member, member function and nested type) inherit into derived class.
- If we create object of derived class then first base class and then derived class constructor gets called.
- Destructor calling sequence is exactly opposite.
- From derived class constructor, by default, base class's parameterless constructor gets called.
- Using constructors base initializer list, we can call any constructor of base class from constructor of derived class.
- In C++, we can not call constructor on object, pointer or reference explicitly. But constructor's base initializer list represent explicit call to the constructor.
- We can read following statement using 2 ways:
 - `class Employee : public Person`
 1. Class Person is inherited into class Employee.
 2. Class Employee is derived from class Person(Recommended).
- Process of acquiring/getting/accessing properties(data members) and behavior (member function) of base class inside derived class is called inheritance.
- Every base class is abstraction for the derived class.

Mode of inheritance

- If we use private/protected/public keyword to control visibility of members of class then it is called access specifier.
- If we use private/protected/public keyword to extend the class then it is called mode of inheritance.
- In below statement, mode of inheritance is public if we don't mention then the default mode of inheritance is private.

```

class Employee : public Person

class Employee : Person
//is equivalent to
class Employee : private Person

```

- In private mode of inheritance, the visibility of base class members that inherit inside the derived class is made as private inside the derived class
- In protected mode of inheritance except private members, the visibility of base class members that inherit inside the derived class is made as protected inside the derived class
- In public mode of inheritance, the visibility of base class members that inherit inside the derived class does not change inside the derived class
- In all types of mode, private members inherit into derived class but we can not access it inside member function of derived class.
- If we want to access private members inside derived class then
 1. Either we should use member function(getter/setter).
 2. or we should declare derived class as a friend inside base class.
- If we want to create object of derived class then constructor of base class and derived must be public

Mode Of inheritance – Private, Protected & Public

Irrespective of Mode of Inheritance				Public Mode of Inheritance		
Access Specifiers	Same Class	Friend Function	Non Member Function	Access Specifiers from Base class	Derived Class	Indirect Derived Class
private	A	A	NA	private	NA	NA
protected	A	A	NA	protected	A	A
public	A	A	A	public	A	A
Private Mode of Inheritance				Protected Mode of Inheritance		
Access Specifiers from Base class	Derived Class	Indirect Derived Class		Access Specifiers from Base class	Derived Class	Indirect Derived Class
private	NA	NA		private	NA	NA
protected	A	NA		protected	A	A
public	A	NA		public	A	A

Types of inheritance

1. Single Inheritance

- class B is derived from class A
- If single base class is having single derived class then it is called single inheritance.

```

class A{
};
class B : public A{
};

```

2. Multiple Inheritance

- class D is derived from class A, B and C
- If multiple base classes are having single derived class then it is called multiple inheritance

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

3. Hierarchical Inheritance

- class B, C and D are derived from class A
- If single base class is having multiple derived classes then such inheritance is called hierarchical inheritance.

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

4. Multilevel Inheritance

- class B is derived from class A, class C is derived from class B and class D is derived from class C.
- If single inheritance is having multiple levels then it is called multilevel inheritance.

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

Hybrid Inheritance

- Combination of any two or more than two types of inheritance is called hybrid inheritance.

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

Points to Remember

- According to client's requirement, if implementation of existing class is logically incomplete / partially complete then we should extend the class i.e we should use inheritance.
- In other words, without changing implementation of existing class, if we want to extend meaning of that class then we should use inheritance.
- According client's requirement, if implementation of base class member function is logically incomplete then we should redefine function in derived class.
- If name of members of base class and derived class are same then derived class members hides implementation of base class members. Hence preference is given to the derived class members.
- This process is called shadowing.
- If we want to access members of base class inside member function of derived class then we should use classname and scope resolution operator.

Diamond Problem

- It is hybrid inheritance. Its shape is like diamond hence it is also called as diamond inheritance.

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

- Data members of indirect base class inherit into the indirect derived class multiple times.
- Hence it effects on size of object of indirect derived class.
- Member functions of indirect base class inherit into indirect derived class multiple times.
- If we try to call member function of indirect base class on object of indirect derived class, then compiler generates ambiguity error.
- If we create object of indirect derived class, then constructor and destructor of indirect base class gets called multiple times.
- All above problems generated by hybrid inheritance is called diamond problem.

- If we want to overcome diamond problem, then we should declare base class virtual i.e. we should derive class B & C from class A virtually. It is called virtual inheritance. In this case, members of class A will be inherited into B & C but it will not be inherited from B & C into class D.

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

RunTime Polymorphism

- During inheritance, members of base class inherit into derived class hence using derived class object, we can access members of base class as well as derived class.
- Members of derived class do not inherit into the base class hence using base class object we can access members of base class only.
- Members of base class inherit into derived class hence derived class object can be considered as base class object.
- Example : Employee object is-a Person object.
- Since Derived class object can be considered as Base class object, we can use it in place of Base class object.

```
Base b1;
Base b2 = b1; //OK
Derived d1;
b1 = d1; //OK
```

```
Base *ptr = NULL;
ptr = new Base(); // OK
ptr = new Derived(); // OK
```

- If we assign derived class object to the base class object then compiler copies state of base class portion from derived class object into base class object. It is called **Object slicing**.
- During Object slicing, mode of inheritance must be public.

```
class Base{
public:
    int n1,n2;
    void printBase(){
        cout<<num1<<" , "<<num2<<endl;
    }
};
```

```

}
class Derived:public Base(){
public:
int n3;
Derived(int n1,int n2,int n3){
    this->n1=n1;
    this->n2=n2;
    this->n3=n3;
}
}
int main( void )
{
Base base;
Derived derived( 500,600,700);
base = derived; //OK : Object Slicing
base.printRecord(); //Base::printRecord() : 500,600
return 0;
}

```

- Members of derived class do not inherit into base class. Hence base class object, can not be considered as derived class object.
- Since base class object, can not be considered as derived class object, we can not use it in place of derived class object.

```

Derived *ptr = NULL;
ptr = new Derived();//Ok

ptr = new Base();//Not ok

```

- Process of converting, pointer of derived class into pointer of base class is called upcasting.
- Upcasting represents object slicing.
- In case of upcasting, explicit type casting is optional.
- Main purpose of upcasting is to reduce object dependency in the code.
- Process of converting pointer of base class into pointer of derived class is called downcasting.
- In Case of downcasting, explicit typecasting is mandatory.
- Note: Only in case of upcasting, we can do downcasting. Otherwise downcasting will fail.

```

int main( void )
{
    Base *ptrBase = new Derived( ); //Upcasting
    ptrBase->printRecord(); //Base::printRecord()

    Derived *ptrDerived = ( Derived*)ptrBase;//Downcasting
    ptrDerived->printRecord();

    return 0;
}

```


