CHAPTER 3

Inheritance

Er. Ganga Gautam

OUTLINES:

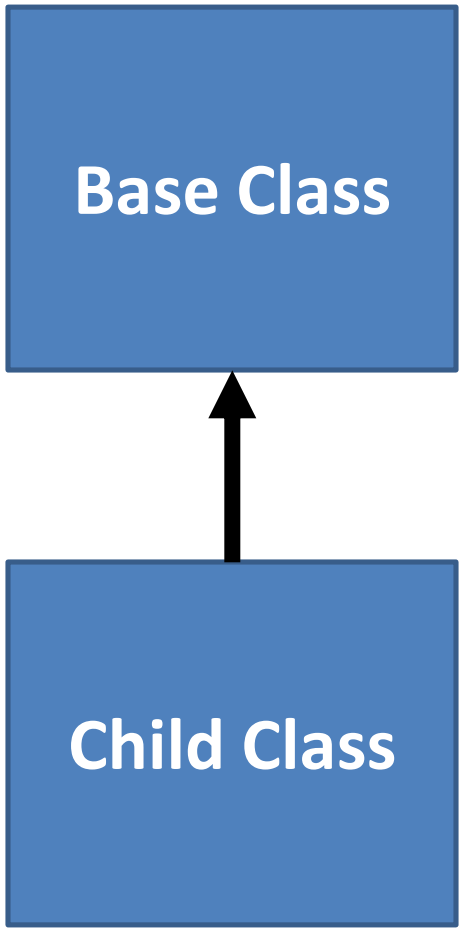
1. **Introduction to Inheritance**
2. **Inheritance Relationship Diagram**
3. **Inheritance Mode: Public, Private & Protected**
4. **Types of Inheritance: Single, Multilevel, Hierarchical, Multiple and Hybrid**
5. **Ambiguity Resolution**
6. **Multipath Inheritance and Virtual Base Class**
7. **Constructor and Destructor in Derived Class**
8. **Subclass, Subtype and Principle of Substitutability**
9. **Composition and its Implementation**
10. **Composition Relationship Diagram**
11. **Software Reusability**

Inheritance

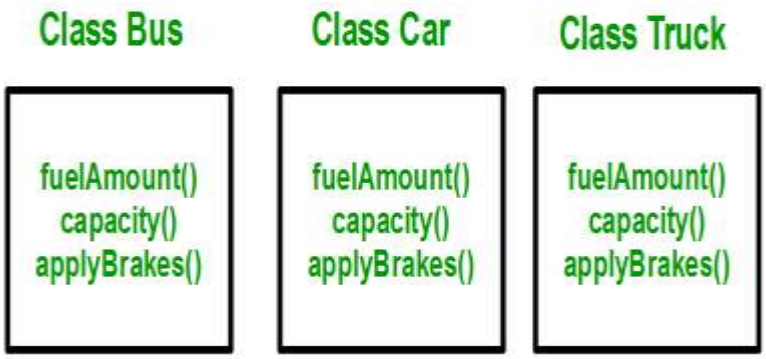
- Inheritance refers to the sharing and transforming the properties of one class to another class in a secured manner.
- So, it is the process by which new classes called child class (or sub class or derived class) are created from existing class called Parent class (or base class).
- The derived class has some or all the features of the base class.

Base class and Derived class

- The existing class from which another class is derived is known as the base class,
- and the newly created class is called its derived class.
- The derived class can inherit all or some properties Base class.
- Base class aka super/parent class
- Derived class aka child/sub class



Why need of inheritance?

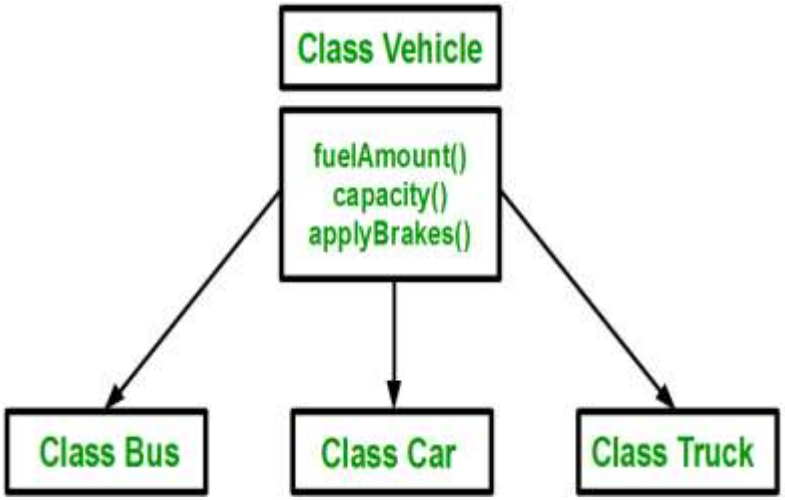


Solution:

- Create a Base class
- Encapsulate all the functions in that class.
- Create three child class

Problems:

- Three separate classes
- Functions() repeated for all classes
- Longer code



Implementing inheritance in C++

- Syntax:

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

- Where:
  - **subclass_name** is the name of the sub class,
  - **visibility_mode** may be either Private or Public or Protected. **base_class_name** is the parent class
  - The **colon (:)** indicates that the sub-class is derived from the base-class.
- Eg:

```
Class Box: public Shape
{
     …..
}
```

Sample program

```cpp
// C++ program to demonstrate implementation of Inheritance
#include<iostream>
using namespace std;

class Parent    //Base class
{
    public:
      int id_p;
};

class Child : public Parent     // Sub class inheriting from Base Class
{
    public:
      int id_c;
};

main()
   {

        Child ob;
        ob.id_c = 7;    //object of child is able to access parent class
        ob.id_p = 91;
        cout << "Child id is " <<  ob.id_c << endl;
        cout << "Parent id is " <<  ob.id_p<< endl;
   }
```

```
Child id is 7
Parent id is 91
```

4.3 Visibility mode in inheritance

1. Private mode

- When a base class is inherited in private mode, **protected and public members of base class become private members** for the derived class.
- And they cannot be accessed outside the derived class.
- The private members of base class are not inherited.

2. Protected mode

- When a base class is inherited in protected mode**,** **protected and public members of base class become protected members** for the derived class.
- The private members of base class are not inherited.

3. Public mode

- When a base class is inherited in public mode, **public members of base class remains public and protected members also remains protected members for the derived class**
- The private members of base class are not inherited.

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

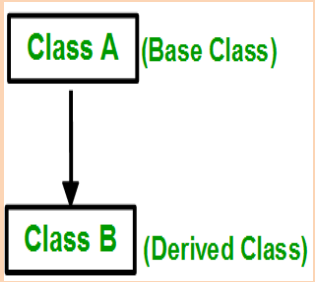
4.4 Making private members inheritable

```
class Test
{
    private:          //optional
        ..... //visible to only the member function within the class
        .....
    protected:
        ..... //visible to only the member function within the class and its derived class
        .....
    public:
        ..... //visible to all the member function within the program
        .....
};
```

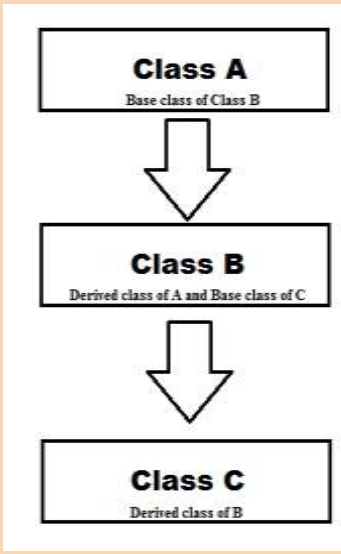
- No any other option.
- Just need to make private members as public members

