Chapter 4
Object Inheritance and Re-usability

The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important feature of Object Oriented Programming.

Sub Class:

The class that inherits properties from another class is called Sub class or Derived Class.

Super Class:

The class whose properties are inherited by sub class is called Base Class or Super class.

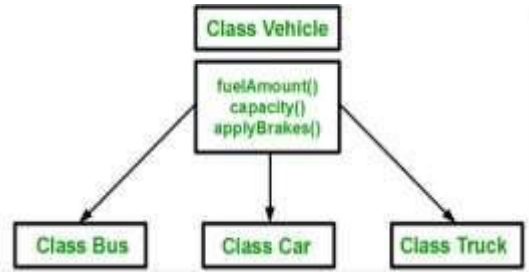
The derived class inherits some or all the traits from base class. The base class is unchanged by this. Most important advantage of inheritance is re-usability. Once a base class is written and debugged, it need not be touched again and we can use this class for deriving another class if we need. Reusing existing code saves time and money. By re-usability a programmer can use a class created by another person or company and without modifying it derive other class from it.

Why and when to use inheritance?

Consider a group of vehicles.

We need to create classes for Bus, Car and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be same for all of the three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown in below:

We can clearly see that above process results in duplication of same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability.



Syntax:

```
class subclass_name : access_mode base_class_name
{
    //body of subclass
};
```

Advantages of Inheritance

1. Inheritance promotes reusability. When a class inherits or derives another class, it can access all the functionality of inherited class.
2. Reusability enhanced reliability. The base class code will be already tested and debugged.
3. As the existing code is reused, it leads to less development and maintenance costs.
4. Inheritance makes the sub classes follow a standard interface.
5. Inheritance helps to reduce code redundancy and supports code extensibility.
6. Inheritance facilitates creation of class libraries

Disadvantages of Inheritance

1. Inherited functions work slower than normal function as there is indirection.
2. Improper use of inheritance may lead to wrong solutions.
3. Often, data members in the base class are left unused which may lead to memory wastage.
4. Inheritance increases the coupling between base class and derived class. A change in base class will affect all the child classes

Example of Inheritance

```
// C++ program to demonstrate implementation of Inheritance
#include <iostream> using
namespace std;
//Base class
class Parent
{
    public:
    int id_p=9;
```

```
};
// Sub class inheriting from Base Class(Parent) class
Child : public Parent
{
        public:
        int
id_c=8;
};
//main function
int main()
{
                Child obj1;
                cout << "Child id is " << obj1.id_c << endl;
        cout << "Parent id is " << obj1.id_p << endl;
                return 0;
}
```

Inheritance Visibility Mode

Depending on Access modifier used while inheritance, the availability of class members of Super class in the sub class changes. It can either be private, protected or public.

Public mode:

If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.

Protected mode:

If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.

Private mode:

If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

	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

```
// C++ Implementation to show that a derived class doesn’t inherit access to private data members.
// However, it does inherit a full parent object.
class A { public:
   int x;
protected:
   int y;
private:
int z;
};

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

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

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

};

Types of Inheritance

In C++, we have 5 different types of Inheritance. Namely,

a. Single Inheritance

b. Multiple Inheritance

c. Hierarchical Inheritance

d. Multilevel Inheritance

e. Hybrid Inheritance (also known as Virtual Inheritance)

1. Single Inheritance:

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

Syntax:

class subclass_name : access_mode base_class

{

//body of subclass

};

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

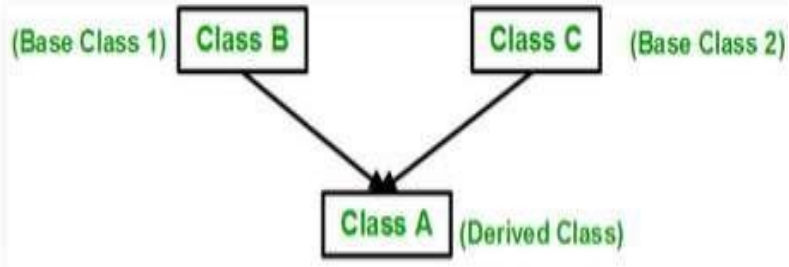
{

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();
}
```

2. Multiple Inheritance:

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



Syntax:

```
class subclass_name : access_mode base_class1, access_mode base_class2, ....
        {
                //body of subclass
        };
```

Example:

Q. 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();
        }
```

Ambiguity/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)
                       {
```

