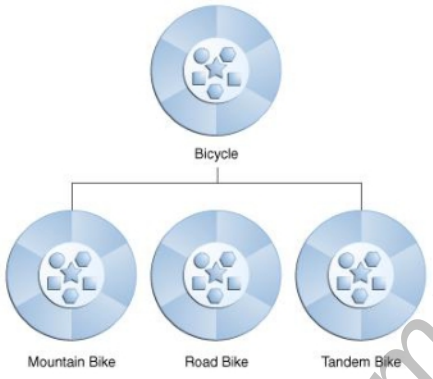
Chapter-4

Object Inheritance & Reusability

Inheritance

- Inheritance is the mechanism of deriving a new class from existing class. The existing class is called **super class or base class or parent class** and the child class is called **subclass or derived class or extended class.**
- The subclass inherits some of the properties from the base class and can add or extends its own property as well.
- All the public and protected properties such as fields, methods etc. can be inherited by the derived class however private members can't be inherited



- Inheritance is also called a **"Kind of" or "isa"** relationship. Example: In the above figure, Mountain Bike is a Bicycle.
- The main advantage of inheritance is the concept of **code reusability**.
- A code is said to be reusable if we can reuse the properties of superclass in other class by using the concept of inheritance.
- A programmer can use a class created by another person or company without any modification at all or with small modification such as deriving other classes from it to fit his situation.
- So, reusing existing codes saves time and money and increase code reliability.

Need of Inheritance

- Consider different types of **Employees** in an organization. We need to create classes for Manager, Supervisor etc. The methods like setName(), setAddress(), setPhoneNumber() etc. will be the same for both classes. Now, 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:

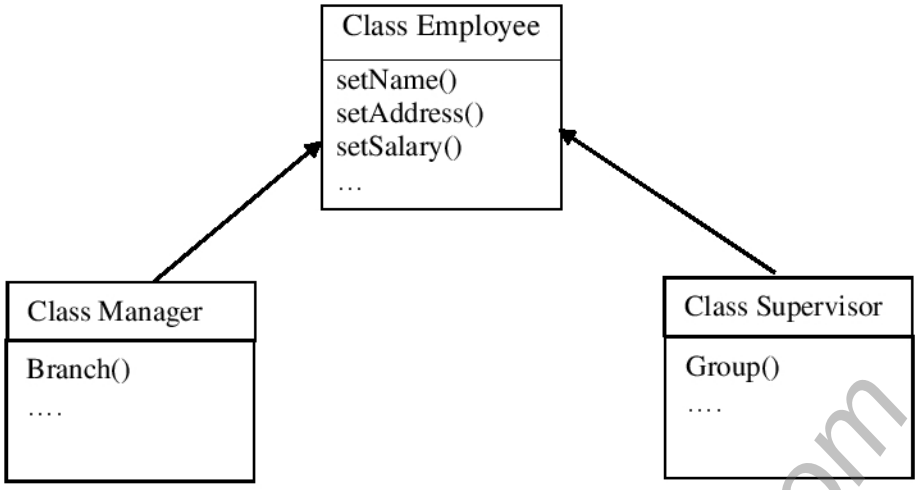
Class Manager

Name()
Address()
Salary()
…

Class Supervisor

Name
Address()
Salary()
…

- So we can clearly see that the above concept results in duplication of the same code 2 times. This increases the chances of error and data redundancy and also time to code and debug.
- To avoid this type of situation, inheritance is used.
- So we create a class Staff and write these three functions in it and inherit the rest of the classes from the Staff class, then we can simply avoid the duplication of data and increase code re-usability. The below diagram illustrates this concept.



- Using inheritance, we need to write the functions only one time instead of two times as we have inherited the rest of the two classes from the base class (Staff).
- Inheritance not only provides code reusability, but also can extend/add new functionalities unique to each derived class. Here, Manager inherits the characteristics of Employee and extends new functionality by adding Branch() and similarly Supervisor by adding function Group().

Implementing inheritance in c++ / Defining a Derived class or sub class

- A derived class is defined by specifying its relationship with the base class in addition to its own detail.
- The general form for deriving a derived class is given below.

```
Class derived_class_name : visibility-mode base_class_name
{
        //Members of derived class
};
```

Where, the colon indicates that the derived_class_name is derived from base class name.

- There are three **visibility mode** using which we can derive child class from existing parent class.
- The visibility mode is optional and if present, it may be private, public or protected. **The default visibility mode is private**. The visibility mode specifies whether the features of base class are privately derived, publicly or protectedly derived.
- **When the base class is privately inherited**, all the public and protected members of the base class become private members of the derived class. So, these cannot be accessed outside the class directly through the derived class object, but might be accessed through public functions in the derived class. Like general private members, these members can be used freely within the derived class.
- **When the base class is publicly inherited**, all the public members of the base class become public members of the derived class. So, they can be accessed through the objects of the derived class. All the protected members become protected members of the derived class.
- **When base class is derived using protected mode**, all the protected and public members of the base class become protected members of the derived class. This means, like a private inheritance, these members cannot be directly accessed through object of the derived class. But can be used freely within the derived class.

Whereas, unlike a private inheritance, they can still be inherited and accessed by subsequent derived classes.

In other words, protected inheritance does not end a hierarchy of classes, as private inheritance does.

Example

```
Class A
{
        //constructors, methods and fields of A
};
Class B: public A //public derivation
{
        //constructors, methods and fields of B
};
Class C: private A // private derivation
{
        //constructors, methods and fields of C
};
Class D: A // private derivation by default
{
        //constructors, methods and fields of D

};
```

Protected access specifier / Making private member inheritable

- When it is necessary to inherit private data member then it can be done by modifying the visibility to public. But doing this make it accessible to all the other functions of the program thus taking away the data hiding features. So for this we use a third visibility mode called **protected.**
- When a member is defined with protected access specifier, these members can be accessed from that class and from the derived class of this base class. But cannot be accessed from any other function or class. i.e. protected members act as public for derived class and private for other classes i.e. outside world.
- The main difference between private access specifier and protected access specifier is that protected members are visible to derived class members whereas private members in the base class are not visible in the derived class. We can't inherit private members. Only the public and protected ones can be inherited.
- **The protected access specifiers are useful in inheritance.** The purpose of making data member protected in a class is to make such members accessible to derived class function which is derived from this class. No function other than base class functions or derived class functions or the friend of derived class can access the protected data of the base class.

Inheritance Types

i. Single inheritance
ii. Multiple inheritance
iii. Hierarchical inheritance
iv. Multilevel inheritance
v. Hybrid inheritance

The below figure illustrates inheritance.

