



LECTURE-26

Inheritance:

Reaccessability is yet another feature of OOP's. C++ strongly supports the concept of reusability. The C++ classes can be used again in several ways. Once a class has been written and tested, it can be adopted by another programmers. This is basically created by defining the new classes, reusing the properties of existing ones. The mechanism of deriving a new class from an old one is called 'INHERTENCE'. This is often referred to as IS-A' relationship because very object of the class being defined "is" also an object of inherited class. The old class is called 'BASE' class and the new one is called'DERIEVED'class.

Defining Derived Classes

A derived class is specified by defining its relationship with the base class in addition to its own details. The general syntax of defining a derived class is as follows:

```
class d_classname : Access specifier baseclass name
{
    ___ // members of derived class
};
```

The colon indicates that the a-class name is derived from the base class name. The access specifier or the visibility mode is optional and, if present, may be public, private or protected. By default it is private. Visibility mode describes the status of derived features e.g.

```
class xyz //base class
{
members of xyz
};
class ABC : public xyz //public derivation
{
members of ABC
};
class ABC : XYZ //private derivation (by default)
{
members of ABC
};
```

In the inheritance, some of the base class data elements and member functions are

inherited into the

derived class. We can add our own data and member functions and thus extend the

base class. Inheritance, when used to modify and extend the capabilities of the existing classes. When a class inherits from a single base class, it is known as single inheritance. Following program shows the single inheritance using public derivation. becomes a very powerful tool for incremental program development.

```
#include<iostream.h>
```

```
#include<conio.h>
```

```
class worker
```

```
{
```

```

int age;
char name [10];
public:
void get ( );
};
void worker : : get ( )
{
cout <<"yout name please"
cin >> name;
cout <<"your age please" ;
cin >> age;
}
void worker :: show ( )
{
        cout <<"In My name is :"<<name<<"In My age is :"<<age;
}
class manager :: public worker //derived class (publicly)
{
int now;
public:
void get ( ) ;
void show ( ) ;
};
void manager : : get ( )
{
worker : : get ( ) ; //the calling of base class input fn.
cout << "number of workers under you";
cin >> now;
cin>>name>>age;
} ( if they were public )
void manager :: show ( )
{
worker :: show ( ) ; //calling of base class o/p fn.
cout <<"in No. of workers under me are: " << now;
}
main ( )
{
clrscr ( ) ;
worker W1;
manager M1;
M1 .get ( ) ;
M1.show ( ) ;
}

```

If you input the following to this program:

Your name please

Ravinder

Your age please

27

number of workers under you

30

Then the output will be as follows:

My name is : Ravinder

My age is : 27

No. of workers under me are : 30

The following program shows the single inheritance by private derivation.

```
#include<iostream.h>
#include<conio.h>
class worker //Base class declaration
{
int age;
char name [10] ;
public:
void get ( ) ;
void show ( ) ;
};
void worker :: get ( )
{
cout << "your name please" ;
cin >> name;
cout << "your age please";
cin >>age;
}
void worker : show ( )
{
    cout << "in my name is: " <<name<< "in" << "my age is : " <<age;
}
class manager : worker //Derived class (privately by default)
{
int now;
public:
void get ( ) ;
void show ( ) ;
};
void manager :: get ( )
{
worker :: get ( ) ; //calling the get function of base
cout << "number of worker under you"; class which is
cin >> now;
}
void manager :: show ( )
{
worker :: show ( ) ;
cout << "in no. of worker under me are : " <<now;
}
main ( )
{
```

```

clrscr ( ) ;
worker wl ;
manager ml;
ml.get ( ) ;
ml.show ( ) ;
}

```

The following program shows the single inheritance using protected derivation

```

#include<conio.h>
#include<iostream.h>
class worker //Base class declaration
{ protected:
int age; char name [20];
public:
void get ( ) ;
void show ( ) ;
};
void worker :: get ( )
{
cout >> "your name please";
cin >> name;
cout << "your age please";
cin >> age;
}
void worker :: show ( )
{
cout << "in my name is: " << name << "in my age is " <<age;
}
class manager:: protected worker // protected inheritance
{
int now;
public:
void get ( ) ;
void show ( ) ;
};
void manager : : get ( )
{
cout << "please enter the name ln";
cin >> name;
cout<< "please enter the age ln"; //Directly inputting the data
cin >> age; members of base class
cout << " please enter the no. of workers under you:";
cin >> now;
}
void manager : : show ( )

{
cout << "your name is : " <<name<<" and age is : " <<age;
cout <<"ln no. of workers under your are : " <<now;
}
main ( )
{
clrscr ( ) ;
manager ml;
ml.get ( ) ;
}