4.5 Types of inheritance

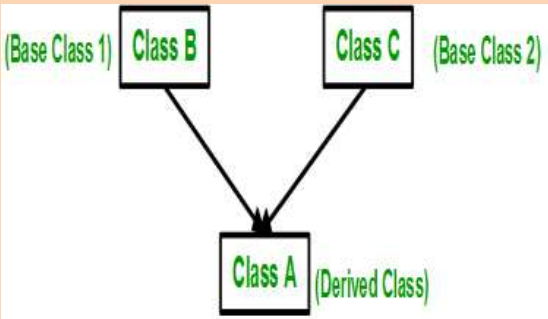
i) Single Inheritance



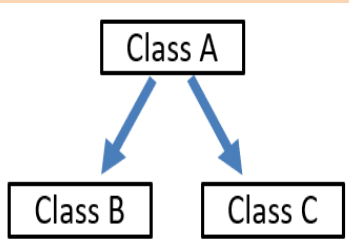
ii) Multilevel Inheritance



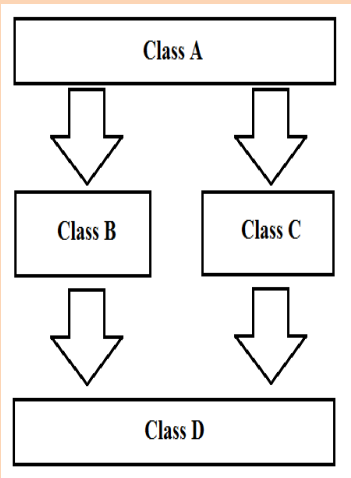
iii) Multiple Inheritance



iv) Hierarchical Inheritance



v) Hybrid Inheritance



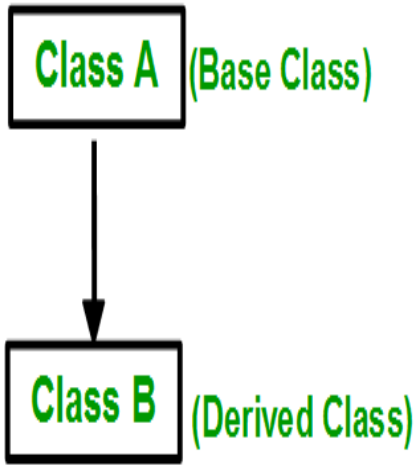
i) Single Inheritance

- 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.
- It is the foundation for all types of inheritance.
- General form:

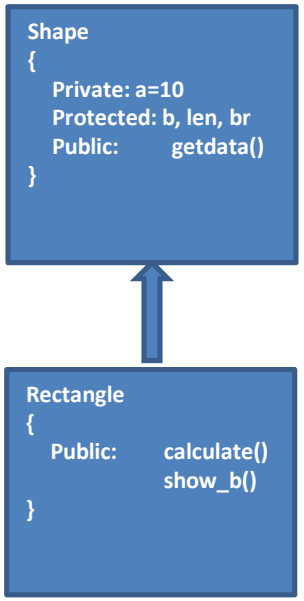
```
class  subclass_name :  access_mode  base_class
{
    //body of subclass
};
```

- Eg:

```
Class Box: public Shape
{
  ....
};
```



i) Single Inheritance : Sample program




```
1  #include<iostream>
2  using namespace std;
3
4  class Shape
5  {
6      int a=10;       //not inherited
7      protected:
8          int b=50;           //inherited by derived class
9          int len,br;         //inherited by derived class
10     public:
11         void getdata(int x, int y)  //inherited by derived class
12         {
13             len=x;
14             br=y;
15         }
16 };
17
```

```
18 class Rectangle:public Shape
19 {
20     public:
21         int calculate()
22         {
23             return len*br;
24         }
25
26         /*
27         void show_a()
28         {
29             cout<<a;
30         }
31         */
32
33         void show_b()
34         {
35             cout<<"Value of b="<<b<<endl;
36         }
37 };
```

```
38
39 main()
40 {
41     Rectangle ob;
42     ob.getdata(5,2);
43     cout<<"Area of rectangle is:"<<ob.calculate()<<endl;
44     //ob.show_a();  //not inherited
45     ob.show_b();
46 }
```

```
Area of rectangle is:10
Value of b=50

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

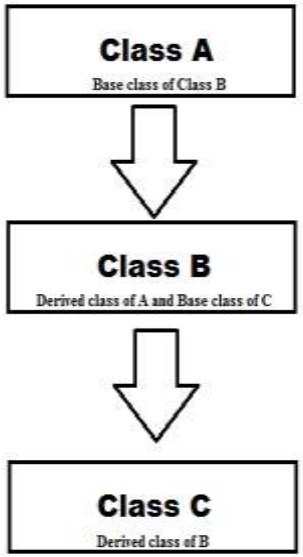
ii) Multilevel Inheritance

- In this type of inheritance, a derived class is created from another derived class.
- **General Form**:

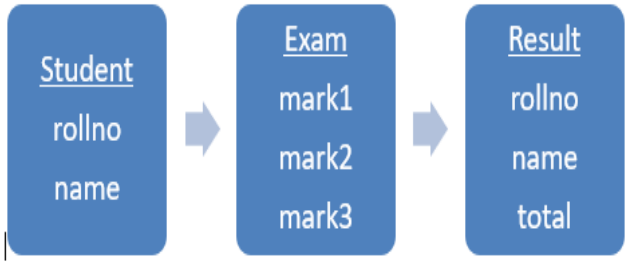
```
class B : access_mode A
{
  //body of subclass B
};

class C: access_mode B
{
  //body of subclass C
};
```

Eg:

```
Class Box: public Shape
{
    .....
};

Class Triangle: public Box
{
   .......
};
```



ii) Multilevel Inheritance : Sample program




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

class Student
{
    protected:
        char name[30];      //inherited by derived class
        int roll;       //inherited by derived class
    public:
        void getdata()  //inherited by derived class
        {
            cout<<"Enter rollno and name of student:"<<endl;
            cin>>roll>>name;
        }
};
```

```cpp
class Exam:public Student
{
    protected:
        float m1,m2,m3;
public:
        void getmark()
        {
            cout<<"Enter mark1, mark2 and mark3"<<endl;
            cin>>m1>>m2>>m3;
        }
};
```

```cpp
class Result:public Exam
{
    float total;
    public:
        void display()
        {
            total=m1+m2+m3;
            cout<<"Roll no:" <<roll<<endl;
            cout<<"Name:" <<name<<endl;
            cout<<"Total mark:" <<total<<endl;
        }
};
```

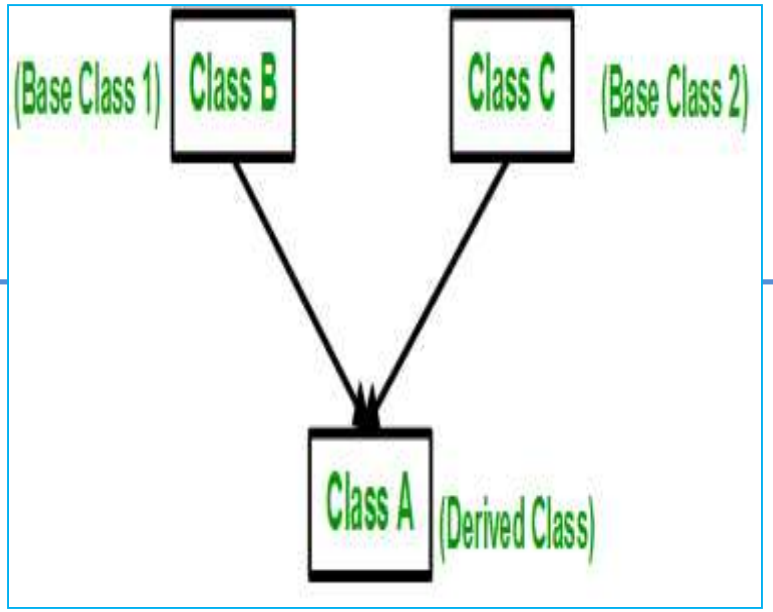
```cpp
main()
{
    Result ob;
    ob.getdata();
    ob.getmark();
    ob.display();
}
```

```
Enter rollno and name of student:
41
shiva
Enter mark1, mark2 and mark3
90
100
80
Roll no:41
Name:shiva
Total mark:270
```