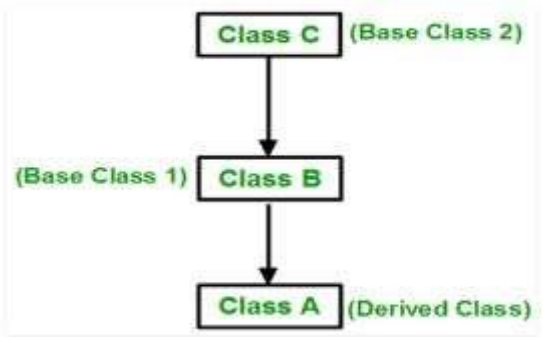
```
                a=x;
               }

};
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.

3. 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

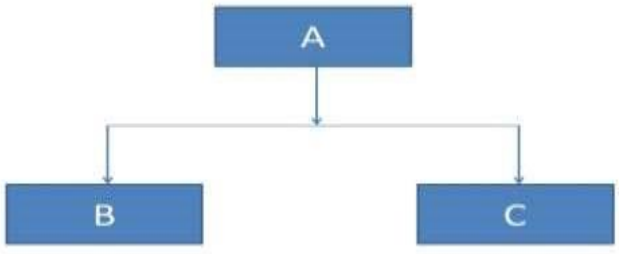
field total and a method to display entire details along with total obtained marks. WAP to input roll number, marks in two different subject and display total.

```
#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();
}
```

4. 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?

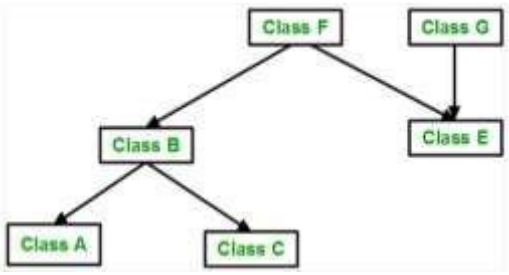
```
#include<iostream>
#include<string.h>
using namespace std;
class Employee
{
private:
char
name[30]
;
float
salary;
public:
void
setName(
char *n)
      {
strcpy(name,n);
      }
      void setSalary(float s)
      {
salary=s;
      }
      char * getName()
```

```
        {
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("Tara 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("Abhigya Thapa");
   s.setSalary(40000);
   s.setGroup_id(5);                cout<<"Name=
"<<s.getName()<<endl;            cout<<"Salary=
"<<s.getSalary()<<endl;       cout<<"Group  ID=
"<<s.getGroup_id()<<endl; }
```

5. Hybrid inheritance

In hybrid inheritance, it can be the combination of single and multiple inheritance or any other combination.



One form can be as below

```
Class                                                    1
{                                                        2
  //member of                                          10
Class A : public/private/protected  {                  12


      }
            // 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

Q. 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
{   private:    float
tot; public:       void
showtotal()
```

```
        {
            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();
    }
```

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.

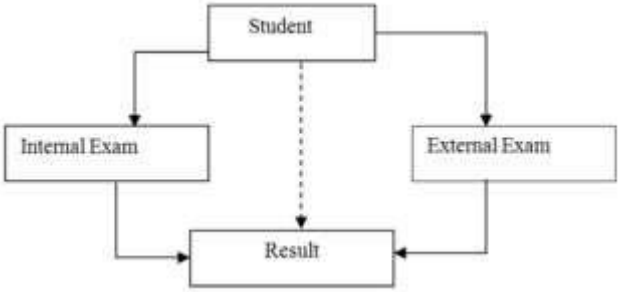
```
#include<iostream>
#include<conio.h>
#include<string.h> using
namespace std;  class
clock
{  protected:
        char model_no[10];
float price;    char
manufacturer[50];    public:
        void getclockdata()
        {
                cout<<"Enter clock
manufacturer:"<<endl;                cin>>manufacturer;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

Virtual Base Class

When two or more class is derived from a common base class, we can prevent multiple copies of the base class being present in an object derived from those objects by declaring the base class as virtual when it is being inherited. Such a base class is known as virtual base class. This can be achieved by preceding the base class' name with the word virtual.

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 A 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(); //ambiguous because multipath exist to reach setroll() from derived class
//so must use virtual base class to overcome this
res.setdatatheory();
    res.setdatapractical();
res.showroll();
res.showmarkstheory();
res.showmarkspractical();   res.showtotal();
}
```

Constructor and Destructor in Inheritance

- Constructor is used to initialize variables and allocation of memory of object
- Destructor is used to destroy objects
- Compiler automatically calls constructer of base class and derived class automatically when derived class object is created.
- If we declare derived class object in inheritance constructor of base class is executed first and then constructor of derived class

- If derived class object goes out of scope or deleted by programmer the derived class destructor is executed first and then base class destructor **Default Constructor (No arguments) in inheritance**
- In this case it is not compulsory to have derived class constructor if base class have a constructor

Example Default Constructor in

