Unit 03 Inheritance

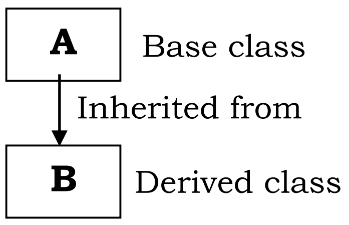
➢ Inheritance:

- We know that, inheritance is one of the most important object oriented concept and C++ language fully supports it.
- The main concept behind inheritance is reusability of code (Software) is possible by adding some extra feature into existing software without modifying it.

Definition:

- "The mechanism of creating new class from existing class is called as Inheritance".
- The existing or old class is called as 'Base class' or 'Parent class' or 'super class' and new class is called as 'Derived class' or 'Child class' or 'sub class'.
- While deriving the new class from exiting one then the derived class will have all the features of existing class and also it has its own features.
- When a new class is derived from exiting class then all public and protected data of base class can easily accessed into derived class where as private data of base class cannot inherited i.e. this data cannot accessed in derived class.

Consider following example:



In above figure class B is inherited from class A. Therefore class A is called as super class or base class of class B where as class B is called as sub class or derived class.

Need or Features of Inheritance:

1) It is always nice to use existing software instead of creating new one by adding some extra features without modifying it. This can be done by only inheritance.
2) Due to inheritance software cost is reduces.
3) Due to inheritance software developing time is also reduces.
4) Inheritance allows reusability of code.
5) Due to inheritance we can add some enhancement to the base class without modifying it this is due to creating new class from exiting one.

Syntax to derive new class from existing class:

```
  class       derived_class_name :  visibility_mode      base_class_name
{
        Declaration of data members and member functions
};
```

Here;

Class	is keyword to define class.
visibility_mode	is access specifier that tells compiler which type of derivation is either public, private or protected.

Note :
If we do not specify visibility mode then by default the derivation is ***Private***

E.g:

```
class          one
{
        // declaration of data members and member functions
};
class   two : public   one
{
        // declaration of data members and member functions
};
```

In above example, '*class two*' is publicly derived from base '*class one*'.

Therefore '*class two*' has ability to access all the public and protected data of base '*class one*'. Whereas private data of base class cannot be inherited i.e. it can't accessible within member functions of derived class.

Types of Derivation of new class:

We can derive or crate new class by three ways:

1) Public derivation (Public inheritance):

If we derive class publicly then;

1) All the public members of base class become public members for derived class.
2) All the protected members of base class become protected members for derived class.
3) Where as private data of base class can't be inherited.

Following example shows public derivation:

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

In above example '*class B'* is publicly derived from base *'class A'*

2) Private derivation (Private inheritance):

- If we do not specify visibility mode then by default the derivation is **Private.**

If we derive class privately then;

1) All the public members of base class become private members for derived class.
2) All the protected members of base class become private members for derived class.
3) Where as private data of base class can't be inherited.

Following example shows private derivation:

```
class        A
{
        ……….
        ……….
        ……….
};
class        B: private   A
{
        ……….
        ……….
        ……….
};
```

In above example '*class B'* is privately derived from base *'class A'*

3) Protected derivation (Protected inheritance):

If we derive class protectedly then;

1) All the public members of base class become protected members for derived class.
2) All the protected members of base class become protected members for derived class.
3) Whereas private data of base class can't be inherited.

Following example shows private derivation:

```
class        A
{
        ……….
        ……….
        ……….
};
class        B: protected      A
{
        ……….
        ……….
        ……….
};
```

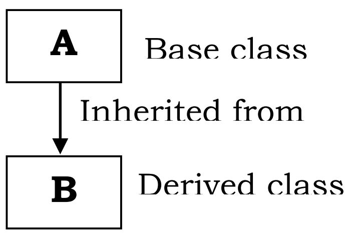
➢ Following table shows types of derivation along with access rights in derived Class:

Derivation Type	Base Class Members	Access in Derived class
Public	Public	Public
	Private	Not accessible
	Protected	Protected
Private	Public	Private
	Private	Not accessible
	Protected	Private
Protected	Public	Protected
	Private	Not accessible
	Protected	Protected

Types (forms) of Inheritance:

1) Single Inheritance:

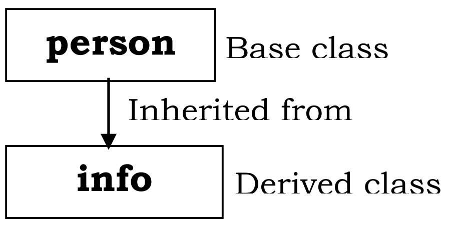
In case of Single inheritance there is only one base class from which we derive only one new class.
e.g.