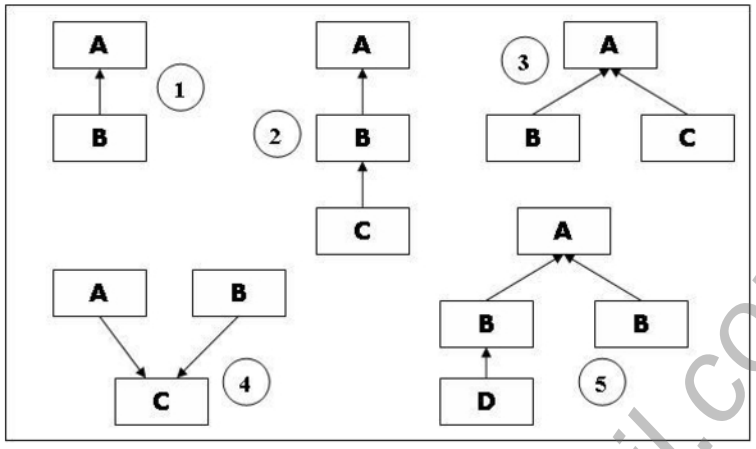
(1) Simple / Single

(2) Multilevel

(3) Hierarchical

(4) Multiple

(5) Hybrid



Single inheritance

- In single inheritance, a class is derived from only one existing class. i.e. only one base class.
- The general form is given below

```
class  A
{
        //member of A
};
class B : public/private/protected A
{
        //own member of B
};
```

Here class A is base class and a new class B is derived (public or private or protected) from A.

Example:

A class Room consists of two fields length and breadth and method int area() to find the area of room. A new class BedRoom is derived from class Room and consist of additional field height and two methods setData(int,int,int) to set the value for three fields and int volume() to find the volume. Now write the c++ program to input the length ,breadth and height and find the area and volume.

```
#include<iostream>
using namespace std;
class Room
{
        protected:
        float length, breadth;
        public:
                int area()
                {
                        return(length*breadth);
                }
};

class BedRoom : public Room
{
        private:
        float height;
        public:
```

```
        void setData(int l,int b, int h)
        {
            length=l;
            breadth=b;
            height=h;
        }

        int volume()
        {
            return(length * breadth * height);
        }
};
int main()
{
    BedRoom b;
    b.setData(3,4,5);
    cout<<"Area of bedroom= "<<b.area()<<endl;
    cout<<"Volume of bedroom="<<b.volume();
}
```

Output:

```
Area of bedroom= 12
Volume of bedroom=60
```

Multilevel inheritance

- In multilevel inheritance, a class is derived from another subclass.
- The general form is

```
class A
{
    //member of A
}
class B :public/private/protected A
{
//own member of B
}
class C :public/private/protected B
{
    //own member of C
}
```

In above case, class B is derived from Class A while class C is derived from derived class B. So, the class A serves as base class for B and B serves as base class for C therefore B is also called as **intermediate class because** it provides a link between class A and C.

Example:

A class Student consists of field roll, a method to assigns roll number. A new class Test is derived from class Student and consists of two new fields sub1 and sub2, a method to initialize these fields with obtained mark. Again, a new class Result is derived from Test and consists of a field total and a method to display entire details along with total obtained marks. WAP to input roll number, marks in two different subject and display total.

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

```
class Student
{
        protected:
                int roll;
        public:
                void setroll(int r)
                {
                        roll=r;
                }
};
class Test: public Student
{
        protected:
        float sub1, sub2;
        public:
                void setmark(float m1, float m2)
                {
                        sub1=m1;
                        sub2=m2;
                }
};

class Result : public Test
{
        private:
                float total;
        public:
                void display()
                {
                        total=sub1+sub2;
                        cout<<"Roll number= "<<roll<<endl;
                        cout<<"Mark in first subject= "<<sub1<<endl;
                        cout<<"Mark in second subject= "<<sub2<<endl;
                        cout<<"Total= "<<total;
                }
};

int main()
{
        int r;
        float s1,s2;
        cout<<"Enter roll number"<<endl;
        cin>>r;
        cout<<"Enter marks in two subject"<<endl;
        cin>>s1>>s2;
        Result res;
        res.setroll(r);
        res.setmark(s1,s2);
        res.display();
}
```

Output:

```
Enter roll number
5
Enter marks in two subject
80
70
Roll number= 5
Mark in first subject= 80
Mark in second subject= 70
Total= 150
```

Multiple inheritance

- In multiple inheritance, a class is derived from more than one existing classes.
- The general form is

```
Class A
{
        //members of A
}
Class B
{
        //members of B
}
Class C : public/private/protected A, public/private/protected B
{
        //members of C
}
```

Example:

Create two classes class1 and class2 each having data member for storing a number, a method to initialize it. Create a new class class3 that is derived from both class class1 and class2 and consisting of a method that displays the sum of two numbers from class1 and class2.

```
#include<iostream>
using namespace std;
class class1
{
        protected:
                int n;
        public:
                void getn(int p)
                {
                        n=p;
                }
};
class class2
{
        protected:
                int m;
        public:
                void getm(int q)
                {
                        m=q;
                }
};
```

```
class class3: public class1, public class2
{
        public:
        void displaytotal()
        {
                int tot;
                tot=n+m;
                cout<<"Sum ="<<tot;
        }
};
int main()
{
        class3 a;
        a.getm(4);
        a.getn(5);
        a.displaytotal();
}
```

Output:

```
Sum =9
--------------------------------
Process exited after 0.07582 seconds with return value 0
```

Problem with multiple inheritance

- Sometime we have to face an **ambiguity problem** in using multiple inheritance when a function with the same name appears in more than one base class.
- Consider the following example

```
#include<iostream>
using namespace std;
class class1
{
        protected:
                int a;
        public:
        void get(int x)
        {
                a=x;
        }

};

class class2
{
        protected:
                int b;
        public:
        void get(int x)
        {
                b=x;
        }
};
class class3: public class1, public class2
{
        public:
        void displaytotal()
        {
                cout<<"Total= "<<(a+b);
```

```
        }
};

int main()
{
        class3 a;
        //a.get();// the request for get() is ambiguous as get() is defined in both class1 and class2
        a.class1::get(5);
        a.class2::get(6);
        a.displaytotal();
}
```

So we see that, when compilers of programming languages that support this type of multiple inheritance encounter, super classes that contain methods with the same name, they sometimes cannot determine which member or method to access or invoke. We can solve this problem by defining a named instance within the derived **class using the class resolution operator** with the function as shown in above example.

Hierarchical inheritance

- In hierarchical inheritance, two or more classes inherits the properties of one existing class.
- The general form is

```
class A
{
        //member of A
}
class B : public/private/protected A
{
        //own member of B
}

class C : pubic / private / protected A
{
        //own member of  C
}
```

In above example, A is base class which includes all the features that are common to sub classes. Class B and Class C are the sub classes that shares the common property of class A and also can define their own property. Note we also can define a subclass that serve as base class for the lower level classes and so on.