```

```

    cout << "\n \n";
    ml.show ();
}

```

Making a Private Member Inheritable

Basically we have visibility modes to specify that in which mode you are deriving the another class from the already existing base class. They are:

- a. **Private:** when a base class is privately inherited by a derived class, 'public members' of the base class become private members of the derived class and therefore the public members of the base class can be accessed by its own objects using the dot operator. The result is that we have no member of base class that is accessible to the objects of the derived class.
- b. **Public:** On the other hand, when the base class is publicly inherited, 'public members' of the base class become 'public members' of derived class and therefore they are accessible to the objects of the derived class.
- c. **Protected:** C++ provides a third visibility modifier, protected, which serve a little purpose in the inheritance. A member declared as protected is accessible by the member functions within its class and any class immediately derived from it. It cannot be accessed by functions outside these two classes.

The below mentioned table summarizes how the visibility of members undergo modifications

when

they are inherited

Base Class Visibility	Derived Class Visibility		
Public	Private	Protected	
Private	X	X	X
Public	Public	Private	Protected
Protected	Protected	Private	Protected

The private and protected members of a class can be accessed by:

- a. A function i.e. friend of a class.
- b. A member function of a class that is the friend of the class.
- c. A member function of a derived class.

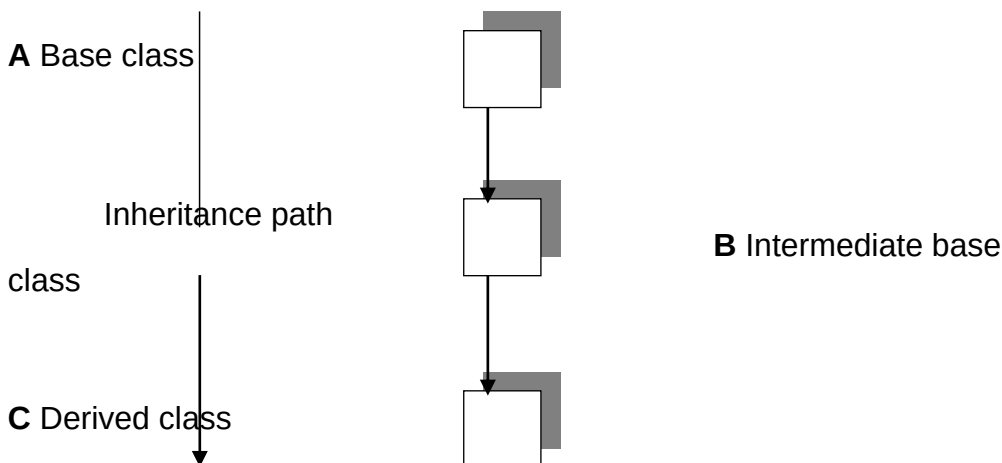
Student Activity

1. Define Inheritance. What is the inheritance mechanism in C++?
2. What are the advantage of Inheritance?
3. What should be the structure of a class when it has to be a base for other classes?

LECTURE-27

Multilevel Inheritance

When the inheritance is such that, the class A serves as a base class for a derived class B which in turn serves as a base class for the derived class C. This type of inheritance is called 'MULTILEVEL INHERITENCE'. The class B is known as the 'INTERMEDIATE BASE CLASS' since it provides a link for the inheritance between A and C. The chain ABC is called 'ITNHERITENCE*PATH' for e.g.



The declaration for the same would be:

```
Class A
{
//body
}
Class B : public A
{
//body
}
Class C : public B
{
//body
}
```

This declaration will form the different levels of inheritance.

Following program exhibits the multilevel inheritance.

```
#include<iostream.h>
#include<conio.h>
class worker // Base class declaration
{
int age;
char name [20] ;
public;
void get( ) ;
```

```

        void show() ;
    }

void worker: get ( )
{
cout << "your name please" ;
cin >> name;
cout << "your age please" ;
}

void worker :: show ( )
{
    cout << "In my name is : " <<name<< " In my age is : " <<age;
}
class manager : public worker //Intermediate base class derived
{ //publicly from the base class
int now;
public:
void get ( ) ;
void show() ;
};

void manager :: get ( )
{
worker ::get ( ) ; //calling get ( ) fn. of base class
cout << "no. of workers under you:";
cin >> now;
}
void manager :: show ( )
{
    worker :: show ( ) ; //calling show ( ) fn. of base class
cout << "In no. of workers under me are: " << now;
}

class ceo: public manager //declaration of derived class
{ //publicly inherited from the
int nom; //intermediate base class
public:
void get ( ) ;
void show ( ) ;
};

void ceo :: get ( )
{
manager :: get ( ) ;
cout << "no. of managers under you are:"; cin >> nom;
}

void manager :: show ( )
{
cout << "In the no. of managers under me are: In";
cout << "nom;
}

