SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
COURSE MATERIAL

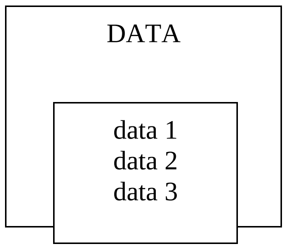
Subject Name : OBJECT ORIENTED PROGRAMMING WITH C++ **UNIT II** **Subject Code : SBS1102**

Classes and objects - class specification, class objects, accessing class members, defining member functions, inline functions, accessing member functions within class, data hiding, class member accessibility, empty classes, constructors, parameterized constructors, constructor overloading, copy constructors, new, delete operators, “this” pointer, friend classes and friend functions.

1.1 Classes and Objects

The most important feature of C++ is the “class”. A class is an extension of the idea of structure used in C. it is a new way of creating and implementing a user-defined data type.

OOP constructs modeled out of data types called *classes*. Defining variables of a class data type is known as a *class instantiation* and such variables are called *objects*.(Object is a n instance of a class.)



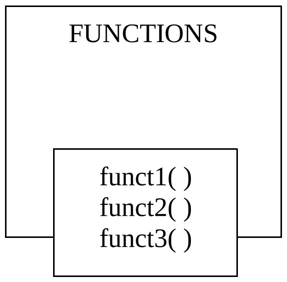


Fig. Class grouping of data and functions

Placing data and functions together in a single unit is the central theme of the OOP. The programmers are entirely responsible for creating their own classes and can also have access to classes developed by the software vendors. The variables and functions enclosed in a class are called *data members* and *member functions* respectively. Member functions define the permissible operations on the data members of a class.

Classes are the basic language construct of C++ for creating the user-defined data types. They are syntactically extension of structures. The difference is that, all the members of a structures are *public* by default where as, the members of classes are private by default. Class follows the principle that the members should be *private* unless it is specifically declared *public*.

1.2 Class specification

A class is a way to bind the data and its associated functions together. It allows the data (and functions) to be hidden, if necessary, from external use.

A class specification has two parts:

1. class declaration
2. class function definitions

The class declaration describes the type and the scope of its members. The class function definitions describe how the class functions are implemented.

The general form of a **class** declaration is:

```
class class_name
{
  private:
        variable declarations;
         function declarations:
 public:
         variable declarations;
          function declarations:
};
```

The **class** decaration is similar to a **struct** declaration.The keyword **class** specifies that what follows is an abstract data of type *class_name*. The body of the class is enclosed within braces and terminated by a semicolon. The class body contains the declarations of variables and functions. These variables and functions are collectively known *members.* They are usually grouped under two sections namely *private* and *public* to denote which of them are *members* of *private* and *public.* The keywords private and **public** and **private** are known as visibility labels. **Note** that these keywords are followed by a colon (:).

The members that have been declared as **private** can be accessed only from within the class. From the other hand the **public** members can be accessed from outside the class also. The *data hiding* is the key future of object oriented programming. The use of the keyword **private** is optional, by default the members of the class are **private** . If both the labels are missing, then by default , all the members are **private**. Such a class is completely hidden from the outside world.

The variables which are declared inside the class are known as *data members* and the functions are known as *member functions.* Only the member functions can have access to the private data members and private functions. The binding of data and functions together into a single class_type variable is referred to as ***encapsulation.***

A simple **example** for **class**

```
Class item
{
            int number;     //variables declared in the
            float cost;     //private section
public:
            void getdata(); //function declared in public section
            void putdata(); //function declared in public section
};
```

class is given some meaningful name,such as **item** . The class contains two data members and two function members. The data members are private by default. The function **getdata( )** can be used to read values for *number* and *cost* and **putdata( )** to print the values. Note that the function is ***declared*** and ***not defined***. Actually the function definition appears after the program. The data members are usually declared as **private** and the member functions as **public.**

1.2.1 Creating objects (Class Objects)

The above example class **item** shows what that class will contain, if we want to create an object of that class, we must use the class name in such a way.

Item x; //memory for x is created

Which creates a variable **x** of type **item.** In C++ , the class variables are known as ***objects***. Therefore **x** is called as an object of type **item.** We may also declare more than one object in one statement.

Eg: **Item x,y,z;**

The declaration of an object is similar to that of variables of any basic data type.

Objects can also be created by placing the objects name immediately after the closing brace, as we do in the case of structures. That is, the definition

```
Class item
   i.  {
……….
……….
……….
} x , y , z ;
```

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

1.2.2 Accessing class members

As pointed out earlier, the private data of a class can be accessed only through the member functions of that class. The **main()** cannot contain statements that access **number** and **cost** directly. The following is the format for calling a member functions.

```
Object_name . function_name (actual-arguments);
```

For example the function call statement

x.getdata(100,75.5);

is an valid and aasigns the value 100 to **number** and 75.5 to **cost** of the object **x** by implementing the **getdata( )** function. The assignments occur in the actual functions.

Similarly,
x.putdata();

would display the values of data members.Remember,a member function can be invoked only by using an object(of the same class).

The statements like,

```
getdata(100,75.5);      //has no meaning
x.number=100;           // illegal
```

The variables declared as **public** can be accessed by the objects directly.

