Unit 4: Inheritance and Reusability

4.1 Introduction to Inheritance

- It is the process of creating new classes by extending the features of existing classes without modifying the existing classes.
- The existing class is also termed as Base Class or Super Class or Parent Class and the new class is also termed as Derived Class or Sub Class or Child Class.
- The derived class possesses all features of the base class, and in addition can have its own set of new features.
- It is one of the most powerful and striking features of OOP which promotes software code reusability by allowing new specialized classes to be built from existing general classes without writing everything from scratch.
- Reusing more reliable "tried and tested" code saves time, effort, and money.
- Additionally, it provides a mechanism for organizing information in a hierarchical form – also known as the inheritance hierarchy.

Syntax for Inheritance in C++:

```
class ABC{
      // features of ABC
};


class XYZ : access_mode ABC{
      // features of XYZ
      // features of ABC is automatically inherited
}
```

Here, ABC and XYZ are names of classes just used as examples. ABC is the base class and XYZ is the derived class which can have its own set of features along with features inherited from the ABC.

In C++, private members of the base class are inherited but are not accessible directly from the derived class. Whereas, protected and public members of the base class are accessible from the derived class.

Access modes (Inheritance Modes)

Access mode is the mode of visibility used during inheritance which can be any of: private, public, or protected.

i) Private Inheritance:

When inheritance is done in private mode, both the protected and public members of the base class become equivalent to private in the derived class. In C++, if no access mode is mentioned, then inheritance is private by default.

ii) Protected Inheritance:

When inheritance is done in protected mode, the protected and public members of the base class both become equivalent to protected in the derived class.

iii) Public Inheritance:

When inheritance is done in public mode, the protected and public members of the base class both remain same in the derived class.

4.2 Subclass, Subtype and Principle of Substitutability

When a new class is created from an existing class via inheritance, the new class becomes a subclass of the base class or parent class and the process is also called subclassing. In general, a subclass may exhibit a different set of behaviors than its parent class.

When the subclass is constructed in such a way that it behaves like the parent class, then the child class can be termed as a subtype of the parent class so that the parent class or the supertype can easily be replaced by its subtype without affecting the program.

Principle of Substitutability

> "Let $\phi(x)$ be a property provable about objects x of type T. Then, $\phi(y)$ should be true for objects y of type S where S is a subtype of T."
>
> - Barbara Liskov (1987)

The Liskov Principle of substitutability states that the relationship of subtypes and supertypes should ensure that any properly provable about supertype objects should also hold for its subtype objects.

In other words, it means that a superclass object should be replaceable by its subclass object without breaking the functionality of the program.

It is a type of strong behavioral subtyping defined by semantic rather than syntactic design consideration.

It makes heavy use of inheritance and runtime polymorphism features.

Different reasons for subclassing

Subclassing or inheritance is done for the following surprisingly different reasons:

1) **Subclassing for Specialization**
   The child class is a special case of the parent class; in other words, the child class is a subtype of the parent class.

2) **Subclassing for Specification**
   The parent class defines behavior that is implemented in the child class but not in the parent class. In other words, the parent class defines interface or contract that must be implemented by all the child classes.

3) **Subclassing for Construction**
   The child class makes use of the behavior provided by the parent class, but is not a subtype of the parent class.

4) **Subclassing for Generalization**
   The child class modifies or overrides some of the methods of the parent class to construct a more general class.

5) **Subclassing for Extension**
   The child class adds new functionality to the parent class, but does not change any inherited behavior.

6) **Subclassing Limitation**
The child class restricts the use of some of the behavior inherited from the parent class.

7) **Subclassing for Variance**
The child class and parent class are variants of each other, and the class-subclass relationship is arbitrary. It is better to implement this scenario as a hierarchical inheritance by introducing a new parent class common to these classes.

8) **Subclassing for Combination**
The child class inherits features from more than one parent class. This is implemented as multiple inheritance.

4.3 Forms of Inheritance

a) Single Inheritance: It is the simplest kind of inheritance where a single derived class is inherited from a single base class.