```

```

main ()
{
clrscr () ;
ceo cl ;
cl.get () ; cout << "\n\n";
cl.show () ;
}

```

Worker

Private:

```

int age;
char name[20];

```

Protected:

Private:

```

int age;
char name[20];

```



Manager:Worker

Private:

```

int now;

```

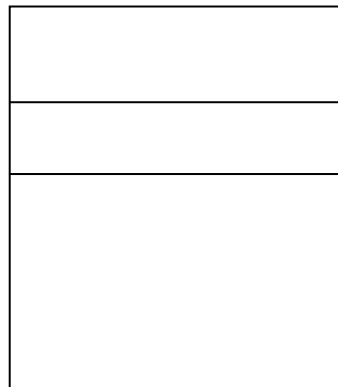
Protected:

Public:

```

void get()
void show()
worker ::get()
worker ::get()

```



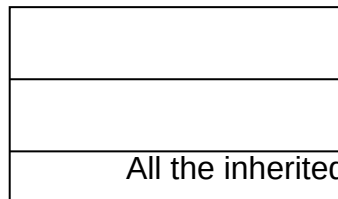
Ceo: Manager

Public:

Protected:

Public:

members



Multiple Inheritances

A class can inherit the attributes of two or more classes. This mechanism is known as 'MULTIPLE INHERITENCE'. Multiple inheritance allows us to combine the features of several existing classes as a starting point for defining new classes. It is like the child

inheriting

the physical feature of one parent and the intelligence of another. The syntax of the derived class is

as follows:

Class base1 Class base2

```
{ {
//body1 // body2
} }
```

Class derived : visibility base1, visibility base2

```
{
//body3
}
```

Where the visibility refers to the access specifiers i.e. public, private or protected. Following program shows the multiple inheritance.

```
#include<iostream.h>
```

```
#include<conio . h>
```

```
class father //Declaration of base class1
```

```
{
int age ;
char name [20] ;
public:
void get ( ) ;
void show ( ) ;
};
void father :: get ( )
```

```
{
cout << "your father name please";
cin >> name;
cout << "Enter the age";
cin >> age;
}
```

```
void father :: show ( )
```

```
{
cout<< "In my father's name is: ' <<name<< "In my father's age is:<<age;
}
```

```
class mother //Declaration of base class 2
```

```
{
char name [20] ;
int age ;
```

```

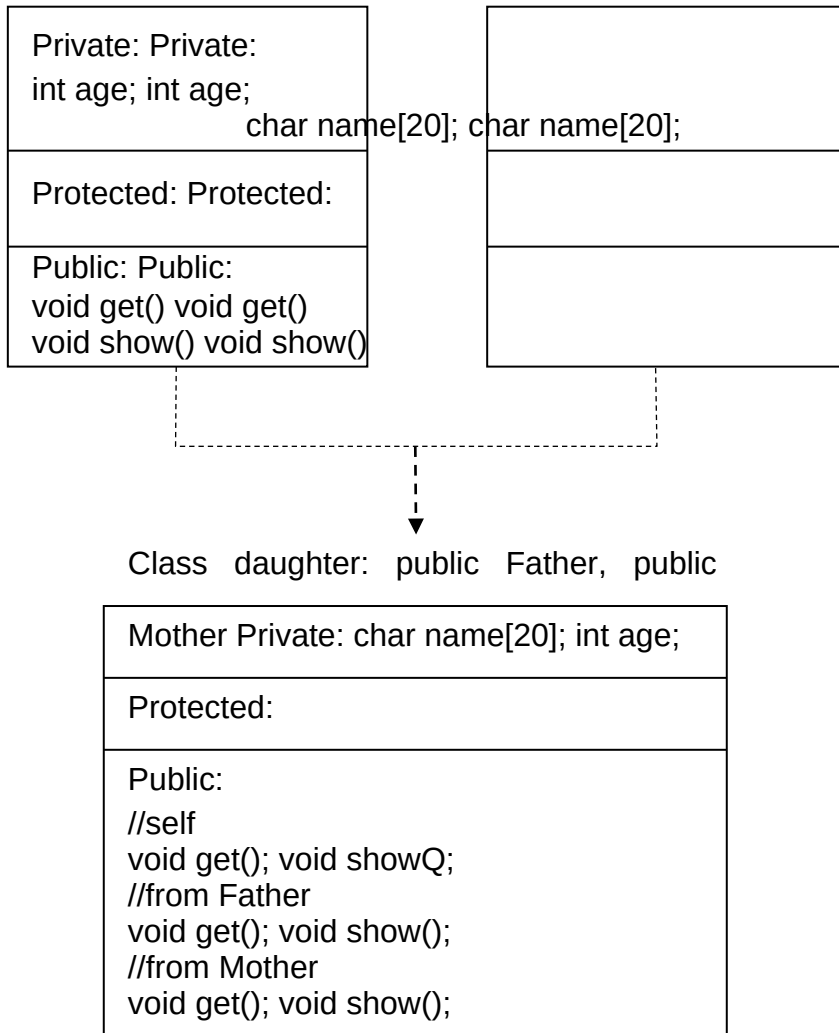
public:
void get ( )
{
cout << "mother's name please" << "In";
cin >> name;
cout << "mother's age please" << "in";
cin >> age;
}
void show ( )
{
cout << "In my mother's name is: " <<name;
cout << "In my mother's age is: " <<age;
}
class daughter : public father, public mother //derived class inheriting
{ //publicly
char name [20] ; //the features of both the base class
int std;
public:
void get ( ) ;
void show ( ) ;
};
void daughter :: get ( )
{
father :: get ( ) ;
mother :: get ( ) ;
cout << "child's name: ";
cin >> name;
cout << "child's standard";
cin >> std;
}
void daughter :: show ( )
{
father :: show ( ) ;
mother :: show ( ) ;
cout << "In child's name is : " <<name;
cout << "In child's standard: " << std;
}
main ( )
{
clrscr ( ) ;
daughter d1;
d1.get ( ) ;
d1.show ( ) ;

}

```

Diagrammatic Representation of Multiple Inheritance is as follows:

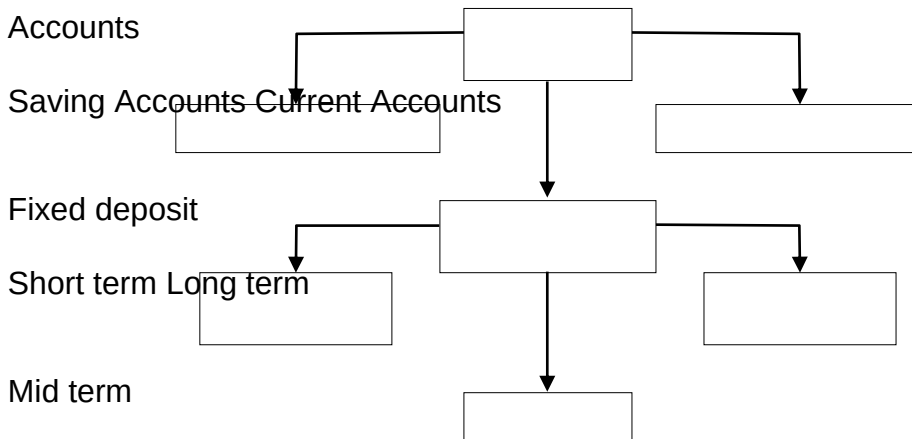
Father Mother



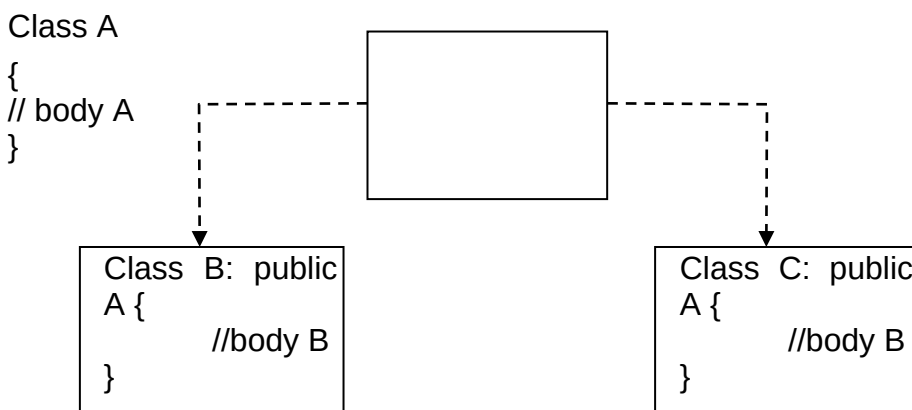
LECTURE-28

Hierarchical Inheritance

Another interesting application of inheritance is to use it as a support to a hierarchical design of a class program. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below that level for e.g.



In general the syntax is given as



In C++, such problems can be easily converted into hierarchies. The base class will include all the features that are common to the subclasses. A sub-class can be constructed by inheriting the features of base class and so on.

// Program to show the hierarchical inheritance

```
#include<iostream.h>
# include<conio. h>
class father //Base class declaration
{
int age;
char name [15];
public:
void get ( )
{
cout<< "father name please"; cin >> name;
```

```

        cout<< "father's age please"; cin >> age;
    }
    void show ( )
    {
        cout << "In father's name is ": "<<name;
    cout << "In father's age is: "<< age;
    }
};
class son : public father //derived class 1
{
    char name [20] ;
    int age ;
    public;
    void get ( ) ;
    void show ( ) ;
};
void son :: get ( )

{
    father :: get ( ) ;
        cout << "your (son) name please" << "in"; cin >>name;
    cout << "your age please" << "In"; cin>>age;
}
void son :: show ( )
{
    father :: show ( ) ;
    cout << "In my name is : " <<name;
    cout << "In my age is : " <<age;
}
class daughter : public father //derived class 2.
{
    char name [15] ;
    int age;
    public:
    void get ( )
    {
        father :: get ( ) ;
            cout << "your (daughter's) name please In" cin>>name;
            cout << "your age please In"; cin >>age;
    }
}
void show ( )
{
    father :: show ( ) ;
    cout << "in my father name is: " << name << "
    In and his age is : "<<age;
}
};
main ( )
{
    clrscr ( ) ;
}

```

```

        son S1;
        daughter D1 ;
        S1. get ( ) ;
        D1. get ( ) ;
        S1 .show( ) ;
        D1. show ( ) ;
    }

```

Hybrid Inheritance

There could be situations where we need to apply two or more types of inheritance to design a program. Basically Hybrid Inheritance is the combination of one or more types of the inheritance. Here is one implementation of hybrid inheritance.

//Program to show the simple hybrid inheritance

```

#include<iostream.h>
#include<conio.h>
class student //base class declaration
{
protected:
int r_no;
public:
void get_n (int a)
{
r_no =a;
}
void put_n (void)
{
cout << "Roll No. : " << r_no;
cout << "\n";
}
};
class test : public student
{ //Intermediate base class
protected : int parti, par 2;

public :
void get_m (int x, int y) {
parti = x; part 2 = y; }
void put_m (void) {
cout << "marks obtained: " << "\n"
<< "Part 1 = " << part1 << "\n"
<< "Part 2 = " << part2 << "\n";
}
};
class sports // base for result
{
protected : int score;
public:
void get_s (int s) {
score = s }
void put_s (void) {
cout << "sports wt. : " << score << "\n\n";
}
};

```

```

    }
};
class result : public test, public sports //Derived from test
& sports
{
int total;
public:
void display (void);
};

void result :: display (void)
{
total = part1 + part2 + score;
put_n ();
put_m ();
put_S ();
        cout << "Total score: " <<total<< "\n"
}
main ()
{
clrscr ();
result S1;
S1.get_n (347) ;
S1.get_m (30, 35);
S1.get_s (7) ;
S1.dciplay () ;
}

```

Student Activity

1. What is the major use of multilevel Inheritance?
2. How are arguments sent to the base constructors in multiple inheritance? Whose responsibility is it.
3. What is the difference between hierarchical and hybrid Inheritance.

LECTURE-29

Virtual Base Classes

We have just discussed a situation which would require the use of both multiple and multi level inheritance. Consider a situation, where all the three kinds of inheritance, namely multi-level, multiple and hierarchical are involved.

Let us say the 'child' has two direct base classes 'parent1' and 'parent2' which themselves

has a

common base class 'grandparent'. The child inherits the traits of 'grandparent' via two separate paths. It can also be inherit directly as shown by the broken line. The grandparent is

sometimes

referred to as 'INDIRECT BASE CLASS'. Now, the inheritance by the child might cause some problems. All the public and protected members of 'grandparent' are inherited into 'child'

twice, first

via 'parent1' and again via 'parent2'. So, there occurs a duplicacy which should be avoided.

The duplication of the inherited members can be avoided by making common base class as

the

virtual base class: for e.g.

```
class g_parent
{
//Body
};
class parent1: virtual public g_parent
{
// Body
};
class parent2: public virtual g_parent
{
// Body
};
class child : public parent1, public parent2
{
// body
};
```

When a class is virtual base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exists between virtual base class and derived class. Note that keywords 'virtual' and 'public' can be used in either order.

//Program to show the virtual base class

```
#include<iostream.h>
#include<conio . h>
class student // Base class declaration
{
protected:
int r_no;
public:
void get_n (int a)
{ r_no = a; }
void put_n (void)
{ cout << "Roll No. " << r_no<< "\n";}
};
```



```

class test : virtual public student // Virtually declared common
{ //base class 1
protected:
int part1;
int part2;
public:
void get_m (int x, int y)
{ part1= x; part2=y;}
void putm (void)
{
cout << "marks obtained: " << "\n";
cout << "part1 = " << part1 << "\n";
cout << "part2 = " << part2 << "\n";
}
};
class sports : public virtual student // virtually declared common
{ //base class 2
protected:
int score;
public:
void get_s (int a) {
score = a ;
}
void put_s (void)
{ cout << "sports wt.: " <<score<< "\n";}
};
class result: public test, public sports //derived class
{
private : int total ;
public:
void show (void) ;
};
void result : : show (void)
{ total = part1 + part2 + score ;
put_n ( );
put_m ( );
put_s ( ) ; cout << "\n total score= " <<total<< "\n" ;
}
main ( )
{
clrscr ( ) ;
result S1 ;
S1.get_n (345)
S1.get_m (30, 35) ;
S1.get-S (7) ;
S1. show ( ) ;
}

//Program to show hybrid inheritance using virtual base classes
#include<iostream.h>
#include<conio.h>
Class A
{

```

```

protected:
int x;
public:
void get (int) ;
void show (void) ;
};
    void A :: get (int a)
{ x = a ; }
void A :: show (void)
{ cout << X ;}
    Class A1 : Virtual Public A
{

```

```

protected:
int y ;
public:
void get (int) ;
void show (void);
};
void A1 :: get (int a)
{ y = a;}
void A1 :: show (void)
{
cout <<y ;
{
class A2 : Virtual public A
{
protected:
int z ;
public:
void get (int a)
{ z =a;}
void show (void)
{ cout << z;}
};
class A12 : public A1, public A2
{
int r, t ;
public:
void get (int a)
{ r = a;}
void show (void)
{ t = x + y + z + r ;
cout << "result =" << t ;
}
};
main ( )
{
clrscr ( ) ;

```

```
A12 r ;  
r.A :: get (3) ;  
r.A1 :: get (4) ;  
r.A2 :: get (5) ;  
r.get (6) ;  
r . show ( ) ;  
}
```

LECTURE-30

Polymorphism:

Introduction

When an object is created from its class, the member variables and member functions are allocated memory spaces. The memory spaces have unique addresses. Pointer is a mechanism to access the memory locations using their address rather than the name assigned to them. You will study the implications and applications of this mechanism in detail in this chapter.

Pointer is a variable which can hold the address of a memory location rather than the value at the location. Consider the following statement

```
int num =84;
```

This statement instructs the compiler to reserve a 2-byte of memory location and puts the value 84 in that location. Assume that the compiler allocates memory location 1001 to num. Diagrammatically, the allocation can be shown as:

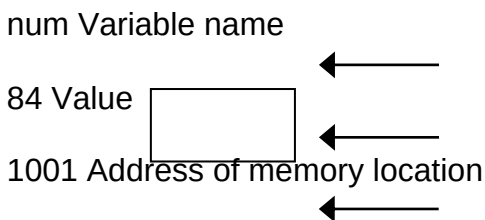
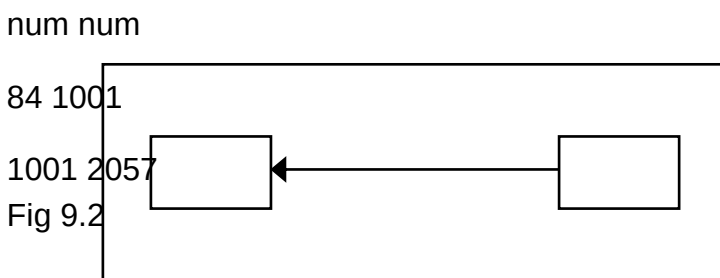


Figure 9.1

As the memory addresses are themselves numbers, they can be assigned to some other variable. For example, ptr be the variable to hold the address of variable num.