Eg:

```
Class xyz
{
        int x;
        int y;
public:
        int z;
        ………..
        ………..
}


void main( )
{
```

```
………..
………..
xyz     p;          //creating an object of class xyz
p.x = 0;            // error, x is private
p.z = 10            //ok z is public
………..
………..
}
```

in the above example it makes an error when the private variables are assigned some values, it does not create any error if the public variables are assigned directly.

1.2.3 Defining member function

Member functions can be defined in two places:

- Outside the class definition
- Inside the class definition

Where ever the class is defined , it does the same job.

1.2.4 Outside the class definition

Member functions that are declared inside the class have to be defined outside the class separately. Their definitions are very much like the normal functions. They should have a function header and a function body.

An important difference between a member function and normal function is that a member function incorporates a membership 'identity label' in the header. This 'label' in the header tells which **class** the function belongs to.

The general form of member function is :

```
return_type class_name : : function –name(argument declaration)
{
function body
}
```

The membership label *class-name* :: tells the compiler that the *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*.

Consider the example below that we have declared the function inside the class (**getdata()** and **putdata()**), and defining the function outside the class using the *scope resolution operator* **(: :).**

Example program:

```
#include<iostream.h>
#include<conio.h>
class item
{
        int number;
        float cost;
public:
        void getdata();          //function declared//
        void putdata();         //function declared//
};


void item : : getdata()             //defining the function getdata()
        {
        cout<<"enter the value of number and cost";
        cin>>number>>cost;
        }

void item : : putdata()            //defining the function putdata()
        {
        cout<<number<<cost;
        }


void main()
{
item x;  //creating an object x of class item
x.getdata();      //calls the function getdata()
x.putdata();    //calls the function putdata()
}
```

1.2.5 Inside the class definition

In the above program the function is defined outside the class using the *scope resolution operator* .Those functions **getdata()** and **putdata()** can also be written inside the class as shown below:

```
#include<iostream.h>
#include<conio.h>
class item
{
        int number;
        float cost;
public :
        void getdata()          //declaring and defining the function inside the class
        {
        cout<<"enter the value of number and cost";
        cin>>number>>cost;
        }

        void  putdata()  //declaring and defining the function putdata()  inside the class
        {
        cout<<number<<cost;
        }
};

void main()
{
item x;
x.getdata();
x.putdata();
}
```

Both the functions does the same job without any changes

1.2.6 Inline function

One of the objective of using functions in a program is to save some memory space , which becomes appreciable when a function is likely to be called many times. However, every time a function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack and returning to the calling function. When a function is small a substantial percentage of execution time may be spent in such overheads.

C++ has a solution to this problem. To eliminate the cost of calls to small function C++ proposes a new feature called *inline function.* **An inline function is a function that is expanded in line when it is invoked.That is,the compiler replaces the function call with the corresponding function code.**

An inline function can be defined as below:

```
inline function_header
{
function body
}
```

Example:

```
inline int cube(int a)
{
return (a*a*a);
}
```

The above inline function can be invoked by the statement like;

```
c = cube(3);
d = cube(2+2);
```

The output of the above statements will be 27 and 64 for c and d respectively.

To make the function inline, just prefix the keyword inline to the function definition. All inline functions must be defined before they are called.

Remember that the **inline** keyword merely sends a request , not a command to the compiler. The compiler may ignore this request if the the function definition is too long or too complicated and compile the function as a normal function.

Some of the situations where the **inline function may not work are:**

- For functions returning values, if a loop, a **switch**, or a **goto** exists.
- For functions not returning values, if a return statement exists.
- If a function contains static variables.
- If **inline** function are recrusive.

Example program:

/* program to multiply and divide two different data type using inline function*/

```
#include<iostream.h>
#include<conio.h>
class inlinefunc
{
        int a,b;
        float c,d;
        public:
         void multiply();
        void divide();


};               inline void inlinefunc:: multiply()
        {
                cout<<"Enter 2 integers:";
                cin>>a>>b;
                int x=a*b;
                cout<<x;
        }
        inline void inlinefunc::divide()
        {
                cout<<endl<<"Enter 2 float values:";
                cin>>c>>d;
                float y=c/d;
                cout<<y;
        }

void main()
{
        inlinefunc f;
        clrscr();
        f.multiply();
        f.divide();
        getch();
}
```

1.2.7 Accessing member functions within class (Nesting of member functions)

A member of class is accessed by the objects of that class using dot operator.a member function of class can call any other function of its own class , this is called as *nesting* of member functions.