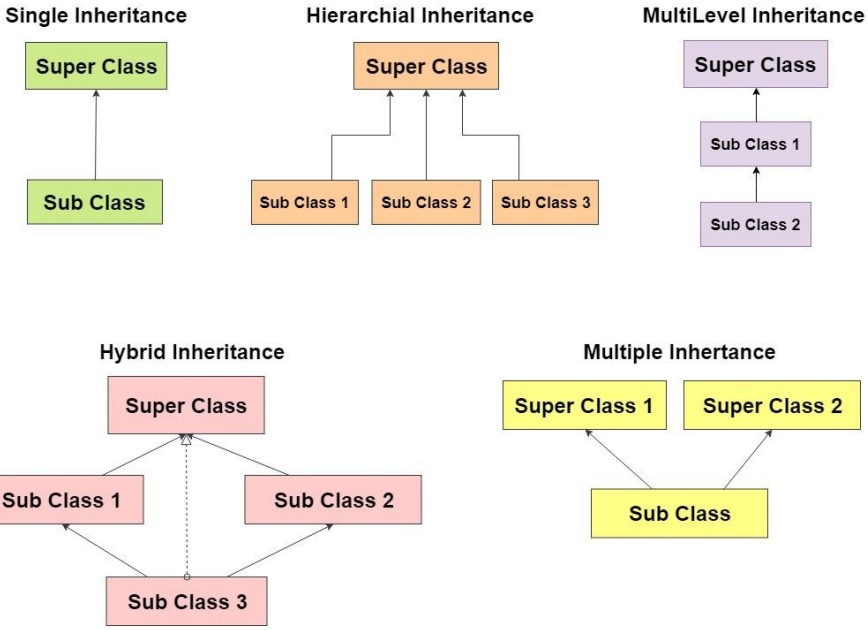
b) Multi-Level Inheritance: When a class is derived from another class which is already derived from another class, this is known as multi-level inheritance.

c) Hierarchical Inheritance: When a single base class is used to inherit two or more derived classes and consequently, these derived classes are used to inherit more derived classes, thus forming a tree structure of inheritance hierarchy, this is known as hierarchical inheritance.

d) Multiple Inheritance: When a single derived class is inherited from two or more base classes, this is known as multiple inheritance.

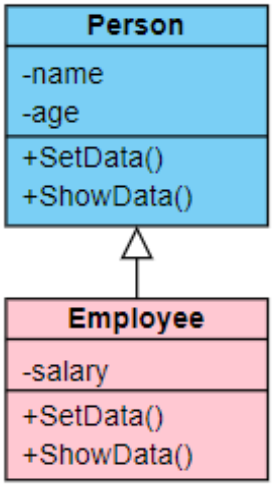
e) Hybrid Inheritance: When multiple forms of inheritance is mixed together, we have hybrid inheritance.

Multi-Path Inheritance: It is one of the most popular examples of Hybrid Inheritance, which is a hybrid of all of the above. First, two or more classes are derived from a single base class as in hierarchical inheritance. Then another derived class is formed from these derived classes.



C++ Examples / Implementation

a) Single Inheritance Example:



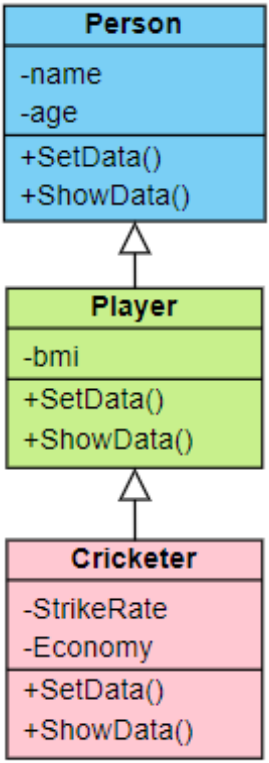

```
#include <iostream>
using namespace std;

class Person{
  protected:
    string name;
    int age;

  public:
    void SetData(string n, int a)
    {
        name = n;
        age  = a;
    }
```

```
    void ShowData()
    {
        cout<<endl<<"Name :" << name;
        cout<<endl<<"Age  :" << age;
    }
};

class Employee : public Person
{
  private:
    float salary;

  public:
    void SetData(string n, int a, float sal)
    {
        Person::SetData(n, a);
        salary = sal;
    }

void ShowData()
    {
        Person::ShowData();
        cout << endl << "Salary :" << salary;
    }
};

int main()
{
    Person   p1;
    Employee e1;

    p1.SetData("Ramesh", 22);
    e1.SetData("Suresh", 23, 33000.0);

    p1.ShowData();
    cout << endl << endl ;
    e1.ShowData();

    return 0;
}
```

Here, the Employee class has been derived publicly from the Person class. Hence, the protected members of Person are inherited as protected in Employee and the public members of Person class are inherited as public in Employee class.