iil) 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 subclassName : accessMode baseclass1, accessMode
 baseclass2, ....
{
  //body of subclass
};
```



Here, the number of base classes will be separated by a comma (,) and access mode for every base class must be specified.

iii) Multiple Inheritance : Sample Program

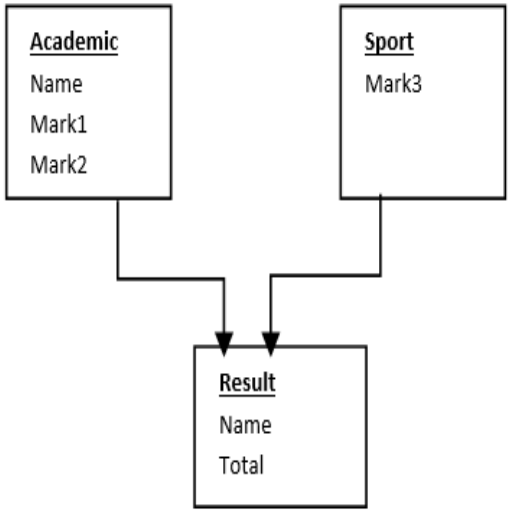
```
#include<iostream>
using namespace std;
class Academic      //base class 1
{
    protected:
        char name[30];      //inherited by derived class
        int m1,m2;      //inherited by derived class
    public:
        void getdata1() //inherited by derived class
        {
            cout<<"Enter name, mark1 and mark2"<<endl;
            cin>>name>>m1>>m2;
        }
};
```

```
class Sport //base class 2
{
    protected:
        int m3; //inherited by derived class
    public:
        void getdata2() //inherited by derived class
        {
            cout<<"Enter sports mark"<<endl;
            cin>>m3;
        }
};
```

```
class Result:public Academic, public Sport      //derived class
{
    int total;
    public:
        void showdata()
        {
            total=m1+m2+m3;
            cout<<"Name:" <<name<<endl;
            cout<<"Total mark:" <<total<<endl;
        }
};
```

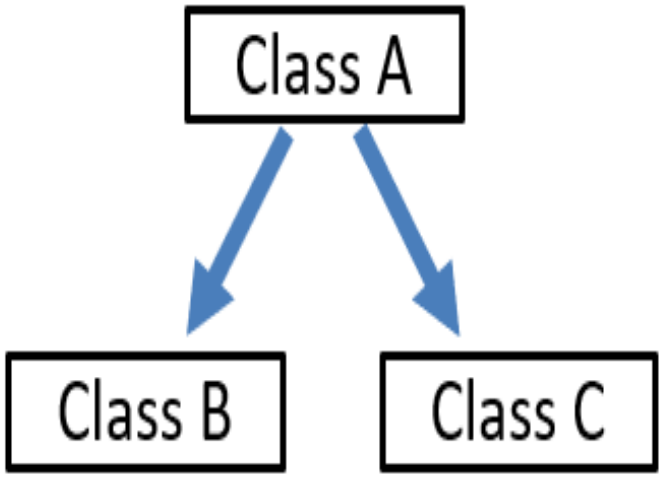
```
main()
{
    Result ob;
    ob.getdata1();
    ob.getdata2();
    ob.showdata();
}
```

```
Enter name, mark1 and mark2
shiva
100
95
Enter sports mark
100
Name:shiva
Total mark:295
```

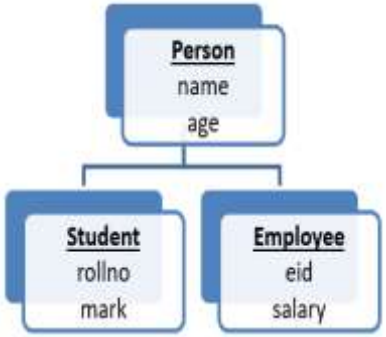


iv) Hierarchical Inheritance

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



iv) Hierarchical Inheritance : Sample program




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

class Person
{
    protected:
        char name[30];      //inherited by derived class
        int age;        //inherited by derived class
    public:
        void getdata1() //inherited by derived class
        {
            cout<<"Enter name and age"<<endl;
            cin>>name>>age;
        }
};
```

```
class Student:public Person
{
    protected:
        int roll,mark;
    public:
        void getdata2()
        {
            cout<<"Enter rollno and mark"<<endl;
            cin>>roll>>mark;
        }

        void show_std_data()
        {
            cout<<"Name="<<name<<endl;
            cout<<"Age="<<age<<endl;
            cout<<"Rollno="<<roll<<endl;
            cout<<"Mark="<<mark<<endl;
        }
};
```

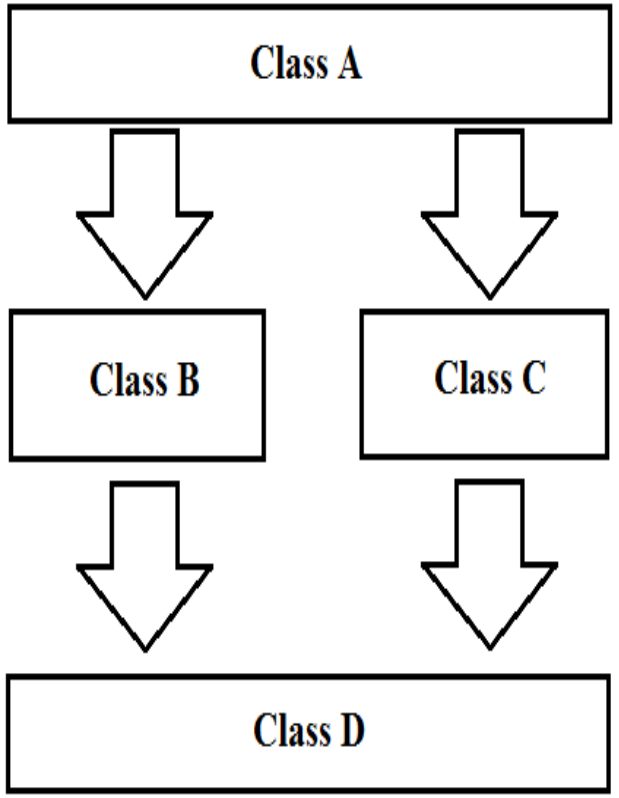
```
class Employee:public Person
{
    protected:
        int eid,salary;
    public:
        void getdata3()
        {
            cout<<"Enter employee-id and salary"<<endl;
            cin>>eid>>salary;
        }

        void show_emp_data()
        {
            cout<<"Name="<<name<<endl;
            cout<<"Age="<<age<<endl;
            cout<<"Employee ID="<<eid<<endl;
            cout<<"Salary="<<salary<<endl;
        }
};
```

```
main()
{
    Student s;
    s.getdata1();
    s.getdata2();
    cout<<"For Derived class Student:"<<endl;
    s.show_std_data();
    Employee e;
    e.getdata1();
    e.getdata3();
    cout<<"For Derived class Employee:"<<endl;
    e.show_emp_data();
}
```