Thus, we can access the value of num by the variable ptr. We can say "ptr points to num" as shown in the figure below.



LECTURE-31

Virtual Functions

Virtual functions, one of advanced features of OOP is one that does not really exist but it appears real in some parts of a program. This section deals with the polymorphic features which are incorporated using the virtual functions.

The general syntax of the virtual function declaration is:

```
class use_detined_name{
private:
public:
virtual return_type function_name1 (arguments);
virtual return_type function_name2(arguments);
virtual return_type function_name3( arguments);
-----
};
```

To make a member function virtual, the keyword virtual is used in the methods while it is declared in the class definition but not in the member function definition. The keyword virtual precedes the return type of the function name. The compiler gets information from the keyword virtual that it is a virtual function and not a conventional function declaration.

For. example, the following declararion of the virtual function is valid.

```
class point {
intx;
inty;
public:
virtual int length ( );
virtual void display ( );
};
```

Remember that the keyword virtual should not be repeated in the definition if the definition occurs outside the class declaration. The use of a function specifier virtual in the function definition is invalid.

For example

```
class point {
intx ;
inty ;
public:
virtual void display ( );
};
virtual void point: : display ( ) //error
{
Function Body
}
```

A virtual function cannot be a static member since a virtual member is always a member of a particular object in a class rather than a member of the class as a whole.

```
class point {
int x ;
int y ;
public:
virtual static int length ( ) //error
```

```

};
int point: : length ( )
{
Function body
}

```

A virtual function cannot have a constructor member function but it can have the destructor member function.

```

class point {
int x ;
int y ;
public:
virtual point (int xx, int yy) ; // constructors, error
void display ( ) ;
int length ( ) ;
};

```

A destructor member function does not take any argument and no return type can be specified for it not even void.

```

class point {
int x ;
int y ;
public:
virtual point (int xx, int yy) ; //invalid
void display ( ) ;
int length ( ) ;

```

It is an error to redefine a virtual method with a change of return data type in the derived class with the same parameter types as those of a virtual method in the base class.

```

class base {
int x,y ;
public:
virtual int sum (int xx, int yy) ; //error
};
class derived: public base {
int z ;
public:
virtual float sum (int xx, int yy) ;
};

```

The above declarations of two virtual functions are invalid. Even though these functions take identical arguments note that the return data types are different.

```

virtual int sum (int xx, int IT) ; //base class
virtual float sum (int xx, int IT) ; //derived class

```

Both the above functions can be written with int data types in the base class as well as in the derived class as

```

virtual int sum (int xx, int yy) ; //base class
virtual int sum (int xx, int yy) ; //derived class

```

Only a member function of a class can be declared as virtual. A non member function (nonmethod) of a class cannot be declared virtual.

```

virtual void display ( ) //error, nonmember function
{
Function body
}

```

Student Activity

1. What are virtual functions
2. What are pure virtual functions
3. Define Virtual destructors.

Late Binding

As we studied in the earlier unit, late binding means selecting functions during the execution. Though late binding requires some overhead it provides increased power and flexibility. The late binding is implemented through virtual functions as a result we have to declare an object of a class either as a pointer to a class or a reference to a class.

For example the following shows how a late binding or run time binding can be carried out

with the

help of a virtual function.

```
class base {
private :
int x;
float y;
public:
virtual void display ( ) ;
int sum ( ) ;
};
class derivedD : public baseA
{
private :
int x ;
float y;
public:
void display ( ) ; //virtual
int sum ( ) ;
};
void main ( )
{
baseA *ptr ;
derivedD objd ;
ptr = &objd ;
Other Program statements
ptr->display ( ) ; //run time binding
ptr->sum ( ) ; //compile time binding
}
```

Note that the keyword virtual is followed by the return type of a member function if a run

time is

to be bound. Otherwise, the compile time binding will be effected as usual. In the above program

segment, only the display () function has been declared as virtual in the base class,

whereas the sum

() is nonvirtual. Even though the message is given from the pointer of the base class to the objects of

the derived class, it will not

access the sum () function of the derived class as it has been declared as nonvirtual.