Example program:

```
#include<iostream.h>
#include<conio.h>
class greater
{
    int num1,num2;

    public:
        void read( )
        {
                cout<<"enter first number";
                cin>>num1;
                cout<<"enter second number";
                cin>>num2;
        }
        int max( )
        {
                if(num1>num2)
                        return num1;
                else
                        return num2;
        }

        void showmax( )
        {
```

```
                    cout<<"maximum="<<max( );
                }
        };

        void main( )
        {
                clrscr( );
                greater g1;
                g1.read( );
                g1.showmax( );
                getch( );
        }
```

1.2.8 Static Data Members

- Intitialized to zero when the first object is created
- Only one copy is created for the entire class and is shared by all the objects
- Visiblity is within the class but its lifetime is throughout the program

Syntax for defining a static data member:
Datatype classname::staticmember_name;
Eg.

```
#include<iostream.h>
#include<conio.h>
class abc
{
static int ct;
public:
int a;

void get()
{
a++;
ct++;
}
void disp()
{
cout<<"a = "<<a<<endl;
cout<<"ct = "<<ct<<endl;

}
};
int abc::ct;
void main()
{
clrscr();
abc o1,o2;
o1.a=0;
o2.a=0;
cout<<"o1.get"<<endl;
o1.get();
cout<<"o1.disp"<<endl;
o1.disp();
cout<<"o2.get"<<endl;
o2.get();
cout<<"o2.disp"<<endl;
o2.disp();
getch();
}
```

1.2.9 Static member function

- A static member function can access only the static data members of a class
- Syntax for calling the static member function from the main
  Classname::staticfunction_name;

Example:

```
#include<iostream.h>
#include<conio.h>
class abc
{
static int ct;
public:
int a;

void get()
{
a++;
ct++;
}
void disp()
{
```

```
cout<<"a = "<<a<<endl;

}
static void show()
{
cout<<"ct = "<<ct<<endl;
}
};
int abc::ct;
void main()
{
clrscr();
abc o1,o2;
o1.a=0;
o2.a=0;
cout<<"o1.get"<<endl;
o1.get();
cout<<"o1.disp"<<endl;
o1.disp();
abc::show();
cout<<"o2.get"<<endl;
o2.get();
cout<<"o2.disp"<<endl;
o2.disp();
abc::show();
getch();
}
```

1.2.10 Array of objects

```
#include<iostream.h>
#include<conio.h>
class abc
{
int a;
public:


void get()
{
```

```
cin>>a;

}
void disp()
{
cout<<"a = "<<a<<endl;

}
};

void main()
{
clrscr();
abc o[5];
for(int i=0;i<3;i++)
{
o[i].get();
}
for(i=0;i<3;i++)
{
o[i].disp();
}

getch();
}
```

1.2.11 Passing objects as arguments

```
#include<iostream.h>
#include<conio.h>
class abc
{
int a;
public:

void get()
{

cin>>a;

}
void disp(abc o)
{
cout<<"a = "<<o.a<<endl;

}
};

void main()
{
clrscr();
abc o;
o.get();

o.disp(o);


getch();
}
```

1.2.12 Returning objects

```
#include<iostream.h>
#include<conio.h>
class abc
{

public:
int a;

void get()
{

cin>>a;

}
abc disp(abc o)
{
abc o1;
o1.a=o.a;
return o1;

}
};

void main()
{
clrscr();
abc o;
o.get();

abc o2=o.disp(o);
cout<<o2.a;


getch();
}
```

1.2.13 Data hiding

Data is hidden inside a class, so that it cannot be accessed by mistake by any function outside the class,which is a key feature of OOP.

C++ imposes a restriction to access both the data and functions of a class.It is achieved by declaring the data part as *private*.All the data and functions defined in a class are private by default.But for the sake of clarity,the members are declared explicitly as *private*.Normally,data members are declared *private* and member functions are declared *public*.

1.2.14 Class member accessibility

Access specifiers in OOP's:

- Private
- Protected
- Public

➢ **Private members**

The private members of a class have strict access control.only the member functions of the same class can access these members.The private members of a class are inaccessible outside the class,thus providing a mechanism for preventing accidental modification of the data members.

Example:

```
class person
{
    private:                    //access specifier
        …………
        int age;
        int getage( );
        ………..
};
        person p1;
        a=p1.age( );            //cannot access private data
        p1.getage( );            //cannot access private function
```

➢ **Protected members**

The access control of the protected members is similar to that of private members and has more significance in inheritance.

Example:

```
class person
{
    protected:                    //access specifier
        …………
        int age;
        int getage( );
        ………..
};

        person p1;
        a=p1.age( );             //cannot access protected data
        p1.getage( );            //cannot access protected function
                                         (same as private)
```

➢ **Public members**

The members of a class which are to be visible(accessible) outside the class,should be decalred in public section.All data members and function declared in the public section of the class can be accessed without any restriction from anywhere in the program.

Example:

```
class person
{
    public:                    //access specifier
        …………
        int age;
        int getage( );
        ………..
};

        person p1;
        a=p1.age( );           //can access public data
        p1.getage( );          //can access public function
```

S.No.	**Access specifier**	**Accessible to**	
		Own class Members	**Objects of a Class**
1.	Private:	Yes	No
2.	Protected:	Yes	No
3.	Public:	Yes	Yes

Fig. Visibily of class members

1.2.15 Empty classes

The main reason for using a class is to encapsulate data and code. It is however, possible to have a class that has neither data nor code i.e., it is possible to have empty classes.

The **declaration** of empty classes is as follows:

```
   i.   class b{};

  ii.   class Empty{};

 iii.   class abc
      {
       };
```

are perfectly legal.