The SetData member function of Person has been overloaded in Employee whereas the ShowData member function of Person has been overridden in Employee.

b) Multi-Level Inheritance Example:

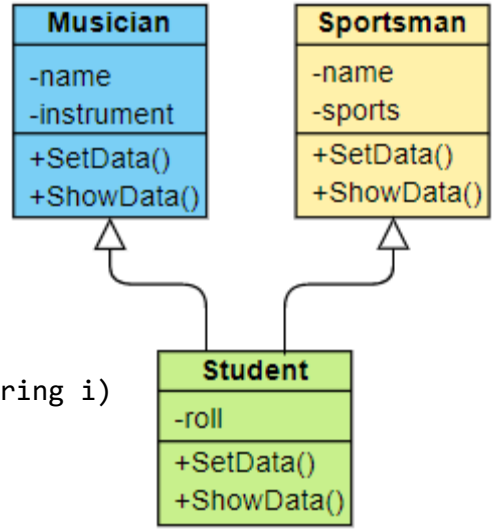



```
#include <iostream>
using namespace std;

class Person{
  protected:
    string name;
    int    age;

  public:
    void SetData(string n, int a)
    {
        name = n;
        age  = a;
    }

    void ShowData()
    {
        cout << endl << "Name: " << name;
        cout << endl << "Age : " << age;
    }
};

class Player : public Person{
  protected:
    float bmi;

  public:
    void SetData(string n, int a, float b)
```

```
    {
        Person::SetData(n, a);
        bmi  = b;
    }

    void ShowData()
    {
        Person::ShowData();
        cout << endl << "BMI  : " << bmi;
    }
};

class Cricketer : public Player{
  protected:
    float StrikeRate;
    float Economy;

  public:
    void SetData(string n, int a, float b, float sr, float be)
    {
        Player::SetData(n, a, b);
        StrikeRate     = sr;
        BallingEconomy = be;
    }

    void ShowData()
    {
        Player::ShowData();
        cout << endl << "Strike Rate     :" << StrikeRate;
        cout << endl << "Balling Economy :" << Economy;
    }
};

int main()
{
    Person p1;
    p1.SetData("Anmol", 23);
    p1.ShowData();
    cout << endl << "---------------------------" << endl;
```

```
    Player q1;
    q1.SetData("Ronaldo", 33, 25.4);
    q1.ShowData();
    cout << endl << "--------------------------" << endl;

    Cricketer c1;
    c1.SetData("Sandeep", 40, 24.3, 120, 7.5);
    c1.ShowData();

    return 0;
}
```

c) Multiple Inheritance example



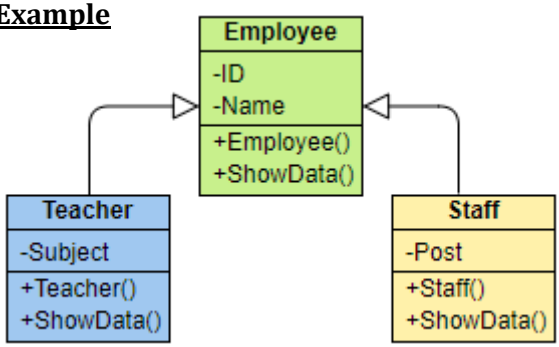

```
#include <iostream>
using namespace std;

class Musician{
  protected:
    string name;
    string instrument;

  public:
    void SetData(string n, string i)
    {
        name = n;
        instrument = i;
    }

    void ShowData()
    {
        cout << endl << name << " plays " << instrument;
    }
};

class Sportsman{
  private:
    string name;
    string sports;
```

```
  public:
    void SetData(string n, string s)
    {
        name   = n;
        sports = s;
    }

    void ShowData()
    {
        cout << endl << name << " likes " << sports;
    }
};

class Student: public Musician, public Sportsman{
  private:
    int roll;

  public:
    void SetData(int r, string n, string i, string s)
    {
        Musician::SetData(n, i);
        Sportsman::SetData(n, s);
        roll = r;
    }

    void ShowData()
    {
        cout << "Roll no: " << roll;
        Musician::ShowData();
        Sportsman::ShowData();
    }
};

int main()
{
    Student s;
    s.SetData(747, "Krishna", "Guitar", "Basketball");
    s.ShowData();
    return 0;
}
```

Ambiguity in Multiple Inheritance

In multiple inheritance, a single class is derived from two or more base classes. If the base classes have data members with same name or member functions with same name and same number and types of arguments, there will be confusion while using the base class members from the derived class. This is known as ambiguity which can be resolved by using the fully qualified name i.e., member name preceded by class name and scope resolution operator (class : : member).