Example:

A company needs to keep record of its following employees:

i) Manager ii) Supervisor

The record requires name and salary of both employees. In addition, it also requires section_name (i.e. name of section, example Accounts, Marketing, etc.) for the Manager and group_id (Group identification number, e.g. 205, 112, etc.) for the Supervisor. Design classes for the above requirement. Each of the classes should have a function called set() to assign data to the fields and a function called get() to return the value of the fields. Write a main program to test your classes. What form of inheritance will the classes hold in this case?

```
#include<iostream>
#include<string.h>
using namespace std;
```

```
class Employee
{
	private:
		char name[30];
		float salary;
	public:
		void setName(char *n)
		{
			strcpy(name,n);
		}
		void setSalary(float s)
		{
			salary=s;
		}


		char * getName()
		{
			return name;
		}
		float getSalary()
		{
			return salary;
		}
};

class Manager: public Employee
{
	private:
		char section_name[50];
	public:
		 void setSection_name(char *sn)
		{
			strcpy(section_name,sn);
		}
		char * getSection_name()
		{
			return section_name;
		}
};
class Supervisor: public Employee
{
	private:
		int group_id;
	public:
		 void setGroup_id(int gid)
		{
			group_id=gid;
		}
		int getGroup_id()
		{
			return group_id;
		}
};
```

```
int main()
{
        Manager m;
        m.setName("Bhesh Bahadur Thapa");
        m.setSalary(50000);
        m.setSection_name("Accounts");
        cout<<"Name= "<<m.getName()<<endl;
        cout<<"Salary= "<<m.getSalary()<<endl;
        cout<<"Section= "<<m.getSection_name()<<endl;

        Supervisor s;
        s.setName("Sagar Kunwar");
        s.setSalary(40000);
        s.setGroup_id(5);
        cout<<"Name= "<<s.getName()<<endl;
        cout<<"Salary= "<<s.getSalary()<<endl;
        cout<<"Group ID= "<<s.getGroup_id()<<endl;
}
```

Output:

```
Name= Bhesh Bahadur Thapa
Salary= 50000
Section= Accounts
Name= Sagar Kunwar
Salary= 40000
Group ID= 5
```

Hybrid inheritance

- In hybrid inheritance, it can be the combination of more than two form of inheritance i.e. single and multiple inheritance or any other combination.
- One form can be as below

```
Class X
{
        //member of X
}
Class A : public/private/protected X
{
        // own member of A
}
Class B: public/private/protected A
{
        //own member of C
}
Class C: public/private/protected A
{
        //own member of C
}
Class D: public/private/protected A
{
        //own member of D
}
```

- Above example consists both **single and hierarchical inheritance hence called hybrid inheritance**.

Example:

The below example shows the hybrid inheritance i.e. combination of multilevel and multiple inheritance

Create a class Student with data member roll_no and two functions to initialize and display it. Derive a new class Test which has two methods to assign and display marks in two subjects. Create a new class Sport with two functions that assign and display the score in sports. Now create another class Result that is derived from both class Test and Sport, having a function that displays the total of marks and score. Write a main program to test your class.

```
#include<iostream>
using namespace std;
class Student
{
        private:
                int roll;
        public:
                void setroll()
                {
                        cout<<"Enter roll number"<<endl;
                        cin>>roll;
                }
                void showroll()
                {
                        cout<<"Roll= "<<roll<<endl;
                }
};

class Test: public Student
{
        protected:
                float com,eng;
        public:
                void setmark()
                {
                        cout<<"Enter marks of computer and English "<<endl;
                        cin>>com>>eng;
                }

                void showmark()
                {
                        cout<<"Computer= "<<com<<endl;
                        cout<<"English= "<<eng<<endl;
                }
};

class Sport
{
                protected:
                float score;
        public:
                void setscore()
                {
                        cout<<"Enter score in sports "<<endl;
                        cin>>score;
                }
```

```
                void showscore()
                {
                        cout<<"Score in sports= "<<score<<endl;
                }
};

class Result: public Test, public Sport
{

        public:
                void showtotal()
                {
                        float tot;
                        tot=com+eng+score;
                        cout<<"Total obtained marks= "<<tot<<endl;
                }
};

int main()
{
        Result res;
        res.setroll();
        res.setmark();
        res.setscore();

        res.showroll();
        res.showmark();
        res.showscore();
        res.showtotal();
}
```

Output:

```
Enter roll number
5
Enter marks of computer and English
55
66
Enter score in sports
4
Roll= 5
Computer= 55
English= 66
Score in sports  4
Total obtained marks= 125
```

Programming Examples

1. Define a shape class (with necessary constructors and member functions) in Object Oriented Programming (abstract necessary attributes and their types). Write a complete code in C++ programming language.
i. Derive triangle and rectangle classes from shape class adding necessary attributes.
ii. Use these classes in main function and display the area of triangle and rectangle.

```
#include<iostream.h>
#include<conio.h>
class shape
{
        protected:
        float breadth, height, area;
        public:
        void getshapedata()
        {
                cout<<"Enter breadth:"<<endl;
```

```
                cin>>breadth;
                cout<<"Enter height:"<<endl;
                cin>>height;
        }
};

class triangle: public shape
{
        public:
        void calarea()
        {
                area=(breadth * height)/2;
        }
        void display()
        {
                cout<<"The area of triangle is"<<area<<endl;
        }
};
class rectangle: public shape
{
        public:
        void calarea()
        {
                area=breadth * height;
        }
        void display()
        {
                cout<<"Area of rectangle is"<<area<<endl;
        }
};

int main()
{
        triangle T;
        rectangle R;
        cout<<"Enter triangle data:"<<endl;
        T.getshapedata();
        cout<<"Enter rectangle data:"<<endl;
        R.getshapedata();
        T.calarea();
        R.calarea();
        T.display();
        R.display();
}
```

2. Define a *student* class (with necessary constructors and member functions) in Object Oriented Programming (abstract necessary attributes and their types). Write a complete code in C++ programming language.

i. Derive a C*omputer Science and Mathematics* class from *student* class adding necessary attributes (at least three subjects).

ii. Use these classes in a main function and display the average marks of computer science and mathematics students.

```
#include<iostream.h>
#include<conio.h>
class student
{
```

```
        protected:
                float english, sum, avg;
        public:
        void getstudentdata()
        {
                cout<<"Enter english marks:"<<endl;
                cin>>english;
        }
};
class computer : public student
{
        float IT, cprog, networks;
        public:
        void getcomputerdata()
        {
                cout<<"Enter marks in IT:"<<endl;
                cin>>IT;
                cout<<"Enter marks in cprog:"<<endl;
                cin>>Cprog;
                cout<<"Enter marks in networks:"<<endl;
                cin>>Networks;
        }
        void average()
        {
                sum=english+IT+cprog+networks;
                avg=sum/4;
                cout<<"Average marks is"<<avg;
        }
};
class mathematics : public student
{
        float calculus, stat, algebra;
        public:
        void getmathdata()
        {
                cout<<"Enter marks in calculus:"<<endl;
                cin>>calculus;
                cout<<"Enter marks in statistics:"<<endl;
                cin>>stat;
                cout<<"Enter marks in Linear Algebra:"<<endl;
                cin>>algebra;
        }
        void average()
        {
                sum=english+calculus+stat+algebra;
                avg=sum/4;
                cout<<"Average marks is"<<avg;
        }
};

int main()
{
        computer C;
        mathematics M;
        cout<<"Enter marks of computer students:"<<endl;
        C.getstudentdata();
        C.getcomputerdata();
```

```
        cout<<"Enter marks of mathematics student:"<<endl;
        M.getstudentdata();
        M.getmathdata();
        C.average();
        M.average();
}
```