From above figure we can implement single inheritance as fallow:

```
class    A
{
   Declaration of data members and member functions
};

class    B : private    A
{
     Declaration of data members and member functions
};
```

Implementation of Single Inheritance is as fallow:
Figure:



```
 #include<iostream.h>
#include<conio.h>
class    person
{
         int     age;
         char    name[20], add[20];
         float   sal;
         public:
         void    pget( );
         void    pshow( );
 };
class    info : private    person
{
         float   ht , wt;
         char    blood[5];
         public:
         void    iget( );
```

```
void  person::pshow( )
{
        cout<<"  \n Age= "<<age;
        cout<<"  \n Name= "<<name;
        cout<<"  \n Address= "<<add;
        cout<<"  \n Sal= "<<sal;
}

void  info::iget( )
{
        person::pget ( );  // nesting of member
function
        cout<<"\n Enter height in inch";
        cin>>ht;
        cout<<"\n Enter weight in Kg";
        cin>>wt;
        cout<<"\n Enter Blood group";
```

```
            void    ishow( );
    };
    void      person::pget( )
    {
      cout<<"\n Enter age of person";
      cin>>age;
      cout<<"\n Enter name of person";
      cin>>name;
      cout<<"\n Enter address of person";
      cin>>add;
      cout<<"\n Enter salary of person";
      cin>>sal;
    }
```
	```
	        cin>>blood;
	}

	void  info::ishow( )
	{
	       person::pshow( ); / / nesting of member
	function
	       cout<<"\Height in inch= "<<ht;
	       cout<<"\n Weight in Kg= "<<wt;
	       cout<<"\n  Blood group="<<blood;
	}
	void  main( )
	{
	        info    x;        // object of info class
	        clrscr( );
	        x.iget( );
	        x.ishow( );
	        getch( );
	}
	```

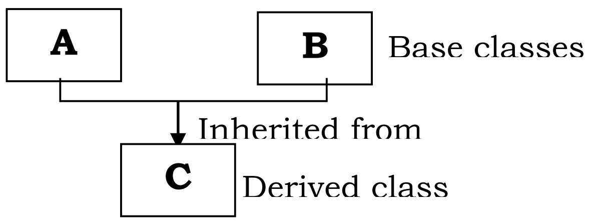
2) Multiple Inheritance:

- In case of multiple inheritance there are multiple base classes from which we derive only one new class.

OR

- Multiple Inheritance is the process of creating a new class from more than one base classes.
- In case of multiple inheritance, the derived class access the data of all base classes.

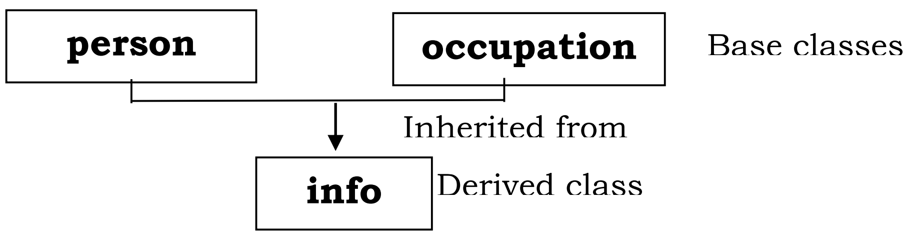
e.g.



From above figure we can implement multiple inheritance as fallow:

```
class     A
{
   Declaration of data members and member functions
};

class     B
{
      Declaration of data members and member functions
};
class     C: public       A, public  B
{
      Declaration of data members and member functions
};
```

Implementation of Multiple Inheritance is as fallow:
Figure:



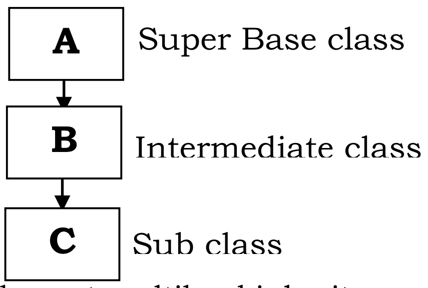
(Program Implementation HOME WORK)

```
#include<iostream.h>
#include<conio.h>
class A
{
	protected:
	int a;
public:
	void get();
};
class B
{
	protected:
	int b;
public:
	void getdata();
};
class C:public A,public B
{
	int c;
public:
	void show();
};
void A::get()
{
	cout<<"enter value of a"<<endl;
	cin>>a;
}
void B::getdata()
{
	cout<<"enter value of b"<<endl;
	cin>>b;
}
void C::show()
{
	c=a+b;
	cout<<"\nadd="<<c;
}
void main()
{
	C x;
	clrscr();
	x.get();
	x.getdata();
	x.show();
	getch();
}
```

3) Multilevel Inheritance:

- In case of multilevel inheritance we can derive new class from another derived classes, further from derived classes we again derive new class and so on.
- Also, there are some levels of inheritances therefore it is called as “Multilevel inheritance”

E.g.