```
inheritance class A { //Base Class public:
A() {                                   //Base Class A Constructor
cout<<"Constructor Class A"<<endl;
}
                                        //Base Class A Destructor
~A()
 {
cout<<"Destructor Class A"<<endl;
}
};
class B:public A          //Derived class
{  public:
B()                                     //Derived Class B Constructor
{ cout<<"Constructor Class
B"<<endl;
}
~B() {                                  //Derived Class B Destructor
cout<<"Destructor Class B"<<endl;
} }; int
main()
{
B obj;                          //Derived class object obj
}
```

In Above Program Class A is base class with one constructor and destructor, Class B is derived from class A having a constructor and destructor. In main() Object of derived class i.e. class B is declared. When class B's object is declared constructor of base class is executed followed by derived class constructor. At end of program destructor of derived class is executed first followed by base class destructor.

Output:

Constructor Class A

Constructor Class B

Destructor Class B

Destructor Class A

Parameterized constructor (with arguments) in inheritance

- In parameterized constructor it is compulsory to have derived constructor if there is base class constructor.
- Derived class constructor is used to pass arguments to the base class.
- If derived class constructor is not available, it is not possible to pass arguments from derived class object to base class constructor

Example of parameterized constructor passing different argument for base class and derived class class A //Base class

```
{
protected:
int a;
A(int x) //Base class constructor with one argument(x)
{ a=x; cout<<"Constructor : Class A : value :
"<<a<<endl;
} }; class B:public A //Derived class B from base
class A
{
protected:
int b;
public:
//derived class constructor passing arguments(y, z)
//y is used in derived class and z is passed base class A
B(int y,int z):A(z)
{ b=y; cout<<"Constructor : Class B value :
"<<b<<endl;
} }; int
main()
{
//derived class object passing arguments to derived & base class
```

```
B obj(5,3);
}
```

Order of execution of Constructor

The base class constructor is executed first and then the constructor in the derived class is executed. In case of multiple inheritance, the base class constructors are executed in the order in which they appear in the definition of the derived class. Similarly, in a multilevel inheritance, the constructors will be executed in the order of inheritance.

Order of execution of Destructor

The derived class destructor is executed first and then the destructor in the base class is executed. In case of multiple inheritances, the derived class destructors are executed in the order in which they appear in the definition of the derived class.

Principal of Substitutability

Substitutability is a principle in object-oriented programming stating that, in a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e. an object of type T may be substituted with any object of a subtype S) without altering any of the desirable properties of the program.

Subtype

The term "subtype" is used to describe the relationship between different types that follow the principle of "substitution". If we consider two types (classes or interfaces) A and B, type B is called as a subtype of A, if the following two conditions are satisfied:

a. The instance (object) of type B can be legally assigned to the variable of type A.
b. The instance (object) of type B can be used by the variable of type A without any observable change in its behavior.

Subclass Vs Subtype

- ✓ To say that A is a subclass of B, declares that A is formed using inheritance.
- ✓ To say that A is a subtype of B, declares that A preserves the meaning of all the operations in B.

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. 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. The complex object is sometimes called the whole, or the parent. The simpler object is often called the part, child, or component. To qualify as a **composition**, an object and a part must have the following relationship:

- The part (member) is part of the object (class)
- The part (member) can only belong to one object (class) at a time
- The part (member) has its existence managed by the object (class)
- The part (member) does not know about the existence of the object (class)

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.

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".

IS-A and HAS-A Relation

One of the advantages of an Object-Oriented programming language is code reuse. There are two ways we can do code reuse either by the implementation of inheritance (IS-A relationship), or object composition (HAS-A relationship).

IS-A Rule or Relationship:

In object-oriented programming, the concept of IS-A is a totally based on inheritance. It is just like saying "A is a B type of thing".

For example, Apple is a Fruit, Car is a Vehicle etc. Inheritance is uni-directional. For example, House is a Building. But Building is not a House. **HAS-A Rule (Container Class/Containership/Composition)** HAS-A rule means division into parts.

Consider the term "A car has an engine", "A bus has an engine". We cannot use engine the super class properties to all child classes because the engine has different for all transportation means. The engine is independent properties that cannot be directly called to the car, bus classes. We can include the class engine or class brake in class car not derive them

It is also called container class or containership because object of one class will be used inside other class which means member of one class is available in other class.

Example for Container Class/composition/has-a rule

```
class A
{ int x;
    public
    :
    A()
    { x=1;
    } void
    dispaly1() {
    cout<<x;
    } };
class
B
{ int y;
    A a; //Object of Class A declared in class B
    public:
    B()     //Constructor Class B
    { y=9;
    } void
    display2()
    {
            a.dispaly1 (); //Calling member of Class A from member of B
        cout<<y;
    }
    }; int
main()
{
    B b; //object of class B b.display2();
}
```

In Above program there are 2 class A & B, Class A have a data member integer x and one default constructor and one member function display1 (), Class B have 2 data member one is integer y and

another is object (a) of class A and one constructor and one member function i.e. display2() and inside diplay2 we call member function of class A i.e.display1().

Here class A's object is declared inside class B, so class B contains members of class A or class B HAS A object of class A (Class B is composite class).

Composition Vs Inheritance

Composition	Inheritance
Composition indicates the operation of an existing structure	Inheritance is a super set of existing structure
Cannot reuse code directly but provide greater functionality	Inheritance can be directly reused the code and function provided by parent class
Code become shorter than inheritance	Code become longer than composition
Very easy to re-implement the behaviors and functions	Difficult to re-implement the behaviors
Example:	Example: