Module-2:

LECTURE-13

CLASS:-

Class is a group of objects that share common properties and relationships .In C++, a class is a new data type that contains member variables and member functions that operates on the variables. A class is defined with the keyword class. It allows the data to be hidden, if necessary from external use. When we defining a class, we are creating a new abstract data type that can be treated like any other built in data type.
Generally a class specification has two parts:-
a) Class declaration
b) Class function definition
the class declaration describes the type and scope of its members. The class function definition describes how the class functions are implemented.

Syntax:-

```
class class-name
{
private:
variable declarations;
function declaration ;
public:
variable declarations;
function declaration;
};
```

The members that have been declared as private can be accessed only from with in the class. On the other hand , public members can be accessed from outside the class also. The data hiding is the key feature of oops. The use of keywords private is optional by default, the members of a class are private.
The variables declared inside the class are known as data members and the functions are known as members mid the functions. Only the member functions can have access to the private data members and private functions. However, the public members can be accessed from the outside the class. The binding of data and functions together into a single class type variable is referred to as encapsulation.

Syntax:-

```
class item
{
int member;
float cost;
public:
void getldata (int a ,float b);
void putdata (void);
```

The class item contains two data members and two function members, the data members are private by default while both the functions are public by declaration. The function getdata() can be used to assign values to the member variables member and cost, and putdata() for displaying their values . These functions provide the only access to the data members from outside the class.

CREATING OBJECTS:

Once a class has been declared we can create variables of that type ~~by using the class name~~.

Example:

item x;

creates a variables x of type item. In C++, the class variables are known as objects. Therefore x is called an object of type item.

item x, y ,z also possible.

```
class item
{
-----------
-----------
-----------
}x ,y ,z;
```

would create the objects x ,y ,z of type item.

ACCESSING CLASS MEMBER:

The private data of a class can be accessed only through the member functions of that ~~class. The main() cannot contains~~ statements that the access number and cost directly.

Syntax:

object name.function-name(actual arguments);

Example:- x. getdata(100,75.5);

It assigns value 100 to number, and 75.5 to cost of the object x by implementing the getdata() function .

similarly the statement

x. putdata ( ); //would display the values of data members.

x. number = 100 is illegal .Although x is an object of the type item to which number belongs , the number can be accessed only through a member function and not by the object directly.

Example:

```
class xyz
{
Int x;
Int y;
public:
int z;
};
---------
----------
xyz p;
p. x =0; error . x is private
p, z=10; ok ,z is public
```

LECTURE-14

DEFINING MEMBER FUNCTION:

Member can be defined in two places

- Outside the class definition
- Inside the class function

OUTSIDE THE CLASS DEFlNAT1ON;

~~Member function that are~~ declared inside a class have to be defined separately outside the class.Their definition are very much like the normal functions.

An important difference between a member function and a normal

function is that a member function incorporates a membership.Identify label in the header. The 'label' tells the compiler which class the function belongs to.

Syntax:

```
return type class-name::function-name(argument declaration )
{
function-body
}
```

The member ship label class-name :: tells the compiler that the function function - name belongs to the class class-name . That is the scope of the function is restricted to the class-name specified in the header line. The :: symbol is called scope resolution operator.

Example:

```
void item :: getdata (int a , float b )
{
number=a;
cost=b;
}
void item :: putdata ( void)
{
cout<<"number=:"<<number<<endl;
cout<<"cost="<<cost<<endl;
}
```

The member function have some special characteristics that are often used in the program development.

- Several different classes can use the same function name. The "membership label" will resolve their scope, member functions can access the private data of the class .A non member function can't do so.
- A member function can call another member function directly, without using the dot operator.

INSIDE THE CLASS DEF1NATION:

Another method of defining a member function is to replace the function declaration by the actual function definition inside the class .

Example:

```
class item
{
                                             Intnumber;
float cost;
public:
void getdata (int a ,float b);
void putdata(void)
{
cout<<number<<endl; cout<<cost<<endl;
}
};
```

A C++ PROGRAM WITH CLASS:

```
# include< iostream. h>
class item
{
int number;
float cost;
public:
void getdata ( int a , float b);
void putdala ( void)
{
cout<<“number:”<<number<<endl;
cout<<”cost :”<<cost<<endl;
}
};
void item :: getdata (int a , float b)
{
number=a;
cost=b;
}
main ( )
{
item x;
                                      cout<<”\nobjectx”<<endl;
x. getdata( 100,299.95);
x .putdata();
item y;
                                     cout<<”\n object y”<<endl;
y. getdata(200,175.5);
y. putdata();
}
```

Output:

```
object x
number 100
```

```
cost=299.950012
object -4
cost=175.5
```

Q.

Write a simple program using class in C++ to input subject mark and prints it. ans:

```
class marks
{
private :
int ml,m2;
public:
void getdata();
void displaydata();
};
void marks: :getdata()
{
cout<<”enter 1st subject mark:”;
cin>>ml;
cout<<”enter 2nd subject mark:”;
cin>>m2;
}
void marks: :displaydata()
{
cout<<”Ist subject mark:”<<ml<<endl ;
cout<<”2nd subject mark:”<<m2;
}
void main()
{
clrscr();
marks x;
x.getdata();
x.displaydata();

}
```

P.T.O

LECTURE-15

NESTING OF MEMBER FUNCTION;

A member function can be called by using its name inside another member function of the same class. This is known as nesting of member functions.

```
#include <iostream.h>
class set
{
int m,n;
public:
void input(void);
void display (void);
void largest(void);
};
int set::largest (void)
{
if(m>n)
return m;
else
return n;
}
void set::input(void)
{
cout<<"input values of m and n:";
cin>>m>>n;
}
void set::display(void)
{
cout<<"largestvalue="<<largest()<<"\n";
}
void main()
{
set A;
A.input( );
A.display( );
}
```

output:

```
Input values of m and n:

3017

largest value= 30
```

Private member functions:

Although it is a normal practice to place all the data items in a private section and all the functions in public, some situations may require contain functions to be hidden from the outside calls. Tasks such as deleting an account in a customer file or providing increment to and employee are events of serious consequences and therefore the functions handling such tasks should have restricted access. We can place these functions in the private section.

A private member function can only be called by another function that is a member of its class. Even an object can not invoke a private function using the dot operator.

```
Class sample
{
int m;
void read (void);
void write (void);
};
```

if si is an object of sample, then

```
s.read();
```

is illegal. How ever the function read() can be called by the function update ( ) to update the value of m.

```
void sample :: update(void)
{
read( );
}
```

```
#include<iostream.h>
class part
{
private:
int modelnum,partnum;
float cost;
public:
void setpart ( int mn, int pn ,float c)
{
                              modelmim=mn;
partnum=pn;
cost=e;
}
void showpart ( )
{
                Cout<<endl<<"model:"<<modelnum<<end1;
Cout<<"num:"<< partnum <<endl
Cout<<"cost:"<<"$"<cost;
}
};
void main()
{
part pl,p2;
p1.setpart(644,73,217.55);
p2.setpart(567,89,789.55);
pl.showpart();
pl.showpart();
}
```

```
output:- model:644
num:73
cost: $217550003
model: 567
num:89
cost: $759.549988
```

```
#indude<iostream.h>
class distance
{
private:
int feet;
float inches;
public:
void setdist ( int ft, float in)
{
feet=ft;
inches=in;
}
void getdist()
{
cout<<”enter feet:”;
cin>>feet;
                    cout<<”enter inches:”;
cin>>inches;
}
void showdist()
{
cout<< feet<<”_”inches«endl;
}
};
void main( )
{
distance dl,d2;
d1.setdist(1 1,6.25);
d2.getdata();
                cout<<endl<<”dist:”<<d 1 .showdist();
cout<<”\n”<<”dist2:”;
d2.showdist();
}
```

output:- enter feet: 12

```
enter inches: 6.25

dist 1:”11’- 6.1.5”

dist 2: 12’- 6.25”
```

LECTURE-16

ARRAY WITH CLASSES:

```
                    #include<iostream.h>
                    #include<conio.h>
                    class employee
                    {
                    private:
                    char name[20];
                            int age,sal;
                    public:
                    void getdata();
                    void putdata();
                    };
void employee : : getdata ()
{
                cout<<"enter name :";
cin>>name;
cout<<"enter age :";
cin>>age;
                cout<<"enter salary:";
                 cin>>sal;
return(0);
}
void employee : : putdata ( )
{
cout<<name <<endl;
cout<<age<<endl;
cout<<sal<<endl;
return(0);
}
int main()
{
```

```
employee emp[5]:
for( int i=0;i<5;i++)
{
emp[i].getdata();
}
                              cout<<endl;
for(i=0;i<5;i++)
{
emp[i].putdata();
}
getch();
return(0);
}
```

ARRAY OF OBJECTS:-

```
#include<iostream.h>
#include<conio.h>
class emp
{
private:
char name[20];
int age,sal;
public:
void getdata( );
void putdata( );
};
void emp : : getdata( )
{
                                      coul<<”enter empname”: .
cin>>name;
                                      cout<<”enter age:”<<endl;
cin>>age;
cout<<”enter salun :”;
```

P.T.O

```
cin>>sal;
}
void emp :: putdata ()
{
       cout<<”emp name:”<<name<<endl;
cout<<”emp age:”<<age<<endl;
cout<<”emp salary:”<<sal;
}



        void main()
        {
        emp foreman[5];
        emp engineer[5];
        for(int i=0;i<5;i ++)
        {
        cout<<”                   for
        foreman:”;  foreman[i]
        . getdata();
        }
        cout<<endl;
        for(i=0;i<5;i++)
        {
        Foreman[i].putdata();
        .
        }
        for(int i=0;i<5;i ++)
        {
        cout<<” for
        engineer:”;
        ingineer[i].getdata();
        }
        for(i=0;i<5;i++)
        {
        ingineer[i].putdata();
        }
        getch();
        return(0);
        }
```

REPLACE AND SORT USING CLASS:-

```
#include<iostream.h>
#include<constream.h>
class sort
{
private:
int nm[30];
public;
void getdata();
void putdata();
}:
void sort :: getdata()
{
int i,j,k;
cout<<”enter 10 nos:” ;
                        for(i=0;i<10;i++)
{
cin>>nm[i];
}
for(i=0;i<9;i++)
{
for(j=i+l:j<10:j++)
{
if(nm[i]>nm[j])
{
k=nm[i];
                                                     nm[i]=nm[j];
nm[j]=k;
}

                                }


        void sort :: putdata()
        {
        int k;
        for(k=0;k<10;k++)
        {
                      cout<<num [k] <<endl ;
}
}
int main()
{
clrscr();
sort s;
s.getdata();
s.putdata();
return(0);
}
```

ARRAY OF MEMBERS:

```
#include<iostream.h>
#include<constream.h>
const int m=50;
class items
{
int item_code[m];
float item_price[m];
int count;
public:
void cnt(void) { count=0;}
void get_item(void);
void display_sum(void);
void remove(void);
void display _item(void);
};


void items :: get_item (void)
{
                                        cout<<”enter itemcode:”;
cin>> item_code[code];
                                        cout<<”enter item cost:”;
cin>>item_price[count];
count ++ ;
}
void items :: display _sum(void)
{
float sum=0;
for( int i=0;i<count;i++)
{
```

P.T.O

```
sum=sum+item_price[i];
}
cout<< “\n total value:”<<sum<<endl;
}
int main ( )
{
items order;
order.cnt();
int x;
do
{
cout<<”\nyou can do the following:”;
                 cout<<”enter appropriate no:”;
cout<<endl<<” 1 :add an item’’;
cout<<endl<<”2: display total value :”;
cout<<endl<<”3 : display an item”;
cout<<endl<<”4 :display all item:”;
cout<<endl<<”5 : quit:”;
                 cout<<endl<<endl<<”what is your option:”;
cin>>x;

switch(x)
{
case 1: order.get_item(); break;
case 2: order.display_sum(); break;
                 cose 3: order.remove(); break;
case 4: order.display_item();break;
case 5: break;
default : cout<<”error in input; try again”;
}
} while(x!=5);
}
```

P.T.O

LECTURE-17

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 member variable has contain special characteristics.
Variable has contain special characteristics:-
1) It is initialized to zero when the first object of its class is created.No other initialization is permitted.
2) Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
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 the objects.
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 rathe
than as a part of an object.
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. A static function can have access to only other static members declared in the 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  (  );  '
test t3;
t3.setcode( );
test:: showcount( );
t1.showcode( ) ;
t2.showcode( );
t3.showcode( );
return(0);
```

output:- count : 2

```
count: 3
object number 1
object number 2
object number 3
```

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 the function.
2. Only the address of the object is transferred to the function
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 calle 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;
int minutes;
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 (time t1,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);
}
```

