Program : **B.Tech**

Subject Name: **Object Oriented Programming & Methodology**

Subject Code: **CS-305**

Semester: **3rd**

Unit-III

Relationships: Inheritance- purpose and its types (IS-A Relationship)

One of the advantages of Object-Oriented programming language is code reuse. This reusability is possible due to the relationship b/w the classes. Object oriented programming generally support 4 types of relationships that are: inheritance, association, composition and aggregation. All these relationship is based on "is a" relationship, "has-a" relationship and "part-of" relationship.

Inheritance is one of the key features of Object-oriented programming in C++. It allows user to create a new class (derived class) from an existing class (base class).The derived class inherits all the features from the base class and can have additional features of its own.

The idea of inheritance implements the **IS-A** relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

The capability of a class to derive properties and characteristics from another class is called **Inheritance**.

NOTE: All members of a class except Private, are inherited

Super Class: In the inheritance the class which is give data members and methods is known as base or super or parent class.

Sub Class: The class which is taking the data members and methods is known as sub or derived or child class.

Real Life Example of Inheritance in C++

The real life example of inheritance is child and parents, all the properties of father are inherited by his son.

Advantages of inheritance are as follows:

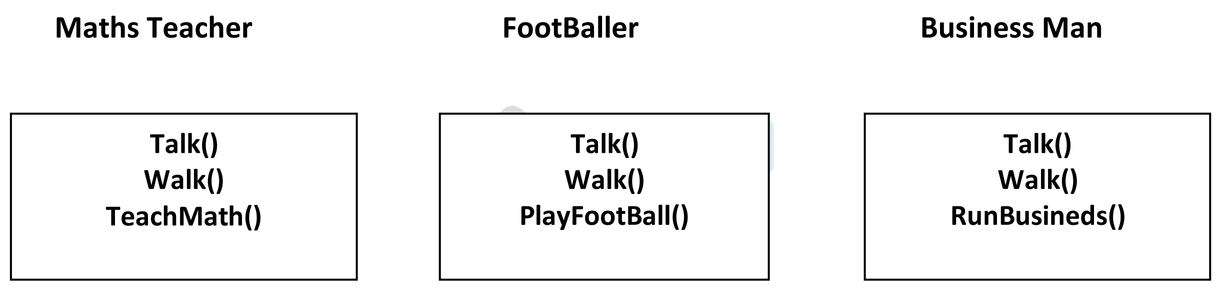
- Inheritance promotes reusability. When a class inherits or derives another class, it can access all the functionality of inherited class.
- Reusability enhanced reliability. The base class code will be already tested and debugged.
- As the existing code is reused, it leads to less development and maintenance 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.
- Application development time is less.
- Applications take less memory.
- Application execution time is less.
- Application performance is enhance (improved).
- Redundancy (repetition) of the code is reduced or minimized so that we get consistence results and less storage cost.

Disadvantages of inheritance are as follows:

- Inherited functions work slower than normal function as there is indirection.
- Improper use of inheritance may lead to wrong solutions.
- Often, data members in the base class are left unused which may lead to memory wastage.
- Inheritance increases the coupling between base class and derived class. A change in base class will affect all the child classes.
- One of the main disadvantages of inheritance in Java (the same in other object-oriented languages) is the increased time/effort it takes the program to jump through all the levels of overloaded classes. If a given class has ten levels of abstraction above it, then it will essentially take ten jumps to run through a function defined in each of those classes

Why inheritance should be used?

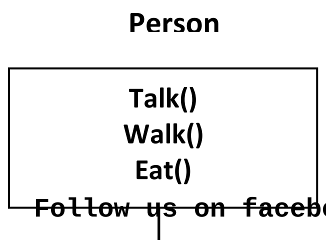
Suppose, in your game, you want three characters - a **maths teacher**, a **footballer** and a **businessman**.
Since, all of the characters are persons, they can walk and talk. However, they also have some special skills. A maths teacher can **teach maths**, a footballer can **play football** and a businessman can **run a business**.
You can individually create three classes who can walk, talk and perform their special skill as shown in the figure below.



In each of the classes, you would be copying the same code for walk and talk for each character.

If you want to add a new feature - eat, you need to implement the same code for each character. This can easily become error prone (when copying) and duplicate codes.

It'd be a lot easier if we had a **Person** class with basic features like talk, walk, eat, sleep, and add special skills to those features as per our characters. This is done using inheritance.



Using inheritance, now you don't implement the same code for walk and talk for each class. You just need to **inherit** them.
So, for Maths teacher (derived class), you inherit all features of a Person (base class) and add a new feature **TeachMaths**. Likewise, for a footballer, you inherit all the features of a Person and add a new feature **PlayFootball** and so on.
This makes your code cleaner, understandable and extendable.
It is important to remember: When working with inheritance, each derived class should satisfy the condition whether it **"is a"** base class or not. In the example above, Maths teacher **is a** Person, Footballer **is a** Person. You cannot have: Businessman **is a** Business.

The general form of defining a derived class is:

class **derived-class_name** : visivility-mode **base-class_name**
{
. . . . // members of the derived class
. . . .
};

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.

C++ offers five types of Inheritance. They are:

When deriving a class from a base class, the base class may be inherited through **public, protected** or **private** inheritance. The type of inheritance is specified by the access-specifier as explained above.

We hardly use **protected** or **private** inheritance, but **public** inheritance is commonly used. While using different type of inheritance, following rules are applied −

- **Public Inheritance** − When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private** members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.
- **Protected Inheritance** − When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class.

- **Private Inheritance** − When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members of the derived class.

Table showing all the Visibility Modes

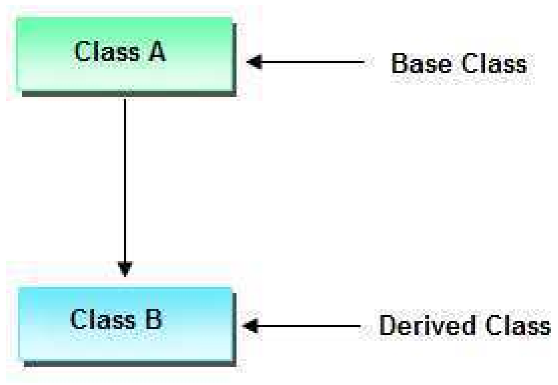
	Derived Class	**Derived Class**	**Derived Class**
Base class	**Public Mode**	**Private Mode**	**Protected Mode**
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

Type of Inheritance

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance (also known as Virtual Inheritance)

Single Inheritance

In single inheritance, there is only one base class and one derived class. The Derived class gets inherited from its base class. This is the simplest form of inheritance. The below figure is the diagram for single inheritance



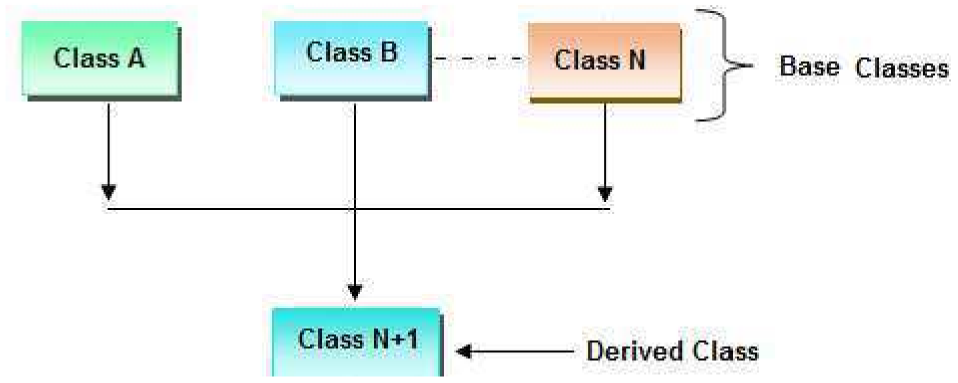
Program

#include<iostream>
using namespace std;

```
class AddData        //Base Class
{
    protected:
        int num1, num2;
    public:
        void accept()
        {
            cout<<"\n Enter First Number : ";
            cin>>num1;
            cout<<"\n Enter Second Number : ";
            cin>>num2;
        }
};
class Addition: public AddData   //Class Addition – Derived Class
{
        int sum;
    public:
        void add()
        {
            sum = num1 + num2;
        }
        void display()
        {
            cout<<"\n Addition of Two Numbers : "<<sum;
        }
};
int main()
{
    Addition a;
    a.accept();
    a.add();
    a.display();
    return 0;
}
```

Multiple Inheritance

In this type of inheritance, a single derived class may inherit from two or more base classes. The below figures is the structure of Multiple Inheritance.