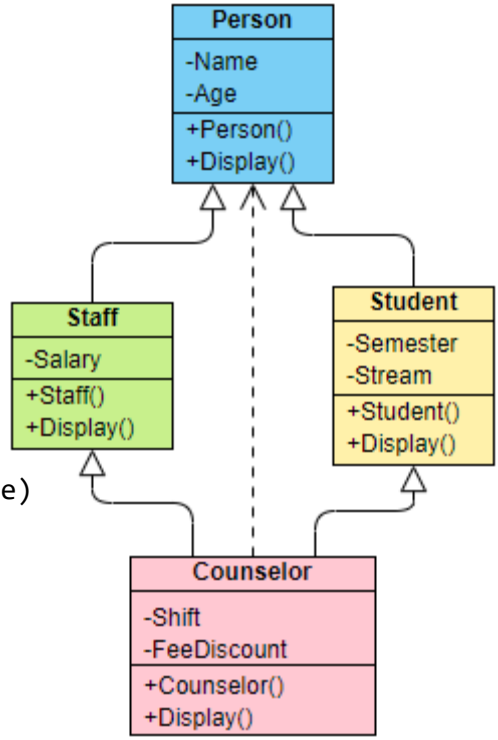
d) Hierarchical Inheritance Example




```
#include <iostream>
using namespace std;

class Employee{
  private:
    int ID;
    string Name;

  public:
    Employee(int id, string name)
    {
        ID   = id;
        Name = name;
    }

    void Display()
    {
        cout << endl << "ID   : " << ID;
        cout << endl << "Name : " << Name;
    }
};

class Teacher : public Employee{
  private:
    string Subject;

  public:
      Teacher(int id, string name, string subject)
                                    : Employee(id, name)
```

```
      {
          Subject = subject;
      }

      void Display()
      {
          Employee::Display();
          cout << endl << "Subject : " << Subject;
      }
};

class Staff : public Employee{
  private:
    string Post;

  public:
      Staff(int id, string name, string post)
                                      : Employee(id, name)
      {
          Post = post;
      }

      void Display()
      {
          Employee::Display();
          cout << endl << "Post : " << Post;
      }
};

int main()
{
    Teacher t1(101, "Mahesh", "Maths");
    Staff   s1(102, "Suresh", "Technician");

    t1.Display();
    cout << endl << endl;
    s1.Display();
    return 0;
}
```

e) Multi-path Inheritance Example




```
#include <iostream>
using namespace std;

class Person{
  protected:
    string  Name;
    int   Age;

  public:
    Person() { }

    Person(string name, int age)
    {
      Name = name;
      Age  = age;
    }

    void Display()
    {
      cout << endl << "Name   : " << Name;
      cout << endl << "Age  : " << Age;
    }
};


class Staff : virtual public Person{
  protected:
    float Salary;

  public:
    Staff() { }

    Staff(string name, int age, float salary)
                                        : Person(name, age)
    {
      Salary = salary;
    }
```

```
    void Display()
    {
      Person::Display();
      cout << endl << "Salary : " << Salary;
    }
};


class Student : virtual public Person{
  protected:
    int  Semester;
    string Stream;

  public:
    Student() { }

    Student(string name, int age, int sem, string str)
                                          : Person(name, age)
    {
      Semester = sem;
      Stream   = str;
    }
    void Display()
    {
      Person::Display();
      cout << endl << "Semester : " << Semester;
      cout << endl << "Stream   : " << Stream;
    }
};


class Counselor : public Staff, public Student{
  private:
    string Shift;
    float  FeeDiscount;

  public:
    Counselor() { }
```

```
Counselor(string name, int age, float sal, int sem,
              string str, string shift, float feeDisc)
    {
      Name      = name;       // Directly from Person
      Age       = age;        // Directly from Person
      Semester  = sem;        // Student
      Stream    = str;        // Student
      Salary    = sal;        // Staff
      Shift     = shift;      // Counselor
      FeeDiscount = feeDisc;        // Counselor
    }

    void Display()
    {
      cout << endl << "Name     : " << Name;
      cout << endl << "Age      : " << Age;
      cout << endl << "Semester : " << Semester;
      cout << endl << "Stream   : " << Stream;
      cout << endl << "Salary   : " << Salary;
      cout << endl << "Shift    : " << Shift;
      cout << endl << "Fee Discount : " << FeeDiscount;
    }
};

int main()
{
    Counselor c1("Ramesh", 20, 25000, 4,
                 "Civil", "morning", 12.5);
    c1.Display();

    getch();
    return 0;
}
```

Ambiguity in Multipath Inheritance:

In Multipath inheritance, a class is derived from two or more immediate parent classes which are again derived from the same parent class which results in multiple copies of the same features from the topmost parent class to the bottommost child class. To resolve this, the intermediate parent classes are inherited from the topmost parent class virtually.