- This type of constructs is useful when developing a skeleton for a class.

  During the initial stages of development of a project, some of the classes are either not fully identified or not fully implemented. In such cases, they are implemented as *empty classes* during the first few implementations of the project. Such empty classes are also called as *stubs*.

  ✓ The significant usage of empty classes can be found with *exception handling*.

- An empty class has size greater than zero.
- The memory allocated for objects of such classes is of *nonzero* size.

Example program:

/*to find the size of empty class object*/

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

class nomembers
{
};

int main( )
{
   nomembers n;                  // n is Object of class nomembers
   cout << "The size of an object of empty class is: " << sizeof(n) << endl;
```

```
    getch( );
    return( );

  }
```

Output:

```
The size of an object of empty class is: 1
```

- Two class objects of empty classes will have distinct addresses.
- Therefore, the objects have different addresses. Having different addresses makes it possible to compare pointers to objects for identity.

Example program:

/* Two class objects of empty classes will have distinct addresses */

Eg1:

```
#include<iostream.h>
#include<conio.h>
class abc
{
};
void main()
{
abc o1,o2;
clrscr();
cout<<sizeof(o1)<<"\t"<<sizeof(o2)<<endl;
cout<<&o1<<"\t"<<&o2;
getch();

}
```

Output:

```
1     1

0x8feefff4     0x8feefff2
```

Eg: 2

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

 class A
       {

        };

 void main()
 {
           clrscr( ):
           A* p1 = new A;
           A* p2 = new A;

           cout << "The size of  p1 object of empty class is: " << sizeof(p1)
<< endl;
           cout << "The size of p2 object of empty class is: " << sizeof(p2)
<< endl;
           cout<<"address of object p1 is"<<&p1<<endl;
           cout<<"address of object p2 is"<<&p2<<endl;

            /* p1 != p2 at this point ...*/

           getch( );
}
```

Output:

```
The size of  p1 object of empty class is:2
The size of p2 object of empty class is:2
address of object p1 is:4002
address of object p2 is:4004
```

1.2.16 Constructors

- In all the C++ programs written using classes,we have used member functions such as **putdata( )** and **setvalue( )** to provide initial values to the private member variables.

- **For example** the following statement,

  A.input( );

  invokes the member function **input( ),**which assigns the initial values to the data items of object **A.**

- Similarly the statement,

  x.getdata(100,29.5);

  passes the initial values as arguments to the function **getdata( )**.Where these values are assigned to the private variables of object **x**.

- All these functions call statements are used with the appropriate objects that have already been created.
- These functions cannot be used to initialize the member variables at the time of creation of their objects.

- In general, if we want to initialize an ordinary variable, we will be initializing as below,

  <u>Ex:</u>

  ```
  int m = 20;
  float x = 5.37;
  ```

  are valid initialization statements for basic data types.

- When a variable of built-in type goes out of scope, the compiler automatically destroys the variable. But it has not happened with the objects we have so far studied.

C++ provides a special member function called the *constructor* which enables an object to initialize itself when it is created. This is known as *automatic initialization* of objects. **It also provides another member function called the *destructor* that destroys the objects when they are no longer required**.

➢ **Constructors**

- A constructor is a 'special' member function whose task is to initialize the objects of its class and allocate the required resources such as memory i.e., normally constructors are used for initializing the class data members.
- A constructor is distinct from other member functions of the class, and it is **special** because its name is the same as the class name.
- The constructor is invoked whenever an object of its associated class is created.
- It is called constructor because it constructs the values of data members of the class.
- The C++ run-time system makes sure that *the constructor of a class is the first member function to be executed automatically when an object of the class is created* i.e., the constructor is executed every time when an object of the class is created.
- Similar to other members, the constructor can be defined either within or outside the body of a class.
- It can access any data members like all other member functions, but cannot be invoked explicitly and must have public status to serve its purpose.
- It is possible to define a class which has no constructor at all. In such a case, the run-time system calls a **dummy constructor** (i.e., which performs no action) when its object is created.

A constructor is **declared and defined** as:

```
/*class with a constructor*/

    class classname
    {
            ...........                  //private members
            ...........
    public :                             //must be public
            classname ( );               //constructor declared
            …………
            ………..
    };
                                         //no return type nor void
    classname : : classname ( )          //constructor defined
    {
          //constructor body definition
    }
```

