Inheritance in C++

- The capability of a class to derive properties and characteristics from another class is called **Inheritance**.
- Inheritance is one of the most important features of Object-Oriented Programming.
- Inheritance is a feature or a process in which, new classes are created from the existing classes.
- The new class created is called “derived class” or “child class” and the existing class is known as the “base class” or “parent class”.
- The derived class now is said to be inherited from the base class.
- When we say derived class inherits the base class, it means, the derived class inherits all the properties of the base class, without changing the properties of base class and may add new features to its own.
- These new features in the derived class will not affect the base class.
- The derived class is the specialized class for the base class.
- **Sub Class:** The class that inherits properties from another class is called Subclass or Derived Class.
- **Super Class:** The class whose properties are inherited by a subclass is called Base Class or Superclass.

Why and when to use inheritance?

- Consider a group of vehicles. You need to create classes for Bus, Car, and Truck.
- The methods fuelAmount(), capacity(), applyBrakes() will be the same for all three classes.
- If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown below figure:
- Using inheritance, we have to write the functions only one time instead of three times as we have inherited the rest of the three classes from the base class (Vehicle).

Implementing inheritance in C++

- For creating a sub-class that is inherited from the base class we have to follow the below syntax.
- **Derived Classes:** A Derived class is defined as the class derived from the base class.

```
class  <derived_class_name> : <access-specifier> <base_class_name>

{

      //body

}
```

Where
class — keyword to create a new class

derived_class_name — name of the new class, which will inherit the base class
access-specifier — either of private, public or protected. If neither is specified, PRIVATE is taken as default
base-class-name — name of the base class

- **Note**: A derived class doesn't inherit ***access*** to private data members. However, it does inherit a full parent object, which contains any private members which that class declares.

Example:

1. class ABC : private XYZ //private derivation

   { }

2. class ABC : public XYZ //public derivation

   { }

3. class ABC : protected XYZ //protected derivation

   { }

4. class ABC: XYZ //private derivation by default

{ }

- When a base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class and therefore, the public members of the base class can only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class.
- On the other hand, when the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the derived class.

Modes of Inheritance

- There are 3 modes of inheritance.

1. **Public Mode**: If we derive a subclass from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in the derived class.
2. **Protected Mode**: If we derive a subclass from a Protected base class. Then both public members and protected members of the base class will become protected in the derived class.
3. **Private Mode**: If we derive a subclass from a Private base class. Then both public members and protected members of the base class will become Private in the derived class.

- **Note:** The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed.
- // C++ Implementation to show that a derived class doesn't inherit access to private data members.

```
    // However, it does inherit a full parent object.
    class A {
    public:
        int x;

    protected:
        int y;

    private:
        int z;
    };

    class B : public A {
        // x is public
        // y is protected
        // z is not accessible from B
    };
class C : protected A {
    // x is protected
    // y is protected
    // z is not accessible from C
};

class D : private A // 'private' is default for classes
{
    // x is private
    // y is private
    // z is not accessible from D
};
```

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

A derived class inherits all base class methods with the following exceptions −

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.
- The friend functions of the base class.

Types Of Inheritance

- Single inheritance
- Multilevel inheritance
- Multiple inheritance
- Hierarchical inheritance
- Hybrid inheritance

Single Inheritance

In single inheritance, a class is allowed to inherit from only one class. i.e. one subclass is inherited by one base class only

class subclass_name : access_mode base_class

{

// body of subclass

};

OR

class A

{

... .

};

class B: public A

{

... .

};

Multiple Inheritance

- Multiple Inheritance is a feature of C++ where a class can inherit from more than one class. i.e one **subclass** is inherited from more than one **base class**.

class subclass_name : access_mode base_class1, access_mode base_class2, ...

{

// body of subclass

};

class B

{

... .

};

class C

{

... .