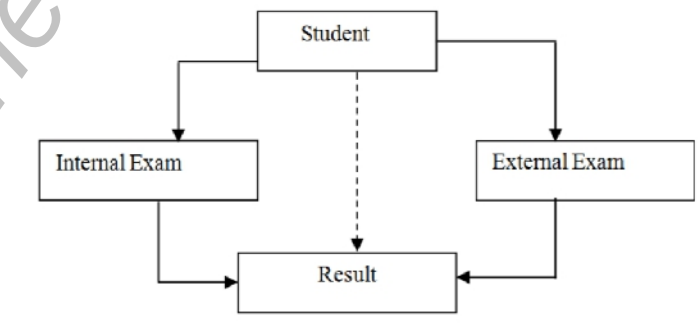
3. Define a *Clock* class (with necessary constructor and member functions) in OOP (abstract necessary attributes and their types). Write a complete code in C++ programming language.
i. Derive *Wall_Clock* class from *Clock* class adding necessary attributes.
ii. Create two objects of *Wall_Clock* class with all initial state to 0 or NULL.

```
#include<iostream>
#include<conio.h>
#include<string.h>
using namespace std;
class clock
{
        protected:
        char model_no[10];
        float price;
        char manufacturer[50];
        public:
        void getclockdata()
        {
                cout<<"Enter clock manufacturer:"<<endl;
                cin>>manufacturer;
                cout<<"Enter model number:"<<endl;
                cin>>model_no;
                cout<<"Enter price:"<<endl;
                cin>>price;
        }
        void clockdisplay()
        {
                cout<<"Model number="<<model_no<<endl;
                cout<<"Manufacturer="<<manufacturer<<endl;
                cout<<"Price="<<price<<endl;
        }
};

class wall_clock: public clock
{
        int hr, min, sec;
        public:
        wall_clock()
        {
                strcpy(model_no,NULL);
                strcpy(manufacturer,NULL);
                price=0.0;
                hr=0;
                min=0;
```

```
                sec=0;
        }
        void getwallclockdata()
        {
                cout<<"Enter hour, minute and seconds:"<<endl;
                cin>>hr>>min>>sec;
        }
        void wallclockdisplay()
        {
                cout<<"Time="<<hr<<":"<<min<<":"<<sec<<endl;
        }
};
int main()
{
        wall_clock W1, W2;
        cout<<"Enter data for W1:"<<endl;
        W1.getclockdata();
        W1.getwallclockdata();
        cout<<"Value of W1:"<<endl;
        W1.clockdisplay();
        W1.wallclockdisplay();

        cout<<"Enter data for W2:"<<endl;
        W2.getclockdata();
        W2.getwallclockdata();
        cout<<"Value of W2:"<<endl;
        W2.clockdisplay();
        W2.wallclockdisplay();
}
```

Multi path inheritance

- When a class is derived from two or more classes, those are derived from the same base class. Such a type of inheritance is known as multipath inheritance.
- The multipath inheritance also consists of many types of inheritance, such as multiple, multilevel, and hierarchical, as shown in Figure below.



- Here, all three kinds of inheritance exist i.e. multilevel, multiple and hierarchical.
- The class Result has two direct base class Internal Exam and External exam which themselves have a common base class Student. So, the class Result inherits the class Student via two separate path called as

multi path inheritance. The class student is sometime called as indirect base class. It also can be inherited directly as shown by the broken line.

Problem with multipath inheritance and virtual base class

- Since all the public and protected members of indirect base class i.e. Student are inherited into final derived class i.e. Result via two paths, first via Internal Exam and second via External exam. This means the class Result would have duplicate sets of members inherited from Student. This introduces **ambiguity** and should be avoided.
- This ambiguity can be avoided by making a **common base class as virtual base class** while declaring the direct or intermediate base classes as shown below.

```
class Student //grand parent
{
};
Class Internal_Exam: virtual public Student //parent1
{
};
Class  External_Exam: virtual public Student//parent2
{
}
Class Result: public Internal_Exam, public External_Exam//child
{
  //only one copy of Student will be inherited
}
```

- So, when a class is made virtual base class, c++ takes care to see that only one copy of that class is inherited, regardless of how many paths exist between the virtual base class and a derived class.

Note: the use of keyword virtual and public can be done in any order

Programming example

Create a class Student with data member roll_no and two functions to initialize and display it. Derive two new classes Theory and Practical from Student. Define suitable functions to assign and display theory and practical marks for two different subjects. Again, derive a new class Result from both class Theory and Practical and add a new function to display the final total marks of student. Write a main program to test your class.

```
#include<iostream>
using namespace std;
class Student
{
        private:
                int roll;
```

```
        public:
                void setroll()
                {
                        cout<<"Enter roll number"<<endl;
                        cin>>roll;
                }
                void showroll()
                {
                        cout<<"Roll= "<<roll<<endl;
                }
};
class Theory: public virtual Student
{
        protected:
                float comth,ength;
        public:
                void setdatatheory()
                {
                        cout<<"Enter Theory marks of computer and English "<<endl;
                        cin>>comth>>ength;
                }
                void showmarkstheory()
                {
                        cout<<"Computer(Theory)= "<<comth<<endl;
                        cout<<"English(THeory)= "<<ength<<endl;
                }
};

class Practical: public virtual  Student
{
                protected:
                float compr,engpr;
        public:
                void setdatapractical()
                {
                        cout<<"Enter Practical marks of computer and English "<<endl;
                        cin>>compr>>engpr;
                }
                void showmarkspractical()
                {
                        cout<<"Computer(Practical)= "<<compr<<endl;
                        cout<<"English(Practical)= "<<engpr<<endl;
                }
};

class Result: public Theory, public Practical
{
        public:
                void showtotal()
                {
                        float tot;
                        tot=comth+ength+compr+compr;
                        cout<<"Total obtained marks= "<<tot<<endl;
                }
};
```

```
int main()
{
        Result res;
        res.setroll(); // is ambiguous because multipath exist to reach setroll()method from derived class
                                                so must implement virtual base class to overcome this
        res.setdatatheory();
        res.setdatapractical();

        res.showroll();
        res.showmarkstheory();
        res.showmarkspractical();
        res.showtotal();
}
```

```
Enter roll number
5
Enter Theory marks of computer and English
70
75
Enter Practical marks of computer and English
20
25
Roll= 5
Computer(Theory)= 70
English(THeory)= 75
Computer(Practical)= 20
English(Practical)= 25
Total obtained marks= 185
```


Benefits/Advantage of Inheritance

- **Software component:** Inheritance provides programmers with the ability to construct reusable software components.
- **Software Reusability:** When a function is inherited from another class, the code that provides that functionality doesn't have to be written.
- **Reliable, Save Time and Effort:** Developing a new program by reusing exiting software component make it less error prone and hence more reliable. Also using reusable code saves both time and effort of programmer from rewriting code that have been written previously.
- **Rapid Prototyping:** When a software is built using large amount of reusable software components. we can totally focus on understanding of new an undiscovered portion of the system. Thus, we can develop a system more quickly and easily leading to style of programming known as rapid prototyping or exploratory programming.
- **Code Sharing:** Inheritance offers code sharing at various level. In one level, many programmers or projects can use the same class. And in another level, two or more derived classes can be derived from a single parent class.
- **Consistency of interface:** When two or more classes derived from the same base class, we can be assured that the behavior(functionality) they inherit will be same in all classes.
- **Extendibility:** We can not only inherit existing properties of parent class but also extend base class logic as per requirement.
- **Data Hiding:** Base class can keep some data private so that it can't be altered by derived class.

