



**Discipline Courses-I**  
**Semester-I**  
**Paper: Programming Fundamentals**  
**Unit-VI**  
**Lesson: Inheritance and virtual Functions**  
**Lesson Developer: Archana Singhal**  
**College/Department: IP College, University of Delhi**

## Table of Contents

### ● Chapter 2: Inheritance and Virtual Functions

- 2.1: Inheritance
  - 2.1.1: What is Inheritance?
  - 2.1.2: Accessing Base class members in Derived class
  - 2.1.3: The Protected Access Specifier
- 2.2: The Public, Private and Protected Inheritance
  - 2.2.1: Access Control of Base class members using Public, Private and Protected Inheritance
  - 2.2.2: Overriding Base class Members
- 2.3: Accessibility of Base class Constructors and Destructors in Inheritance
- 2.4: Multiple Inheritance
  - 2.4.1: What is Multiple Inheritance
  - 2.4.2: Accessing Base class Constructors in Multiple Inheritance
  - 2.4.3: Ambiguity in Multiple Inheritance
- 2.5: Advantages of Inheritance
- 2.6: Virtual Functions
  - 2.6.1: What are virtual functions?
  - 2.6.2: Why do we need Virtual Functions?
  - 2.6.3: Virtual Base classes
- 2.7: Abstract classes and Pure Virtual functions
  - 2.7.1: What are abstract classes?
- Summary
- Exercises
- Glossary
- References

## Chapter 2. Inheritance and Virtual Functions

In this chapter we would be focusing upon inheritance and virtual functions. Inheritance is one of the major elements of Object Oriented Programming Languages. Inheritance would allow us to create new classes using already existing classes. This is called code reusability. This code reusability feature would result in saving time and efforts.

Virtual functions are used to implement run-time polymorphism. In run-time polymorphism call to overridden function is decided at run time rather than compile time.

### Learning Objectives

After this lesson you would be able to:

- Explain what is inheritance?
- Explain the advantages of Inheritance?
- Create derived classes using base classes.
- Differentiate between public, private and protected inheritance.
- Enforce runtime polymorphism using virtual functions.
- Implement abstract classes.

## 2.1 Inheritance

### **2.1.1 What is Inheritance?**

Inheritance is a very important feature of Object Oriented Programming languages. Inheritance is a process of deriving new classes from already existing classes. It defines the relationship among classes.

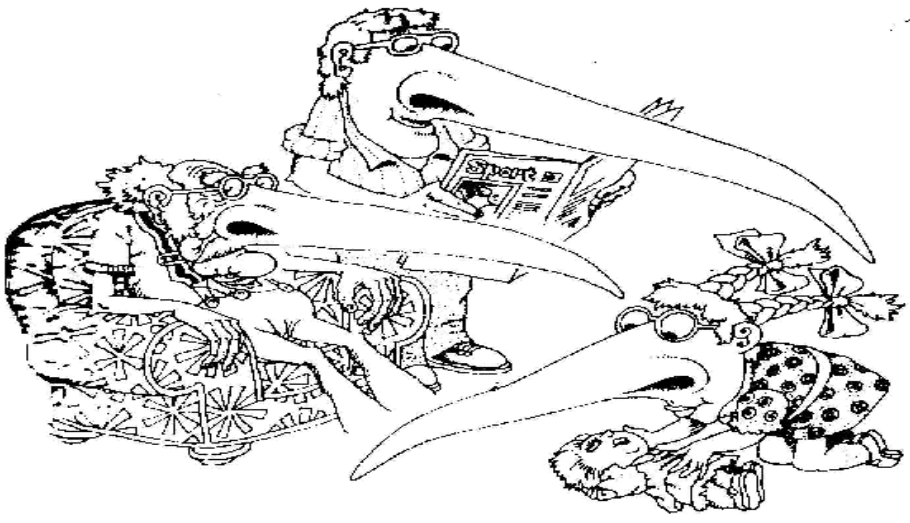
As the name suggests a class is able to inherit characteristics from other classes. One class can share the structure and behavior defined in one or more classes. Or we can say that a class is able to pass on its state and behaviors to its children. The child class can then extend the state and behaviors to reflect the differences it represents. So child class is a more specialized version of the parent. Inheritance process is also called generalization/specialization process.

In this process of Inheritance a class being inherited will be a base class or super class, while subclass or derived is the class that is inheriting from the super class. In Figure 2.1, X is a base class and Y, Z are the derived classes.

General syntax of deriving a class from an already existing class is given as follows:

```
class Derived:access Base
{
    //Body of derived class.
}
```

Where Base is a base class, and Derived is a class derived from base class Base, and access is an access specifier.

<b>Value addition: Did you Know?</b>
<b>Heading text: Inheritance</b>
<b>Body text:</b>  <p>A subclass may inherit the structure and behaviour of its superclass.</p>
<b>Source:</b> Object Oriented Design- by Grady Booch, The Benjamin/Cummings Publishing Company

Let us take following example to illustrate the concept of Inheritance:

//X is a base class

```
class X
{
    //Body
};
```

//Y is a derived class

```
class Y:public X
{
    //Body
}
```

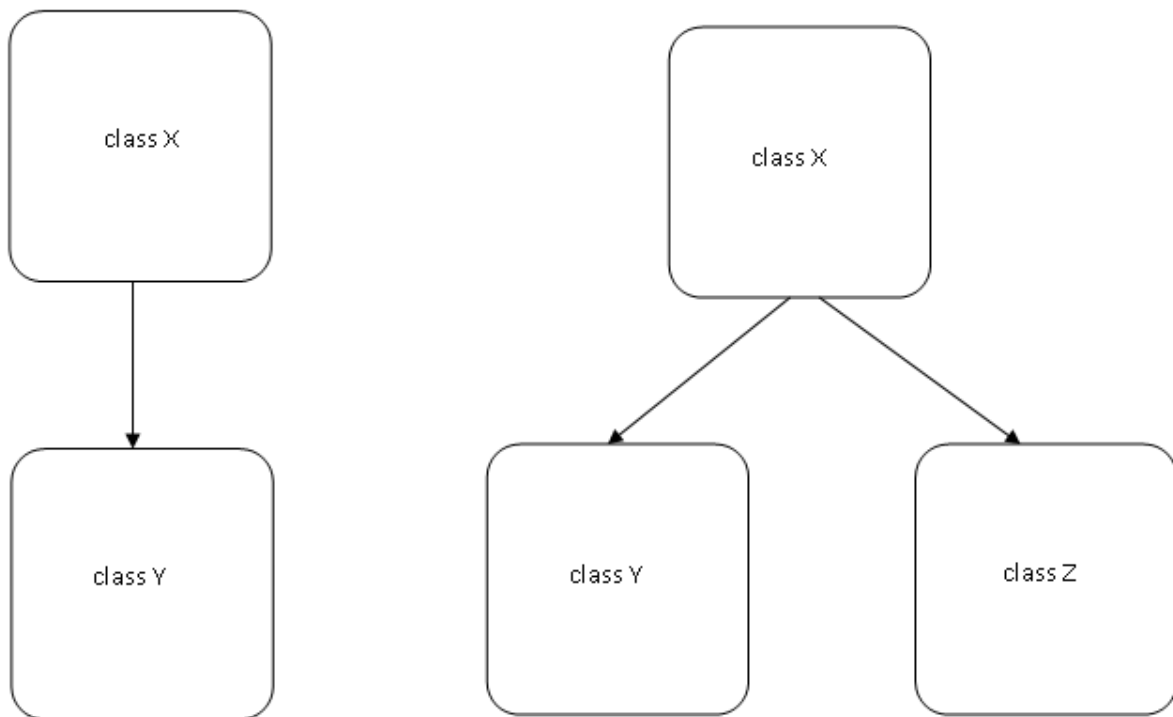


Figure 2.1 Inheritance

Similarly class Z can be derived from X.

```
class Z:public X
{
    //Body
}
```

In the above example X is a base class and Y and Z are derived classes. Classes Y and Z have been derived using public access specifier. In general public access specifier is used while deriving a new class from a base class. We can also use other access specifier protected and private. This concept would be discussed in detail in subsequent sections.