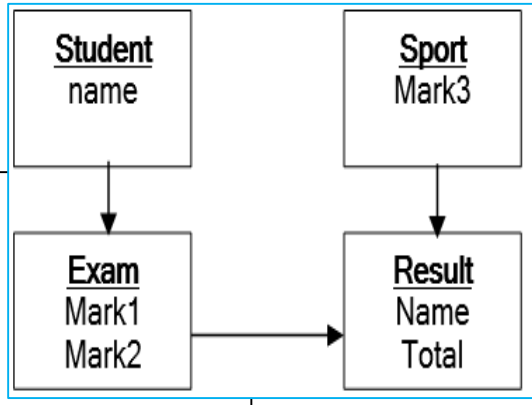
```
Enter name and age
Ashok
26
Enter rollno and mark
20
98
For Derived class Student:
Name=Ashok
Age=26
Rollno=20
Mark=98
Enter name and age
Bishal
30
Enter employee-id and salary
1001
50000
For Derived class Employee:
Name=Bishal
Age=30
Employee ID=1001
Salary=50000
```

v) Hybrid Inheritance

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



v) Hybrid Inheritance : Sample program




```
#include<iostream>
using namespace std;
class Student
{
   protected:
      char name[30];    //inherited by derived class
      int roll;       //inherited by derived class
   public:
      void getdata1()   //inherited by derived class
      {
         cout<<"Enter name and rollno"<<endl;
         cin>>name>>roll;
      }
};
```

```
class Exam:public Student
{
   protected:
      int m1,m2;
   public:
      void getdata2()
      {
         cout<<"Enter mark1 and mark2"<<endl;
         cin>>m1>>m2;
      }
};
```

```
class Sport
{
   protected:
      int m3;
   public:
      void getdata3()
      {
         cout<<"Enter sports mark"<<endl;
         cin>>m3;
      }
};
```

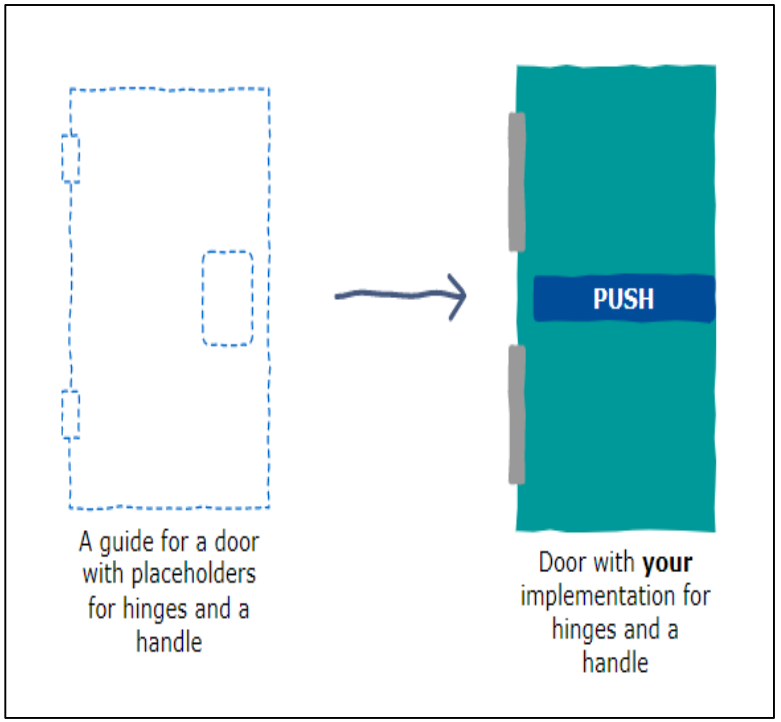
```
class Result: public Exam, public Sport
{
   int tot;
   public:
      void display()
      {
         tot=m1+m2+m3;
         cout<<"Name="<<name<<endl;
         cout<<"Rollno="<<roll<<endl;
         cout<<"Total Mark="<<tot<<endl;
      }
};
```

```
main()
{
   Result r;
   r.getdata1();
   r.getdata2();
   r.getdata3();
   r.display();
}
```

```
Enter name and rollno
Manish
25
Enter mark1 and mark2
100
100
Enter sports mark
100
Name=Manish
Rollno=25
Total Mark=300
```

4.6 Abstract Base Class

- If we were to build a simple door, we would need a guide to follow.
- This guide may require us to include a handle and a hinge.
- The advantage of having this guide is that it helps us to not forget to include these two features in our implementation.
- This is how we should think about an Abstract Base Class (ABC).



4.6 Abstract Base Class

```
This is display() method od derived class
This is show() method of ABC
```

- An Abstract Base Class (ABC) is a class that is designed to be specifically used as a base class. It is not used to create objects.
- Allows base class to provide only an interface for the derived classes.
- Prevents anyone from creating an instance of this class. No object of an ABC can be created.
- A class is made abstract by defining at least one virtual function pure. A pure virtual function is one with an initialize of =0 on its declaration as:

```
virtual returntype functionname()=0;
```

- **General form Abstract class is;**

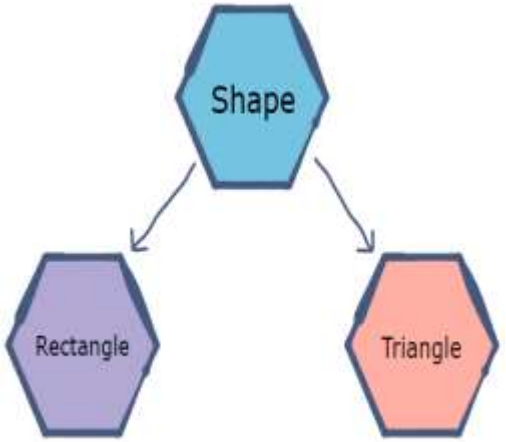
```
class Baseclassname
{
 public:
                    //pure virtual function
};
```

```
#include<iostream>
using namespace std;
class Vehicle     //Abstract Base Class
{
   public:
      virtual void display()=0;  //pure virtual function
      void show()
      {
         cout<<"This is show() method of ABC"<<endl;
      }
};

class Bike:public Vehicle
{
   public:
      void display()
      {
         cout<<"This is display() method od derived class"<<endl;
      }
};

main()
{
   //Vehicle v;   // this shows error message
   Bike ob;
   ob.display();  //invokes display() of derived class
   ob.show();     //invokes show() of base class
}
```

4.6 Abstract Base Class : Sample program

```
#include <iostream>
using namespace std;
class Shape
{
   protected:
      int h,w;
   public:
      virtual int Area() = 0; // Pure virtual function
                              //is declared as follows.
      void getdata()
      {
         cout<<"Enter height and width"<<endl;
         cin>>w>>h;
      }
};

class Rectangle: public Shape {
   public:
      int Area() //Rectangle implements Area()
      {
         return (w * h);
      }
};
```

```
class Triangle: public Shape {
    public:
       int Area()  //Triangle implements Area()
       {
          return (w * h)/2;
       }
};
```




```
int main() {
    Rectangle R;
    Triangle T;
    R.getdata();
    cout << "The area of the rectangle is: " << R.Area() << endl;
    T.getdata();
    cout << "The area of the triangle is: " << T.Area() << endl;
}
```

```
Enter height and width
10
5
The area of the rectangle is: 50
Enter height and width
10
20
The area of the triangle is: 100

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

4.7 Ambiguity in inheritance

- The ambiguity is the situation in which the main() function (or the calling function) cannot give decision to call the function.
- If the base class and derived class have the same function name, then this creates ambiguity.
- The compiler cannot decide which function to call.
- There are few cases where ambiguity occurs:

  i) Ambiguity due to function overriding

  ii) Ambiguity in multiple inheritance

  iii) Ambiguity in multipath inheritance

i) Ambiguity due to function overriding

- This type of ambiguity occurs when there are same function names in the base as well as in its derived classes.
- The compiler cannot decide which function to call.
- So, ambiguity occurs due to the function overriding.

i) Ambiguity due to function overriding: Sample Program