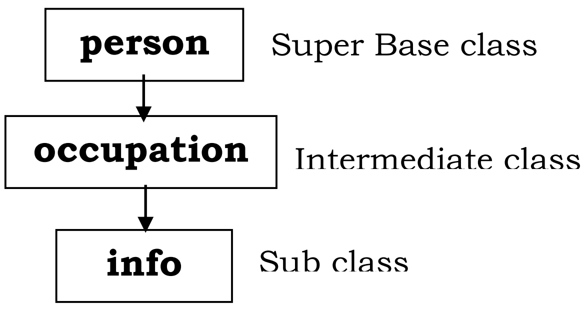


From above figure we can implement multilevel inheritance as fallow:

```
class    A
{
   Declaration of data members and member functions
};

class    B : public    A
{
      Declaration of data members and member functions
};
class    C: public       B
{
      Declaration of data members and member functions
};
```

Implementation of Multilevel Inheritance is as fallow:

Figure:



(Program Implementation HOME WORK)

Example:-

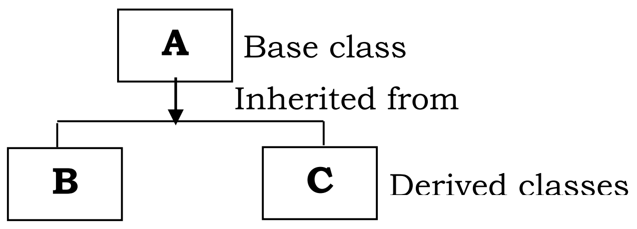
```
#include<iostream.h>
#include<conio.h>
class A
{
protected:
int a;
public:
void get();
};
class B:public A
{
protected:
int b;
public:
void getdata();
};
class C:public B
{
```

```
int c;
public:
void show();
};
void A::get()
{
cout<<"enter value of a"<<endl;
cin>>a;
}
void B::getdata()
{
cout<<"enter value of b"<<endl;
cin>>b;
}
void C::show()
{
c=a+b;
cout<<"\nadd="<<c;
}
void main()
{
C x;
clrscr();
x.get();
x.getdata();
x.show();
getch();
}
```

4) Hierarchical Inheritance:

- In case of hierarchical inheritance there is only one base class from that class we can derive multiple new classes.
- The base class provides all the features for derived classes which are common to all sub classes.
- The hierarchical inheritance comes into picture when certain feature of one level is shared by many other derived classes.

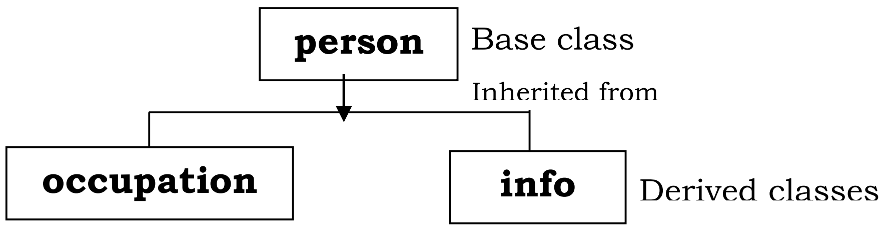
e.g.



From above figure we can implement hierarchical inheritance as fallow:

```
class    A
{
  Declaration of data members and member functions
};

class    B : public    A
{
    Declaration of data members and member functions
};
class    C: public      A
{
    Declaration of data members and member functions
};
```

Implementation of Hierarchical Inheritance is as fallow:
Figure:



(Program Implementation HOME WORK)

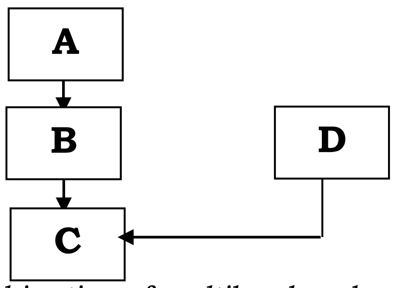
Example:-

```
#include<iostream.h>
#include<conio.h>
class A
{
   protected:
   int a,b,c;
   public:
   void get()
   {
   cout<<"enter value of a&b";
   cin>>a>>b;
   }
};
class B:public A
{
  public:
  void getdata()
  {
  c=a+b;
  cout<<"add="<<c<<endl;
  }
};
class C:public A
{
  public:
  void show()
  {
  c=a*b;
  cout<<"mul="<<c<<endl;
  }
};
void main()
{
B p;
  C x;
  clrscr();
  p.get();
  p.getdata();
  x.get();
  x.show();
  getch();
}
```

5) Hybrid Inheritance:

- Hybrid inheritance is the combination of two or more forms of inheritances.
- To implement hybrid inheritance, we have to combine two or more forms of inheritances and such a combination of different forms of inheritances is called "Hybrid inheritance".