### **2.1.2 Accessing Base Class Members in Derived Class**

```
//X is a base class
#include<iostream>
using namespace std;
```

```
class X
{
    int i,j;
    public:
        X(){i=1;j=1;}
```



## Inheritance and virtual Functions

```
void display1()// displays values of i and j;
{
    cout<<endl<<"i="<<i;
    cout<<endl<<"j="<<j;
}
};
//Y is a derived class

class Y : public X
{
    int z;
public:
    Y(){z=1;}
    void display2(){
        cout<<endl<<"i="<<i;//Error. Can't access base class private data
        cout<<endl<<"j="<<j;//Error. Can't access base class private data
        cout<<endl<<"z="<<z;
    }
};
```

In the above example X is a base class and Y is a derived class.

Since i and j are in private section of class X they cannot be directly accessed by derived class member functions. If function display2() in class Y try to access i and j it will be an error and program would not compile.

To display i and j of base class X, display2() can be rewritten as follows:

```
void display2()
{
    display1();//OK
    cout<<endl<<"z="<<z;
}
```

Since i and j are in private section of class X they cannot be directly accessed by derived class member functions as explained earlier so function display1() is being used to display value of i and j.

```
int main()
{
    X obj1;
    obj1.display1();
    Y obj2;
    obj2.display2();
    getchar();
    return 0;
}
```

In the main() method the statement

```
Y obj2;
```

Creates an object of derived class Y. When object of derived class is created space is allocated for data inherited from base class and for the data additionally defined in derived class as well. So total space allocated for obj2 will be the space required for i, j and z.

Output of the above program after changing definition of display2() is:

```
i=1
j=1
i=1
j=1
z=1
```

### **2.1.3 The Protected Access Specifier**

Earlier we were only discussing about public and private members of a class. Private members of a class cannot be directly accessed outside the class. Even a derived class public member functions cannot access private members of base class.

To solve this problem one solution is to declare data of base class as public. But in this case data is not safe because it can be accessed anywhere outside the class.

Another solution is to declare base class data as **protected**. Data declared in protected section can be accessed by derived class member functions but not outside the class directly. So data is safe from accidental damage and can be accessed by derived class member functions as well.

Let us modify the above example using key word protected as follows:

```
//X is a base class
#include<iostream>
using namespace std;

class X
{
protected:
int x,y;
public:
    X(){x=1;y=1;}
    void display1()// displays values of x and y;
    {
        cout<<endl<<"x="<<x;
        cout<<endl<<"y="<<y;
    }
};

//Y is a derived class

class Y : public X
{
    int z;
public:
```

```

Y(){z=1;}
void display2(){
    cout<<endl<<"x="<<x;
    cout<<endl<<"y="<<y;
    cout<<endl<<"z="<<z;
}

};

int main()
{
    X obj1;
    obj1.display1();
    Y obj2;
    obj2.display2();//Ok.
    getchar();
    return 0;
}

```

Now this program would compile without error and give the following output:

```

x=1
y=1
x=1
y=1
z=1

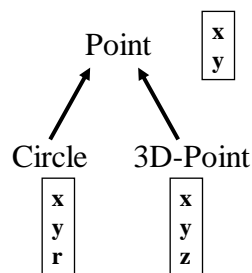
```

## Value addition: Did you Know?

### Heading text: Inheritance

#### Body text:

## Inheritance Concept



```

class Point{
    protected:
        int x, y;
    public:
        void set (int a, int b);
};

```

```

class Circle : public Point{
    private:
        double r;
};

```

```

class 3D-Point: public Point{
    private:
        int z;
};

```

4

Inheritance



**Source:** [www.cse.ohio-state.edu](http://www.cse.ohio-state.edu)

### Value addition: Did you Know?

#### Heading text: Inheritance

##### Body text:

Public members (data and functions) of a base class can be directly accessed by member functions of derived class and by the object of derived class as well.

Consider the following example:

```
#include<iostream>
using namespace std;
class X
{
protected:
    int i,j;
public:
    int k;
    X(){i=1;j=2;k=3;}
    void display()// displays values of i and j;
    {
        cout<<endl<<" i="<<i;
        cout<<endl<<" j="<<j;
        cout<<endl<<" k="<<k;
    }
};

//Y is a derived class

class Y: public X
{
public:
    void display_sum()
    {
        cout<<endl<<" Sum="<<i+j;
    }
};

int main()
{
    Y obj;
    obj.k=10; //OK public member of base class
    obj.display();//OK public member of base class
    obj.display_sum();
    getchar();
    return 0;
}
```

Output:

```
i=1
j=2
k=10
sum=3_
```

## 2.2 Public, Protected and Private Inheritance

### **2.2.1 Access Control of Base Class Members using Public, Private and Protected Inheritance**

In the earlier section we were deriving new classes using public inheritance. But other key words like private and protected access specifier can also be used as discussed in earlier section. Now let us explain the meaning of Public, Private and Protected inheritance.

When we derive new classes using public which is a general practice, the members of a base class are inherited with same access rights as defined in base class. For example if members are public in base class, they will also be public in derived class. The protected members of base class will remain protected in derived class and so on. If we use Private access specifier all the members inherited from base class will behave as private members in derived class. Similarly in Protected inheritance everything except Private is inherited as Protected. This has been illustrated in the table given in figure 2.2.

	Base Class members	Inherited in Derived Class
Public Inheritance	Public Private Protected	Public Private(not accessible) Protected
Private Inheritance	Public Private Protected	Private Private(not accessible) Private
Protected Inheritance	Public Private Protected	Protected Private(not accessible) Protected

Figure 2.2 Public, Private and Protected Inheritance

## Inheritance and virtual Functions

Let us derive the class Y from class X using protected key word.

```
#include<iostream>
using namespace std;
class X
{
    protected:
        int i,j;
    public:
        X(){i=1;j=2;}
        void display()// displays values of x and y;
        {
            cout<<endl<<"i="<<i;
            cout<<endl<<"j="<<j;
        }
};
//Y is a derived class

class Y: protected X
{
    public:
        void swap()
        {
            int temp=i;// OK protected
            i=j;
            j=temp;
            display();//OK protected member
        }
};

int main()
{
    Y obj;
    obj.display();//Error inherited as protected member
    obj.swap();
    getch();
    return 0;
}
```

When we use protected inheritance all the public members of base class are inherited as protected members as discussed earlier. So the statement

`obj.display();` is not valid. So when you compile this program it will give an error message that `display()` is not accessible.

Now `display()` can be used only inside public member functions of class Y.

Similarly if we use private keyword in place of protected all the inherited members of base class X becomes private in derived class Y and cannot be directly accessed outside the class with class object.

**Value addition: Did you Know?****Heading text: Friendship in Inheritance****Body text:**

- The friend functions and friend classes of the base class are not inherited.

As we know that private members of a class cannot be directly accessed outside the class. Even a derived class cannot access private members of its base class. But a class or a function can access private data of the class if defined as friend. If a function or a class is a friend of a base class they will not be a friend of its derived class by default. This concept is shown in the following example.