Example:

```
class integer
 {
        int m , n;
 public :
        integer ( );     //constructor declared

 };

 integer : : integer ( )  //constructor defined
 {
 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** the declaration

integer eg1;   // object eg1 created

not only creates an object **eg1** of type **integer**, but also initializes **m** and **n** to zero. *There is no need to write any statement to invoke the constructor function* (as we do with the normal function).

The **constructor functions** have some special **characteristics:**

- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return types, not even void. Therefore, they cannot return any value.
- They cannot be inherited by any class.
- It is normally used to initialize data members of class.
- They make 'implicit calls' to the operators **new** and **delete** when memory allocation is required.
- Like other C++ functions , they can have default arguments.
- Constructors cannot be **virtual**.

- We cannot refer their address.
- An object with a constructor (or destructor) cannot be used as a member of a unio

Note: When a constructor is declared for a class, initialization of the class objects become mandatory.

➢ **Default constructor**

- A constructor that **accepts** no parameters i.e., which does not take parameters explicitly is called ***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 to create object **a**.

❖ **Parameterized constructors**

- The constructor **integer ( )**, defined in the above example program, initializes the data members of all the objects to zero.
- However in practice it may be necessary to initialize the data elements of different objects with different values when they are created.
- C++ permits us to achieve this objective by passing arguments to the constructor function when the objects are created.
- **The constructors that can take arguments are called *parameterized constructors.***

From the above **example** the constructor **integer( )** may be modified to take arguments as shown below:

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

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

- When a constructor has been parameterised,the object declaration statement such as,

  integer eg1;
  may not work.

- We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways:

  - *By calling the constructor explicitly.*
  - *By calling the constructor implicitly.*

- **Calling** the constructor **explicitly**

  The **declaration** is:

  integer eg1 = integer(0,100); //explicit call

this object creates an integer object eg1 and passes the values 10 and 20 to it i.e., in the above example parameters x and y are passed to the variables m and n.

- **Calling** the constructor **implicitly**

  The **declaration** is:

  integer eg1(10,20); //implicit call

here above an object **eg1** is created of the class **integer** and the values 10 and 20 initialized to **m** and **n** by passing those values as arguments.

This method is also called as *shorthand method*, is used very often as it is shorter, looks better and easy to implement.

Note: When the constructor is parameterized, we must provide appropriate arguments for the constructor.

Example program for parameterized constructor

```
#include<iostream.h>
class integer
{
        int m , n:
public:
        integer (int , int );                    //constructor declared
        void display(void)
        {
                cout<<"m"<<m<<endl;
                cout<<"n"<<n<<endl;
        }
};

integer : : integer (int x , int y)             //constructor defined
{
m = x;
n = y;
}

        void main()
        {

                integer  eg1(10,20);                    //constructor called implicitly

                integer eg2 = integer(25,75);           //constructor called explicitly

                cout<<"object1"<<endl;
                eg1.display( );

                cout<<"object2"<<endl;
                eg2.display( );

        }
```

<u>Output</u>

```
Object1
m=10
n=20

Object2
m=25
n=75
```

- The constructor's functions can also be defined as **inline** functions.

<u>Example</u>

```
class integer
{
        int m , n:
        public:
            integer (int x , int y )            //inline constructor
            {
                m=x;
                y=n;
            }

};
```

- **Copy constructor**

  - It holds the copy of another constructor.
  - A copy constructor takes a reference to an object of the same class as itself as an argument.
  - A copy constructor may be written as

integer (integer & i);

- **A copy constructor is used to declare and initialize an object from another object.**
- For example the statement:

integer y(x);

would define the object **y** and at the same time initialize it to the values of **x** .

- Another form of this statement is

  integer y = x ;

  the process of initializing through a copy constructor is known as *copy initialization*.

- The statement,

  y=x;

  will not invoke the copy constructor.

- However, if **x** and **y** are objects, this statement is legal and simply assigns the values of **x** to **y**, member-by-member.

<u>Example program:</u>

```
#include<iostream.h>
#include<conio.h>
class code
{
        int id;
        public:
        code( )                  //default constructor
        {
        }
        code(int a)              //parameterized constructor
        {
           id=a;
        }
        code(code &x)            //copy constructor
        {
          id=x.id;               //copy in the value
        }
        void display(void)
        {
          cout<<id;
        }
};
void main()
{
        clrscr( );

        code A(100);        //object A is created and initialized
        code B(A);          //copy constructor is called
        code C=A;           //copy constructor called again

        code D;             //object D is created and not initialized
        D=A;                //copy constructor not called, just assigns values of object A to D

        cout<<"id of A:"<<endl;
        A.display( );

        cout<<"id of B:"<<endl;
        B.display( );

        cout<<"id of C:"<<endl;
        C.display( );

        cout<<"id of D:"<<endl;
        D.display( );

        getch( );
}
```

➢ **Constructors overloading**

- **A class having more than one constructor is called as multiple constructor**
- So far we have used three kinds of constructors. They are

```
integer();              // no argument constructor (null constructor)
integer( int , int);    // argument constructor (parameterized constructor)
integer(interger & i);  //copy constructor
```

- In the *first case*, the constructor itself supplies the data values and no values are passed by the calling program.
- In the *second case*, the function call passes the appropriate values from **main( )**.

- In the *third case* , the constructor takes a reference to an object of the same class as itself as an argument.
- C++ permits us to use all these constructors in the same class.

<u>For example</u> we could define a class as follows:

```
class integer
{
        int m, n;
public:
        integer( )              // constructor 1
        {
                m = 0;
                n = 0;
        }
        integer ( int a ,int b)    //constructor 2
        {
                m = a;
                n = b;
        }
         integer(integer & i)     //constructor 3
        {
                m = i. m;
                 n = i.n;
        }
};
```

- This declares three constructors for an **integer** object.
- The first constructor receives no arguments, the second one receives two integer arguments and third receives one **integer** object as an argument .

- **For example**, the declaration

        integer   a1;

  would automatically invoke the first constructor and set the both **m** and **n** of **a1** to zero. The statement

        integer a2 (20,40)

  would call the second constructor and set both **m** and **n** of **a2** to 20 and 40 respectively, finally the statement

integer a3 (a2);

would invoke the third constructor which copies the values of **a2** and **a3** . that is , it sets the value of every data element of **a3** to the value of the corresponding data element of **a2.** as mentioned earlier, such a constructor is called the ***copy constructor.***

- Sharing the same name by two or more functions is referred to as ***function overloading*.**
- When more than one constructor is defined inside the class it is called as ***constructor overloading.***

Example program:

```
#include<iostream.h>
    class code
    {
    int m,n;

 Public:
    code( );                    //constructor 1
    code(int a,int b);              //constructor 2
    code(float c,int d);    //constructor 3
     void display();
    };
    code::code( )
    {
    m=0;
    n=0;
    }
    code::code(int a,int b)
    {
    m=a;
    n=b;
    }
    code::code(float c,int d)
    {
    m=c;
    n=d;
    }
    Void display()
    {
    cout<<m;
    cout<<n;
    }
    void main( )
    {
    clrscr( );

    code c;          //would automatically invokes 1st constructor
    code b(10,20);          //would automatically invokes 2nd constructor
    code d(1.20,20);        //would automatically invokes 3rd constructor
    c.display();
    b.display();
    d.display();

    getch( );
    }
```

1.2.17 Friend functions

- It is known that the private and protected members cannot be accessed from outside the class. i.e., non_member functions cannot have an access to the private and protected data of the class.
- This feature leads to considerable inconvenience in programming.
- However, there can be a situation where user wants a function to operate on objects of two different classes (would like two classes to share a particular function).
- At such times, it is required to allow functions outside a class to access and manipulate the private members of the class.
- **For example**,consider a case where two classes, **manager** and **scientist,** have been defined. If we would like to use a function **incometax()** to operate on the objects of the both the classes.
- In that situation , C++ allows the common function to be made *friendly* with both the classes, thereby allowing the function to have access to the private data of these classes.
- To make an outside function ‘ *friendly*’ to a class, we have to simply declare this function as a **friend** of the class as shown below:

Example:

```
class ex
{
……………..
……………
public:
……………..
…………
friend void xyz( );        //declaration of friend function
};
```

- The function declaration should be prefixed by the keyword ***friend*** as shown in the above example.
- Whereas, the function definition does not use neither the keyword **friend** nor the **::** (*scope resolution Operator*).
- This function can be defined elsewhere in the program like a normal c++ program.
- The functions that are declared with the keyword **friend** are known as *friend functions*.

A **friend function** possesses certian special **characteristics**:

- The scope of the friend function is not limited to the class in which it has been declared as a **friend.**
- 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 normal function without the help of any object.
- A dot operator is not needed to execute this function.
- Unlike class member functions,it cannot access the class members directly.

- However, it can use the object name and dot operator to access the each member of the class (private and public members).

Example:

(a.x)

- Friend function can be declared either in the private or public part of the class without affecting its meaning
- Usually it has the objects as arguments.

Example program:

```
///program involving friend function//

#include<iostream.h>

class ABC;            //advance declaration like function prototype

class XYZ
{
        private:
                int a;
        public:
                void getval()
                {
                  cin>>a;
                 }                  keyword

      friend void max(ABC  m,XYZ   n) ;        //friend function  declaration
};

class ABC
{
        private:
                int x;
        public:
                void getval()
                {
                 cin>>x;
                }
```

```
    friend void max(ABC  m,XYZ  n);              //friend function declaration
};

        //friend function of class ABC and XYZ

void max(ABC   m , XYZ   n)                    //defining the friend function
{                                              //no friend keyword and no :: operator
        if(m.x>=n.a)
        {
                cout <<"Greater is"<<m.x;       //x is the private member of class XYZ
        else
                cout <<"Greater is"<<n.a;       //a is the private member of class ABC

        }
}

 void main( )
{
        clrscr( );

        ABC  obj1;
        Obj1. getval(10 );

        XYZ   obj2;
        obj2. getval ( 20);

        max(obj1 ,obj2);

        getch( );
}
```

<u>Output</u>

```
        Greater is 20
```

Here in the above program a common function **max( )** is used by the two classes **XYZ** and **ABC** by sharing the private data's.

- Though a **friend** functions add flexibility to the language and make programming convenient, in certain situations they are controversial.

- It goes against the philosophy that only member functions can access a class's private data.
- Friend functions should be used only when it is required.
- If a program uses many **friend** function, it would be better to redesign such programs.
- However, **friend** functions are very useful in certain situations.
- One such example is , a **friend** function is used to increase the versatility of *overloaded operators.*

1.2.18 Friend classes

- Just as we have the possibility to define the **friend fuction**,we can also define a class as **friend** of another **class**.
- So,that all the protected and private members of that class can be accessible.

Example program:

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

class rect;                    //advance declaration like function prototype (it is optional)


class sqr
{
         private:
               int side;
          public:
               void set_side(int x)
               {
                       side=x;
               }

friend class rect;
};

class rect
{
   private:
         int width;
    public:
        int area()
        {
                return (width*height)
        }

        void convert(sqr a);
};

void rect::convert(sqr a)
{
        width=a.side;
        height=aside;
}

void main( )
{
        sqr s;
        rect r;
        s.set_side(4);
        r.convert(s);
        cout<<r.area( );
}
```