E.g.

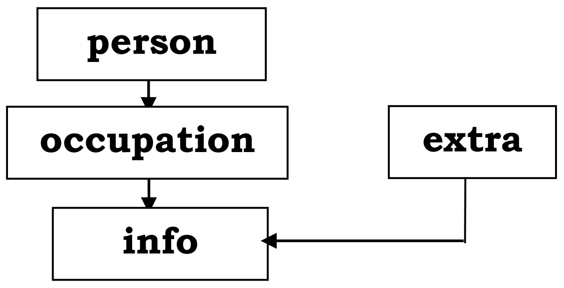


The above inheritance is a combination of multilevel and multiple inheritances therefore it is "Hybrid Inheritance".

From above figure we can implement hybrid inheritance as fallow:

```
class    A
{
         Declaration of data members and member functions
};
class    B : public    A
{
         Declaration of data members and member functions
};
class    D
{
         Declaration of data members and member functions
};

class    C : public    B, public D
{
         Declaration of data members and member functions
};
```

<u>Implementation of Hybrid Inheritance is as fallow:</u>
Figure:



The above inheritance is a combination of multilevel and multiple inheritances therefore it is "Hybrid Inheritance".

(Program Implementation HOME WORK)

```
#include<iostream.h>
#include<conio.h>
class A
{
protected:
int a,b;
public:
void get()
{
cout<<"enter value of a&b";
cin>>a>>b;
}
};
```

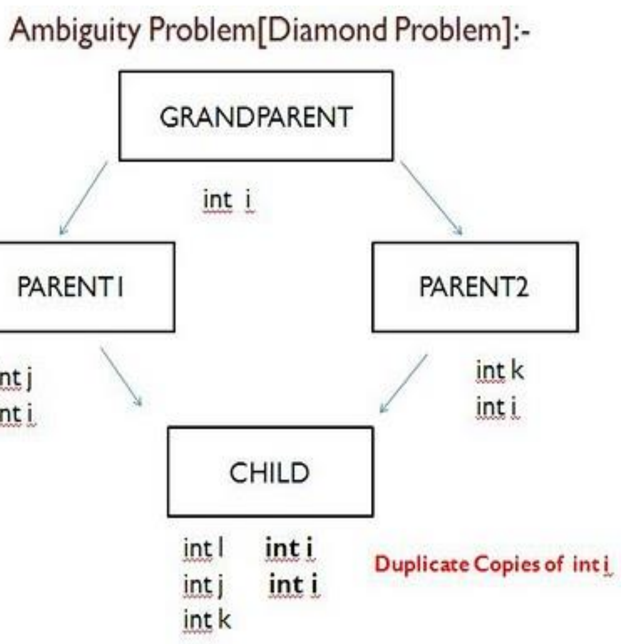
```
class B:public A
{
public:
void show()
{
cout<<"a="<<a<<endl<<"b="<<b<<endl;
}
};


class D
{
protected:
int x,y;
public:
void getdata()
{
cout<<"enter value of x&y";
cin>>x>>y;
}
};
class C:public B,public D
{
public:
void display()
{
cout<<"x="<<x<<endl<<"y="<<y<<endl;
}
};
void main()
{
clrscr();
C q;
q.get();
q.show();
q.getdata();
q.display();
getch();
}
```

6) Multipath Inheritance: (Virtual Inheritance)

- Multipath inheritance is a special type of hybrid inheritance, which is combination of hierarchical inheritance and multiple inheritance.
- It is also called as Virtual Inheritance.
- In Multipath Inheritance there is a one base class ‘*GRANDPARENT*’. Two derived classes ‘*PARENT1*’ and ‘*PARENT2*’ which are inherited from ‘*GRANDPARENT*’. Third Derived class ‘*CHILD*’ which is inherited from both ‘*PARENT1*’ and ‘*PARENT2*’.

Problem in Multipath Inheritance

- There is an ambiguity problem. When you run program with such type inheritance. It gives a compile time error [Ambiguity]. If you see the structure of Multipath Inheritance then you find that there is shape like Diamond.