4.4 Inheritance Merits and Demerits

Merits (Advantages) of Inheritance

- Inheritance promotes reusability. When a class is derived from another class, it acquires all the functionalities of the inherited class.
- Reusability enhances reliability due to the fact that the base class code is be already tested and debugged.
- As the existing code is reused, it leads to less development and maintenance costs.
- Inheritance makes the sub classes follow a standard interface.
- Inheritance helps to reduce code redundancy and supports code extensibility.
- Inheritance facilitates creation of class libraries.

Demerits (Disadvantages) of Inheritance

- Improper use of inheritance may lead to wrong solutions.
- Call to inherited functions may be slower than normal function calls.
- All data members / member functions from the base class may not be used from the derived class which leads to memory wastage.
- Inheritance increases the coupling between base class and derived class. A change in base class will affect all the child classes.

Even though there are some demerits of inheritance, the merits outpace its limitations.

4.5 Composition (Containership)

In real life, complex objects are often built from smaller and simpler objects. For example, a car is built using a metal frame, an engine, tires, a transmission system, a steering wheel, and a large number of other parts. Such a process of building complex objects from simpler ones is called composition or containership in C++.

Generally, simple classes have member variables that are built-in data types like int, float, double, char, etc. While this is generally sufficient for designing and implementing small, simple classes, but it gradually becomes difficult to carry out for more complex classes, especially for those built from many sub-parts. In order to facilitate the building of complex classes from simpler ones, C++ allows us to do object composition in a very simple way by using classes as data types allowing their objects to become member variables in other classes.

A class can have one or more objects of other classes as members i.e., a class is written in such a way that the object of another existing class becomes a member of the new class. This relationship between classes is known as composition. It is also known as containership, containment, part-whole relationship, or has-a relationship. Like inheritance, this is another common form of software reusability.

Example: Smartphone class containing an object of class DigitalCamera

```
#include <iostream>
using namespace std;

class DigitalCamera{
  private:
    float MegaPixels;

  public:
    void SetData(float m)
    {
        MegaPixels = m;
    }
```

```
    void Display()
    {
        cout << "Mega Pixels : " << MegaPixels;
    }
};

class SmartPhone{
private:
    string company;
    string model;
    DigitalCamera RearCamera;
public:
    void SetData(string c, string m, float mp)
    {
        company = c;
        model   = m;
        RearCamera.SetData(mp);
    }

    void Display()
    {
        cout << endl << "Company :" << company;
        cout << endl << "Model   :" << model;
        cout << endl << "Camera  :" << endl;
        RearCamera.Display();
    }
};

int main()
{
    SmartPhone myPhone;
    myPhone.SetData("Samsung", "Note 10", 128);
    myPhone.Display();
    return 0;
}
```

4.6 The is-a rule and has-a rule

Both “is-a rule” and “has-a” rule are kinds of relationships between two concepts which are fundamental to apply software reusability in Object Oriented programming.

Is-a rule

The “is-a” relationship holds between two concepts when one concept is a specialized version of another concept.

It is the relation which asserts that instances of a subclass must be more specialized forms of the superclass such that instances of a subclass can be used where an instance of the super class is required (principle of substitutability).

Example: A Student “is-a” Person

In OOP, it is achieved by using inheritance.

Has-a rule

The “has-a” relationship holds when a concept is a component of another concept and the two are not in any sense the same thing.

Example: A car “has-a” wheel

It is the relation which asserts that instances of a class possess members of another type.

In OOP, it is achieved by using composition or containership.

4.8 Software Reusability

Software reusability is an important and useful, but difficult to achieve, aspect in programming. The major hurdle in reusability is the interconnectedness of different segments of code in most software developed using conventional techniques.

However, using the object-oriented technologies, such interconnectedness can be decreased to a greater extent by developing independent components which can be tested in isolation. Using the concepts of inheritance and composition, already developed components which are already tried and tested, can be used to construct new classes, thus promoting the concept of reusability.