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

class A
{
    int Private_dataA;
public:
    friend class C;
    A(){Private_dataA=0;}
    A(int d){Private_dataA=d;}
    void display(){cout<<endl<<"Inside A";}
};

class B: public A
{
    int Private_dataB;
public:
    B(){Private_dataB=0;}
    B(int i){Private_dataB=i;}
    void display(){cout<<endl<<"Inside class B:";}
};

class C
{
public:
    void show(A a,B b)
    {
        cout<<endl<<"Private data of class A:"<<a.Private_dataA;//OK C is friend of A
        cout<<endl<<"Private data of class B:"<<b.Private_dataB;//Error
    }
};

int main()
{
    A a;
    B b;
    C c;
```

```
c.show(a,b);//Error
getchar();
return 0;
}
```

This program would not compile. Because class C is the friend of base class A. that friendship is not inherited in class B so it will not be not a friend of class B. That's why it cannot access private data of class B. To compile this program make class B also friend of class C. the output would be:  
Private data of class A:0  
Private data of class B:0

**Source:**

## **2.2.2 Overriding Base class members**

### **Method Overriding:**

While defining a derived class we can use same function name as used in base class. This feature is called method-overriding. Consider the following example:

```
//X is a base class
#include<iostream>
using namespace std;

class X
{
protected:
int x,y;
public:
X(){x=10;y=10;}
void display()// displays values of x and y;
{
cout<<endl<<"x="<<x;
cout<<endl<<"y="<<y;
}
};

//Y is a derived class

class Y : public X
{
int z;
public:
Y(){z=5;}
void display()//same function name as defined in base class
{
cout<<endl<<"x="<<x;
cout<<endl<<"y="<<y;
cout<<endl<<"z="<<z;
}
};
```



## Inheritance and virtual Functions

```
int main()
{
    X obj1;
    obj1.display(); //invoke base class definition
    Y obj2;
    obj2.display(); //invoke derived class definition
    getchar();
    return 0;
}
```

This program would give the following output:

```
x=10
y=10
x=10
y=10
z=5
```

In the above program base class X is defining a function `display()` to display data of base class. The derived class Y is also defining the same function `display()` to display the data inherited from base class and its own additional data. The `display()` of derived class Y overrides the definition of function `display()` of class X. So whenever we access `display()` with derived class object it will invoke the definition given in derived class. So the statement

`Obj2.display()` will invoke the derived class definition of `display()`;

But definition of base class `display()` is still available in class Y and can be invoked through class scope using scope resolution operator (`::`) as shown below:

```
class Y: public X
{
    int z;
public:
    Y(){z=5;}
    void display(){
        X::display(); //invokes base class display()
        cout<<endl<<"z="<<z;
    }
};
```

This will have the same effect as the earlier version of function `display()` in class Y and give the same output.

<b>Value addition: Did you Know?</b>
<b>Heading text: Inheritance and static functions</b>
<b>Body text:</b>

- The static functions of the base class can also be overridden in derived classes..

```
//X is a base class
#include<iostream>
using namespace std;

class X
{
protected:
    static int a;
public:
    static void display_static_data()// displays values of a;
    { cout<<endl<<" Inside X";
      cout<<endl<<" a="<<a;
    }
};

//Y is a derived class
class Y : public X
{
    static int b;
public:
    static void display_static_data()// OK. Displays values of a & b;
    { cout<<endl<<" Inside Y";
      cout<<endl<<" a="<<a;
      cout<<endl<<" b="<<b;
    }
};

int X::a=1;
int Y::b=2;
int main()
{
    X::display_static_data();
    Y::display_static_data();
    getchar();
    return 0;
}
```

This program will compile without error and produce the following output:

```
Inside X
a=1
Inside Y
a=1
b=2
```

### Overriding Data Members of Base Class:

## Inheritance and virtual Functions

In the previous section we have discussed method-overriding in which derived class have same function name as base class functions. It is also possible to give same name to data members also. This feature is called data hiding. In this case base class data that has been redefined is not directly available in derived class. We can access that data through class scope as in the case of functions.

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

class X
{
protected:
    int i,j;

public:
    X(){i=10;j=10;}
    void display()// displays values of i and j;
    {
        cout<<endl<<"i="<<i;
        cout<<endl<<"j="<<j;
    }
};

//Y is a derived class

class Y : public X
{
    int i;
public:
    Y(){i=5;}
    void display()
    {
        cout<<endl<<"i="<<i;//display derived class i
        cout<<endl<<"j="<<j;//display base class j
        cout<<endl<<" base class i="<<X::i;//display base class i
    }
};

int main()
{
    X obj1;
    obj1.display();//invoke base class definition
    Y obj2;
    obj2.display();//invoke derived class definition
    getchar();
    return 0;
}
```

Output is:

i=10

j=10

i=5

j=10

Base class i =10

In the above program class Y is again redefining i as a data member. Now two copy of i are available in class Y. one inherited from base class and second redefined in derived class. Definition of i in derived class is hiding the definition of i inherited from base class. That's why base class definition is not directly available in derived class. If we have to access i of base class that will be done through class scope as shown in the statement

```
cout<<endl<<" base class i="<<X::i;//display base class i
```

X::i will invoke the base class definition of i.

## 2.3 Accessibility of Constructors and Destructors in Inheritance

### **2.3.1 Accessibility of Base class Constructors in Derived classes**

Consider the following program:

```
//X is a base class
#include<iostream>
using namespace std;
class X
{
protected:
int i,j;
public:
    X(){i=10;j=10;}
    X(int x,int y){i=x;j=y;}
    void display()// displays values of i and j;
    {
        cout<<endl<<"i="<<i;
        cout<<endl<<"j="<<j;
    }
};

//Y is a derived class
class Y : public X
```

## Inheritance and virtual Functions

```
{
public:
    void display_sum()
    {
        cout<<endl<<"Sum of i and j is:"<<i+j;//display sum of base class i &j
    }
};

int main()
{
    Y obj;//invokes base class constructor to initialize i and j
    obj.display();//invokes base class definition
    obj.display_sum();//invokes derived class definition
    getchar();
    return 0;
}
```

In the above program consider the statement

`Y obj;`

This statement creates a derived class object `obj`. Since no constructor in derived class has been defined it will automatically invoke base class default constructor to initialize `i` and `j` inherited from base class and generates the following output:

Output is:

```
i=10
j=10
Sum of i and j is :20
```

Now let us modify `main()` as follows:

```
int main()
{
    Y obj(1,2);//Error
    obj.display();
    getchar();
    return 0;
}
```

This program will not compile and give an error message that no matching function call to `Y::Y(int,int)`. Even if a parameterized constructor has been defined in base class it is not accessible in derived class. Since parameterized constructors are not inherited. To make this program compile let us modify it as given below:



**Value addition: Did you Know?****Heading text: Inheritance****Assignment compatibility between Base class and Derived****class:**

1. Derived class object can be assigned to base class object
2. Derived class object can be assigned to base class reference
3. Derived class object address can be assigned to base class pointer

```
#include<iostream>
using namespace std;
class X
{
protected:
int i,j;
public:
X(){i=10;j=10;}
X(int x,int y){i=x;j=y;}
void display()// displays values of i and j;
{
cout<<endl<<"i="<<i;
cout<<endl<<"j="<<j;
}
};
//Y is a derived class

//Y is a derived class
class Y : public X
{
int k;
public:
Y():X(){k=5;}
Y(int x, int y,int z):X(x,y){k=z;}
void display(){X::display();
cout<<endl<<"k="<<k;}
void display_sum()
{
cout<<endl<<"Sum of i, j and k is:"<<i+j+k;//display sum of
base class i &j
}
};

int main()
{
X x, *p;
Y y(1,2,3);
x=y; //O.K. subclass object assignment to base class object
x.display();
p=&y; //O.K. subclass object address assignment to base class object
```

## Inheritance and virtual Functions

```
p->display();  
X &r=y;//O.K. subclass object assignment to base class reference  
r.display();  
getchar();  
return 0;  
}
```

Output:

```
i=1  
j=2  
i=1  
j=2  
i=1  
j=2
```

**Source:**

```
//X is a base class  
#include<iostream>  
using namespace std;  
class X  
{  
protected:  
int i,j;  
public:  
    X(){i=0;j=0;}  
    X(int x,int y){i=x;j=y;}  
    void display()// displays values of i and j;  
    {  
        cout<<endl<<"i="<<i;  
        cout<<endl<<"j="<<j;  
    }  
};
```

```
//Y is a derived class  
class Y : public X  
{  
public:  
    Y(){i=5;j=5;}  
    Y(int x, int y){i=x;j=y;}  
    void display_sum()  
    {
```

## Inheritance and virtual Functions

```
        cout<<endl<<"Sum of i and j is:"<<i+j;//display sum of base class i &j
    }

};

int main()
{
    Y obj1;
    Y obj2(1,2);//O.K.
    obj1.display();
    obj1.display_sum();
    obj2.display();
    obj2.display_sum();
    getchar();
    return 0;
}
```

Output is:

```
i=5
j=5
Sum of i and j is 10
i=1
j=2
Sum of i and j is 3
```

Note that we have also defined default constructor in derived class because once you write a parameterized constructor in derived class, even default constructor of base class will not be automatically invoked while creating derived class object as in earlier program.