P.T.O

LECTURE-18

FRIENDLY FUNCTIONS:-

We know private members can not be accessed from outside the class. That is a non - member function can't have an access to the private data of a class. However there could be a case where two classes manager and scientist, have been defined we should like to use a function income-tax to operate on the objects of both these classes.
In such situations, c++ allows the common function lo be made friendly with both the classes , there by following the function to have access to the private data of these classes .Such a function need not be a member of any of these classes.
To make an outside function "friendly" to a class, we have to simply declare this function as a friend of the classes as shown below :

```
class ABC
{
---------
---------
public:
--------
----------
friend void xyz(void);
};
```

The function declaration should be preceded by the keyword friend , The function is defined else where in the program like a normal C ++ function . The function definition does not use their the keyword friend or the scope operator **::** . The functions that are declared with the keyword friend are known as friend functions. A function can be declared as a friend in any no of classes. A friend function, as though not a member function , has full access rights to the private members of the class.

A friend function processes certain special characteristics:

a. It is not in the scope of the class to which it has been declared as friend.
b. Since it is not in the scope of the class, it cannot be called using the object of that class. It can be invoked like a member function without the help of any object.
c. Unlike member functions.

Example:

```
#include<iostream.h>
class sample
{
int a;
int b;
public:
void setvalue( ) { a=25;b=40;}
friend float mean( sample s);
}
float mean (sample s)
{
return (float(s.a+s.b)/2.0);
}

                    int main ( )
                    {
```

```
sample x;
x . setvalue( );
                                    cout<<"mean value="<<mean(x)<<endl;
return(0);

}
```

output:

```
mean value : 32.5
```

A function friendly to two classes

```
#include<iostream.h>
class abc;
class xyz
{
int x;
public:
void setvalue(int x) { x-= I; }
friend void max (xyz,abc);
};
class abc
{
int a;
public:
void setvalue( int i) {a=i; }
friend void max(xyz,abc);
};


void max( xyz m, abc n)

{
if(m . x >= n.a)
cout<<m.x;
else
cout<< n.a;
}

int main( )

{
abc j;
j . setvalue( 10);
xyz s;
s.setvalue(20);
max( s , j );
return(0);
}
```

SWAPPING PRIVATE DATA OF CLASSES:

~~#include<iostream.h>~~

```
class class-2;
class class-1
{
```

P.T.O

```
int value 1;
public:
void indata( int a) { value=a; }
void display(void) { cout<<value<<endl; }
                                        friend void exchange ( class-1 &, class-2 &);
};
class class-2
{
int value2;
public:
void indata( int a) { value2=a; }
void display(void) { cout<<value2<<endl; }
friend void exchange(class-l & , class-2 &);
};
void exchange ( class-1 &x, class-2 &y)
{
int temp=x. value 1;
x. value I=y.valuo2;
y.value2=temp;
}
int main( )
{
class-1 c1;
class-2 c2;
c1.indata(l00);
                                           c2.indata(200);
cout<<"values before exchange:"<<endl;
c1.display( );
c2.display( );
exchange (c1,c2);
cout<<"values after exchange :"<< endl;
c1. display ( );
c2. display ( );
return(0);
}
```

output:

```
values before exchange
100
200
values after exchange
200
100
```

P.T.O

PROGRAM FOR ILLUSTRATING THE USE OF FRIEND FUNCTION:

```
#include< iostream.h>
class account1;
class account2
{
private:
int balance;
public:
account2( ) { balance=567; }
void showacc2( )
{
cout<<”balanceinaccount2 is:”<<balance<<endl;
friend int transfer (account2 &acc2, account1 &acc1,int amount);
};
class acount1
{
private:
int balance;
public:
account1 ( ) { balance=345; }


        void showacc1 ( )
        {
        cout<<”balance in account1 :”<<balance<<endl;
        }
        friend int transfer (account2 &acc2, account1 &acc1 ,int amount);
        };
        int transfer ( account2 &acc2, account1 & acc1, int amount)
        {
        if(amount <=accl . bvalance)
        {
        acc2. balance + = amount;
        acc1 .balance - = amount;
        }
                  else
        return(0);
        }
        int main()
        {
        account1 aa;
        account2 bb;


            cout << “balance in the accounts before transfer:” ;
            aa . showacc1( );
            bb . showacc2( );
            cout << “amt transferred from account1 to account2
            is:”; cout<<transfer ( bb,aa,100)<<endl;
```