};

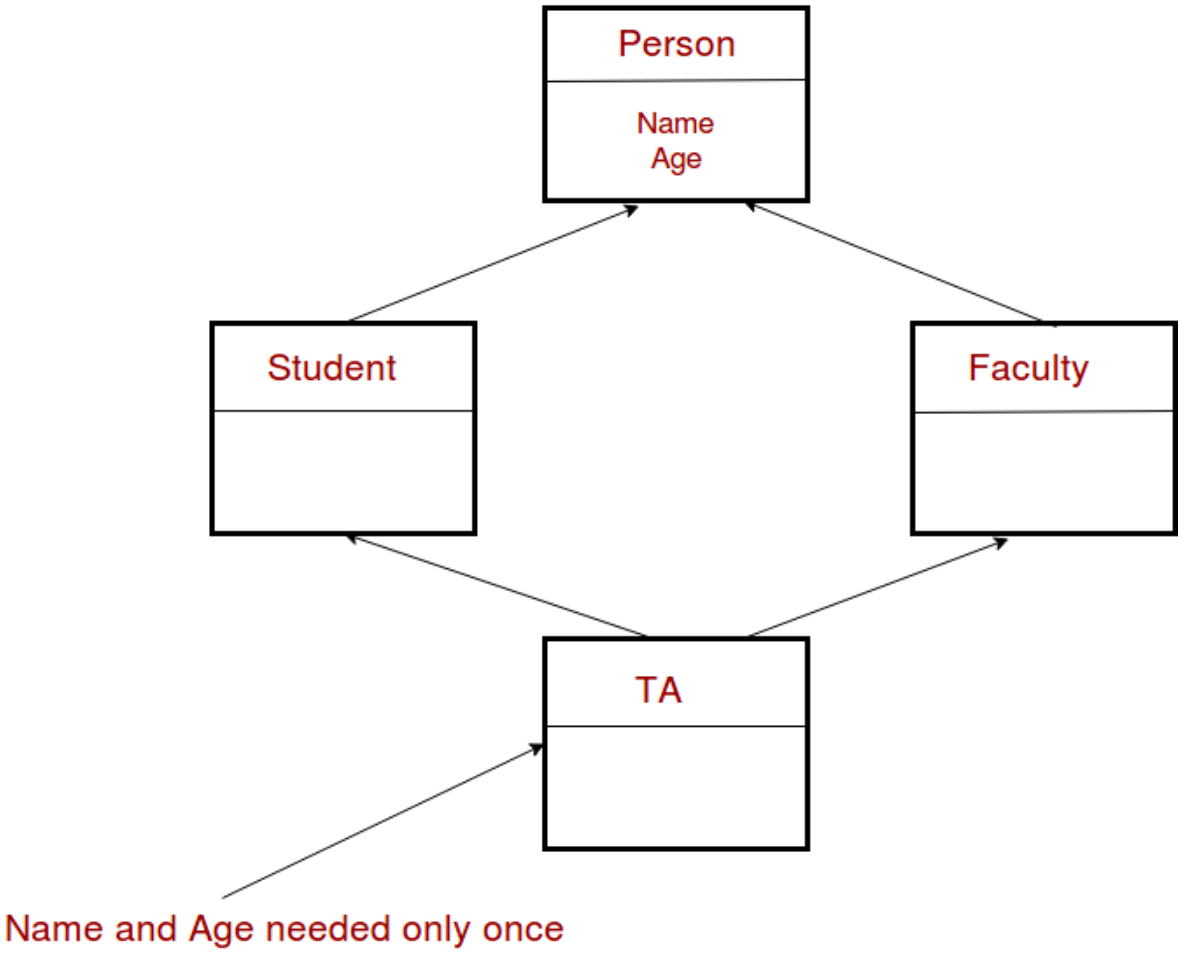
class A: public B, public C

{

... ...

};

- Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes.
- The constructors of inherited classes are called in the same order in which they are inherited.
- The destructors are called in reverse order of constructors
- **The diamond problem** The diamond problem occurs when two superclasses of a class have a common base class. For example, in the following diagram, the TA class gets two copies of all attributes of Person class, this causes ambiguities.