- **Overriding:** with the help of inheritance, we can override the method of base class for meaningful execution of program. Method overriding means base and derived class having method of same signature.

Demerits/Disadvantage of Inheritance

- Some of the demerits of inheritance are

1. **Execution Speed:**
   - There is no doubt that carefully hand-crafted system(tailored) is faster that some general-purpose software tools.
   - Similarly, Inherited methods are slower as compared to specialized code /normal method. This is because inherited methods must deal with arbitrary number of sub classes.
2. **Program Size:**
   - The use of reusable software components may require a size penalty. This is because some data members and functions in the software library may be of no use in our current project. This leads to memory wastage.
3. **Program Complexity:**
   - Improper use of inheritance may lead to wrong solution. Also, it is sometime difficult to understand the flow of control of program that uses inheritance as it may requires multiples scans up and down the inheritance graph.

Method overriding

- The process of redefining the inherited methods of the base class in the derived class is called method overriding.
- The method in the derived class should have same signature and same return type as that of base class in order to override it.
- Since it refers to the subclass object and subclass method overrides the Parent class method, subclass method is invoked at runtime, it is also called **as Run Time Polymorphism**.[Will be discussed in next chapter]

Example:

```
#include<iostream>
using namespace std;
class Base
{
        public:
        void display()
        {
                cout<<"This is hello world from base class";
        }
};
```

```
class Derived: public Base
{
        public:

                void display()
                {
                        cout<<"This is hello world from derived class";
                }
};

int main()
{
        Derived d;
        d.display();
}
```

```
This is hello world from derived class
--------------------------------
```

Overloading vs overriding

	Overloading	Overriding
Definition	Methods having same name but each must have different number of parameters or parameters having different types & order.	Sub class have method with same name and exactly the same number and type of parameters and same return type as super class method.
Meaning	More than one method shares the same name in the class but having different signature.	Method of base class is re-defined in the derived class having same signature.
Behaviour	To Add/Extend more to method's behaviour.	To Change existing behaviour of method.
Polymorphism	Compile Time	Run Time
Inheritance	Not Required	Always Required
Method Signature	Must have different signature	Must have same signature.

Constructors in Derived class

- Constructors are basically used for initializing objects. If we have not defined any parameterized constructor in base class, then it is not necessary for defining constructor in derived class.
- **But if we define a parameterized constructor in base class then we must define constructor in derived class also and pass the arguments to the base class constructors.**
- When both base and derived classes contain constructors then Constructor are invoked **in order of** derivation from base to subclass when object of derived class is created.

Example: The below example show the construction execution order in case of multilevel inheritance

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

```
class A
{
	public:
		A()
		{
			cout<<"Constructor of class A"<<endl;
		}
};
class B: public A
{
	public:
		B()
		{
			cout<<"Constructor of class B"<<endl;
		}
};
class C: public B
{
	public:
		C()
		{
			cout<<"Constructor of class C"<<endl;
		}
};

int main()
{
	C a;
}
```

```
Constructor of class A
Constructor of class B
Constructor of class C
```

Again, consider the case of **multiple inheritance.**

```
#include<iostream>
using namespace std;
class A
{
	public:
		A()
		{
			cout<<"Constructor of class A"<<endl;
		}
};
class B
{
	public:
		B()
		{
			cout<<"Constructor of class B"<<endl;
		}
};
```

```
class C: public B,public A
{
	public:
		C()
		{
			cout<<"Constructor of class C"<<endl;
		}
};

int main()
{
	C a;
}
```

```
Constructor of class B
Constructor of class A
Constructor of class C
```

Invoking base class constructor from derived class

- **Constructed can't be inherited but can be invoked from derived class.**
- The constructor of the derived class receives a list of arguments and invokes base class constructor for initializing base class.
- The base constructors are called and executed before executing the statements in the body of derived constructor.
- Let Derived be a derived class, Base1 and Base2 be two base class, then the general form for defining constructors in derived class is given below

```
Demo(arg1, arg2, arg3….) : Base1(arg1), Base2(arg2)
{
	//body of derived class
}
```

- Base1(arg1), Base2(arg2) invokes the base class constructors. So while creating object of derived class Demo, we must pass a list of arguments that can be used for initializing derived class as well as base classes. Here arg1 and arg2 are the arguments for base class constructor which must be provided while creating the object of derived class.

Example:

A class Room consists of two fields length and breadth, a constructor and a method to return the area of room. A new class BedRoom is derived from class Room and consist of additional field height, a constructor and function to find the volume. Now write the c++ program to input the length, breadth and height and find the area and volume.

```
#include<iostream>
using namespace std;
class Room
{
	private:
		int length,breadth;
	public:
		Room(int l, int b)
		{
				length=l;
				breadth=b;
		}

		int area()
		{
			return(length*breadth);
		}
};

class BedRoom: public Room
{
	private:
		int height;
	public:
		BedRoom(int l, int b, int h):Room(l,b)
		{
			height=h;
		}
		int volume()
		{
			return(area() * height);
		}
};

int main()
{
	BedRoom b(4,5,6);
	cout<<"Area= "<<b.area()<<endl;
	cout<<"Volume= "<<b.volume();
}
```

```
Area= 20
Volume= 120
--------------------------------
```

Note:
We also can use following syntax to initialize class objects.

```
	Constructor (argument list): initialization section
	{

	}
```

The initialization section can be used to provided initial values to the base constructors and also to initialization its own class members.