P.T.O

```
              cout<< “ balance in the accounts after the transfer:”;
        aa . showacc 1 ( );
        bb. showacc 2( );
        return(0);
}
```

output:

```
balance in the accounts before transfer
balance in account 1 is 345
balance in account2 is 567
and transferred from account! to account2 is 100
balance in account 1 is 245
balance in account2 is 667
```

P.T.O

LECTURE-19

RETURNING OBJECTS:

```
# include< iostream,h>
class complex
{
float x;
float y;
public:
void input( float real , float imag)
{
x=real;
y=imag;
}
                    friend complex sum( complex , complex);
void show ( complex );
};
complex sum ( complex c1, complex c2)
{
                      complex c3;
c3.x=c1.x+c2.x;
c3.y=c1.y+c2.y;
return c3;}


                void complex :: show ( complex c)
     {
     cout<<c.x<<" +j "<<c.y<<endl;
     }
     int main( )
     {
     complex a, b,c;
     a.input(3.1,5.65);
     b.input(2.75,1.2);
     c=sum(a,b);
     cout <<" a="; a.show(a);
     cout <<" b= "; b.show(b);
     cout <<" c=" ; c.show(c);
     return(0);
     }
```

output:

```
a =3.1 + j 5.65
b= 2.75+ j 1.2
c= 5.55 + j 6.85
```

POINTER TO MEMBERS;

It is possible to take the address of a member of a class and assign it to a pointer. The address of a member can be obtained by applying the operator & to a “fully qualified” class member name.

A class member pointer can be declared using the operator **::** * with the class name.

For Example:

```
class A
{
private:
int m;
public:
void show( );
};
```

We can define a pointer to the member m as follows :

int A **::** * ip = & A **::** m

The ip pointer created thus acts like a class member in that it must be invoked with a class object. In the above statement. The phrase A **::** * means “pointer - to - member of a class” . The phrase & A **::** m means the “ Address of the m member of a class”

The following statement is not valid :

int *ip=&m ; // invalid

This is because m is not simply an int type data. It has meaning only when it is associated with the class to which it belongs. The scope operator must be applied to both the pointer and the member.

The pointer ip can now be used to access the m inside the member function (or

friend function).

Let us assume that “a” is an object of “ A” declared in a member function . We can

access "m" using the pointer ip as follows.

```
cout<< a . * ip;
cout<< a.m;
ap=&a;
cout<< ap-> * ip;
cout<<ap->a;
```

The deferencing operator ->* is used as to accept a member when we use pointers to both the object and the member. The dereferencing operator. .* is used when the object itself is used with the member pointer. Note that * ip is used like a member name.

We can also design pointers to member functions which ,then can be invoked using the deferencing operator in the main as shown below.

(object-name.* pointer-to-member function)

(pointer-to -object -> * pointer-to-member function)

The precedence of ( ) is higher than that of .* and ->* , so the parenthesis are necessary.

DEREFERENCING OPERATOR:

```
#include<iostream.h>
class M
{
int x;
int y;
public:
                        void set_xy(int a,int b)
{
x=a;
y=b;
}
friend int sum(M);
};
int sum (M m)
{
                       int M :: * px= &M :: x; //pointer to member x




        int M :: * py- & m ::y;//pointer to y
        M * pm=&m;
        int s=m.* px + pm->py;
        return(s);
        }
        int main ( )
        {
        M m;
                         void(M::*pf)(int,int)=&M::set-xy;//pointer to function set-xy (n*pf)( 10,20);
        //invokes set-xy
        cout<<"sum=:"<<sum(n)<<cncil;
        n *op=&n; //point to object n
        ( op->* pf)(30,40); // invokes set-xy
        cout<<"sum="<<sum(n)<<end 1 ;
        return(0);
        }
        output:
        sum= 30
        sum=70
```

P.T.O

LECTURE-20

CONSTRUCTOR:

A constructor is a special member function whose task is to initialize the objects of its class .
It is special because its name is the same as the class name. The constructor is invoked when ever a
object of its associated class is created. It is called constructor because it construct the values of data
members of the class.

A constructor is declared and defined as follows:

```
//'class with a constructor
class integer
{
int m,n:
public:
integer! void);//constructor declared
------------
------------
};
integer :: integer(void)
{
m=0;
n=0;
}
```

When a class contains a constructor like the one defined above it is guaranteed that an
object created by the class will be initialized automatically.

For example:-

Integer int1; //object int 1 created
This declaration not only creates the object int1 of type integer but also initializes its
data members m and n to zero.

A constructor that accept no parameter is called the default

constructor. The default constructor for class A is A :: A( ). If no such constructor is
defined, then the compiler supplies a default constructor .
Therefore a statement such as :-
A a ;//invokes the default constructor of the compiler of the
compiler to create the object "a" ;

Invokes the default constructor of the compiler to create the object a.

The constructor functions have some characteristics:-
   They should be declared in the public section .
   They are invoked automatically when the objects are created.
   They don't have return types, not even void and therefore
they cannot return values.
   They cannot be inherited , though a derived class can call

the base class constructor .
Like other C++ function , they can have default arguments,
Constructor can't be virtual.
An object with a constructor can't be used as a member of union.

Example of default constructor:

```
#include<iostream.h>
#include<conio.h>
class abc
{
private:
char nm[];
public:
                                    abc ( )
{
cout<<"enter your name:";
cin>>nm;
}
void display( )
{
cout<<nm;
}
};

int main( )
{
clrscr( );
abc d;
d.display( );
getch( );
return(0);
}
```

PARAMETERIZED CONSTRUCTOR:-

the co~~nstructors that can take arguments are c~~alled parameterized constructors.
Using parameterized constructor we can initialize the various data elements of different objects with different values when they are created.
Example:-

```
class integer
{
int m,n;
public:
integer( int x, int y);
                                          --------
                                          ---------
};
```

```
integer:: integer (int x, int y)
{
m=x;n=y;
}
```

the argument can be passed to the constructor by calling the constructor implicitly.

```
integer int 1 = integer(0,100); // explicit call
integer int 1(0,100); //implicite call
```

CLASS WITH CONSTRUCTOR:-

```
#include<iostream.h>
class integer
{
int m,n;
public:
integer(int,int);
void display(void)



                {
                cout<<"m=:"<<m ;
                cout<<"n="<<n;
                }
        };
                integer :: integer( int x,int y) // constructor defined
                {
                m=x;
                n=y;
                }
                int main( )
                {
                integer int 1(0, 100); // implicit call
                integer int2=integer(25,75);
                cout<<" \nobjectl "<<endl;
                int1.display( );
                cout<<" \n object2 "<<endl;
                int2.display( );
                }
```

```
output:
   object 1
m=0
n=100
    object2
m=25
n=25
```

Example:-

```
#include<iostream.h>
#include<conio.h>
class abc
{
private:
char nm [30];
int age;
public:
abc ( ){ }// default
abc ( char x[], int y);
void get( )
{
                 cout<<”enter your name:”;
cin>>nm;
cout<<” enter your age:”;
cin>>age;
}
void display( )
{
cout<<nm«endl;
cout«age;
}
};
abc : : abc(char x[], int y)
{
strcpy(nm,x);
age=y;
}
void main( )
{
abc 1;
abc m=abc("computer",20000);
l.get();
l.dispalay( );
m.display ( );
getch( );
}
```

OVERLOADED CONSTRUCTOR:-

```
#include<iostream.h>
#include<conio.h>
class sum
{
private;
int a;
int b;
int c;
float d;
double e;
public:
sum ( )
{
cout<<”enter a;”;
cin>>a;
cout<<”enter b;”;
cin>>b;
cout<<”sum= “<<a+b<<endl;
}
sum(int a,int b);
                                    sum(int a, float d,double c);
};
sum :: sum(int x,int y)
{
a=x;
b=y;
}
sum :: sum(int p, float q ,double r)
{
a=p;
d=q;
e=r;
}
void main( )
{
clrscr( );
sum 1;
sum m=sum(20,50);
sum n= sum(3,3.2,4.55);
getch( );
}
```

output:

```
enter a : 3
enter b : 8
sum=11
sum=70
sum=10.75
```

COPY CONSTRUCTOR:

A copy constructor is used to declare and initialize an object from another object.
~~Example:-~~

the statement

integer 12(11);
would define the object 12 and at the same time initialize it to the values of 11.
Another form of this statement is : integer 12=11;
The process of initialization through a copy constructor is known as copy initialization.
Example:-

```
#incliide<iostream.h>
class code
{
int id;
public
code ( ) { } //constructor
code (int a) { id=a; } //constructor
code(code &x)
{
Id=x.id;
}
void display( )
{
cout<<id;
}
};
int main( )
{
code A(100);
code B(A);
code C=A;
code D;
D=A;
cout<<" \n id of A :"; A.display( );
cout<<" \nid of B :"; B.display( );
cout<<" \n id of C:"; C.display( );
cout<<" \n id of D:"; D.display( );
}
```

output :-

```
id of A:100
id of B:100
id of C:100
id of D:100
```

DYNAMIC CONSTRUCTOR:-

~~The constructors~~ can also be used to allocate memory while creating objects . This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in the saving of memory.