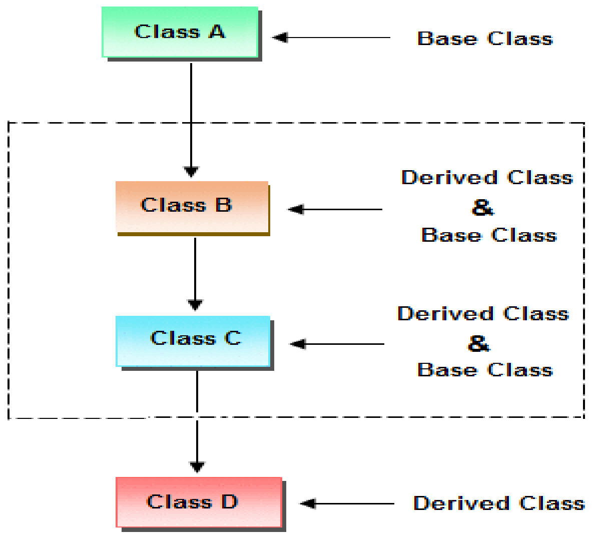
Program

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

class stud {
protected:
    int roll, m1, m2;

public:
    void get()
    {
        cout << "Enter the Roll No.: "; cin >> roll;
        cout << "Enter the two highest marks: "; cin >> m1 >> m2;
    }
};
class extracurriculam {
protected:
    int xm;

public:
    void getsm()
    {
        cout << "\nEnter the mark for Extra Curriculam Activities: "; cin >> xm;
    }
};
class output : public stud, public extracurriculam {
    int tot, avg;

public:
    void display()
    {
        tot = (m1 + m2 + xm);
        avg = tot / 3;
        cout << "\n\n\tRoll No    : " << roll << "\n\tTotal      : " << tot;
        cout << "\n\tAverage    : " << avg;
    }
};
```

```
int main()
{
    output O;
    O.get();
    O.getsm();
    O.display();
    return 0;
}
```

Multilevel Inheritance

The classes can also be derived from the classes that are already derived. This type of inheritance is called multilevel inheritance.

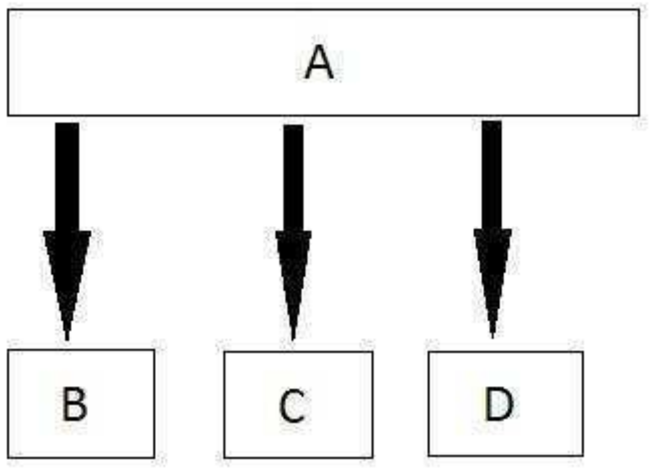


```
#include<iostream>
using namespace std;
class Student
{
            protected:
                        int marks;
            public:
                        void accept()
                        {
                                     cout<<" Enter marks";
                                    cin>>marks;
                        }
};
 class Test :public Student
{
             protected:
                        int p=0;
            public:
                        void check()
                        {
```

