# Lecture Notes

# on

# Object Oriented Programming

# 3CS4-06


## Unit IV

**Department of Computer Science & Engineering**
**Jaipur Engineering College & Research Centre, Jaipur**

## Vision of the Institute

To become a renowned centre of outcome based learning and work toward academic, professional, cultural and social enrichment of the lives of individuals and communities.

## Mission of the Institute

**M1:** Focus on evaluation of learning outcomes and motivate students to inculcate research aptitude by project based learning.
**M2:** Identify, based on informed perception of Indian, regional and global needs, the areas of focus and provide platform to gain knowledge and solutions.
**M3:** Offer opportunities for interaction between academia and industry.
**M4:** Develop human potential to its fullest extent so that intellectually capable and imaginatively gifted leaders can emerge in a range of professions.

## Vision of the Department

To become renowned Centre of excellence in computer science and engineering and make competent engineers & professionals with high ethical values prepared for lifelong learning.

## Mission of the Department

**M1:** To impart outcome based education for emerging technologies in the field of computer science and engineering.
**M2:** To provide opportunities for interaction between academia and industry.
**M3:** To provide platform for lifelong learning by accepting the change in technologies.
**M4:** To develop aptitude of fulfilling social responsibilities.

# Program Outcomes (PO)

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and Computer Science & Engineering specialization to the solution of complex Computer Science & Engineering problems.
2. **Problem analysis**: Identify, formulate, research literature, and analyze complex Computer Science and Engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex Computer Science and Engineeringproblems and design system components or processes that meet the specified needs with appropriateconsideration for the public health and safety,and the cultural, societal, and environmentalconsiderations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of Computer Science and Engineering experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex Computer Science Engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional Computer Science and Engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional Computer Science and Engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the Computer Science and Engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings in Computer Science and Engineering.
10. **Communication**: Communicate effectively on complex Computer Science and Engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the Computer Science and Engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation andability to engage in independent and life-long learning in the broadest contextof technological changein Computer Science and Engineering.

# Program Educational Objectives (PEO)

**PEO1:** To provide students with the fundamentals of Engineering Sciences with more emphasis in Computer Science & Engineering by way of analyzing and exploiting engineering challenges.
**PEO2:**To train students with good scientific and engineering knowledge so as to comprehend, analyze, design, and create novel products and solutions for the real life problems inComputer Science and Engineering
**PEO3:** To inculcate professional and ethical attitude, effective communication skills, teamwork skills, multidisciplinary approach, entrepreneurial thinking and an ability to relate engineering issues with social issues for Computer Science & Engineering.
**PEO4:** To provide students with an academic environment aware of excellence, leadership, written ethical codes and guidelines, and the self-motivated life-long learning needed for a successful professional career in Computer Science & Engineering.
**PEO5**: To prepare students to excel in Industry and Higher education by Educating Students along with High moral values and Knowledge in Computer Science & Engineering.

## Course Outcomes

1. Understand the paradigms of object oriented programming in comparison of procedural oriented programming.
2. Apply the class structure as fundamental, building block for computational programming.
3. Apply the major object-oriented concepts to implement object oriented programs in C++.
4. Implement the concept of abstraction inheritance, polymorphism, dynamic binding and generic structure in building reusable code.

## Mapping of Course Outcomes with Program Outcomes

H=3, M=2, L=1

| Semester | Subject | Code | L / T / P | CO | PO1 | PO2 | PO3 | PO4 | PO5 | PO6 | PO7 | PO8 | PO9 | PO10 | PO11 | PO12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| III | Object Oriented Programming | 3CS4-06 | L | Understand the paradigms of object oriented programming in comparison of procedural oriented programming. | 3 | 3 | 3 | 2 | 2 | 2 | 1 | 1 | 0 | 1 | 1 | 3 |
| | | | L | Apply the class structure as fundamental, building block for computational programming. | 3 | 3 | 3 | 3 | 2 | 2 | 1 | 1 | 1 | 2 | 1 | 3 |
| | | | L | Apply the major object-oriented concepts to implement object oriented programs in C++. | 3 | 3 | 3 | 3 | 2 | 1 | 2 | 2 | 1 | 2 | 1 | 3 |
| | | | L | Implement the concept of abstraction inheritance, polymorphism, dynamic binding and generic structure in building reusable code. | 3 | 3 | 3 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 3 |

# Syllabus

## RAJASTHAN TECHNICAL UNIVERSITY, KOTA

**Syllabus**

**II Year-III Semester: B.Tech. Computer Science and Engineering**

**3CS4-06: Object Oriented Programming**

**Credit-3**
**3L+0T+0P**

**Max. Marks : 150 (IA:30,ETE:120)**
**End Term Exam: 3 Hours**

| SN | CONTENTS | Hours |
|---|---|---|
| 1 | Introduction to different programming paradigm, characteristics of OOP, Class, Object, data member, member function, structures in C++, different access specifiers, defining member function inside and outside class, array of objects. | 8 |
| 2 | Concept of reference, dynamic memory allocation using new and delete operators, inline functions, function overloading, function with default arguments, constructors and destructors, friend function and classes, using this pointer. | 8 |
| 3 | Inheritance, types of inheritance, multiple inheritance, virtual base class, function overriding, abstract class and pure virtual function | 9 |
| 4 | Constant data member and member function, static data member and member function, polymorphism, operator overloading, dynamic binding and virtual function | 9 |
| 5 | Exception handling, Template, Stream class, File handling. | 6 |
| | **TOTAL** | 40 |

## UNIT IV

### STATIC DATA MEMBER:

A data member of a class can be qualified as static . The properties of a static member variable are similar to that of a static variable. A static membervariable has contain specialcharacteristics.

Variable has contain special characteristics:-

1) It is initialized to zero when the first object of its class is created.No other initialization ispermitted.
2) Only one copy of that member is created for the entire class and is shared by all the objects of thatclass, no matter how many objects arecreated.
3) It is visible only with in the class but its life time is the entire program. Static variables are normally used to maintain values common to the entire class. For example a static data member can be used as a counter that records the occurrence of all theobjects.

int item :: count; // definition of static data member

Note that the type and scope of each static member variable must be defined outside the class definition .This is necessary because the static data members are stored separately rather than as a part of anobject.

Example:-

```
#include<iostream.h>
        class item
        {
                static int count; //count is static
                int number;
        public:
                void getdata(int a)
.                {
                        number=a;
                        count++;
                 }
                void getcount(void)
                 {
                        cout<<"count:";
                        cout<<count<<endl;
                 }
        };
int item :: count ; //count defined
        int main( )
        {
        item a,b,c;
        a.get_count( );
        b.get_count( );
        c.get_count( ):
        a.getdata():
        b.getdata();
```

```
        c.getdata( );
        cout«"after reading data : "«endl;
            a.get_count( );
            b.gel_count( );
            c.get count( );
    return(0);
    }
```

The output would be

```
    count:0
    count:0
    count:0
```

After reading data

```
    count: 3
    count:3
    count:3
```

The static Variable count is initialized to Zero when the objects created . The count is incremented whenever the data is read into an object. Since the data is read into objects three times the variable count is incremented three times. Because there is only one copy of count shared by all the three object, all the three output statements cause the value 3 to be displayed.

**STATIC MEMBER FUNCTIONS:-**

A member function that is declared static has following properties :-

1. Astaticfunctioncanhaveaccesstoonly otherstaticmembersdeclaredinthe same class.
2. A static member function can be called using the class name as follows:-
   class - name :: function -name;

Example:-

```
#include<iostream.h>
    class test
    {
        int code;
        static int count; // static member variable
    public:
        void set(void)
        {
        code=++count;
        }
        void showcode(void)
        {
        cout<<"object member : "<<code<<end;
        }
    static void showcount(void)
        { cout<<"count="<<count<<endl; }
        };
int test:: count;
int main()
{
```

```
test t1,t2;
t1.setcode( );
t2.setcode();
test :: showcount();          '
testt3;
t3.setcode( );
test:: showcount( );
t1.showcode( );
t2.showcode( );
t3.showcode( );
return(0);
```

**output:-** count : 2
count: 3
object number1
object number2
object number3

## OBJECTS AS FUNCTION ARGUMENTS

Like any other data type, an object may be used as A function argument. This can cone in two ways

1. A copy of the entire object is passed to thefunction.
2. Only the address of the object is transferred to thefunction

The first method is called pass-by-value. Since a copy of the object is passed to the function, any change made to the object inside the function do not effect the object used to call the function.
The second method is called pass-by-reference . When an address of the object is passed, the called function works directly on the actual object used in the call. This means that any changes made to the object inside the functions will reflect in the actual object .The pass by reference method is more efficient since it requires to pass only the address of the object and not the entire object.

Example:-

```
#include<iostream.h>
        class time
        {
                int hours;
                intminutes;
        public:
                void gettime(int h, int m)
                {
                                hours=h;
                                minutes=m;
                }
        void puttime(void)
                {
                cout<< hours<<"hours and:";

                cout<<minutes<<"minutes:"<<end;
                }
```

```
        void sum( time ,time);
    };
void time :: sum (timet1,time t2)       .
    {
    minutes=t1.minutes + t2.minutes;
    hours=minutes%60;
    minutes=minutes%60;
    hours=hours+t 1.hours+t2.hours;
    }

    int main()
    {
    time T1,T2,T3;
    T1.gettime(2,45);
    T2.gettime(3,30);
    T3.sum(T1,T2);
    cout<<"T1=";
    T1.puttime( );
    cout<<"T2=";
    T2.puttime( );
    cout<<"T3=";
    T3.puttime( );
    return(0);
    }
```

**OPERATOR OVERLOADING:-**

Operator overloading provides a flexible option for the creation of new definations for most of the C++ operators. We can overload all the C++ operators except the following:

- Class members access operator (. ,.*)
- Scope resolution operator (::)
- Size operator(sizeof)
- Condition operator (?:)

Although the semantics of an operator can be extended, we can't change its syntax, the grammatical rules that govern its use such as the no of operands precedence and associativety. For example the multiplication operator will enjoy higher precedence than the addition operator.

When an operator is overloaded, its original meaning is not lost. For example, the operator +, which has been overloaded to add two vectors, can still be used to add two integers.

**DEFINING OPERATOR OVERLOADING:**

To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied . This is done with the help of a special function called operator function, which describes the task.

Syntax:-

```
return-type class-name :: operator op( arg-list)
        {
            function body
        }
```

Where return type is the type of value returned by the specified operation and op is the operator being overloaded. The op is preceded by the keyword operator, operator op is the function name.

operator functions must be either member function, or friend function. A basic defference between them is that a friend function will have only one argument for unary operators and two for binary operators, This is because the object used to invoke the member function is passed implicitly and therefore is available for the member functions. Arguments may be either by value or by reference.

operator functions are declared in. the class using prototypes as follows:-

```
vector operator + (vector); /./ vector addition
vector operator-( ); //unary minus
friend vector operator + (vuelor, vector); // vector add
friend vectoroperator -(vector);       // unary minus
vector operator - (vector&a);          //substraction
int operator==(vector);         //comparision
friend int operator = =(vector ,vrctor); // comparision
```

vector is a data type of class and may represent both magnitude and direction or a series of points called elements.

The process of overloading involves the following steps:-

1. Create a class that defines the data type that is used in the overloading operation.
2. Declare the operator function operator op() in the public part of the class
3. It may be either a member function or friend function.
4. Define the operator function to implement the required operations.

Overloaded operator functions can be invoked by expressions such as

op x or x op;

for unary operators and

x op y

for binary opearators.

operator op(x);

for unary operator using friend function

operator op(x,y);

for binary operator usinf friend function.

**Unary – operator overloading(using member function):**

```
class abc
{
int m,n;
public:
abc()
{
m=8;
n=9;
}
void show()
{
cout<<m<<n;
}
operator --()
{
--m;
--n;
}
};
void main()
{
abc x;
x.show();
--x;
```

```
x.show();
}
```

**Unary – - operator overloading(using friend function):**

```
class abc
{
int m,n;
public:
abc()
{
m=8;
n=9;
}
void show()
{
cout<<m<<n;
}
friend operator --(abc &p);
};
operator -- (abc &p)
{
--p.m;
--p.n;
}
};
void main()
{
abc x;
x.show();
operator--(x);
x.show();
}
```

**Unary operator+ for adding two complex numbers (using member function)**

```
class complex
{
float real,img;
public:
complex()
{
        real=0;
        img=0;
}
complex(float r,float i)
{
real=r;
img=i;
}
void show()
{
cout<<real<<"+i"<<img;
}
complex operator+(complex &p)
{
        complex w;
        w.real=real+q.real;
        w.img=img+q.img;
        return w;
}
};
void main()
{
complex s(3,4);
complex t(4,5);
complex m;
m=s+t;
s.show();
t.show();
m.show();
}
```

**Unary operator+ for adding two complex numbers (using friend function)**

```
class complex
{
float real,img;
public:
complex()
{
        real=0;
        img=0;
}
complex(float r,float i)
{
real=r;
img=i;
```

```
}
void show()
{
cout<<real<<"+i"<<img;
}
friend complex operator+(complex &p,complex &q);
};
complex operator+(complex &p,complex &q)
{
        complex w;
        w.real=p.real+q.real;
        w.img=p.img+q.img;
        return w;
}
};
voidmain()
{
complex s(3,4);complex t(4,5);
complexm;
m=operator+(s,t);
s.show();t.show();
m.show();
}
```

Overloading an operator does not change its basic meaning. For example assume the + operator can be overloaded to subtract two objects. But the code becomes unreachable.

```
class integer
{
intx, y;
        public:
        intoperator + ( ) ;
}
int integer: : operator + ( )
{
        return (x-y) ;
}
```

Unary operators, overloaded by means of a member function, take no explicit argument and return no explicit values. But, those overloaded by means of a friend function take one reference argument (the object of the relevant class).

Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.

**Table 7.2**

| Operator to Overload | Arguments passed to the Member Function | Arguments passed to the Friend Function |
|---|---|---|
| Unary Operator | **No** | **1** |
| Binary Operator | **1** | **2** |

## Type Conversions

In a mixed expression constants and variables are of different data types. The assignment operations causes automatic type conversion between the operand as per certain rules.

The type of data to the right of an assignment operator is automatically converted to the data type of variable on the left.

Consider the followingexample:

```
intx;
float y = 20.123;
x=y;
```

This converts float variable y to an integer before its value assigned to x. The type conversion is automatic as far as data types involved are built in types. We can also use the assignment operator in case of objects to copy values of all data members of right hand object to the object on left hand. The objects in this case are of same data type. But of objects are of different data types we must apply conversion rules for assignment.

There are three types of situations that arise where data conversion are between incompatible types.

1. Conversion from built in type to classtype.
2. Conversion from class type to built intype.
3. Conversion from one class type toanother.

## Basic to Class Type

A constructor was used to build a matrix object from an int type array. Similarly, we used another constructor to build a string type object from a char* type variable. In these examples constructors performed a defacto type conversion from the argument's type to the constructor's class type

Consider the following constructor:

```
string :: string (char*a)
{
        length = strlen (a);
        name=new char[len+1];
        strcpy (name,a);
}
```

This constructor builds a string type object from a char* type variable a. The variables length and name are data members of the class string. Once you define the
constructor in the class string, it can be used for conversion from char* type to string type.

**Example**

```
string si , s2;
char* namel = “Good Morning”;
char* name2 = “ STUDENTS” ;
s1 = string(namel);
s2 = name2;
```

The program statement

```
si = string (namel);
```

first converts name 1 from char* type to string type and then assigns the string type values to the object s1. The statement

```
s2 = name2;
```

performs the same job by invoking the constructor implicitly.
Consider the following example

```
class time
{
        int hours;
        intminutes;
                public:
        time (int t) // constructor
        {
                hours = t  /  60;                  //t is inputted in minutes
                minutes = t %60;
                }
        };
```

In the following conversion statements :

```
timeTl;                 //object  Tl  created
int period =160;
Tl= period;             //int to classtype
```

The object Tl is created. The variable period of data type integer is converted into class type time by invoking the constructor. After this conversion, the data member hours ofTl will have value 2 arid minutes will have a value of 40 denoting 2 hours and 40 minutes.

Note that the constructors used for the type conversion take a single argument whose type is to be converted.

In both the examples, the left-hand operand of = operator is always a class object. Hence, we can also accomplish this conversion using an overloaded =operator.

## Class to Basic Type

The constructor functions do not support conversion from a class to basic type. C++ allows us to define a overloaded casting operator that convert a class type data to basic type. The general form of an overloaded casting operator function, also referred to as a conversion function, is:

```
operator typename ( )
{
        //Program statmerit .
}
```

This function converts a class type data to typename. For example, the operator double( ) converts a class object to type double, in the following conversion function:

```
vector:: operator double ( )
{
        double sum = 0 ;
        for(int I = 0; ioize;
        sum =  sum + v[i]   * v[i ];      //scalar magnitude
        returnsqrt(sum);
}
```

The casting operator should satisfy the following conditions.

- It must be a classmember.
- It must not specify a return type.
- It must not have any arguments. Since it is a member function, it is invoked by the object and therefore, the values used for, Conversion inside the function belongs to the object that invoked the function. As a result function does not need anargument.

In the string example discussed earlier, we can convert the object string to char* as follows:

```
string:: operator char*( )
{
        return (str) ;
}
```

### One Class to Another Class Type

We have just seen data conversion techniques from a basic to class type and a class to basic type. But sometimes we would like to convert one class data type to another class type.

**Example**

```
Obj1 = Obj2 ; //Obj1 and Obj2 are objects of different classes.
```

Objl is an object of class one and Obj2 is an object of class two. The class two type data is converted to class one type data and the converted value is assigned to the Objl. Since the conversion takes place from class two to class one, two is known as the source and one is known as the destination class.

Such conversion between objects of different classes can be carried out by either a constructor or a conversion function. Which form to use, depends upon where we want the type-conversion function to be located, whether in the source class or in the destinationclass.

We studied that the casting operator function

```
Operator typename( )
```

Converts the class object of which it is a member to typename. The type name may be a built-in type or a user defined one(another class type) . In the case of conversions between objects,

typename refers to the destination class. Therefore, when a class needs to be converted, a casting operator function can be used. The conversion takes place in the source class and the result is given to the destination class object.

Let us consider a single-argument constructor function which serves as an instruction for converting the argument's type to the class type of which it is a member. The argument belongs to the source class and is passed to the destination class for conversion. Therefore the conversion constructor must be placed in the destinationclass.

**Table 7.3**

| **Conversion** | **Conversion takes place in** | |
|---|---|---|
| | **Source class** | **Destination class** |
| Basic to class | Not applicable | Constructor |
| Class to Basic | Casting operator | Not applicable |
| Class to class | Casting operator | Constructor |

When a conversion using a constructor is performed in the destination class, we must be able to access the data members of the object sent (by the source class) as an argument. Since data members of the source class are private, we must use special access functions in the source class to facilitate its data flow to the destinationclass.

Consider the following example of an inventory of products in a store. One way of keeping record of the details of the products is to record their code number, total items in the stock and the cost of each item. Alternatively we could just specify the item code and the value of the item in the stock. The following program uses classes and shows how to convert data of one type to another.

```
#include<iostream.h>
#include<conio.h>
class stock2;
class stock1
{
int code, item;
float price;
public:
stockl (int a, int b, float c)
{
code=a;
item=b;
price=c;
}
void disp( )
{
cout<<"code"<<code <<"\n";
cout<<"Items"<<item <<"\n";
cout<<"Price per item Rs . "<<price <<"\n";
}
int getcode()
{return code; }
int getitem()
{return item; }
int getprice( )
{return price;}
```

```
operator float( )
{
return ( item*price );
}
};

class stock2
{
int  code;
float val;
public:
stock2()
{
code=0; val=0;
}
stock2(int x, float y)
{
code=x; val=y;
}
void disp( )
{
cout<< “code”<<code << “\n”;
cout<< “Total Value Rs . “ <<val<<”\n”
}
stock2 (stockl p)
{
code=p . getcode ( ) ;
val=p.getitem( ) * p. getprice ( ) ;
}
};

void main ( )
{          '
Stockl il(101, 10,125.0);
Stock2 12;
float tot_val;
tot_val=i1 ;
i2=il;
cout<<” Stock Details-stockl-type” <<”\n”;
i 1 . disp ( ) ;
cout<<” Stock value”<<”\n”;
cout<< tot_val<<”\n”;
cout<<” Stock Details-stock2-type”<< “\n”;
i2 .disp( );
getch ( ) ;
}
```

You should get the following output.

Stock Details-stock1-type

code 101

Items 10

Price per item Rs. 125

Stock value

1250

Stock Details-stock2-type
code 10 1
Total Value Rs. 1250

**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 these 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:

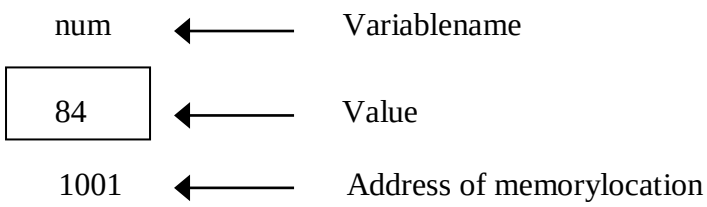


**Figure 9.1**

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 figurebelow.

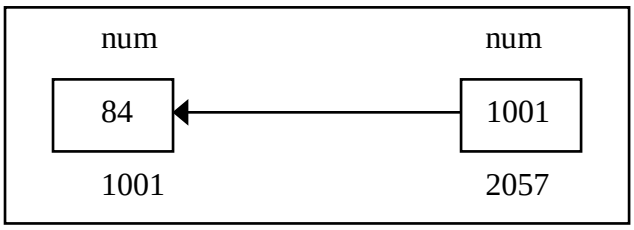


Fig 9.2

**Pointers to Objects**

An object of a class behaves identically as any other variable. Just as pointers can be defined in case of base C++ variables so also pointers can be defined for an object type. To create a pointer variable for the following class

```
class employee{
        int code;
        char name [20] ;
      public:
        inline void getdata ( )= 0 ;
        inline void display ( )= 0 ;
};
```

The following codes is written

```
employee *abc;
```

This declaration creates a pointer variable abc that can point to any object of employee type.

**this Pointer**

C++ uses a unique keyword called "this" to represent an object that invokes a member function. 'this' is a pointer that points to the object for which this function was called. This unique pointer is called and it passes to the member function automatically. The pointer this acts as an implicit argument to all the member function, fore.g.

```
class ABC
{
  int a ;
        -----
        -----
 };
```

The private variable ‘a’ can be used directly inside a member function, like

```
a=123;
```

We can also use the following statement to do the same job.

```
this → a = 123
```

e.g.

```
class stud
{
        int a;
        public:
        void set (int a)
        {
        this → a = a; //here this point is used to assign a class level
        }       ‘a’ with the argument ‘a’
        void show ( )
        {
        cout << a;
        }
};
main ( )
{
        stud S1, S2;
```

```
        S1.bet (5) ;
        S2.show ();
}
o/p = 5
```

**Pointers to Derived Classes**

Polymorphism is also accomplished using pointers in C++. It allows a pointer in a base class to point to either a base class object or to any derived class object. We can have the following Program segment show how we can assign a pointer to point to the object of the derived class.

```
class base
{
        //Data Members
        //Member Functions
};
class derived : public base
{
        //Data Members
        //Member functions
};

void main ( ) {
        base *ptr; //pointer to class base
        derived obj ;
        ptr =&obj;      //indirect reference obj to thepointer
        //Other Program statements

}
```

The pointer ptr points to an object of the derived class obj. But, a pointer to a derived class object may not point to a base class object without explicit casting.

For example, the following assignment statements are not valid

```
void main ( )
{
        base obja;
        derived *ptr;
        ptr = &obja; //invalid.... .explicit casting required
        //Other Program statements
}
```

A derived class pointer cannot point to base class objects. But, it is possible by using explicit casting.

```
void main ( )
{
        base obj ;
        derived*ptr;            // pointer of the derived class
        ptr =   (derived *) &obj;           //correctreference
        //Other Program statements
}
```

**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.

Forexample

```
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 ( );
intlength ( ) ;
```

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 virtuall method in the base class.

```
class base {
int x,y ;
public:
virtual int sum (int xx, int yy ) ; //error
} ;
class derived: public base {
intz ;
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 aredifferent.

```
virtual  int  sum  (int xx,intIT) ;//baseclass
virtual float sum (int xx, int IT) ; //derivedclass
```

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

```
virtual intsum (int xx,int yy) ;//base class virtual
intsum (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 declaredvirtual.

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

**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 aclass.

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- >di splay ( ) ; //run time binding ptr-
        >sum ( ) ; //compile time binding
}
```

Note that the keyword virtual is be followed by the return type of a member function if a run time is to bebound. 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 willnot

access the sum ( ) function of the derived class as it has been declared as nonvirtual. The sum ( ) function compiles only the static binding.

The following program demonstrates the run time binding of the member functions of a class. The same message is given to access the derived class member functions from the array of pointers. As function are declared as virtual, the C++ compiler invokes the dynamic 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 ( inti = 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 theclass.

```
#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<< end1;
}
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
```

**Virtual destructors:**

Just like declaring member functions as virtual, destructors can be declared as virtual, whereas constructors can not be virtual. Virtual Destructors are controlled in the same way as virtual functions. When a derived object pointed to by the base class pointer is deleted, destructor of the derived class as well as destructor of all its base classes are invoked. If destructor is made as non virtual destructor in the base class, only the base class's destructor is invoked when the object is deleted.

```
#icnlude<iostream.h>
#include<string.h>
class father
{
protected:
char *fname;
public:
father(char *name)
{
fname=new char(strlen(name)+1);
strcpy(fname,name);
}
virtual ~father()
{
delete fname;
cout<<"~father is invoked…";
}

virtual void show()
{
cout<<"father name…"<<fname;
}
};

class son: public father
{
protected:
char *s_name;
public:
son(char *fname,char *sname):father(fname)
{
sname=new char[strlen(sname)+1];
strcpy(s_name,sname);
}
~son()
{
delete s_name;
cout<<"~son() is invoked"<<endl;
}
void show()
{
cout<<"father's name"<<fname;
cout<<"son's name:"<<s_name;
```

```
}
};
void main()
{
father *basep;
basep =new father (“mona”);
cout<<”basep points to base object…”
basep->show();
delete basep;
basep=new son(“sona”,”mona”);
cout<<”base points to derived object…”;
basep->show();
delete basep;
}
```

**Overloading of >> and << operator**

```
#define size 5
class vector
{
int v[size];
public:
vector();
friend vector operator*(int a,vector b);
friend vector operator *(vector b,int a);
friend istream &operator>>(istream &,vector &);
friend ostream &operator<<(ostream &,vector &);
};
vector :: vector()
{
for(int i=0;i<size;i++)
v[i]=0;
}
vector::vector(int *x)
{
for (int i=0;i<size;i++)
v[i]=x[i];
}
vector operator*(int a,vector b)
{
vector c;
for(int i=0;i<size;i++)
c.v[i]=a*b.v[i];
return c;
}

vector operator*(vector b,int a)
{
vector c;
```

```
for(int i=0;i<size;i++)
c.v[i]=a*b.v[i];
return c;
}
istream &operator>>(istream &din,vector &b)
{
for(int i=0;i<size;i++)
din>>b.v[i];
}
ostream &operator<<(ostream &dout,vector &b)
{
for(i=0;i<size;i++)
dout<<a[i];
returndout;
}
int x[size]={2,4,6};
int main()
{
vector m;
vector n=x;
cout<<"enter elements of vector m";
cin>>m;
cout<<m;
vector p,q;
p=2*m;
q=n*2;
cout<<p;
cout<<q;

}
```