Ambiguity Problem in Multipath inheritance (Virtual Inheritance):

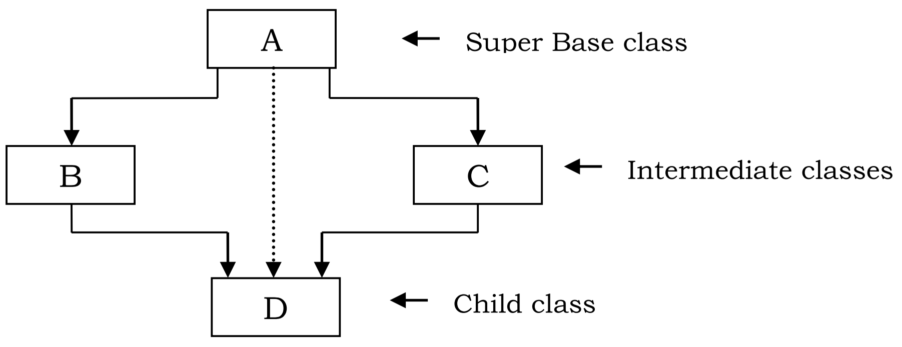
- Suppose *‘GRANDPARENT’* has a data member **‘int i**'. *‘PARENT1’* has a data member **‘int j’**. Another ‘*PARENT2’* has a data member **‘int k’**. *‘CHILD’* class which is inherited from *‘PARENT1’* and ‘*PARENT2’*. It has a data member **‘int l’**.
  Therefore ‘CHILD’ class have data members
  int l (CHILD class data member its own)
  int j (one copy of data member PARENT1)
  int k (one copy of data member PARENT2)
  int i (two copies of data member GRANDPARENT)

- This is ambiguity problem. The *‘CHILD’* class has two copies of Base class.
- There are two duplicate copies of **‘int i'** of base class. One copy through PARENT1 and another copy from PARENT2. This problem is also called as DIAMOND Problem.

Solution of Ambiguity Problem in multipath inheritance:

- To avoid duplicate copies of Base class, we declare intermediary inherited classes with the prefix keyword "virtual".
- We use “virtual” keyword so it is also called as "Virtual Inheritance".
- By making intermediate classes as “virtual”, it avoids duplicate copies of Base class members in child class i.e. only one copy of base class member is sent to child class. **<u>And such intermediately classes are called as “Virtual Base” classes.</u>**

<u>Following figure shows multipath inheritance:</u>



In above, Figure *'Class A'* is super base class where as *'class B'* and *'class C'* are intermediate classes which are derived from base *'class A'*. Further *'class C'* is derived from two base classes *'B' and 'class C'*.
To avoid multiple copies of data members of 'class A' in 'class D', we make 'class B' and 'class C' as virtual base class for Class D.

Following program shows the implementation of multipath inheritance with virtual base class:

```
#include<iostream.h>
#include<conio.h>
class      A
{
    protected:
    int  i;
};
class    B: virtual  public  A    //virtual base
class
{
    protected:
    int  j;
};
class   C: virtual  public A     //virtual base
class
{
    protected:
    int k;
};
```

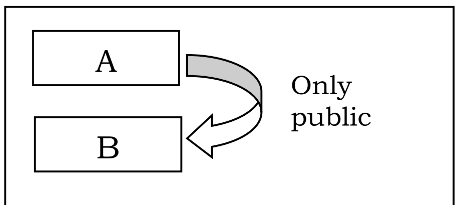
```
class D : public   B,   public C
{
    int l,z;
    public:
    void get( );
};
void  D::get( )
{
          cout<<"\nEnter four Numbers: ";
          cin>>i>>j>>k>>l;
          z=i+j+k+l;
          cout<<"\nAddition= "<<z;
}
void  main( )
{
           clrscr( );
           D       p;
           p.get( );
           getch( );
}
```

Container Class:

- The class contain object of another class as data member is called "container class"
- The concept of taking or declaring object of one class as data member inside another class is called as "Composition"
- The container class only accepts public data of existing class. It cannot accept private and protected data of existing class.
- Following example shows container class:

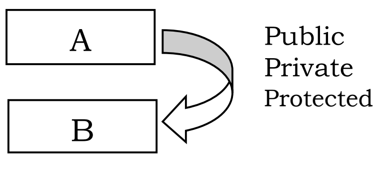
```
class           A
{
          …………
          …………
          …………
}
class           B          //  container class
{
          A        x;     //object of class A as data member
          …………
          …………
          …………
};
```

- Following figure shows concept of container class , friend class and inheritance:



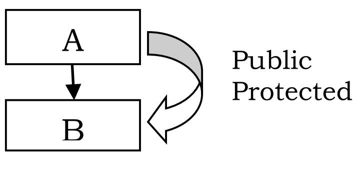
In above fig. class 'B' is container class. Therefore it accept only public data of class 'A'

(About container Class)



In above fig. class 'B' is friend class of 'A'. Therefore it accept all data i.e. public, private and protected data of class 'A'

(About Friend Class)