```
                        if(marks>60)
                        {
                                p=1;
                        }
                }
 };
class Result :public Test
{
        public:
                void print()
                {
                        if(p==1)
                                cout<<"\n You have passed";
                        else
                                cout<<"\n You have not passed";
                }
};
 int main()
{
        Result r;
        r.accept();
        r.check();
        r.print();
        return 0;
}
```

Hierarchical Inheritance

In this type of inheritance, multiple derived classes get inherited from a single base class. The below fig is the structure of Hierarchical Inheritance.



Example

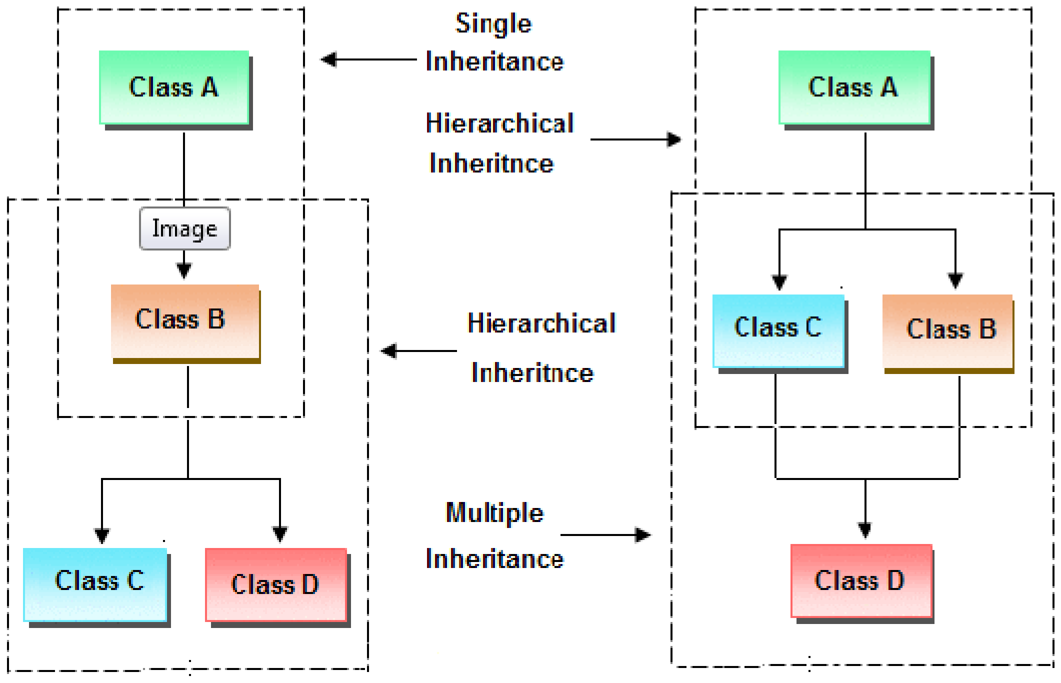
```
#include <iostream>
#include <string.h>
```

```
using namespace std;

class member {
    char gender[10];
    int age;

public:
    void get()
    {
        cout << "Age: "; cin >> age;
        cout << "Gender: "; cin >> gender;
    }
    void disp()
    {
        cout << "Age: " << age << endl;
        cout << "Gender: " << gender << endl;
    }
};
class stud : public member {
    char level[20];

public:
    void getdata()
    {
        member::get();
        cout << "Class: "; cin >> level;
    }
    void disp2()
    {
        member::disp();
        cout << "Level: " << level << endl;
    }
};
class staff : public member {
    float salary;

public:
    void getdata()
    {
        member::get();
        cout << "Salary: Rs."; cin >> salary;
    }
    void disp3()
    {
        member::disp();
        cout << "Salary: Rs." << salary << endl;
    }
};
int main()
{
    member M;
    staff S;
    stud s;
    cout << "Student" << endl;
    cout << "Enter data" << endl;
    s.getdata();
    cout << endl<< "Displaying data" << endl;
    s.disp();
    cout << endl<< "Staff Data" << endl;
    cout << "Enter data" << endl;
    S.getdata();
    cout << endl<< "Displaying data" << endl;
    S.disp();
}
```

Hybrid Inheritance

This is a Mixture of two or More Inheritance and in this Inheritance, a Code May Contains two or three types of inheritance in Single Code. In the below figure, the fig is the diagram for Hybrid inheritance.

It is a combination of two types of inheritance namely the multiple and hierarchical Inheritance. The following is a schematic representation.



Hybrid Inheritance is a method where one or more types of inheritance are combined together and used.

Program