```
#include<iostream>
using namespace std;
class A
{
   public:
      int a;
      void getdata()
      {
         cout<<"Enter value of a:"<<endl;
         cin>>a;
      }
};
class B:public A
{
   public:
      int b,sm;
      void getdata() // function overridden
      {
         cout<<"Enter value of b:"<<endl;
         cin>>b;
      }
      void display()
      {
         sm=a+b;
         cout<<"The sum is:"<<sm<<endl;
      }
};
```

```
main()
{
   B ob;
   ob.getdata();  //invokes function in derived class B
   ob.getdata();  //invokes function in derived class B
   ob.display();
}
```

```
Enter value of b:
5
Enter value of b:
5
The sum is:1182051749

--------------------------------
Process exited after 2.184 seconds with return value 0
Press any key to continue . . .
```

Here, the compiler is confused which function *getdata()* to call.
To resolve this ambiguity, we can use scope resolution operator (::) in the main function while calling them.

i) Ambiguity due to function overriding: Resolved

```
1  #include<iostream>
2  using namespace std;
3  class A
4  {
5     public:
6        int a;
7        void getdata()
8        {
9           cout<<"Enter value of a:"<<endl;
10          cin>>a;
11       }
12 };
13 class B:public A
14 {
15    public:
16       int b,sm;
17       void getdata() // function overridden
18       {
19          cout<<"Enter value of b:"<<endl;
20          cin>>b;
21       }
22       void display()
23       {
24          sm=a+b;
25          cout<<"The sum is:"<<sm<<endl;
26       }
27 };
```

```
28 main()
29 {
30    B ob;
31    ob.A::getdata();  //invokes function in derived class A
32    ob.B::getdata();  //invokes function in derived class B
33    ob.display();
34 }
```

```
Enter value of a:
5
Enter value of b:
10
The sum is:15
```

Thus, ambiguity is resolved

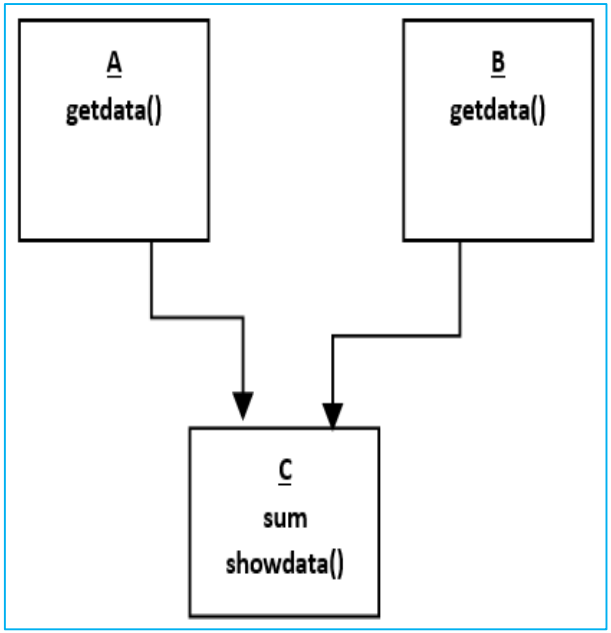
ii) Ambiguity in Multiple inheritance

- In multiple inheritance, two base class may have functions with same name.
- If the object of the derived class needs to access one of them, the compiler is confused which one to invoke.
- This is ambiguity in multiple inheritance.
- This can be resolved by using scope resolution operator while invoking.
- The syntax is:

```
Objectname.classname::functionname();
```

- Eg:

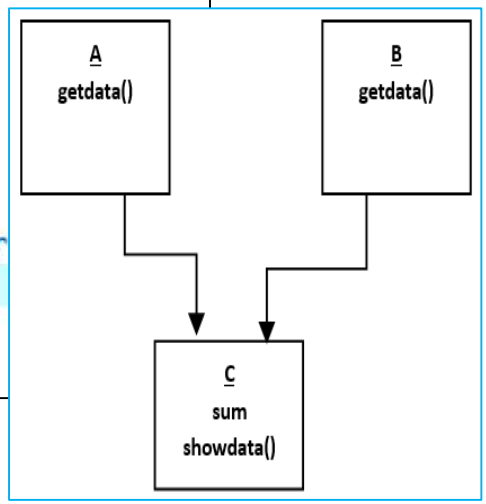
```
ob.A::getdata();
```



ii) Ambiguity in Multiple inheritance: Sample Program

```
1  #include<iostream>
2  using namespace std;
3  class A
4  {
5     public:
6        int a;
7        void getdata()
8        {
9           cout<<"Enter first no:"<<endl;
10          cin>>a;
11       }
12 };
13
14 class B
15 {
16    public:
17       int b;
18       void getdata()
19       {
20          cout<<"Enter second no:"<<endl;
21          cin>>b;
22       }
23 };
24
```

```
25 class C:public A,public B
26 {
27    public:
28       int sm;
29       void showdata()
30       {
31          sm=a+b;
32          cout<<"The sum is:"<<sm<<endl;
33       }
34 };
35
36 main()
37 {
38    C ob;
39    ob.getdata();  //gives error
40    ob.getdata();
41    ob.showdata();
42 }
```

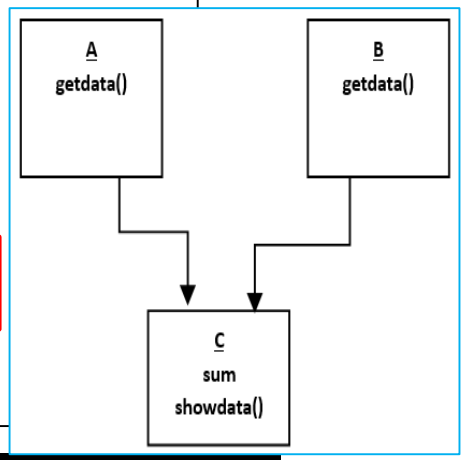


- To resolve this, we use scope resolution operator while invoking.

ii) Ambiguity in Multiple inheritance: Resolved

```
1  #include<iostream>
2  using namespace std;
3  class A
4  {
5     public:
6        int a;
7        void getdata()
8        {
9           cout<<"Enter first no:"<<endl;
10          cin>>a;
11       }
12 };
13
14 class B
15 {
16    public:
17       int b;
18       void getdata()
19       {
20          cout<<"Enter second no:"<<endl;
21          cin>>b;
22       }
23 };
24
```

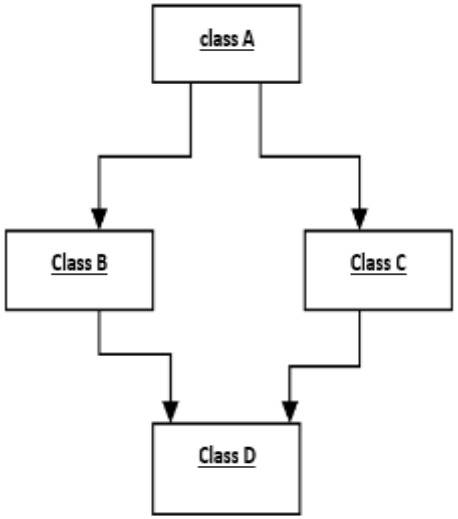
```
25 class C:public A,public B
26 {
27    public:
28       int sm;
29       void showdata()
30       {
31          sm=a+b;
32          cout<<"The sum is:"<<sm<<endl;
33       }
34 };
35
36 main()
37 {
38    C ob;
39    ob.A::getdata();
40    ob.B::getdata();
41    ob.showdata();
42 }
```



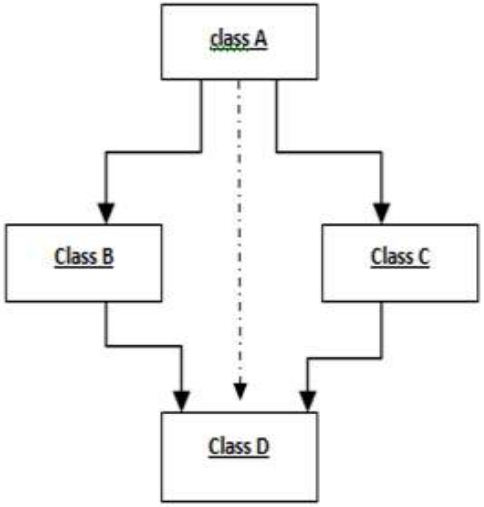