Example:

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

```
class Room
{
	private:
		int l,b;
	public:
		Room(int x, int y):l(x),b(y){ }
		int area()
		{
			return(l*b);
		}
};

int main()
{
	Room r(4,5);
	cout<<"Area= "<<r.area();
}
```

```
Area= 20
--------------------------------
```

Destructors in derived class

- When both base and derived classes contain constructors then Constructor are invoked in order of derivation from base to subclass when object of derived class is created.
- The destructor of the class whose constructor is called last is called first and destructor of the class whose constructor is called first is called at last.
- The below example shows the order of destructor execution in case of multilevel inheritance.

```
#include<iostream>
using namespace std;
class A
{
    public:
		A()
		{
			cout<<"Constructor of class A"<<endl;
		}
		~A()
		{
			cout<<"Destructor of class A"<<endl;
		}

};

class B: public A
{
    public:
		B()
		{
			cout<<"Constructor of class B"<<endl;
		}
		~B()
		{
			cout<<"Destructor of class B"<<endl;
		}
```

```
};

class C: public B
{
    public:
        C()
        {
            cout<<"Constructor of class C"<<endl;
        }
        ~C()
        {
            cout<<"Destructor of class C"<<endl;
        }
};

int main()
{
    C a;
}
```

```
Constructor of class A
Constructor of class B
Constructor of class C
Destructor of class C
Destructor of class B
Destructor of class A
```

- **The below example shows the execution order of destructor in case of multiple inheritance.**

```
#include<iostream>
using namespace std;
class A
{
        public:
                A()
                {
                        cout<<"Constructor of class A"<<endl;
                }
                ~A()
                {
                        cout<<"Destructor of class A"<<endl;
                }
};



class B
{
        public:
                B()
                {
                        cout<<"Constructor of class B"<<endl;
                }
                ~B()
                {
                        cout<<"Destructor of class B"<<endl;
                }
```

```
};


class C: public B, public A
{
	public:
		C()
		{
			cout<<"Constructor of class C"<<endl;
		}
		~C()
		{
			cout<<"Destructor of class C"<<endl;
		}
};

int main()
{
	C a;
}
```

```
Constructor of class B
Constructor of class A
Constructor of class C
Destructor of class C
Destructor of class A
Destructor of class B
```

IS-A Relationship(inheritance) vs HAS-A Relationship (Composition)

- Association is a structural relationship that defines how objects are related with each other.
- Broadly there are two kinds of relationship in which object can be associated with each other.
    1. IS-A relationship
    2. HAS-A relationship

IS-A Relationship

- A relationship of type is-a (is a relationship), in which one class is a subtype of another class. In this, one class expands (details) the capabilities of another class. IS-A relationship asserts the instance of a sub class musb be more specialized form f the super class. Thus, instance of a subclass can be used where quantities of super class type are required. In OOP, the concept of IS-A relationship is based on **inheritance.**
- For example, Student is a person, Staff is a Employee, Apple is a Fruit, Car is a Vehicle etc. So if A is a B then we say A has isa relationship with B and this is modeled in OOP through inheritance such that A is super class and B is subclass. In above, Parent is Base class while Person is child class and similar for other.

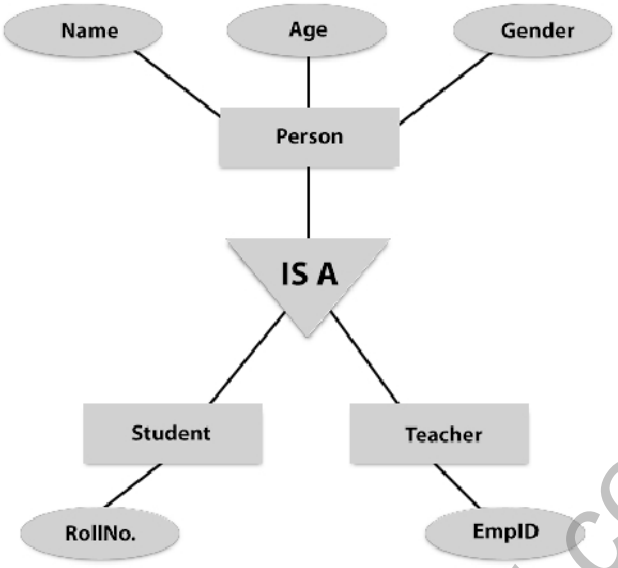


Fig: IS-A relationship

- The following program demonstrates the concept

```
class Person
{
};
class Student : public Person
{
}
class Teacher : public Person
{
}
```

In the above example, class Student and Teacher holds ISA relationship with class Person so this is implemented using concept of inheritance. Here Student and Teacher are both sub class of super class Person.

HAS-A Relationship (Composition/Containership)

- Composition is the design technique in object-oriented programming to implement **has-a** relationship between objects.
- Using composition, A relationship of type has-a (has a relationship) is defined, in which one class **contains** instances of another class as member.
- For example: Human has a Heart, College has a department, Car has a Engine
- In Composition, parent owns child object so child entity can't exist without parent entity. We can't directly or independently access child entity. Also the containing object is responsible for creation and life time of contained object.

- A good real-life example of a composition is the relationship between a person's body and a heart.
- 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 can not be part of someone else's body at the same time.
- In a composition relationship, the object is responsible for the existence of the parts. Most often, this means the part is created when the object is created, and destroyed when the object is destroyed. But more broadly, it means the object manages the part's lifetime in such a way that the user of the object does not need to get involved. For example, when a body is created, the heart is created too. When a person's body is destroyed, their heart is destroyed too. Because of this, composition is sometimes called a "death relationship".

Implementing Composition in C++

- 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.
- 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" engine. Your computer "has-a" CPU. You "have-a" heart. Person has a Address. The complex object is sometimes called the whole, or the parent. The simpler object is often called the part, child, or component.
- Object Composition is useful in a C++ context because it allows us to create complex classes by combining simpler, more easily manageable parts. This reduces complexity, and allows us to write code faster and with less errors because we can **reuse code** that has already been written (**code reusability**), tested, and verified as working.

```
#include<iostream>
#include<stdlib.h>
using namespace std;
class Address
{
        private:
                char city [100];
                char street[100];
        public:
                void setAddress()
                {
                        cout<<"Enter city and streen"<<endl;
                        cin>>city>>street;
                }
                void displayAddress()
                {
                        cout<<"City="<<city<<endl;
```

```
                cout<<"Street="<<street<<endl;
        }
};
class Person
{
    private:
        char name[100];
    public:
        Address a;//composition or containershipe or Address is part of Person or Person has
                                                                                Address
        void setName()
        {
            cout<<"Enter name"<<endl;
            cin>>name;
        }

        void showName()
        {
            cout<<"Name="<<name<<endl;
        }
};
int main()
{
    Person p; // This only not create Person object but also create Address Object
    p.setName();
    p.a.setAddress(); // Since Person is composed of Address we can reuse functionality of Address
    p.showName();
    p.a.displayAddress(); // Since Person is composed of Address we can reuse functionality of
                                                                                Address
    delete(p); //This doesn't only destroy object P, but also destroys Address Object
}
```

- In the above example, the association between class Person and Address is HAS-A i.e Person has a Address, which is implemented via composition concept of OOP. For this, the class Person is composed of instance of class Address. So, using concept of composition, it clearly shows how the existing code of class Address can be reused for class Person. Instead of rewriting the code for setting and displaying address in class Person, we have created class Person composed of instance of class Employee. Now, we are able to set and display address via Person object. Also, it is important that creation of object of person is responsible to creates Address object.