```
#include<iostream>
```

```
Using namespace std;

class arithmetic
{
       protected:
             int num1, num2;
      public:
             void getdata()
            {
                  cout<<"For Addition:";
                  cout<<"\nEnter the first number: ";
                  cin>>num1;
                  cout<<"\nEnter the second number: ";
                  cin>>num2;
          }
};
class plus:public arithmetic
{
           protected:
                 int sum;
           public:
                void add()
               {
                    sum=num1+num2;
               }
};
class minus
{
              protected:
                     int n1,n2,diff;
              public:
                    void sub()
                    {
                         cout<<"\nFor Subtraction:";
                         cout<<"\nEnter the first number: ";
                         cin>>n1;
                         cout<<"\nEnter the second number: ";
                         cin>>n2;
                         diff=n1-n2;
                    }
};
class result:public plus, public minus
{
                 public:
                      void display()
                {
                    cout<<"\nSum of "<<num1<<" and "<<num2<<"= "<<sum;
                    cout<<"\nDifference of "<<n1<<" and "<<n2<<"= "<<diff;
                }
};
int main()
{
        result z;
        z.getdata();
        z.add();
        z.sub();
        z.display();
        return 0;
}
```

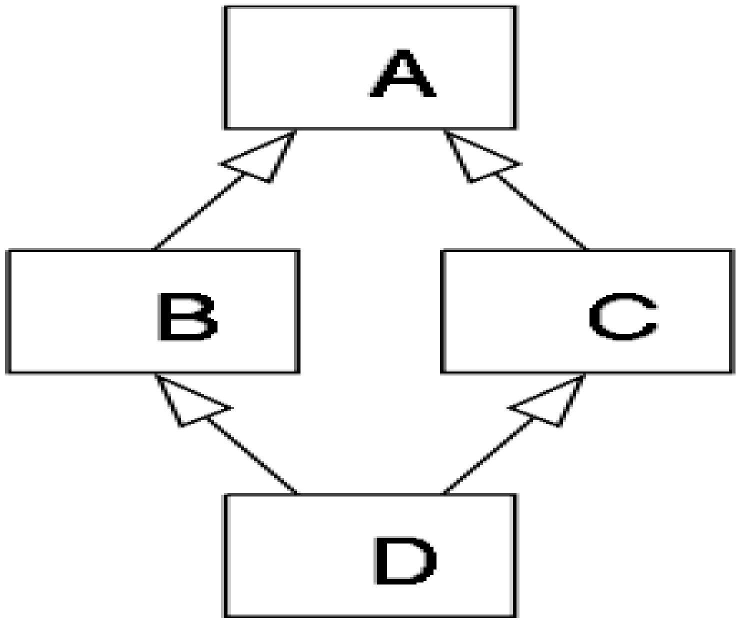
Diamond Problem in Inheritance

Let's understand this with one example.

```
class A {
   void display()
   {
      //some code
   }
}

class B : public A{
   void display()
   {
      //some code
   }
}

class C : public A{
   void display()
   {
      //some code
   }
}

class D : public B, public C{
   //contains two display() functions
}
```

Suppose there are four classes A, B, C and D. Class B and C inherit class A. Now class B and C contains one copy of all the functions and data members of class A.
Class D is derived from Class B and C. Now class D contains two copies of all the functions and data members of class A. One copy comes from class B and another copy comes from class C.
Let's say class A have a function with name display(). So class D have two display() functions as I have explained above. If we call display() function using class D object then ambiguity occurs

because compiler gets confused that whether it should call display() that came from class B or from class C. If you will compile above program then it will show error.



How to Remove Diamond Problem in C++?

We can remove diamond problem by using **virtual** keyword. It can be done in following way.

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

class A {
  public:
    void display()
    {
       cout<<"A Class Display Called.."<<endl;
    }
};

class B : virtual public A{
    public:
    void display()
    {
        cout<<"B Class Display Called.."<<endl;

    }
};

class C :virtual public A{
    public:

    void display()
    {
        cout<<"C Class Display Called.."<<endl;
    }
};

class D : public B, public C{
    //contains two display() functions
};

int main()
{
        A *a=new D();
        B *b=new D();
        C *c=new D();
        a->display();
        b->display();
        c->display();

        return 0;
}
```