```
Enter first no:
10
Enter second no:
20
The sum is:30
```

iii) Ambiguity in Multipath inheritance

- Such ambiguity occurs when a derived class have two base classes and these two base classes again have one common super-base class.
- In such case, the compiler is confused which path to follow while inheriting the state and behavior of the super-base class.
- So, grand-child class inherits the properties of class grand-parent class for twice.
- This case can be resolved by using the concept of "Virtual Base class".



After resolving,



General form is:

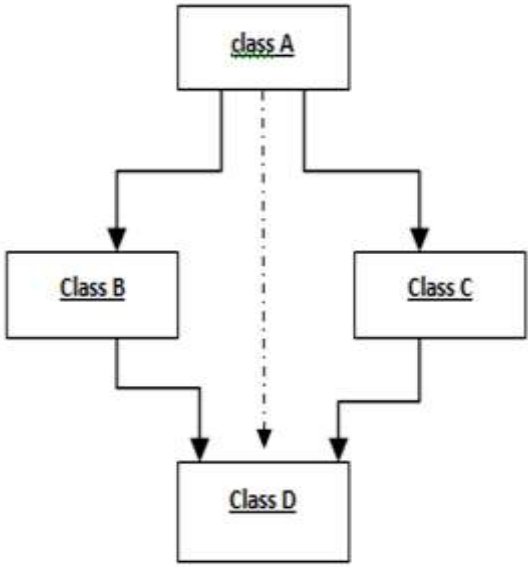
```
class A
{
   ........
   ........
};

class B: virtual public A
{
   ........
   ........
};

class C: virtual public A
{
   ........
   ........
};
class D: public B, public C
{
   ........
   ........
};

main()
{
   .......
   .......
}
```

iii) Ambiguity in Multipath inheritance: Resolved

```
#include<iostream>
using namespace std;
class A
{
    public:
        int a;
};

class B: virtual public A
{
    public:
        int b;
};

class C: virtual public A
{
    public:
        int c;
};
```

```
class D: public B, public C
{
    public:
        int d;
        void sum()
        {
            d=a+b+c;
        }
        void display()
        {
            cout<<"Sum of a,b,c is:"<<d<<endl;
        }
};

main()
{
    D ob;
    ob.a=10;
    ob.b=20;
    ob.c=30;
    ob.sum();
    ob.display();
}
```

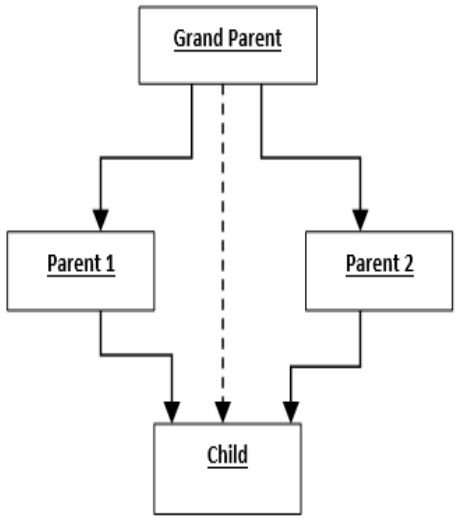
```
Sum of a,b,c is:60

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



4.8 Virtual Base class

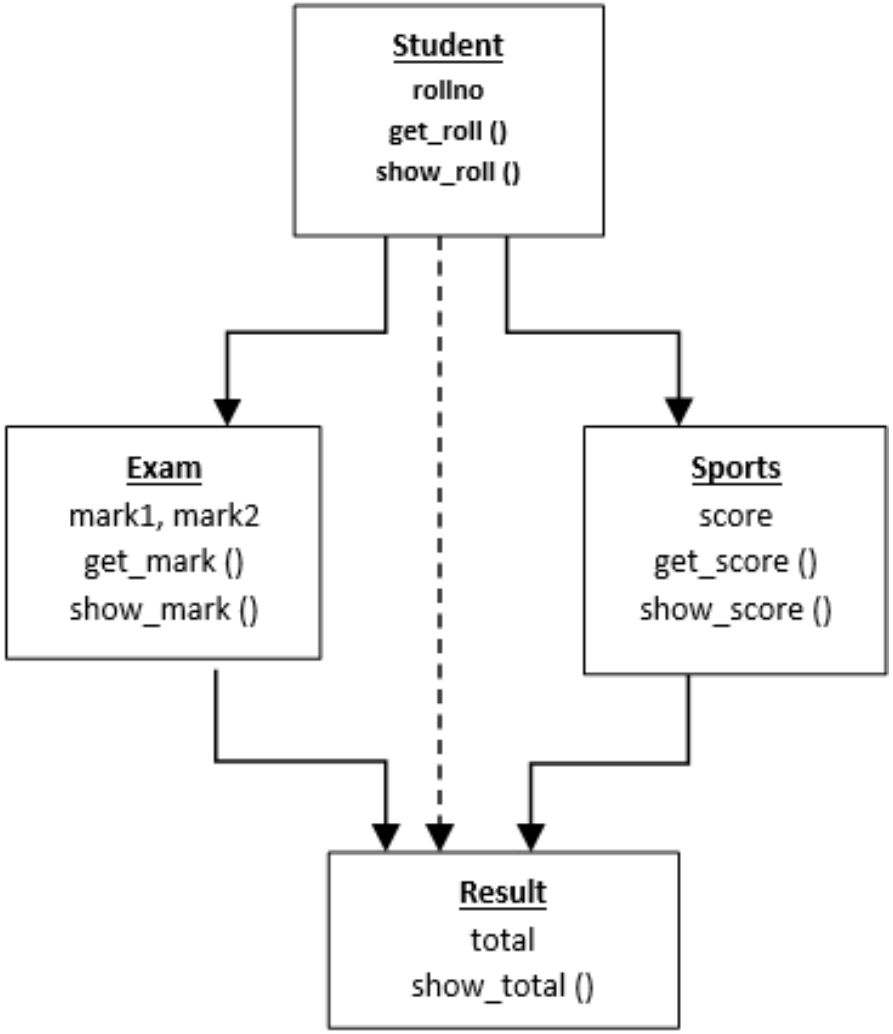
- In the diagram, all the Public and Protected members of Grand-parent are inherited in the child twice:
- first via Parent1 and second via Parent2.
- This means child would have duplicate set of members inherited from Grand-parent.
- This introduces ambiguity(duplication) and should be avoided.
- The duplication of inherited members due to their multipaths can be avoided by making common base class as virtual class.
- Such class is called Virtual Base Class.
- This helps us to inherit directly as shown in the broken line.

```
class Grand_parent
{
   ........
   ........
};

class Parent1: virtual public Grand_parent
{
   ........
   ........
};

class Parent2: virtual public Grand_parent
{
   ........
   ........
};
class Child: public Parent1, public Parent2
{
   ........
   //only one copy of Grand_parent will be inherited
   ........
};

main()
{
   ........
   ........
}
```



4.8 Virtual Base class : Sample Program

WAP to implement the below inheritance and resolve the ambiguity if it occurs.



4.8 Virtual Base class : Sample Program

```
#include<iostream>
using namespace std;
class Student
{
    protected:
       int roll;
    public:
       void get_roll()
       {
          cout<<"Enter rollno:"<<endl;
          cin>>roll;
       }
       void show_roll()
       {
          cout<<"Rollno is:"<<roll<<endl;
       }
};
```

```
class Exam: virtual public Student
{
    protected:
       int m1,m2;
    public:
       void get_mark()
       {
          cout<<"Enter mark1 and mark2:"<<endl;
          cin>>m1>>m2;
       }
       void show_mark()
       {
          cout<<"Mark1 ="<<m1<<endl;
          cout<<"Mark2 ="<<m2<<endl;
       }
};
```

```
class Sports: virtual public Student
{
    protected:
       int sc;
    public:
       void get_score()
       {
          cout<<"Enter score in sports:"<<endl;
          cin>>sc;
       }
       void show_score()
       {
          cout<<"Score ="<<sc<<endl;
       }
};
```

```
class Result: public Exam, public Sports
{
    public:
       int tot;
       void show_total()
       {
          tot=m1+m2+sc;
          cout<<"Total= "<<tot<<endl;
       }
};
```

```
main()
{
    Result r;
    r.get_roll();
    r.get_mark();
    r.get_score();
    r.show_roll();
    r.show_mark();
    r.show_score();
    r.show_total();
}
```

```
Enter rollno:
22
Enter mark1 and mark2:
100
100
Enter score in sports:
100
Rollno is:22
Mark1 =100
Mark2 =100
Score =100
Total= 300

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

4.9. Merits and Demerits of Inheritance

Merits	Demerits
1) Reusability	1) Execution speed
2) Code sharing	2) Program size
3) Consistency of interface	3) Program complexity
4) Construction of software components	
5) Rapid prototyping	
6) Information hiding	

4.10. Constructors and Destructors in Derived class

- If the base class contains a zero-argumented constructor or no constructor, then the derived class does not require a constructor,
- But if the base class contains a parameterized constructor, then it is essential for the derived class to have a constructor.

How invoked?

- Firstly, the constructor in the Base class is executed and then that of the Derived class.
  - The derived class constructor passes arguments to the base class constructor.
- Destructors are executed in reverse order of constructor execution.
  - The destructor in the derived class is executed first and then that of the base class.

4.10. Constructors and Destructors in Derived class

- Syntax for Derived class constructor

```
Constructor(parameters): Base class(parameters)
```

- Eg:

```
Rectangle( int j): Polygon(j)
```

Sample program 1:
Case 1 (derived class constructor not required)

WAP to show that derived class constructor is not required if the base class has no arguments in its constructor.

```
1  #include<iostream>
2  using namespace std;
3  class Shape
4  {
5     public:
6        int a,b;
7        Shape()     //base class constructor
8        {
9           a=10;
10          b=5;
11       }
12 };
13
14 class Rectangle:public Shape
15 {
16    //Derived class constructor not required
17    public:
18       void Showarea()
19       {
20       cout<<"Area:"<<a*b<<endl;
21       }
22 };
```

```
24 main()
25 {
26    Rectangle ob;
27    ob.Showarea();
28 }
```

```
Area:50

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

Sample program 2:
Case 2 (derived class constructor required)

WAP to show that derived class constructor is required if the base class has arguments in its constructor.

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

class Polygon
{
   int x;
   public:
      Polygon(int i) //base constructor
      {
         x=i;
         cout<<"Base class constructor invoked"<<endl;
         cout<<"Value of x="<<x<<endl;
      }
   ~Polygon()
   {
      cout<<"Destructor of Base class"<<endl;
   }
};
```

```
class Rectangle:public Polygon
{
   int y;
   public:
      Rectangle(int j):Polygon(j)   //derived constructor
      {
         y=j;
         cout<<"Derived class constructor invoked"<<endl;
         cout<<"Value of y="<<y<<endl;
      }

      ~Rectangle()
      {
         cout<<"Destructor of Derived class"<<endl;
      }
};
```

```
main()
{
   Rectangle ob(10);
}
```

```
Base class constructor invoked
Value of x=10
Derived class constructor invoked
Value of y=10
Destructor of Derived class
Destructor of Base class
```

Sample program 3:
Case 2 (derived class constructor required)

WAP to show that derived class constructor is required if the base class has arguments in its constructor.

```
1  #include<iostream>
2  using namespace std;
3  class Shape
4  {
5     public:
6        int a,b;
7        Shape(int x,int y)       //base class constructor
8        {
9           a=x;
10          b=y;
11       }
12 };
```

```
13 class Rectangle:public Shape
14 {
15    public:
16       Rectangle(int x, int y):Shape(x,y)  //Derived class constructor required
17       {
18          a=x;
19          b=y;
20       }
21       void Showarea()
22       {
23       cout<<"Area:"<<a*b<<endl;
24       }
25 };
```

```
26 main()
27 {
28     Rectangle ob(10,5);
29     ob.Showarea();
30 }
```

```
Area:50

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

Sample program 4:
Case 2 (derived class constructor required)

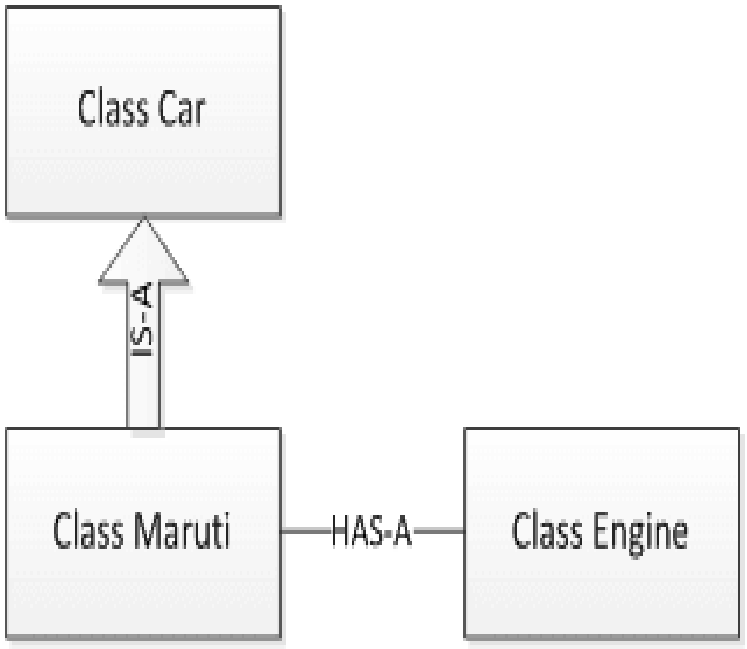
WAP to show that derived class constructor is required if the base class has arguments in its constructor.

```
#include<iostream>
using namespace std;
class Shape
{
   public:
      int l;
      Shape (int k)     //parameterized constructor of parent class.
      {
          l = k;
      }
      void showlength()
      {
          cout<<"Value of l = "<<l<<endl;
      }
};
class Rectangle: public Shape
{
  int b,a;
   public:
      //constructor of child class calling constructor of base class.
      Rectangle(int x, int y):Shape(x)
      {
          b = y;
      }
      void showbreadth()
      {
          cout<<"Value of b = "<<b<<endl;
      }
       void showarea()
       {
          a=l*b;
          cout<<"Area of Rectangle = "<<a<<endl;
       }
};
int main()
{
  Rectangle ob(10,5);
  ob.showlength();
  ob.showbreadth();
  ob.showarea();
}
```

```
Value of l = 10
Value of b = 5
Area of Rectangle = 50

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

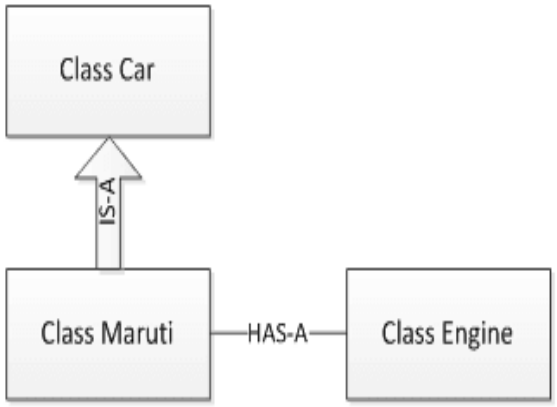
4.11. IS-A rule and HAS-A rule

- Two ways of code reuse:
  - Inheritance (IS-A rule)
    - Maruti is a car.
  - Composition (HAS-A rule)
    - Maruti has engine



IS-A rule

- As shown above, ***Car*** class has a couple of instance variable and few methods. ***Maruti*** is a specific type of Car which inherits ***Car*** class means Maruti IS-A Car.



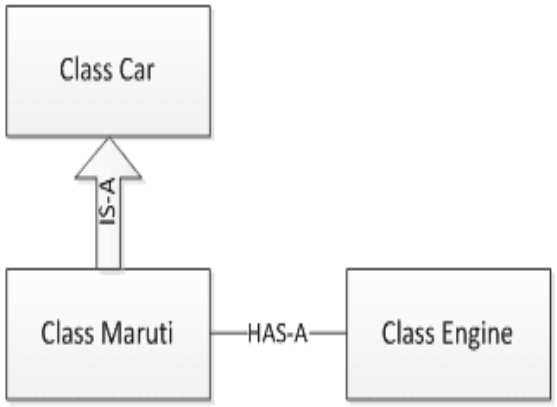