WAP to concatenate two strings(name and address of a person) using concept of containership.

```
#include<iostream>
#include<string.h>
using namespace std;
class Name
{
    char nm[100];
```

```
        public:
                void setName()
                {
                        cout<<"Enter name"<<endl;
                        cin>>nm;
                }
                char * getName()
                {
                        return(nm);
                }
};
class Address
{
        char ad[100];
        public:
                void setAddress()
                {
                        cout<<"Enter address"<<endl;
                        cin>>ad;
                }
                char * getAddress()
                {
                        return(ad);
                }
};
class Person
{
        public:
                Name n; //Person Contains Name
                Address a; //Person Contains Address
                void displayNameAddress()
                {
                        char tmp[100]="";
                        strcat(tmp,n.getName());
                        strcat(tmp,a.getAddress());
                        cout<<"Concatenated Name and Address="<<tmp<<endl;
                }
};
int main()
{
        Person p;
        p.n.setName();
        p.a.setAddress();
        p.displayNameAddress();
}
```

```
Enter name
ram
Enter address
pokhara
Concatenated Name and Address=rampokhara
```

Composition(HAS-A) vs Inheritance(IS-A)

Inheritance	Composition
It is the functionality by which one object acquires the characteristics of one or more other objects.	Using an object within another object is known as composition.
Class inheritance is defined at compile-time.	Object composition is defined dynamically at run-time.
Exposes both public and protected members of the base class.	The internal details are not exposed to each other and they interact through their public interfaces.
No access control in inheritance.	Access can be restricted in composition.
It exposes a subclass to details of its parent's implementation, so it often breaks encapsulation.	Objects are accessed solely through their interfaces, so it won't break encapsulation

DB Difference Between.net

Subclass vs Subtype

- In OOP. We can use two concept to inherit base class functionality. These are **subclass and subtype**

Sub class

- The class that inherits properties from another class is called Subclass or Derived Class.
- Consider a relationship of data of parent class to a data of sub class then
  - Object of subclass must posses all data members associated with parent class.
  - Object of subclass must implement all functionality of parent class through inheritance
  - Object of child class
- For example: If class B is derived from class A then we say class B is sub class of super class A. Subclasses allow o1ne to reuse the code inside classes - both instance variable declarations and method definitions. Thus, they are useful in supporting code reuse *inside* a class.

Example:

```
class Employee
{
    public:
            void setName(){}
            void setAddress(){}
            ……….
}
```

```
class Lecture: public Employee//Lecture is Subclass of Super Class Employee
{

}
int main()
{
    Lecture l;
    l.setName();
    ……….
}
```

- In the above example, the relationship Lecture and Employee is **Lecture is a Employee (isa relationship),** which can be modeled using inheritance in OOP. Hence instead of rewriting code in Lecture to achieve already defined functionality of class Employee, we can create a subclass Lecture from super class Employee. So we can now reuse all the code inside the class Employee using subclass concept of inheritance.

Issues with Subclass

- When we reuse using subclasses, sometime subclasses may behave differently as compared to their base class. The following example demonstrates this situation.

Example

```
class Rectangle
{
        protected:
        int len,bre;
        public:
        void setDimension()
        {
                cout<<"Enter length and breadth"<<endl;
                cin>>len>>bre;
        }
        void displayArea()
        {
                cout<<"Area of rectangle="<<(len*bre)<<endl;
        }
};
class Square: public Rectangle
{

};
```

- As we know the geometrical relationship between Rectangle and Square that Square is a Rectangle i.e. Square is specialization of Rectangle. This isa relationship again can be modeled using inheritance in OOP. Based on the previous concept if we use concept of subclass for inheritance in this case then a strange behavior is observed during implementation with this logic.
- For rectangle, the above functionality setDimension() and displayArea() are quite logical in case of super class Rectangle but for base class Square achieving the above functionality by inheritance is meaningless as it doesn't make any sense for a Square. This issues in subclass leads to the concept of subtype for inheritance.

Subtype

- Subtyping on the other hand is useful in **supporting reuse externally i.e. reuse of interface**, giving rise to a form of **type polymorphism**. That is, once a data type is determined to be a subtype of another, any function or procedure that could be applied to elements of the supertype can also be applied to elements of the subtype.
- So, subtype is all about inher**itance of interface/functionality independent** from specific implementation such that it preserves the purpose/meaning of original supertype while subclass is all about reuse of code. The C++ interfaces are implemented using **abstract base classes**.[Discussed in Next Chapter]
- Subtyping is a form of type polymorphism in which a subtype is a datatype that is related to another data type i.e. super type by some notions of **substitutability.**
- The principle of substitutability states that if we have two class A and B such that B is a subclass of class A, it should be possible to substitute instance of class B for instances of class A in all situations without any observable effect. i.e. a supertype should be substitutable by any of its sub types without breaking the meaning of application.
- So subtype can be used to refer a subclass relationship in which principal of substitution is maintained t distinguish such forms from the general sub class relationship, which may or may not satisfy the principle of substitutability.

Example

```
#include<iostream>
using namespace std;
class Shape //abstract base class i.e. interface
{
        public:
                virtual void setDimension()=0;   //pure virtual function
                virtual void displayArea()=0; //pure virtual function
};
class Rectangle:public Shape
{
        private:
                int len,bre;
        public:
                void setDimension()
                {
                        cout<<"Enter length and breadth"<<endl;
                        cin>>len>>bre;
                }
```

```
                void displayArea()
                {
                        cout<<"Area of Rectangle="<<(len*bre)<<endl;
                }
};
class Square:public Shape
{
        private:
                int s;
        public:
                void setDimension()
                {
                        cout<<"Enter length of a side of a square"<<endl;
                        cin>>s;
                }
                void displayArea()
                {
                        cout<<"Area of Square="<<(s*s)<<endl;
                }
};
int main()
{
        Shape *sh;// Super Type
        Rectangle re; //Sub Type
        Square sq;          //Sub Type
        sh=&re;          //Super Type sh Substituted with Subtype re
        sh->setDimension();
        sh->displayArea();
        sh=&sq; //Super Type sh Substituted with Subtype sq
        sh->setDimension();
        sh->displayArea();
}
```

So, in the above program, it clearly shows the implementation of principle of substitutability i.e. superclass type shall be substituted with the sub class type.

Forms of inheritance

Inheritance can be used in varieties of ways. The various forms of inheritance based on how we use them are described below

1. **Subclassing for Combination**
   - In this form of inheritance, the child class inherits features from more than one parent class.
   - This form of inheritance is achieved through multiple inheritance.

- Example:

```
class A
{
};
class B
{
};

Class C: public A, public B
{
};
```

Here class C is derived from two parent classes A and B. So multiple inheritance takes place here.