- In the above program, we have declared **rect** as a *friend* of **sqr**, so that **rect** member functions could have access to the protected and private members of **sqr**.
- **rect** is considered as a *friend class* by **sqr**, but **rect** does not consider **sqr** to be a friend.
- So,**rect** can access the protected and private members of **sqr**.But not the reverse way.
- If want we can declare in reverse way also (i.e., **sqr** as friend class to **rect**).
- Also, *forward declaration* of **sqr** is made. This is because, the definition of **sqr** is included later.
- So, if we did not include a previous empty declaration for **sqr**,this class would not be visible from within the definition of **rect**.

1.2.19 new and delete operators

➢ new operator

- Pointers provide the necessary support for C++'s powerful dynamic memory allocation system.
- Dynamic allocation is the means by which a program can obtain memory while it is running.
- **I**t is necessary to declare arrays to some approximate size.
- It is not always possible to predict the size of the array and therefore in many cases it can lead to wastage of memory if the amount of data is much less than the maximum.
- It would be desiable to start the program and then allocate memory as the need arises.
- This capability is provided by the **new** operator.

The **syntax** for the new operator is:

pointer-variable = **new** data-type;

- Where pointer variable is a pointer of type data-type,which can be char, int, float or any user defined data type.
- The type of variable mentioned on the left hand side and he type mentioned on the right hand side should match.
- The **new** operator allocates sufficient memory to hold a data object of type data-type and returns the address of the object.
- The data-type may be any valid data type.
- The pointer-variable holds the address of the memory space allocated.

Examples:

1. char *cPtr;
cPtr = **new** char[10];

The above declaration allocates 10 bytes to the pointer cPtr.

2. int *p;
p = **new** int;

Subsequently the statement

*p=30;

Assigns 30 to the newly created int object.

3. int *p=**new** int(30):

➢ **<u>delete operator</u>**

- The delete operator is used to release the memory, which was allocated, using the new operator.

The **<u>syntax</u>** of the delete operator is:

delete pointer-variable;

<u>Example:</u>

```
int *p;
p = new int;

 delete  p;
```

The above code releases the allocated memory to the pointer p.

- If we want to free a dynamically allocated *array*,

we must use the **<u>form</u>**:

delete [size]pointer-variable;

<u>Example:</u>

```
char *p;
p = new char[10];
delete [ 10]p;
```

1.2.20 <u>**this** pointer</u>

“When a member function is called, how does C++ know which object it was called on?”. The answer is that C++ utilizes a hidden pointer named “**this**”

- C++ uses a unique keyword called **this** to represent an object that invokes a member function.
- The **this pointer** is a hidden pointer inside every class member function that points to the class object for which member functions was called i.e., the **this** pointer points to the object which made the call to the member function of the class.

Ex:

a.max();

this acts as an implicit argument to all the member functions i.e., this pointer points to the address of the object **a** automatically.

- The **this** is a pointer which is used to access the nonstatic data members and member function of the class i.e., The **this** pointer is only available for nonstatic data members so static member functions do not have a **this** pointer.

Syntax:

this->data-member;

this->member-function;

- Consider an **example**

```
class abc
{
int x;
};
```

- In the above example we can assign some values to **x** i.e., the private member **x** can be directly assigned a value inside the member function as ,

```
void max( )
{
        x=245;                //this pointer acts implicitly
    }
```

- The above assignment can also be written using **this** pointer as

```
    void max( )
    {
        this →x=245           //this pointer written explicitly
    }
```

- Here in the above statement **this** function is written *explicitly*. But in the former it acts *implicitly*.
- For convenience, **this** pointer is not used so often.
- However, we have been implicitly using the pointer **this**, when overloading the operators.
- i.e., when a binary operator is overloaded using a member function, we pass only one argument to the function. The other argument is implicitly passed using the pointer **this**.
- **this** pointer is also used to return the object it points to.

<u>Ex:</u>

return ***this;**
It will return the object that invoked the function.(for prev. eg. It will return object **a**)

- **this** pointer is used while comparing two or more objects inside a member function.
- The **this** pointer stores the refernce (address) of the class instance, to enable pointer access of the members to the member functions or data member of the class i.e., it can be used to find the address of the object in which the function is a member.
- Programmer can not modify this pointer, programmer can call member function of class.
- Most of the time, you never need to explicitly reference the "this" pointer. However, there are a few occasions where it can be useful:

1) If you have a constructor (or member function) that has a parameter of the same name as a member variable, you can disambiguate them by using "this":

Example:

```
class something
                    {
                       private:
                          int x;
                      public:
                           something(int x)
                             {
                             this x=x;
                             }
                    };
```

Note that our constructor is taking a parameter of the same name as a member variable. In this case, "x" refers to the parameter, and "**this**->x" refers to the member variable.