Consider the following program:

```
//X is a base class
#include<iostream>
using namespace std;

class X
{
protected:
    int i,j;
public:
    X(){i=0;j=0;}
    X(int x,int y){i=x;j=y;}
    void display()// displays values of i and j;
    {
        cout<<endl<<"i="<<i;
        cout<<endl<<"j="<<j;
    }
};

//Y is a derived class
```

```
class Y : public X
{
public:
Y(int x, int y){i=x;j=y;}
void display_sum()
{
    cout<<endl<<"Sum of i and j is:"<<i+j;//display sum of base class i &j
}
};

int main()
{
Y obj;//Error
obj.display();
getchar();
return 0;
}
```

The above program will not compile and give an error message that Y::Y() is not available. Since no default constructor has been defined in derived class Y.

### Value addition: Did you Know?

#### Heading text: Order of execution of constructors

While creating derived class object its immediate base class constructor is called first to initialize its base class data which in turn invokes the constructor of its immediate base class if any and so on.

```
#include<iostream>
using namespace std;
class A
{
public:
    A(){cout<<endl<<"Constructing class A object";}
    virtual void display()
    {
        cout<<endl<<"Inside class A";
    }
};

class B: public A
{
public:
    B(){cout<<endl<<"Constructing class B object";}
    void display()
    {
        cout<<endl<<"Inside Derived class B";
    }
};
```

```

class C: public B
{
public:
    C(){cout<<endl<<"Constructing class C object";}
    void display()
    {
        cout<<endl<<"Inside Derived class C";
    }
};

int main()
{
    C c;
    c.display();
    getchar();
    return 0;
}

```

A is a base class and B is its subclass. C is a subclass derived from B. when we create an object of class C it will invoke the constructor of class C which invokes constructor of class B before executing and constructor of class B will in turn invoke the constructor of class A. So constructor of class A will be executing first then constructor of class B will be executed. Finally constructor of class C will be executed. The following output will be produced by the program:

Output:

```

Constructing class A object
Constructing class B object
Constructing class C object
Inside Derived class C

```

### **Source:**

Base class constructors can be explicitly invoked in derived classes to initialize base class members as shown in the following program:

```

//X is a base class
#include<iostream>
using namespace std;

class X
{
protected:
    int i,j;
public:
    X(){i=0;j=0;}
    X(int x,int y){i=x;j=y;}
    void display()// displays values of i and j;
}

```



## Inheritance and virtual Functions

```
{
    cout<<endl<<"i="<<i;
    cout<<endl<<"j="<<j;
}
};
//Y is a derived class
class Y : public X
{
    int k;
public:
    Y():X(){k=5;}
    Y(int x, int y, int z):X(x,y){k=z;}
    void display()
    {
        X::display();
        cout<<endl<<"k="<<k;//display derived class k
    }
};

int main()
{
    Y obj1;
    Y obj2(1,2,3);//O.K.
    obj1.display();
    obj2.display();
    getch();
    return 0;
}
```

In the above program the statement

`Y():X(){k=5}`

Invokes the base class default constructor to initialize base class data, and the statement

`Y(int x,int y, int z):X(x,y){k=z;}`

Makes a call to base class parameterized constructor with x and y to initialize base class i and j respectively.

Output of the above program is as follows:

Output:

```
i=0
j=0
k=5
i=1
j=2
k=3
```

**Value addition: Program showing the concept of Inheritance**

**Heading text: Inheritance**

**Body text:**

0

```
#include<iostream>
#include<string>
using namespace std;
class Person
{
protected:
    char name[20];
    int age;
public:
    Person()
    {age=0;strcpy(name,"");}
    Person(char n[],int a)
    {
        age=a;
        strcpy(name,n);
    }
    void Input()
    {
        cout<<endl<<"Enter Name:";
        cin>>name;
        cout<<"Enter Age:";
        cin>>age;
    }
    void display()
    {
        cout<<endl<<"Name:"<<name;
        cout<<endl<<"Age:"<<age;
    }
};

class Student : public Person
{
protected:
    int RollNo;
    char course[20];
public:
    Student():Person()
    {RollNo=0;}
};
```

## Inheritance and virtual Functions

```
Student(int a, char n[], int r,char c[]):Person(n,a)
{
    RollNo=r;
    strcpy(course,c);
}

void Input()
{
    cout<<endl<<"Input Student Data"<<endl;
    Person::Input();
    cout<<"Enter RollNo:";
    cin>>RollNo; cout<<"Enter Course:";
    cin>>course;
}

void display()
{
    Person::display();
    cout<<endl<<"RollNo:"<<RollNo;
    cout<<endl<<"Course:"<<course<<endl;;
}

};

class Employee : public Person
{
protected:
    int EmpNo;
    double salary;
public:
    Employee():Person()
    {EmpNo=0;salary=0.0;}
    Employee (int a, char n[], int e,double s):Person(n,a)
    {
        EmpNo=e;
        salary=s;
    }

    void Input()
    {
        cout<<endl<<"Input Employee Data:";
        Person::Input();
        cout<<"Enter EmpNo:";
        cin>>EmpNo;
        cout<<"Enter Salary:";
        cin>>salary;
    }

    void display()
    {
```

## Inheritance and virtual Functions

```
        Person::display();
        cout<<endl<<"EmpNo:"<< EmpNo;
        cout<<endl<<"Salary:"<< salary;
    }

};

int main()
{
    Student s;//O.K.
    s.Input();
    s.display();
    Employee e;
    e.Input();
    e.display();
    getchar();
    return 0;
}
```

The output of the above program is:

Input Student Data:

Enter Name:Ravi  
Enter Age:21  
Enter Rollno:20  
Enter Course:B.Sc.

Name:Ravi  
Age:21  
Rollno:20  
Course:B.Sc.

Input Employee Data:

Enter Name:John  
Enter Age:41  
Enter Empno:2  
Enter Salary:30000

Name:John  
Age:41  
Empno:2  
Salary:30000

**Source:**

## 2.4 Multiple Inheritance

### **2.4.1 What is Multiple Inheritance?**

In the earlier sections we were deriving sub classes from only one base class. Sometimes there is a requirement when a class wants to share the attributes and behavior of more than one class plus have its own additional attributes and behavior. In C++ it is possible to derive a new class from more than one base class. This is called Multiple Inheritance.

In figure 2.4 class X and class Y are base classes and Z is a derived class. This is an example of multiple inheritance because Y is a subclass of more than base class i.e. inheriting the attributes and behavior of class X as well as class Y.