In above fig. class 'B' is derived class from class 'A'. Therefore it accept public and protected data of class 'A'

(About Inheritance)

Following program shows implementation of container class:

```
#include<iostream.h>
#include<conio.h>
class     emp
{
   public:
    int      empno;
    char    name[20];
    float    sal;
};
class      accept    // container class
{
       emp   p;     //object of 'emp' class
   public:
       void get( );
       void show( );
};
void accept::get( )     //access data by 'p'
object
{
  cout<<"\n Enter emp No: ";
  cin>>p.empno;
  cout<<"\n Enter emp Name: ";
  cin>>p.name;
  cout<<"\n Enter emp Salary: ";
  cin>>p.sal;
          }
void accept::show( )      //access data by 'p'
object
{
             cout<<"\n Emp No: "<<p.empno;
             cout<<"\n Emp Name: "<<p.name;
             cout<<"\n Salary: "<<p.sal;
}
void   main( )
{
              clrscr( );
              accept  x;
              x.get( );
              x.show( );
              getch( );
}
```

Invocation of constructor and destructor in Inheritance:

➢ Invocation of constructor in inheritance:

- We know that, constructor is used to initialize the object.
- Without initializing base class members, the derived class members cannot initializes therefore compiler automatically (implicitly) invoke base class constructor first and then the derived class constructor is invoked.
- In short, base class constructor invoked first and then derived class constructor invoked. That is order of execution of constructor in inheritance is **base to derive**.
- *Note that: If derived class contains parameterized constructor then it is necessary for the base class to have default constructor.*
- Every constructor in derived class first implicitly make call to default constructor of base class.

➢ Invocation of Destructor in inheritance:

- We know that, destructor is used to free or release the memory occupied by objects.
- As we have seen, the constructors are executed from base class to derived class whereas the destructors are executed in the reverse order i.e. from **derived class to base class**.

Following program shows invocation of constructor & destructor in inheritance:

```
#include<iostream.h>
#include<conio.h>
class one
{
    public:
    one( )
    {
      cout<<"\nBase class default constructor ";
    }
    ~one( )
    {
      cout<<"\nBase class destructor ";
    }
};
class two : public one
{
  public:
     two( )
     {
    cout<<"\nDerived class default
constructor.";
    }
  two(int     x)
  {
     cout<<"\nDerived class Parameterized
constructor.";
  }
  ~two( )
   {
     cout<<"\nDerived class Destructor";
   }
};
void main( )
{
   clrscr( );
   two     a;
   two     b(10);
   getch( );
}
OUTPUT:   Base class default constructor
          Derived class default constructor.
          Base class default constructor
          Derived class Parameterized constructor.
          Derived class Destructor
          Base class destructor
          Derived class Destructor
          Base class destructor
```
(Note: Check output by pressing Alt+f5 key also)

Virtual destructor:

- C++ language allows to declare a destructor as 'virtual' by preceding the declaration of destructor with keyword 'virtual'
- By declaring base class destructor as 'virtual' then the base class offers to redefine of the destructor in any derived classes.
- If an object is destroyed explicitly by using 'delete' operator to the base class pointer to object then base class destructor function is called.
- When we declare base class destructor as 'virtual' then automatically it makes all derived class destructor as 'virtual' even though they do not have the same name as the base class destructor. Now, if an object in the hierarchy is destroyed explicitly by using 'delete' operator to base class pointer or to derived class object then destructor for appropriate class is called.
- Remember that when a derived class object is destroyed then base class part of the derived class object is also destroyed.
- The base class destructor is automatically executes <u>after</u> the derived class destructor.
- The conclusion is that base class must define a destructor which is used in cases where derived classes do not define their own destructors.
- Syntax to declare virtual destructor:

```
virtual    ~    class_Name( );
```

Here,

'virtual'	is keyword used to make function as virtual
~	is tiled symbol used to declare destructor.
'class_name'	is name of the class.

<u>Following program demonstrate the concept of virtual destructor:</u>

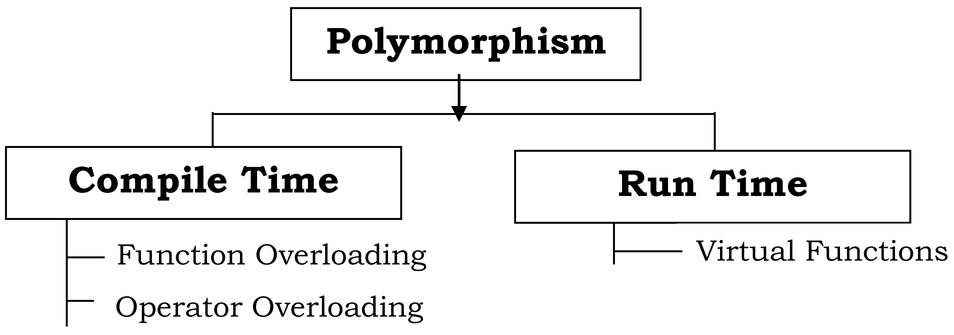
```
#include<iostream.h>
#include<conio.h>
class   base
{
    public:
    virtual void get( )=0;            //pure virtual mem.fun
    virtual ~base( );         //virtual destructor of base
class
};
class derive : public base
{
    public:
    void get( );
   virtual ~derive( );    //virtual destructor of derived
class
};
void derive::get( )
{
  cout<<"\n Derived class member function  ";
}

base::~base( )
{
  cout<<"\nBase class Destructor ";
}
```

```
derive::~derive( )
{
  cout<<"\nDerived class Destructor ";
}
void main( )
{
   clrscr( );
   base *p;
   derive q;
   p=&q;
   p->get( );
   delete p;
   getch( );
}
```

OUTPUT:

```
Derived class member function
Derived class Destructor
Base class Destructor
```

VIRTUAL FUNCTIONS:



Introduction:

We know that, while doing programming the functions are gets selected by the compiler at two times.

1) At Compile time of program 2) At runtime of program

Definition:

- "The functions which are get selected (Invoked) by the compiler at run time of program are called Virtual Functions"
- When both classes i.e. base class as well as derived class having same function prototype at that time functions declared in the base class must be declared as virtual such that compiler select such virtual function at run time.
- To declare the function as 'virtual', we have to just specify keyword 'virtual' at the declaration of function.
- When function is made virtual, C++ compiler determines which function is used at run time, based on type of object pointed to the base class pointer.
- Runtime polymorphism is achieved with the help of virtual functions.

Characteristics/Features/Rules of Virtual Functions:

1) Virtual function must be member function of class.
2) The keyword 'virtual' should not be repeated in the definition if the definition occurs outside the class.
3) Virtual function can't be static
4) Virtual functions are accessed by using objects pointer only.
5) Virtual function can be friend of another class.
6) If we declare function as virtual then we must have to define it, even though it may not be used.
7) We cannot have virtual constructor but we have virtual destructors.
8) If a virtual function is defined in base class then it need not be necessary to redefine it in derived class.

Syntax to declare Virtual function:

```
virtual    return_type   funct_Name(arg1,arg2,……,argN);
```

Here,

virtual is keyword to make function as virtual

return_type is value return from function.

funct_Name is name of function which is an Identifier.

E.g. Following program shows implementation of virtual function.

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
class   student
{
  int    roll;
  char   name[20],add[20];
  public:
   virtual  void  get( );    // declaration of virtual
function
   virtual  void  show( );  // declaration of virtual
function
};
class result : public student
{
    int mark[6],total;
    float per;
```

```
void student::show( )
{
   cout<<"\n Roll Number = "<<roll;
   cout<<"\n Name = "<<name;
   cout<<"\n Address= "<<add;
}
void result::get( )
{
   int i;
   total=0;
   student::get( );      // nesting of mem. function
   cout<<"\nEnter mark of 6 subjects ";
   for(i=0;i<=5;i++)
   {
     cin>>mark[i];
     total=total+mark[i];
```

```
    public:
    void get( );
    void show( );
};
void student::get( )
{
  cout<<"\n Enter roll Number ";
  cin>>roll;
  cout<<"\n Enter Name ";
  cin>>name;
  cout<<"\n Enter Address";
  cin>>add;
}
```

```
  }
  per=(float)total/6;
}
void result::show( )
{
  student::show( );    // nesting of mem. function
  cout<<"\n Total Marks = "<<total;
  cout<<"\n Persentage = "<<per;
}
void main( )
{
  student *p;
  result q;
  p=&q;
  p->get( );
  p->show( );
  getch( );
}
```

PURE VIRTUAL FUNCTIONS: (Do nothing function or Dummy functions)

- "A pure virtual function is a virtual function that function does not have any definition related to base class".
- In such cases the compiler requires each derived class to either define the function or re-declare it as a pure virtual function.
- "The virtual function equated to zero such virtual function is called pure virtual function"
- Pure virtual functions are like empty bucket that the derived class functions have to fill it.
- Pure virtual function does not have definition. These functions are defined by its derived classes according their own manner.
- *NOTE: The class containing pure virtual function is called as "Abstract Class" and object of abstract class cannot be instantiated or created.*
- An abstract class cannot used to declare any object of its own class.

Syntax to declare pure virtual function:

```
virtual     return_type    funct_Name(arg1,arg2,……,argN) = 0 ;
```

Here,

virtual	is keyword to make function as virtual
return_type	is value return from function.
funct_Name	is name of function which is an Identifier.

NOTE: In above syntax virtual function is equated to zero that's why it is pure virtual function or do nothing function.

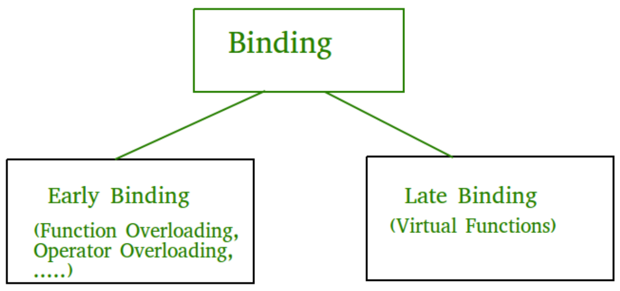
```
#include<iostream.h>
#include<conio.h>
class   first
{
     public:
     int   no;
     virtual void get()=0;       //pure virtual funct.
     virtual void show()=0;     //pure virtual funct.
 };
class   second : public first
{
        char   name[20];
        public:
        void   get( );
        void   show();
};
void  second::get( )
{
      cout<<"Enter Roll no ";
      cin>>no;
      cout<<"Enter your name ";
      cin>>name;
}
```

```
void  second::show( )
{
      cout<<"\n RollNo = "<<no;
      cout<<"\n your name =
"<<name;
}

void  main( )
{
     first     *x;
     second   y;
     x=&y;
     x->get( );
     x->show( );
     getch( );
}
```

In above example, *class 'first'* contains the pure virtual function therefore it is called as '**Abstract class**' and **object** of **abstract class** cannot be created or instantiated.

➢ <u>Early binding and Late binding in C++</u>

Binding refers to the process of converting identifiers (such as variable and performance names) into addresses. Binding is done for each variable and functions. For functions, it means that matching the call with the right function definition by the compiler. It takes place either at compile time or at runtime.



1. **<u>Early Binding (compile-time time polymorphism)</u>** As the name indicates, compiler (or linker) directly associate an address to the function call. It replaces the call with a machine language instruction that tells the mainframe to leap to the address of the function.

By default early binding happens in C++. Late binding (discussed below) is achieved with the help of **virtual keyword).**

```
#include<iostream.h>
#include<conio.h>
class Base
{
public:
   void show() { cout<<" In Base \n"; }
};

class Derived: public Base
{
public:
   void show() { cout<<"In Derived \n"; }
};

int main(void)
{
   Base *bp = new Derived;

                        // The function call decided at compile time (compiler sees type of pointer and
                        //calls base class function.
   bp->show();

   return 0;
}
```

<u>Output:</u>

```
In Base
```

2. **<u>Late Binding : (Run time polymorphism</u>**) In this, the compiler adds code that identifies the kind of object at runtime then matches the call with the right function definition (Refer this for details). This can be achieved by declaring a virtual function.

```
#include<iostream.h>
#include<conio.h>
class Base
{
public:
   virtual void show() { cout<<" In Base \n"; }
};

class Derived: public Base
```

```
{
public:
   void show() { cout<<"In Derived \n"; }
};

int main(void)
{
   Base *bp = new Derived;
   bp->show();  // RUN-TIME POLYMORPHISM
   return 0;
}
```

Output:

```
In Derived
```

Early Binding vs Late Binding	
The process of using the class information to resolve method calling that occurs at compile time is called Early Binding.	The process of using the object to resolve method calling that occurs at run time is called the Late Binding.
Time of Binding	
Early Binding happens at compile time.	Late Binding happens at run time.
Functionality	
Early Binding uses the class information to resolve method calling.	Late Binding uses the object to resolve method calling.
Synonyms	
Early Binding is also known as static binding..	Late Binding is also known as dynamic binding.
Occurrence	
Overloading methods are bonded using early binding.	Overridden methods are bonded using late binding.
Execution Speed	
Execution speed is faster in early binding.	Execution speed is lower in late binding.

Assignment: 03

1. What is static and dynamic binding? How static and dynamic binding is achieved in C++?
2. What is reusability? How it is possible in C++?
3. What is inheritance? List out its significance in developing program (software).
4. Explain accessibility of base class members in derived class.
5. In how many ways we derive new class? Explain each with one example.
6. What is inheritance? Explain different types (forms) of inheritance with example.
7. When do we make class as “virtual”? Explain with suitable example.
8. How constructor and destructor are invoked in inheritance?
9. What is container class? Explain it with suitable example.
10. Write short note on: “Virtual Destructor”
11. Differentiate between container class and inheritance.
12. What is Virtual Function? List out its characteristics and give one example.
13. What is pure virtual function? List out its characteristics and give one example.