2) Occasionally it can be useful to have a function return the object it was working with. Returning ***this** will return a reference to the object that was implicitly passed to the function by C++.

One use for this feature is that it allows a series of functions to be "chained" together, so that the output of one function becomes the input of another function! The following is somewhat more advanced and can be considered optional material at this point.

- the “**this**” pointer is a hidden parameter of any member function. Most of the time, you will not need to access it directly. It’s worth noting that “**this”** is a const pointer — you can change the value of the object it points to, but you can not make it point to something else!

- Presence of **this** pointer is not included in the sizeof calculations.

Example program:

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

class person
{
            char name[20];
            int age;

        public:

            person( )
            {
            }

            person(char *a,int b)
            {
                        strcpy(name,a);
                        age=b;
            }

            greater(person & r)
            {
                        if(r.age>age)
                                    return r;
                        else
                                    return *this;
            }
            void display( )
            {
                    cout<<name<<age;
            }
    };

    void main( )
    {
            person x("aaa",20),y("bbb",10),z("ccc",15),m;
            m=y.greater(x);
            m.display( );
            m=z.greater(y);
            m.display( );
    }
```

Output:

```
name=aaa
age=20

name=bbb
age=10
```

1.22 FUNCTION OVERLOADING:-

We can use the same function name to create functions that perform a variety of different tasks. This is known as function polymorphism in oop. Using the concept of function overloading, we can design a family of functions with one function name but with different argument lists. The function would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type.

Example:-

Declarations:-

1. int add (int a, int b);

2.int add (int a, int b, int c);

3.double add (double x, double y);

4.double add (int p, double q);

5.double add (double p, int q);

Function calls:-

```
cout << add (0.75, 5);          // uses 5

cout << add (5, 10);            // 1

cout << add (15, 10.0); // 4 cout << add (12.5, 7.5); // 3 cout << add (5, 10.15); //2

#include<iostream.h> class funoverloading

{

int a, b, c; public:

void add ( )

{

cin>>a>>b; c = a + b; cout << c;

}

int add (int a, int b);

{

c = a + b;
return c;

}

void add (int a)

{

cin>>b;

c = a + b; cout<<c;

}

}

}
```

```
void main ( )
{
funoverloading F; int x;
F. add ( );
x = F. add (10, 5); cout<<x;
F. add (5);
}
```

Output:-

```
5 3
c = 8
c = 15
5
c = 10
```

1.23 OPERATOR OVERLOADING:-

C ++ has the ability to provide the operators with a special meaning for a data type. The mechanism of giving such special meanings to an operator is known as operator overloading.

The process of overloading involves the following steps:

Create a class that defines the data type that is to be used in the overloading operation.

Declare the operator function operator op()in the public part of the class

It may be either a member function or a friend function

Define the operator function to implement the required operations.

Syntax:-

```
returntype classname:: operator op (argument list)
{
Function body
}.
```

Example:-

```
void space:: operator-( )
{
x=-x;
}
```

1.24 OVERLOADING UNARY OPERATORS:-

Let us consider the unary minus operator. A minus operator when used as a unary takes just one operand. We know that this operator changes the sign of an operand when applied to a basic data item. We will see here how to overload this operator so that it can be applied to an object in much the same way as is applied to an int or float variable. The unary minus when applied to an object should change the sign of each of its data items.

```
#include<iostream.h>
{
int x;
int y;
int z;
public:
void getdata(int a, int b, int c); void display(void);
void operator-(); //overload unary minus
};
void space::getdata(int a, int b, int c)
```

```
{
x=a;
y=b;
z=c;
}
void space::display(void)
{
cout<<x<<" "; cout<<y<<" "; cout<<z<<" ";
}
void space::operator-()
{
x=-x;y=-y;z=-z;
}
int main()
{
space S; S.getdata(10,-20,30);cout<<"S="; S.display();
- S; cout<<"S="; S.display(); return 0;
}
```

Output:-

```
S= 10 -20 30
S= -10 20 -30
```

Note:

The function operator-() takes no argument. Then, what does this operator function do? It changes the sign of data members of the object S. Since this function is a member function of the same class, it can directly access the members of the object which activated it.

1.25 OVERLOADING BINARY OPERATORS:-

The same mechanism which is used in overloading unary operator can be used to overload a binary operator.

```
# include<iostream.h> class complex
{
float x; float y; public:
complex()
{
}
complex(float real, float imag)
{
x=real;
y=imag;
}
complex operator+(complex); void display(void);
};
complex complex::operator+(complex c)
{
complex temp; temp.x=x+c.x; temp.y=y+c.x; return(temp);
}
void complex::display(void)
{
cout<<x<<"j"<<y<<"\n";
}
int main()
{
```

complex C1,C2,C3; C1=complex(2.5,3.5) C2=complex(1.6,2.7) C3= C1 + C2; cout<<”C1 = ”; C1.display(); cout<<”C2 = ”; C2.display(); cout<<”C3 = ”; C3.display();

return 0;

}

Output:-

```
C1=2.5 +j3.5
C2=1.6 + j2.7

C3= 4.1 +j6.2
```