Let us illustrate this concept through the following example:

```
//X is a base class
#include<iostream>
using namespace std;

class X
{
protected:
int i,j;
public:
    X(){i=0;j=0;}//default constructor
    X(int x,int y){i=x;j=y;}
    void display1()// displays values of i and j;
    {
        cout<<endl<<"i="<<i;
        cout<<endl<<"j="<<j;
    }
};
```



## Inheritance and virtual Functions

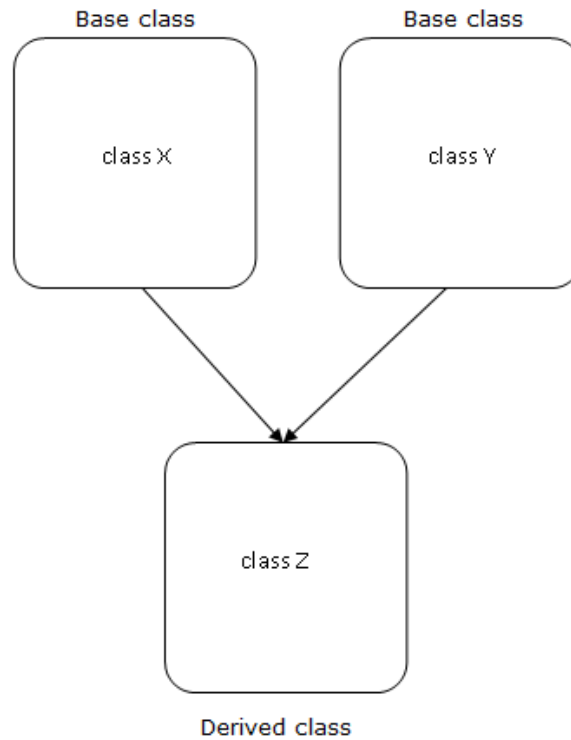


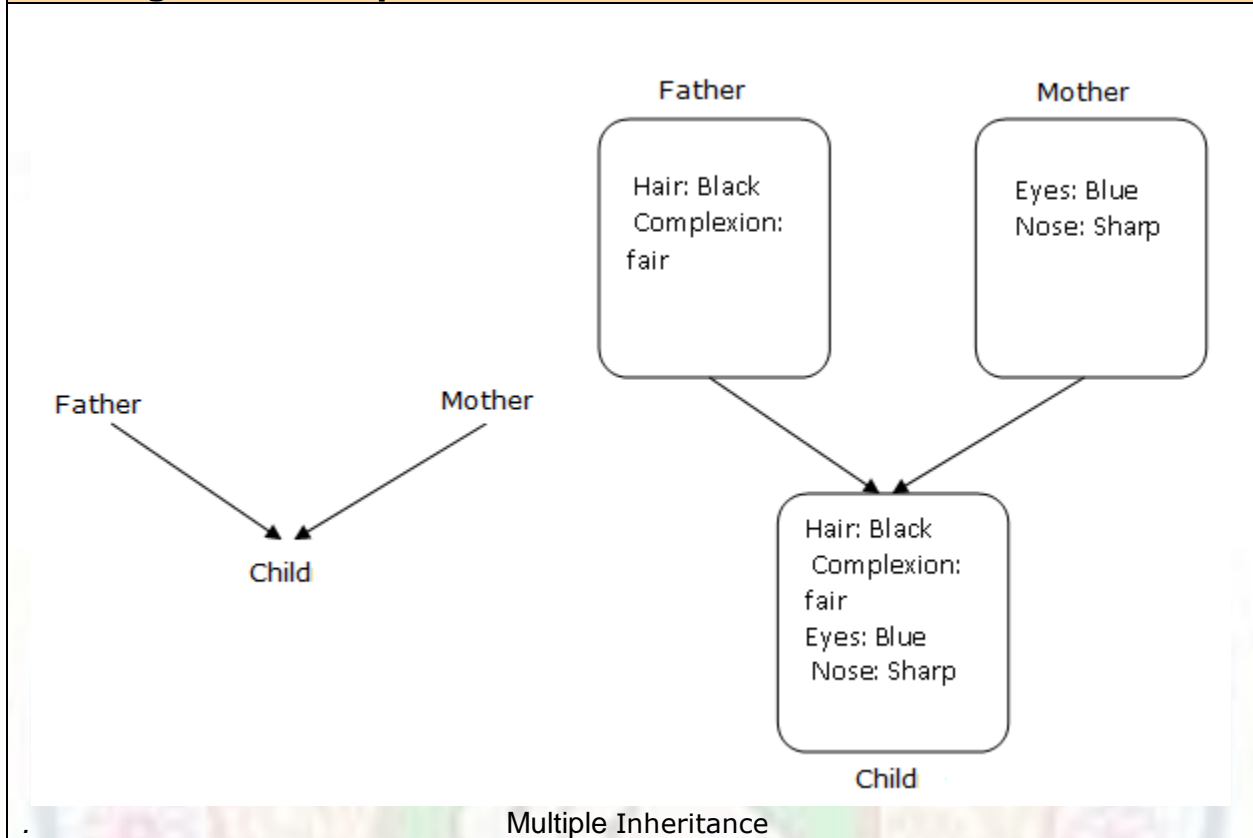
Figure 2.3 Multiple Inheritance

//Y is a base class

```
class Y
{
protected:
    int k;
public:
    Y(){k=0;} //default constructor
    Y(int x){k=x;}
    void display2()
    {
        cout<<endl<<"k="<<k;
    }
};
```

**Value addition: Image**

**Heading text : Multiple Inheritance**



```

//Z is derived from X and Y
class Z:public X, public Y
{
    Int I;

public:
    Z(){i=0;j=0;k=0;l=0;} //default constructor
    Z(int x,int y, int z, int w){i=x;j=y;k=z;l=w;}
    void display ()
    {
        display1();
        display2();
        cout<<endl<<"I="<<l;
    }
};

int main()
{
    Z obj1;//invokes default constructor
  
```

```
obj1.display();  
Z obj2(1,2,3,4); //invokes parameterized constructor  
obj2.display();  
getchar();  
return 0;  
}
```

Output is:

```
i=0  
j=0  
k=0  
l=0  
i=1  
j=2  
k=3  
l=4
```

### Value addition: Did you Know?

#### Heading text: Order of execution of Destructors

##### Body text:

While destroying objects of derived classes the order of invocation of destructors is reverse as the order of invocation of constructors. Derived class destructor is called first before its base class.

```
#include<iostream>  
using namespace std;  
  
class A  
{  
  
public:  
    A(){cout<<endl<<"Constructing class A object";}  
    ~A(){cout<<endl<<"Destructing class A object";}  
  
    virtual void display()  
    {  
        cout<<endl<<"Inside class A";  
    }  
};  
  
class B: public A  
{  
  
public:  
    B(){cout<<endl<<"Constructing class B object";}  
    ~B(){cout<<endl<<"Destructing class B object";}  
  
    void display()
```

```
{
cout<<endl<<"Inside Derived class B";
}
};

class C: public B
{

public:
    C(){cout<<endl<<"Constructing class C object";}
    ~C(){cout<<endl<<"Destructing class C object";}

    void display()
    {
        cout<<endl<<"Inside Derived class C";
    }
};

int main()
{
    C *c=new C();
    c->display();
    delete c;
    getchar();
    return 0;
}
```

Output:

```
Constructing class A object
Constructing class B object
Constructing class C object
Inside Derived class C
Destructing class C object
Destructing class B object
Destructing class A object
```

As you can see from the output while destructing the object of class C, the destructor of derived class C is called first then the destructor of its immediate base class B is called, and finally destructor of class A is called which is a base class of B. This is reverse order of constructor invocation.