2. Subclassing for Variance

- In this form of inheritance, the child class and parent class are alternative of each other and the class and subclass relationship is arbitrary.
- This form of inheritance is employed when two or more classes have similar implementations but do not seem to possess any hierarchical relationship.
- For example: Say behavior i.e., functionality provided by class A and class B are nearly identical. So instead of defining the functionality in both class, one can be arbitrarily chosen as Parent class and other as Child class. Now, the common code can be inherited by child class and can overriding some functionality specific to class B.

```
class A
{
    public:
            void test1(){
                    cout<<"I have common role in base and derived class"<<endl;
            }
            void test2()
            {
                    cout<<"I am from Base Class"<<endl;
            }
    }
};
```

```
class B: public A
{
    public:
        void test2()//overriding for Specific Implementation for class B
        {
            cout<<"I am from Derived Class"<<endl;
        }
};
```

- In above example, B can use test1() through inheritance, which is common to both A and B while override test2() or specific implementation.

3. Subclassing for Limitation

- In this form of inheritance, the child class restricts the use of some of the behavior inherited from the parent class.
- Subclassing or limitation occurs when the behavior of the sub class is smaller or more restrictive than behavior of parent class.
- For example: If there are two class A and B such that B is sub class of class A and we want some behavior of A only be available in parent class not in class B then this can be achieved using concept of method overriding. The overridden method shows some meaningful error message if restricted or undesired method are used.

```
class A
{
    public:
        void test()
        {
            cout<<"Restrictive Method only available in base class"<<endl;
        }
};
class B: public A
{
    public:
        void test()
        {
            cout<<"Restrictive Method not Availabe Here";
        }
};
```

In above example, if test() method is accessed via object of B, then the overridden method is invoked as resulting following output. Clearly class B inherits limited features from class A.

```
Restrictive Method only available in base class

------------------------------
```

4. **Subclassing for Extension**

- In this form of inheritance, the child class add new functionality to the parent class but doesn't change(override) any inherited behavior
- This form of inheritance totally add new functionalities in the derived class.

Example:

```
Class A
{
        public:
        void test1(){}
}
Class B: public A
{
        public:
        void test2(){}
}
```

In the above example, B inherits class A and is now able to reuse the functionality provided by test1() method and extends new functionality provided by test2() method.

5. **Subclassing for Generalization**

- In this form of inheritance, the child class expands or modifies/overrides some or at least one of the methods of parent class.
- Example: Consider class Window with display() function for displaying black and white background. Now we can create new sub class ColoredWindow from parent class Window that overrides the display() method to draw the background other than black and white i.e. colored background

```
class Window
{
    public:
    void display(){}//display black and white background
};
Class ColoredWindow: Window
{
    public:
    void display(){}//Override method to display colored background
}
```

6. **Subclassing for Construction**
   - In this form of inheritance, the child class make use of behavior provided by the parent class but is not subtype of parent class.
   - Here, a class can inherit almost all of its desired functionality from a parent class and **may change only the names of the methods use to interface to the class, or modifying the arguments in certain fashion.**
   - It simplify the construction of newly formed abstraction but is not a form of subtype and hence violates principle of substitutability.

   For example:

```
#include<iostream>
using namespace std;
class Storable
{
        protected:
        void writeBytes ()
        {
                cout<<"Write Operation Completed"<<endl;
        }
};
class StoreStruct: private Storable //Private inheritance
{
         public:
         void writeStructs () // the name used to interface the base class method is changed here
         {
                writeBytes();//uses the behavior of parent class for actual writing operation
                                //Note: writeBytes() now here is logically private member of
                                        StoreStruct class due to private inheritance

         }
};
int main()
{
        StoreStruct s;
        //s.writeBytes();//Not allowed due to private inheritance
        s.writeStructs(); //
}
```

   Here the parent class Storable may implement only the ability to write binary data. A new subclass StoreStruct is constructed for every structure that is saved. The subclass implements a save procedure for the struct data type, which uses the behavior of the parent type to do the actual storage.

   Also, we clearly see that the derived class add new method writeStructs() which provides an interface to call the base class method writeBytes(). Due to private inheritance, writeBytes() is hidden from external world so we use the public extended method writeStructs() to access it.

7. **Subclassing for Specification**
   - In this form of inheritance, the parent class defines behavior that is implemented in the child class but not in parent class.
   - Here, the super class defines an interface (abstract base class) that just declares the required methods (abstract method) specification without definition.

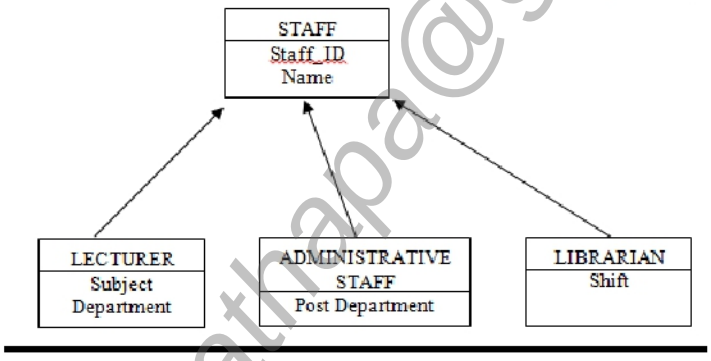
- Now it is the responsibility of child class that is subclass of above interface to define all the abstract methods.
- Example: Previously Discussed

8. Subclassing for Specialization (Subtyping)

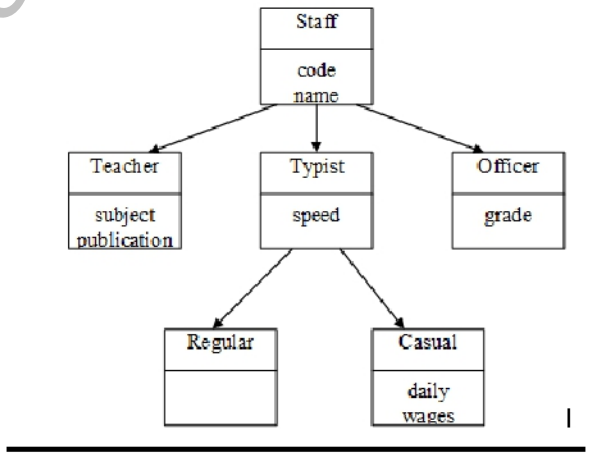
- In this form of inheritance, the child class is a subtype of parent class. This form of inheritance support principle of substitutability.
- The child class is a special case of parent class. The subclass can inherit all the behavior of the parent class and **override** it in order to specialize the class in some way.
- Example: Previously Discussed

Assignment-4

1. Develop a complete program for an institution, which wishes to maintain a database of its staff. The database is divided into number of classes whose hierarchical relationship is shown in the following diagram. Specify all the classes and define **constructors** and functions to create database and retrieve the individual information as per the requirements.



2. An Education institute wishes to maintain a database of its employees. The database is divided into a number of classes whose hierarchical relationships are shown in figure. The figure also shows the minimum information required for each class. Specify all the classes and define functions to create the database and retrieve individual information as an when required. . What form of inheritance will the classes hold in this case?



3. The following figure shows minimum information required for each class. Specify all the classes and define functions to assign and retrieve individual information. What form of inheritance will the classes hold in this case?