- Constructor of ‘Person’ is called two times. Destructor of ‘Person’ will also be called two times when object ‘ta1’ is destructed.
- So object ‘ta1’ has two copies of all members of ‘Person’, this causes ambiguities
- *The solution to this problem is ‘virtual’ keyword*.
- We make the classes ‘Faculty’ and ‘Student’ as virtual base classes to avoid two copies of ‘Person’ in ‘TA’ class.
- One important thing to note in the above output is, *the default constructor of ‘Person’ is called*.
- When we use ‘virtual’ keyword, the default constructor of grandparent class is called by default even if the parent classes explicitly call parameterized constructor.

Multilevel Inheritance

- In this type of inheritance, a derived class is created from another derived class.

```
class C
{
... .. ...
```

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

Hierarchical Inheritance

- In this type of inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class.

```
class A
{
   // body of the class A.
}
class B : public A
{
   // body of class B.
}
class C : public A
{
   // body of class C.
}
class D : public A
{
   // body of class D.
}
```

Hybrid (Virtual) Inheritance

- Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.

A special case of hybrid inheritance: Multipath inheritance

- A derived class with two base classes and these two base classes have one common base class is called multipath inheritance. Ambiguity can arise in this type of inheritance.
- **There are 2 Ways to Avoid this Ambiguity:**
    - Using the scope resolution operator we can manually specify the path from which data member a will be accessed. (obj.ClassB::a = 10;)
    - ***Note:*** Still, there are two copies of ClassA in Class-D.
    - **Avoiding ambiguity using the virtual base class:**

Virtual base class in C++

- Virtual base classes are used in virtual inheritance in a way of preventing multiple “instances” of a given class appearing in an inheritance hierarchy when using multiple inheritances.

Constructors of Derived class

- Whenever we create an object of a class, the default constructor of that class is invoked automatically to initialize the members of the class.
- If we inherit a class from another class and create an object of the derived class, it is clear that the default constructor of the derived class will be invoked but before that the default constructor of all of the base classes will be invoke, i.e the order of invocation is that the base class’s default constructor will be invoked first and then the derived class’s default constructor will be invoked.
- **Why the base class’s constructor is called on creating an object of derived class?**
- What happens when a class is inherited from other? The data members and member functions of base class comes automatically in derived class based on the access specifier but the definition of these members exists in base class only. So when we create an object of derived class, all of the members of derived class must be initialized but the inherited members in derived class can only be initialized by the base class’s constructor as the definition of these members exists in base class only. This is why the constructor of **base class is called first to initialize all the inherited members.**

Inheritance and Friendship in C++

- **Inheritance in C++:** This is an OOPS concept. It allows creating classes that are derived from other classes so that they automatically include some of the functionality of its base class and some functionality of its own.
- **Friendship in C++:** Usually, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, a friend class has the access to the protected and private members of the first one. Classes that are ‘friends’ can access not just the public members, but the private and protected members too.

- In C++, friendship is not inherited. If a base class has a friend function, then the function doesn't become a friend of the derived class(es).
- If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.
- By using the keyword friend compiler knows the given function is a friend function.
- For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.
- The function can be defined anywhere in the program like a normal C++ function. The function definition does not use either the keyword **friend or scope resolution operator**.

Const member functions in C++

- Constant member functions are those functions that are denied permission to change the values of the data members of their class. To make a member function constant, the keyword const is appended to the function prototype and also to the function definition header.
- Like member functions and member function arguments, the objects of a class can also be declared as const. An object declared as const cannot be modified and hence, can invoke only const member functions as these functions ensure not to modify the object. A const object can be created by prefixing the const keyword to the object declaration. Any attempt to change the data member of const objects results in a compile-time error.

C++ Operators Overloading

- Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type.
- Operator overloading is used to overload or redefines most of the operators available in C++.
- It is used to perform the operation on the user-defined data type.
- **Operator that cannot be overloaded are as follows:**
- Scope operator (::)
- Sizeof
- member selector(.)
- member pointer selector(*)
- ternary operator(?:)

Syntax of Operator Overloading

```
return_type class_name  : : operator op(argument_list)
{
    // body of the function.
}
```

- Where the **return type** is the type of value returned by the function.
- **class_name** is the name of the class.
- **operator op** is an operator function where op is the operator being overloaded, and the operator is the keyword.