Association

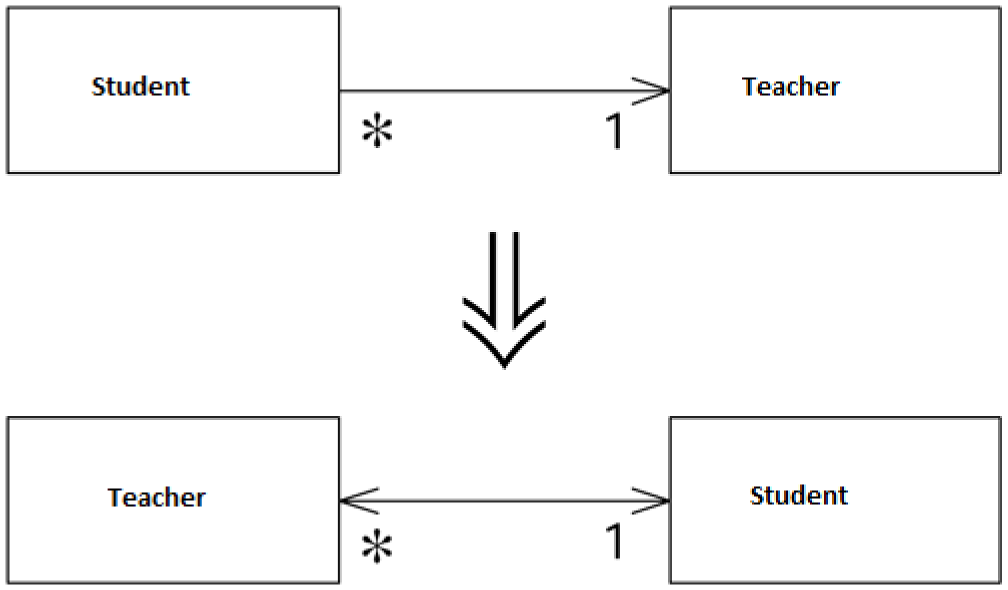
Association is a semantically weak relationship (a semantic dependency) between otherwise unrelated objects. An association is a "using" relationship between two or more objects in which the objects have their own life time and there is no owner. As an example, imagine the relationship between a doctor and a patient. A doctor can be associated with multiple patients and at the same time, one patient can visit multiple doctors for treatment and/or consultation. Each of these objects has their own life-cycle and there is no owner. In other words, the objects that are part of the association relationship can be created and destroyed independently.

In UML an association relationship is represented by a single arrow. An association relationship can be represented as (also known as cardinality) one-to-one, one-to-many and many-to-many. Essentially, an association relationship between two or more objects denotes a path of communication (also called a link) between them so that one object can send a message to another.

Association is a relationship which describes the reasons for the relationship and the rules that govern the relationship.

Let’s take an example of Teacher and Student.

Unidirectional Association is a specialized form of association where one object is associated with another object, but the reverse is not true. It is like one way communication.



Let’s take examples of multiple students can associate with a single teacher.

Bidirectional Association is a type of association where one object is related with other objects and the reverse is also true. It is like two way communication.

Let’s take examples of multiple students can associate with a single teacher and a single student can associate with multiple teachers.

But there is no ownership between the objects and both have their own life cycle. Both can create and delete independently.

Association is a simple structural connection or channel between classes and is a relationship where all objects have their own lifecycle and there is no owner.

Let’s take an example of Department and Student. Multiple students can associate with a single Department and single student can associate with multiple Departments, but there is no ownership between the objects and both have their own lifecycle. Both can create and delete independently.

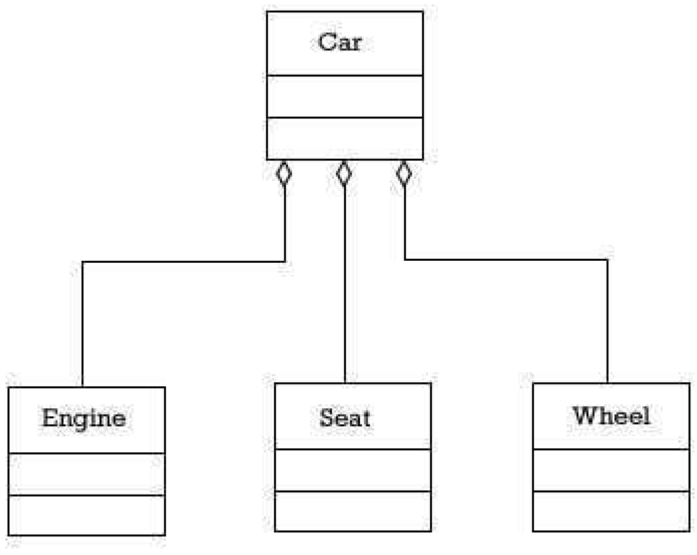
Aggregation (HAS-A Relationship)

Aggregation is a specialized form of association between two or more objects in which the objects have their own life-cycle but there exists an ownership as well. Aggregation is a typical whole/part relationship but it may or may not denote physical containment -- the objects may or may or be a part of the whole. In aggregation the objects have their own life-cycle but they do have ownership as well. As an example, an employee may belong to multiple departments in the organization. However, if the department is deleted, the employee object wouldn't be destroyed. Note that objects participating in an aggregation relationship cannot have cyclic aggregation relationships, i.e., a whole can contain a part but the reverse is not true.

It is a special form of Association where:

- It represents **Has-A** relationship.
- It is a **unidirectional association** i.e. a one way relationship. For example, department can have students but vice versa is not possible and thus unidirectional in nature.
- In Aggregation, **both the entries can survive individually** which means ending one entity will not effect the other entity.

car object is an aggregation of engine, seat, wheels and other objects.



When do we use Aggregation??

Code reuse is best achieved by aggregation.

C++ Aggregation Implementation

Let's see an example of aggregation where Employee class has the reference of Address class as data member. In such way, it can reuse the members of Address class.

```
 #include <iostream>
using namespace std;
 class Address {
     public:
    string addressLine, city, state;
      Address(string addressLine, string city, string state)
     {
        this->addressLine = addressLine;
        this->city = city;
        this->state = state;
     }
 };
 class Employee
{
      private:
        Address* address;  //Employee HAS-A Address
        public:
        int id;
        string name;
        Employee(int id, string name, Address* address)
       {
           this->id = id;
           this->name = name;
           this->address = address;
       }
      void display()
       {
           cout<<id <<" "<<name<< " "<<
             address->addressLine<< " "<< address->city<< " "<<address->state<<endl;
       }
     };
  int main(void) {
      Address a1= Address("C-146, Sec-15","Noida","UP");
      Employee e1 = Employee(101,"Nakul",&a1);
      e1.display();
       return 0;
  }
```

Composition

Composition is a specialized form of aggregation in which if the parent object is destroyed, the child objects would cease to exist. It is actually a strong type of aggregation and is also referred to as a "death" relationship. As an example, a house is composed of one or more rooms. If the house is destroyed, all the rooms that are part of the house are also destroyed as they cannot exist by themselves.

A good real-life example of a composition is the relationship between a person’s body and a heart. Let’s examine these in more detail.

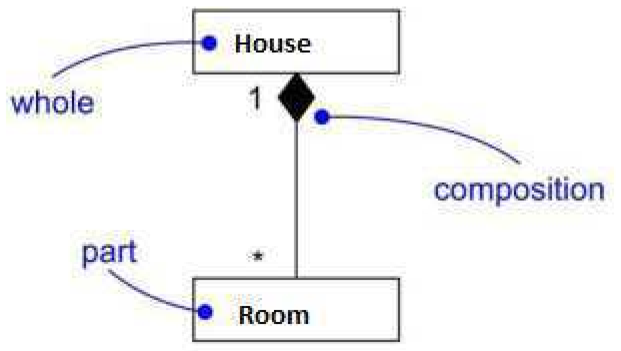
Composition relationships are part-whole relationships where the part must constitute part of the whole object. For example, a heart is a part of a person’s body. The part in a composition can only be part of one object at a time. A heart that is part of one person’s body cannot be part of someone else’s body at the same time.

Object Composition

In real-life, complex objects are often built from smaller, simpler objects. For example, a car is built using a metal frame, an engine, some tires, a transmission, a steering wheel, and a large number of other parts. A personal computer is built from a CPU, a motherboard, some memory, etc… Even you are built from smaller parts: you have a head, a body, some legs, arms, and so on. This process of building complex objects from simpler ones is called **object composition**.

Broadly speaking, object composition models a “has-a” relationship between two objects. A car “has-a” transmission. Your computer “has-a” CPU. You “have-a” heart. The complex object is sometimes called the whole, or the parent. The simpler object is often called the part, child, or component.

Example “engine is part of car”, “heart is part of body”.



Implementation

```
#include <iostream>
#include <string>

using namespace std;

class Birthday{

public:
    Birthday(int cmonth, int cday, int cyear){
        cmonth = month;
        cday = day;
        cyear = year;

    }
    void printDate(){
        cout<<month <<"/" <<day <<"/" <<year <<endl;

    }
private:
    int month;
    int day;
    int year;

};

class People{

public:
    People(string cname, Birthday cdateOfBirth):name(cname), dateOfBirth(cdateOfBirth)
    {

    }
    void printInfo(){
        cout<<name <<" was born on: ";
        dateOfBirth.printDate();
    }
private:
    string name;
    Birthday dateOfBirth;

};


int main() {

    Birthday birthObject(7,9,97);
    People infoObject("Shantilal", birthObject);
    infoObject.printInfo();
    return 0;
 }
```

Comparison Chart

	Association	**Aggregation**	**Composition**
Owner	No owner	Single owner	Single owner
Life time	Have their own lifetime	Have their own lifetime	Owner's life time
Child object	Child objects all are independent	Child objects belong to a single parent	Child objects belong to a single parent

Concept of interfaces and Abstract classes

To make it short, there is no 'interface' in C++. Conceptually, an interface is a pure virtual class, with no implementation whatsoever.

Abstract classes are the way to achieve abstraction in C++. Abstraction in C++ is the process to hide the internal details and showing functionality only. Abstraction can be achieved by two ways:

1. **Abstract class**
2. **Interface**

Abstract class and interface both can have abstract methods which are necessary for abstraction.

Abstract Class

An abstract class is a class that is designed to be specifically used as a base class. An abstract class is a class that has at least a pure virtual method. You can't create instances of that class, but you can have implementation in it, that is shared with the derived classes.

In C++ class is made abstract by declaring at least one of its functions as pure virtual function. A pure virtual function is specified by placing "= 0" in its declaration. Its implementation must be provided by derived classes.

Implementation

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

// Base class
class Shape {
  public:
    // pure virtual function providing interface framework.
    virtual int getArea() = 0;
    void setWidth(int w) {
      width = w;
    }

    void setHeight(int h) {
      height = h;
    }

  protected:
    int width;
    int height;
};

// Derived classes
class Rectangle: public Shape {
  public:
    int getArea() {
      return (width * height);
    }
};

class Triangle: public Shape {
  public:
    int getArea() {
      return (width * height)/2;
    }
};

int main(void) {
  Rectangle Rect;
  Triangle  Tri;

  Rect.setWidth(5);
  Rect.setHeight(7);

  // Print the area of the object.
```

```
   cout << "Total Rectangle area: " << Rect.getArea() << endl;

   Tri.setWidth(5);
   Tri.setHeight(7);

   // Print the area of the object.
   cout << "Total Triangle area: " << Tri.getArea() << endl;

   return 0;
}
```

Interface

C++ has no built-in concepts of interfaces. You can implement it using an abstract class which contains only pure virtual functions. Since it allows multiple inheritance, you can inherit this class to create another class which will then contain this interface (I mean, object interface :) in it.

Characteristics

1. An interface has no implementation.
2. An interface class contains only a virtual destructor and pure virtual functions.
3. An interface class is a class that specifies the polymorphic interface i.e. pure virtual function declarations into a base class. The programmer using a class hierarchy can then do so via a base class that communicates only the interface of classes in the hierarchy.

Implementation of Interface

```
#include<iostream>
Using namespace std;
class Base
{
Public:
    virtual void func1(int)=0;
    virtual void func1(int, int)=0;
};

class Derived : public Base
{
    public:
     virtual void func1(int someValue)
     {
         cout<<someValue;
      }

    private:
     virtual void func1(int someValue1, int someValue2)
    {
       cout<<0;
    }
};

class Derived2 : public Base
{
    public:
    virtual void func1(int someValue1, int someValue2)
    {
        cout<<someValue1+someValue2;
    }

    private:
     virtual void func1(int someValue)
     {

      }
 };

int main()
{
    Base *b;
    Derived d;

    b=&d;
    cout<<b->func(1,2); //this line would access the method even if it is private
return 1;
}
```

Interface vs. Abstract Class

An interface does not have implementation of any of its methods, it can be considered as a collection of method declarations. In C++, an interface can be simulated by making all methods as pure virtual. In Java, there is a separate keyword for interface.

Interfaces provide a convenient means of resolving the tension between what a class is and what it can do. Keeping interface and implementation separate in C++ programs keeps designs clean and fosters reuse.