**Source:**

### **2.4.2 Accessing Base class constructors in Multiple Inheritance**

We can explicitly invoke base class constructors in case of multiple inheritance also. The class Z in the above program can be modified as follows:

```
//Z is derived from X and Y
class Z:public X, public Y
{
public:
    Z():X(),Y(){l=0;} //invoking default constructors of base class X and Y
    Z(int x,int y, int z, int w):X(x,y),Y(z){l=w;} //invoking parameterized constructors of base
class X & Y
    void display ()
    {
        display1();
        display2();
        cout<<endl<<"l="<<l<<endl;
    }
};
```

In the statement  
`Z():X(),Y(){l=0;}`

The default constructor `Z()` of class `Z` is invoking base class default constructors `X()`, `Y()` in sequence to initialize members inherited from `X` and `Y` respectively.  
The name of the base class constructors follow the colon (`:`) and are separated by commas.

Similarly the statement  
`Z(int x,int y, int z, int w):X(x,y),Y(z){l=w;}`

Invokes base class parameterized constructors in sequence.

### **2.4.3 Ambiguity in Multiple Inheritance**

To illustrate the type of ambiguity that can occur in multiple inheritance, let us consider the following example:

```
//X is a base class
#include<iostream>
using namespace std;

class X
{
protected:
    int i,j;
public:
    X(){i=0;j=0;} //default constructor
    X(int x,int y){i=x;j=y;}
    void display()// displays values of i and j;
    {
```



## Inheritance and virtual Functions

```
        cout<<endl<<"i="<<i;
        cout<<endl<<"j="<<j;
    }
};

//Y is a base class

class Y
{
    protected:
        int k;
    public:
        Y(){k=0;} //default constructor
        Y(int x){k=x;}
        void display()
        {
            cout<<endl<<"k="<<k;
        }
};

//Z is a derived from X and Y
class Z:public X,public Y
{
    public:
        Z():X(),Y(){k=0;} //invoking default constructors of base class X and Y
        Z(int x,int y, int z):X(x,y),Y(z){} //invoking parameterized constructors of base class X & Y

};

int main()
{
    Z obj;//invokes default constructor
    obj.display();//Error. Ambiguity.
    getchar();
    return 0;
}
```

In the above program both the base classes X as well as Y are defining a display() function and class Z is not having any definition for function display(). if you try to call display() as given in following statement

```
obj.display();
```

## Inheritance and virtual Functions

There will be an error and program will not compile. Since two version of display() are inherited in class Z, one from class X and another from class Y. To resolve this ambiguity we have to use scope resolution operator (::) as given below:  
obj.X::display() or obj.Y::display()//OK

Please note that if class Z overrides function inherited from these two base classes then there will be no ambiguity in the following program.

```
//X is a base class
#include<iostream>
using namespace std;

class X
{
protected:
int i,j;
public:
    X(){i=0;j=0;}//default constructor
    X(int x,int y){i=x;j=y;}
    void display()// displays values of i and j;
    {
        cout<<endl<<"i="<<i;
        cout<<endl<<"j="<<j;
    }
};

//Y is a base class

class Y
{
protected:
    int k;
public:
    Y(){k=0;} //default constructor
    Y(int x){k=x;}
    void display()
    {
        cout<<endl<<"k="<<k;
    }
};

//Z is a derived from X and Y
class Z: public X, public Y
{
public:
```

```
Z():X(),Y(){k=0;} //invoking default constructors of base class X and Y
Z(int x,int y, int z):X(x,y),Y(z){} //invoking parameterized constructors of base class X & Y
void display ()
{
    X::display();//invokes display of class X
    Y::display();//invokes display of class Y
}

};
int main()
{
    Z obj;//invokes default constructor
    obj.display();//OK. No Error! It calls overridden version of class Z.
    getchar();
    return 0;
}
```

In the main() the statement

obj.display() will invoke the definition of derived class Z and there will be no ambiguity.

## 2.5 Advantages of Inheritance

One of the most important advantages of Inheritance is code reusability. It allows programmers to re-use the previously written code without starting from scratch. It will save time and efforts and result in developing more efficient program.

In the Person class example we don't need to create new data members in the Employee and Student class to hold the Name and age because we can inherit them from the base class Person. Only class specific data have been added to these classes.

## 2.6 Virtual Functions

### **2.6.1 What are Virtual Functions?**

Virtual functions are the functions defined in base class and redefined in different derived classes of that base class. These functions are accessed through base class pointers. To discuss the significance of virtual function let us first write a program in which a normal function redefined in derived class is being accessed through base class pointer.

Consider the following program:

## Inheritance and virtual Functions

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

class Base
{
public:
void display()
{
cout<<endl<<"Inside Base class";
}
};

class Derived1: public Base
{
public:
void display()
{
cout<<endl<<"Inside Derived class one";
}
};

class Derived2: public Base
{
public:
void display()
{
cout<<endl<<"Inside Derived class two";
}
};

int main()
{
Base *b;// base class pointer
Derived1 d1;
b=&d1;
b->display();
Derived2 d2;
b=&d2;
b->display();
getchar();
return 0;
}
```

In the above program Base is a base class and Derived1 and Derived2 are two classes derived from class Base. The display() function is defined in class Base and redefined in class Derived1 and class Derived2.

In main method the statement  
Base \*b; creates a base class pointer.

Another statement

## Inheritance and virtual Functions

`b = &d1;` assigns address of derived class object `d1` to base class pointer `b`. Is it a valid statement? Will this program compile?

The answer is yes. A base class pointer can always hold the address of its derived class object. So this is a valid assignment.

Similarly the statement

`b = &d2;` is a valid statement.

The program would compile without error and produce the following output:

Inside Base class

Inside Base class

This is not the expected output as we were expecting the following output:

Inside Derived class one

Inside Derived class two

When we write the statement

`b->display()`, it will always invoke base class function `display()` whether it is having address of object of `Derived1` class or `Derived2` class. Since compiler is making the decision based on the type of pointer through which function is being called.



**Value addition: Did you Know?****Heading text: Static functions as Virtual functions**

**Static functions cannot be declared as virtual.** Static function only access static data which is common to all objects of that class. They can be accessed without object with class scope. That's why they don't need to be virtual which facilitates run-time decision of function invocation.

```
#include<iostream>
using namespace std;
class X
{
protected:
    static int a;
    int x,y;
public:
    X(){x=10;y=10;}
    void display()// displays values of x and y;
    {
        cout<<endl<<"x="<<x;
        cout<<endl<<"y="<<y;
    }
    virtual static void display_a()// Error! Can't be Virtual.
    {
        cout<<endl<<"a="<<a;
    }
};

//Y is a derived class
class Y : public X
{
    int z;
public:
    Y(){z=5;}
    void display(){
        X::display();
        cout<<endl<<"z="<<z;
    }
};

int X::a=1;
int main()
{
    Y y;
    y.display();//invoke derived class definition
    X::display_a();//Or write Y::display_a()
    getch();
    return 0;
}
```

To compile this program remove key word virtual from function display\_a() than this

will produce the following output:

```
x=10
y=10
z=5
a=1_
```

**Source:**

Let us modify the same program given above using key word virtual as shown below:

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

class Base
{
public:
virtual void display()
{
cout<<endl<<"Inside Base class";
}
};

class Derived1: public Base
{
public:
void display()
{
cout<<endl<<"Inside Derived class one";
}
};

class Derived2: public Base
{
public:
void display()
{
cout<<endl<<"Inside Derived class two";
}
};

int main()
{
Base *b;// base class pointer
Derived1 d1;
b=&d1;
b->display();
Derived2 d2;
b=&d2;
b->display();
}
```

```
getchar();  
return 0;  
}
```

The output of the above program is:

```
Inside Derived class one  
Inside Derived class two_
```

In the above program function `display()` has been defined as virtual in base class `Base`. This enforces runtime polymorphism feature of the language. Now in the statements

```
b=&d1;  
b->display()
```

In runtime polymorphism the decision of the function to be invoked will be made at run time depending on the address, the object is referring to. This feature is also termed as late binding. The key word `virtual` enforces the compiler to delay the binding of the function till run time. So `b->display()` will invoke the `display()` of `Derived1` class because `b` is referring to `d1` which is an object of `Derived1` class.

Similarly the statements

```
b=&d2;  
b->display()
```

Would cause to invoke `display()` of derived class `Derived2`, since `d2` is an object of class `Derived2`.

### Value addition: Did you Know?

#### Heading text:Virtual function

#### Body text:

Constructors cannot be defined as virtual. Since constructors are invoked when an object of a class is created. At the time of creation of an object, its exact type has to be known. This cannot be delayed till run-time. That's why virtual constructors are not possible.

```
class X  
{  
    public:  
        virtual X();//Error. Not allowed  
}
```

#### Source:

### 2.6.3 Why do we need Virtual functions?

Virtual functions are required when we want to group different type of objects together and use them through a single pointer.

## Inheritance and virtual Functions

Suppose we wish to create a list of birds and want to display their features. We create a base class Bird to keep common attributes and make subclasses to deal with different type of birds as shown in figure 2.5. Each class also defines a function display() to display its features. The display() function has been defined as virtual in base class Bird. We will create an array of base class pointers to hold objects of different type of birds, because base class pointer can hold address of different derived class objects. Then we call display to display features of each bird object as shown below:

```
Bird * p[3];  
for(int i=0;i<3;++i)  
p->display();
```

In the array if pointer p is pointing to class Parrot it will display the features of parrot, if it is pointing to class Hen it will display the features of Hen and so on.

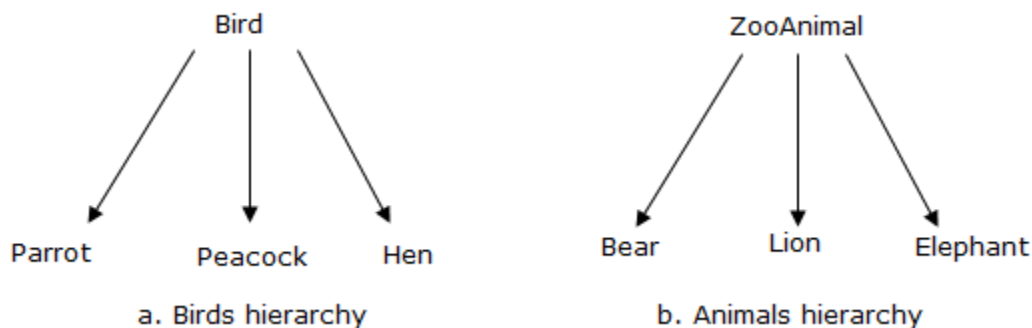


Figure 2.5 Inheritance

Similarly we can create an array of animals using pointer of class ZooAnimal and display their features as discussed above.

### Value addition: Did you Know?

#### Heading text: Virtual function

##### Body text:

Destructors can be defined as virtual. Since Destructors are used to destroy an already existing object of a class. Depending on the address of the object appropriate destructor can be invoked to destroy an object.

```
#include<iostream>  
using namespace std;  
  
class X  
{  
    public:  
        X(){};  
        virtual ~X(){cout<<endl<<"destroying X...";}//OK  
};  
class Y:public X  
{  
    public:  
        Y(){};  
        ~Y(){cout<<endl<<"destroying Y...";}//OK  
};
```

```
};
int main()
{
    Y obj;
    X * p;
    p=&obj;
    delete p;
    getchar();
    return 0;
}
```

In main the statement `delete p;` will invoke destructor of class Y since p is containing address of class Y object. Destructor of class Y in turn will invoke destructor of class X. the output of the above program would be:

```
destroying Y...
destroying X..._
```

**Source:**

## 2.6.3 Virtual Base classes

To explain the concept of virtual base class let us consider the following example:

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

class A
{
public:
void display(){cout<<endl<<"Inside display";}
};

class B:public A
{
};

class C:public A
{
};

class D:public B, public C
{
};
```



## Inheritance and virtual Functions

```
int main()
{
    D d;
    d.display();//Error!
    getchar();
    return 0;
}
```

In the above example A is a base class and B and C are two subclasses derived from class A. D is a sub class derived from class B and C both as shown in figure 2.6.

When you compile this program it will not compile and give an error message that call to function display() is ambiguous. This is because two version of display() are available in class D, one through class B and another through class C. To resolve this ambiguity we have to derive subclasses B and C from A using keyword virtual. In this case class A will be called a virtual base class and only one copy of A will be available in D. This is shown in the following modified program:

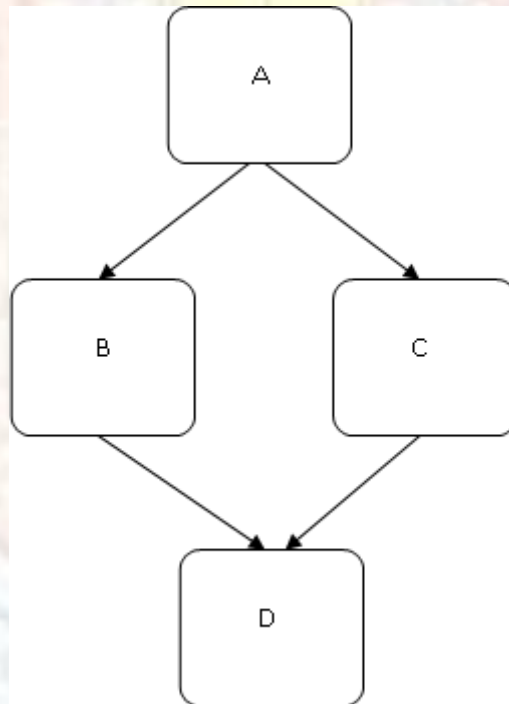


Figure 2.6 Multiple Inheritance

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

class A
{
public:
    void display(){cout<<endl<<"Inside display";}
```

```
};

//class B is derived using keyword virtual
class B:virtual public A
{

};

//class C is derived using keyword virtual
class C: virtual public A
{

};
class D: public B, public C
{

}
int main()
{
D d;
d.display();//OK!
getchar();
return 0;
}
```

Now this program would compile without error and produce the following output:

Output:

Inside display

Value addition: Did you Know?
Heading text:Polymorphism
<b>Body text:</b>  Since Polymorphism has a lot of overhead, it is not used in STL (Standard Template Library) to optimize performance.
Source:

## 2.7 Abstract Classes and Pure Virtual functions

### 2.7.1 What are Abstract classes?

## Inheritance and virtual Functions

Abstract class is a class for which no object can be created because this is not completely defined. To implement abstract class we derive subclasses of this abstract class to complete its definition and then use it through derived class objects.

In C++ abstract classes are implemented with the help of pure virtual functions. A pure virtual function is a function without definition i.e. a function without body. A pure virtual function is defined as follows:

```
virtual return-type name(parameter list)=0
```

Consider the following example:

```
class base
{
public:
virtual void f()=0;//pure virtual function
}
```

In the class base function f() is a pure virtual function as no definition has been given to this function. Since base class is incomplete, it is not allowed to create an object of this class.

The following program illustrates this concept.

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

class Shape
{
public:
virtual void draw()=0;
}

class Circle : public Shape
{
public:
void draw(){cout<<endl<<"This is a class Circle.";}
};

class Rectangle : public Shape
{
public:
void draw(){cout<<endl<<"This is a class Rectangle.";}
};

class Triangle : public Shape
{
public:
void draw(){cout<<endl<<"This is a class Triangle.";}
};

int main()
{
Shape *p;
```

## Inheritance and virtual Functions

```
Circle C;  
p=&C;  
p->draw();  
Rectangle R;  
p=&R;  
p->draw();  
Triangle T;  
p=&T;  
p->draw();  
getchar();  
return 0;  
}
```

In the above program class Shape is defined as an abstract class as it is having one pure virtual function draw(). We cannot create an object of class shape as it is partially defined class. But a pointer to an abstract class can be created as given in the following statement:  
Shape \*p;

Classes Circle, Rectangle and Triangle are subclasses completing the partial implementation of class Shape so we can create the objects of these classes.

The output of the above program is:

```
This is a class Circle.  
This is a class Rectangle.  
This is a class Triangle.
```

If a subclass of an abstract class do not complete its partial implementation than this subclass also becomes abstract and object of that class also cannot be created.

Consider the following example:

```
class X  
{  
public:  
void display(){cout<<endl<<"Inside X";}  
virtual void f()=0;  
virtual void g()=0;  
}  
  
class Y:public X  
{  
public:  
void display(){cout<<endl<<"Inside Y";}  
void f(){cout<<endl<<"Inside f() of class Y";}  
}  
  
int main()  
{  
Y y;//Error Y is also an abstract class  
}
```

Class X is having two pure virtual functions f() and g(). The subclass Y of class X provides definition to function f() but not to function g(). The function g() is inherited as pure virtual

function. So class Y is also not complete and becomes an abstract class. If you try to create an object of class Y it will give an error message that object of abstract class cannot be created. To implement this class Y either we must provide definition to both f() as well as g() or derive a subclass from Y and provide definition to pure virtual function g();

```
class Z:public Y
{
public:
void display()
{cout<<endl<<"Inside Z";}

void g(){cout<<endl<<"Inside g() of class Z";}
}
int main()
{
Z z;//OK
z.f();
Z.g();
}
```

### Summary

- Inheritance is a process of deriving new classes from already existing classes. It defines the relationship among classes.
- Private members of base class are not inherited i.e. they can be directly used by derived class member functions.
- Protected data of base class is inherited in derived class i.e. it can be directly accessed by derived class member functions as it is part of derived class.
- While defining a derived class we can use same function name as used in base class. This feature is called method-overriding.
- The most important advantages of Inheritance is code reusability.
- Derived can also give same name to data members as given in base class. This feature is called data hiding.
- Base class constructors and destructors are not inherited in derived classes except default constructor in some special cases.
- When a class is inheriting features from more than one base classes this is called Multiple inheritance.
- In C++ runtime Polymorphism provides an ability to decide which function to invoke at run time depending on the address of the object. This is implemented through the use of virtual functions.
- Virtual functions are the functions defined in base class and redefined in different derived classes of that base class. These functions are accessed through base class pointers.
- Abstract class is a class for which no object can be created because this is not completely defined.
- In C++ abstract classes are implemented using pure virtual functions.



- A pure virtual function is a function without definition.
- To implement an abstract class its subclass must implement it fully.

### Exercises

- 2.1 What is inheritance? What are its advantages? Explain with the help of suitable example.
- 2.2 Differentiate between public, private and protected inheritance with the help of suitable example.
- 2.3 Differentiate between private and protected members of a class.
- 2.4 What is multiple inheritance in C++? Explain.
- 2.5 What are virtual functions? Why do we need virtual functions? Explain with examples.
- 2.6 State whether the following statements are True or False?
- Private members of base class are not inherited in derived classes.
  - A class in C++ cannot be derived from more than one base class.
  - Friend functions of base class are not inherited.
  - Default constructor of base class can be implicitly invoked while creating a derived class object.
- 2.7 Find the error(s) if any in the following code. Also correct the error(s) and give the output.

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

class X
{
protected:
    int a,b;
public:
    X(){a=0;b=0;}
    X(int x, int y){a=x;b=y;}
    void read(){
        cout<<"Enter the value of a:";
        cin>>a;
        cout<<"Enter the value of b:";
        cin>>b;
    }
    void print(){
        cout<<"a="<<a<<" "<<"b="<<b;
```

## Inheritance and virtual Functions

```
    }  
}  
  
class Y:public X  
{  
    int c;  
public:  
    Y(){c=0;}  
    Y(int i, int j, int k):X(i,j){c=k;}  
    void read(){  
        X::read();  
        cout<<"Enter the value of c:";  
        cin>>c;  
    }  
    void print(){  
        X::print();  
        cout<<"c="<<c;  
    }  
}  
  
int main()  
{  
    Y O;  
    O.print();  
    return 0;  
}  
  
ii)  
  
#include<iostream>  
using namespace std;  
  
class X  
{  
protected:  
    int a,b;  
public:  
    X(){a=0;b=0;}  
    X(int x, int y){a=x;b=y;}  
    virtual void read(){  
        cout<<"Enter the value of a:";  
        cin>>a;  
        cout<<"Enter the value of b:";  
        cin>>b;  
    }  
    void print(){  
        cout<<"a="<<a<<" "<<"b="<<b;  
    }  
}
```

## Inheritance and virtual Functions

```
}

class Y:public X
{
    int c;
public:
    Y(){c=0;}
    Y(int i, int j, int k):X(i,j){c=k;}
    void read(){
        X::read();
        cout<<"Enter the value of c:";
        cin>>c;
    }
    void print(){
        X::print();
        cout<<"c="<<c;
    }
}

int main()
{
    X *p;
    Y O;
    p=&O;
    p->read();
    p->print();
    return 0;
}

iii)
#include<iostream>
using namespace std;

class A
{
public:
    virtual void display()=0;
    virtual void draw()=0;
}

class B: public A
{
    int i;
public:
    void display(){cout<<endl<<"Inside B";}
    void display_i(){cout<<endl<<"i="<<i;}
};

class C: public A
{

```

```
void display(){cout<<endl<<"Inside C";}  
void draw(){cout<<endl<<"What you want to draw?";}  
};  
  
int main()  
{  
  B b;  
  C c;  
  b.display();  
  c.display();  
  getchar();  
  return 0;  
}
```

### Glossary

**Abstract class:** A class with partial implementation is called an abstract class. No object of this class can be created.

**delete:** it is an operator used to free the space used in dynamic initialization of the objects.

**Function:** Function groups a set of statement into a unit and give it a name which can be invoked any number of times whenever required.

**Friend:** it is a key word of the language. A friend class or friend function can access private data of the another class.

**Keyword:** They are reserved words of the language.

**Life time:** life time of a program is the duration for which a program is running.

**new:** it is an operator used for dynamic initialization of the objects.

**Object:** Object is an instance of some class. It is also defined as a class variable.

**Private:** It is also a key word and defines access right for various members of a class. Private members are accessed differently from Public members.

**Protected:** It's a key word and associated with access rights of various members of class. Protected members cannot be directly accessed outside the class but derived class public member functions can directly access protected data of base class.

**Public:** It's a key word and defines how to access various members of a class. Public members can be directly accessed outside the class.

**Scope:** The area where a particular data can be accessed defines its scope.

**Virtual Function:** They are accessed through base class pointer, defined in base class and redefined in various subclasses of that base class.

**Pure Virtual Function:** Virtual function without body is called pure virtual function.

**void:** it's a key word. When it is applied with a data type it can contain any type of data.

## References

- [1] **Bjarne Stroustrup**, The C++ Programming Language, , Addison Wesley Publishing Company.
- [2] **Bruce Eckel**, Thinking in C++, Prentice Hall, Upper Saddle River, New Jersey
- [3] **E. Balaguruswamy**, Object Oriented Programming with C++ (4<sup>th</sup> ed.), Tata McGraw Hill
- [4] **Grady Booch**, Object Oriented Design with Application, The Benjamin/Cummings Publishing Company.
- [5] **Herbert Schildt**, C++: The complete Reference (3<sup>rd</sup> ed), Tata McGraw Hill
- [6] **Robert Lafore**, Object Oriented Programming in Microsoft C++, Galgotia Publications
- [7] **Stanly B. Lippman**, Josee Lajoie, C++ Primer, 3<sup>rd</sup> Edition, Addison Wesley