Allocate of memory to objects at the time of their construction is known as dynamic constructors of objects. The memory is allocated with the help of new operator.

Example:-

```
#include<iostream.h>
#include<string.h>
class string
{
char *name;




                    int length;
              public:
              string ( )
{
length=0;
name= new char [length+1]; /* one extra for \0 */
}
string( char *s) //constructor 2
{
length=strlen(s);
name=new char [length+1];
strcpy(name,s);
}
void display(void)
{
cout<<name<<endl;
}
void join(string &a .string &b)
{
length=a. length +b . length;
delete name;
name=new char[length+l]; /* dynamic allocation */
strcpy(name,a.name);
strcat(name,b.name);
}
};
int main( )
{
char * first = “Joseph” ;
string name1(first),name2(“louis”),naine3( “LaGrange”),sl,s2;
sl.join(name1,name2);
s2.join(s1,name3);
namel.display( );
name2.display( );
name3.display( );
s1.display( );
s2.display( );
}
```
output :-

```
Joseph
Louis
language
Joseph Louis
Joseph Louis Language
```

LECTURE-21

DESTRUCTOR:-

A destructor, us the name implies is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde.

For Example:-

```
~ integer( ) { }
```

A destructor never takes any argument nor does it return any value. It will be invoked implicitly by the compiler upon exit from the program to clean up storage that is no longer accessible. It is a good practice to declare destructor in a program since it releases memory space fo future use.

Delete is used to free memory which is created by new.

Example:-

```
matrix : : ~ matrix( )
{
for(int i=0; i<11;i++)
delete p[i];
delete p;
}
```

IMPLEMENTED OF DESTRUCTORS:-

```
#include<iostream.h>
int count=0;
class alpha
{
public:
alpha( )
{
count ++;
cout<<"\n no of object created :"<<endl;
}
~alpha( )
{
cout<<"\n no of object destroyed :" <<endl;
coutnt--;
}
};



int main( )
{
cout<<" \n \n enter main \n:";
alpha A1,A2,A3,A4;
{
cout<<" \n enter block 1 :\n";
alpha A5;
}
{
cout<<" \n \n enter block2 \n";
alpha A6;
}
cout<<\n re-enter main \n:";
return(0);
}
```

output:-

```
enter main
no of object created 1
no of object created 2
no of object created 3
no of object created 4
enter block 1
no of object created 5
no of object destroyed 5
enter block 2
no of object created 5
no of object destroyed 5
re-enter main
no of object destroyed 4
no of object created 3
no of object created 2
no of object created 1
```

Example :-

```
#include<iostream.h>
int x=l;
class abc
{
public:
abc( )
{
x--;
                    cout<<"construct the no"<<x<<endl;
}
~abc( )
{
cout<<"destruct the no:"<<x<<endl;
x--;
}
};
int main( )
{
                      abc I1,I2,I3,I4;
cout«Il«12«13«l4«endl;
return(0);
}
```

LECTURE-22 & 23

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. Fo 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 vector operator -(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;
}
};
void main()
{
complex s(3,4);complex t(4,5);
complex m;
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:
int operator + ( ) ;
}
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**

LECTURE-24

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 following example:

```
int x;
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 class type.
2. Conversion from class type to built in type.
3. Conversion from one class type to another.

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.

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

In the following conversion statements :

```
time Tl; //object Tl created
int period = 160;
Tl = period; //int to class type
```

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.

LECTURE-25

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
return sqrt(sum);
}
```

The casting operator should satisfy the following conditions.

It must be a class member.

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 an argument.

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 destination class.

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 destination class.

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 destination class.

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”;
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
```

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, for e.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 the pointer
//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; //correct reference
//Other Program statements
}
```

Student Activity

1. Define Pointers.
2. What are the various operators of pointer? Describe their
3. usage. How will you declare a pointer in C++?