The sum ()

function compiles only the static binding.

```

#include <iostream.h>
#include <conio.h>
class baseA {
public :
virtual void display () {
cout<< "One \n";
}
};
class derivedB : public baseA
{
public:
virtual void display(){
cout<< "Two\n"; }
};
class derivedC: public derivedB
{
public:
virtual void display () {
cout<< "Three \n"; }
};
void main () {
//define three objects
baseA obja;
derivedB objb;
derivedC objc;
base A *ptr [3]; //define an array of pointers to baseA
ptr [0] = &obja;
ptr [1] = &objb;
ptr [2] = &objc;
for ( int i = 0; i <=2; i ++ )
ptr [i]->display ( ); //same message for all objects
getche ( ) ;
}

```

Output
One
Two
Three

The program listed below illustrates the static binding of the member functions of a class. In program there are two classes student and academic. The class academic is derived from class student. The two member function getdata and display are defined for both the classes. *obj is defined for class student, the address of which is stored in the object of the class academic. The functions getdata () and display () of student class are invoked by the pointer to the class.

```

#include<iostream.h>
#include<conio.h>
class student {
private:
int rollno;
char name [20];
public:
void getdata ( );
void display ( );

```



```

};
class academic: public student {
private:
char stream;
public:
void getdata ( );
void display ( ) ;
};
void student:: getdata ( )
{
cout<< "enterrollno\n";
cin>>rollno;
cout<< "enter name \n";
cin>>name;
}
void student:: display ( )
{
cout<< "the student's roll number is "<<rollno<< "and name is"<<name ;
cout<< endl;
}
void academic :: getdata ( )
{
cout<< "enter stream of a student? \n";
cin >>stream;
}
void academic :: display ( ) {
cout<< "students stream \n";
cout <<stream<< endl;
}
void main ( )
{
student *ptr ;
academic obj ;
ptr=&obj;
ptr->getdata ( ) ;
ptr->display ( ) ;
getche ( ) ;
}
output
enter rollno
25
enter name
raghu
the student's roll number is 25 and name is raghu

```

The program listed below illustrates the dynamic binding of member functions of a class. In this program there are two classes student and academic. The class academic is derived from student. Student function has two virtual functions getdata () and display (). The pointer for student class is defined and object . for academic class is created. The pointer is assigned the address of the object and function of derived class are invoked by pointer to student.

```

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

```

```

private:
introllno;
char name [20];
public:
virtual void getdata ( );
virtual void display ( );
};
class academic: public student {
private :
char stream[10];
public:
void getdata { };
void display ( ) ;
};
void student: : getdata ( )
{
cout<< "enter rollno\n";
cin >> rollno;
cout<< "enter name \n";
cin >>name;
}
void student:: display ( )
{
cout<< "the student's roll number is"<<rollno<< "and name is"<<name;
cout<< endl;
}
void academic: : getdata ( )
{
cout << "enter stream of a student? \n";
cin>> stream;
}
void academic:: display ( )
{
cout<< "students stream \n";
cout<< stream << endl;
}
void main ( )
{
student *ptr ;
academic obj ;
ptr = &obj ;
ptr->getdata ( );
ptr->dlsplay ( );
getch ( );
}
output
enter stream of a student?
Btech
students stream
Btech

```

Pure Virtual Functions

Generally a function is declared virtual inside a base class and we redefine it the derived classes. The function declared in the base class seldom performs any task.

The following program demonstrates how a pure virtual function is defined, declared and

invoked

from the object of a derived class through the pointer of the base class. In the example there are two classes employee and grade. The class employee is base class and the grade is

derived class. The

functions getdata () and display () are declared for both the classes. For the class employee the

functions are defined with empty body or no code inside the function. The code is written

for the

grade class. The methods of the derived class are invoked by the pointer to the base class.

```
#include<iostream.h>
#include<conio.h>
class employee {
int code
char name [20] ;
public:
virtual void getdata ( ) ;
virtual void display ( ) ;
};
class grade: public employee
{
char grd [90] ;
float salary ;
public :
void getdata ( ) ;
void display ( ) ;
};
void employee :: getdata ( )
{
}
void employee:: display ( )
{
}
void grade :: getdata ( )
{
cout<< " enter employee's grade ";
cin> > grd ;
cout<< "\n enter the salary " ;
cin>> salary;
}
void grade :: display ( )
{
cout<<" Grade salary \n";
cout<< grd<< " "<< salary<< endl;
```