```
#include<iostream>
using namespace std;
class Car
{
   protected:
      char color[20];
      int max;
   public :
      void getInfo()
      {
         cout<<"Enter color and maxSpeed"<<endl;
         cin>>color>>max;
      }

};
```

```
class Maruti:public Car
{
   public:
      void showInfo()
      {
         cout<<"color="<<color<<endl;
         cout<<"MaxSpeed="<<max<<endl;
      }
};
main()
{
   Maruti m;
   m.getInfo();
   m.showInfo();
}
```

```
Enter color and maxSpeed
Blue
150
color=Blue
MaxSpeed=150

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

HAS-A rule

- Maruti class uses Car object's getInfo() method via composition. We can say that Maruti class HAS-A color and maxSpeed.




```cpp
#include<iostream>
using namespace std;
class  Maruti
{
   protected:
      char color[20];
      int max;
   public :
      void getInfo()
      {
         cout<<"Enter color and maxSpeed"<<endl;
         cin>>color>>max;
      }

      void showInfo()
      {
         cout<<"color="<<color<<endl;
         cout<<"MaxSpeed="<<max<<endl;
      }
};
```

```cpp
class Engine
{
   public:
      Maruti m;   //Embeded object
};
main()
{
   Engine e;
   e.m.getInfo();
   e.m.showInfo();
}
```

```
Enter color and maxSpeed
Red
200
color=Red
MaxSpeed=200

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

4.12 Composition

- In composition, we simply create objects of our existing class inside a new class.
- And this is called composition because the new class is composed of objects of existing classes.
- This technique of including user defined object as a part of newly defined object is called composition.

4.12 Composition : Sample Program 1

```
1   #include<iostream>
2   using namespace std;
3   class First
4   {
5      int m1;
6      public:
7         First()
8         {
9            m1=0;
10        }
11
12        void setdata(int x)
13        {
14           m1=x;
15        }
16
17        void calculate()
18        {
19           m1=m1*100;
20           cout<<"Mark1="<<m1<<endl;
21        }
22  };
```

```
23  class Second
24  {
25     int m2;
26     public:
27        First f;     //embedded object
28        Second()
29        {
30           m2=0;
31        }
32
33        void getdata(int y)
34        {
35           m2=y;
36           cout<<"Mark2="<<m2<<endl;
37        }
38  };
```

```
39  main()
40  {
41     Second s;
42     s.getdata(47);
43     s.f.setdata(37);
44     s.f.calculate();
45  }
```

```
Mark2=47
Mark1=90

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

4.12 Composition : Sample Program 2

```
#include<iostream>
using namespace std;
class First
{
   int m1;
   public:
      First()
      {
         m1=0;
      }

      void setdata(int x)
      {
         m1=x;
      }

      void calculate()
      {
         m1=m1*100;
         cout<<"Mark1="<<m1<<endl;
      }
};
```

```
class Second
{
   int m2;
   public:
      First f;     //embedded object
      Second()
      {
         m2=0;
      }

      void getdata1(int y)
      {
         m2=y;
         cout<<"Mark2="<<m2<<endl;
      }
};
```

```
class Third
{
   int m3;
   public:
      Second se;      //embedded object
      Third()
      {
         m3=0;
      }

      void getdata2(int x)
      {
         m3=x;
         cout<<"Mark3="<<m3<<endl;
      }
};
```

```
main()
{
   Third th;
   th.getdata2(10);
   th.se.getdata1(20);
   th.se.f.setdata(30);
   th.se.f.calculate();
}
```

```
Mark3=10
Mark2=20
Mark1=3000

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

4.14 Principle of substitutability

- Substitutability means the feature of a program in which certain things can be substituted in other section or part of a program without changing the effect.
- Principle of Substitutability states that:

  In a computer program, if B is a subtype of A, then objects of type A may be replaced with objects of type B without altering any of the desirable properties of the program. It means an object of type A may be substituted with any object of a subtype B.

Sub-class

- Subclass refers to the class which is derived from its parent class. Such as:

```
Class A
{
  ……..
};
Class B : public A
{
  ……..
};
```

- Here, class B is a subclass of class A.
- However, it does not follow the principle of substitutability.
- Simply, the base class cannot be replaced by its subclass.
- Here, class B specializes class A to a particular use, by reusing some of its behavior and perhaps overriding parts.
- a subclass is a class that inherits from a superclass and extends or modifies its behavior.

Sub-type

- Subtype refers to the class which follows Liskov's substitution principle.
- These types of subtypes can replace the parent class without making any errors.
- A subtype *inherits* all features from its supertypes.

```
Class A
{
  ……..
};
Class B : public A
{
  ……..
};
```

- Here, Subtyping means writing a class B which conforms to A's interface, as well as possibly adding some new methods of its own.
- As per Liskov,s substitution principle, we can supply B in any context where an A is expected.
- a subtype is a type that is derived from a supertype and can be substituted for the supertype.

Software Reusability

- Software reusability refers to the ability to reuse existing code components in new applications.
- Object-Oriented Programming (OOP) provides powerful mechanisms to enhance software reusability.
- By embracing tools for achieving software reusability, developers can save time, improve code quality, and enhance maintainability

key concepts and techniques for achieving software reusability in OOP:

- Encapsulation
- Inheritance
- Polymorphism
- Design Patterns
- Libraries and Frameworks

Benefits of Software Reusability:

- Saves development time and effort: Reusing existing code reduces the need for reinventing the wheel.
- Improves code quality: Reusable components are often thoroughly tested and debugged, leading to more reliable code.
- Enhances maintainability: Changes made to reusable components propagate across multiple applications, ensuring consistency and ease of maintenance.

End of chapter 3