Classes and Objects

By: Richa Jain

Class

- A class is a way to bind the data and its associated functions together.
- When defining a class, we are creating a new abstract data type that can be treated like built-in data type.
- A class specification has two parts:

1. Class declaration

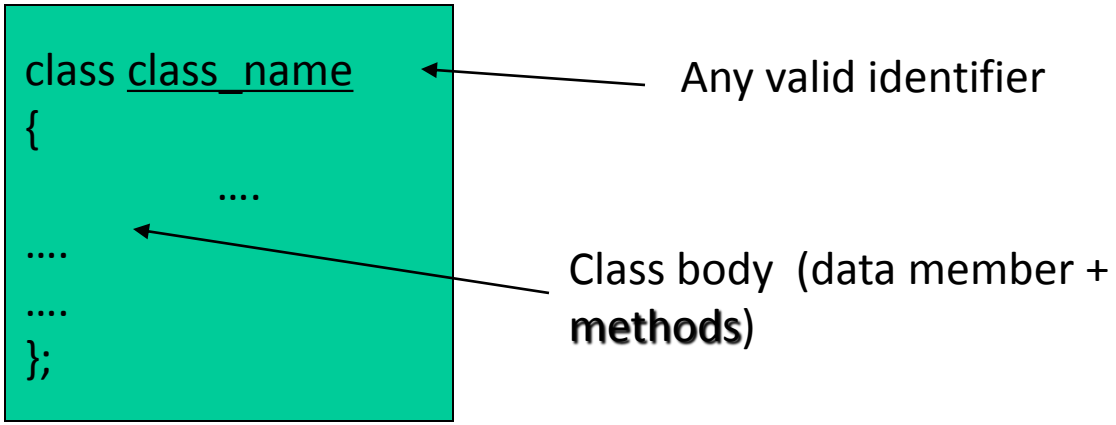
   describes the type and scope of its member

2. Class function definitions

   describe how the class functions are implemented

Classes in C++

- A class definition begins with the keyword *class*.
- The body of the class is contained within a set of braces, { } ; (notice the semi-colon).



General Structure of a class

- Class name or name of class
- Class members
  - Data Members
  - Member functions
- Access Specifiers
- Declaring objects

General form of Class declaration

```
class classname
{
 private:
variable declarations;
 function declarations;
 public:
 variable declarations;
 function declarations;
} obj1, obj2,…..objN;
```

Access Specifier

Class Members

Class Name

- Name given to a particular class.
- Serves as a name specifier for the class using which we can create objects.
- The class is specified by keyword “class”

Data Members

- Are the variables declared inside the class
- We can declare any number of data members of any type in a class.
- We can say that variables in C are data members in C++.
- E.g. int rn;

Member functions

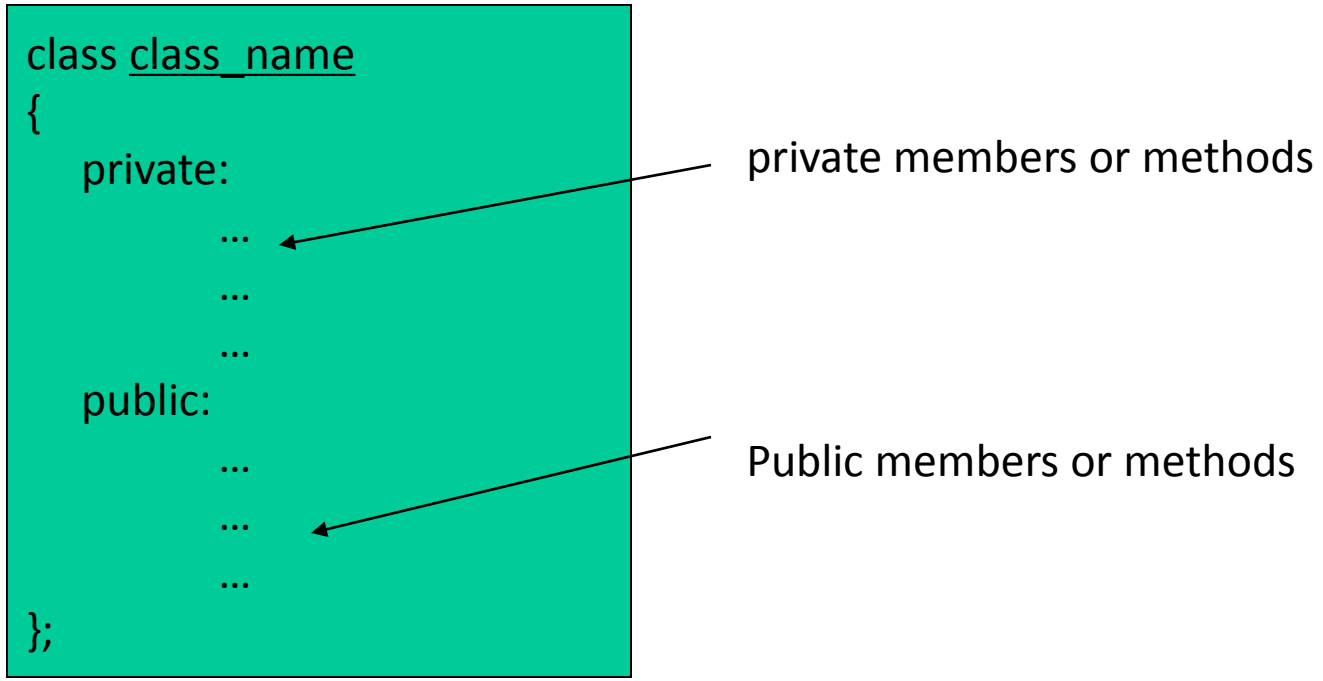
- Are functions that are declared inside the class
- Various operations(functions) that can be performed to data members of that class.
- We can declare any number of member functions of any type in a class.
- E.g. void read();

Access Specifiers

- Used to specify access rights for the data members and member functions of the class.
- Depending upon the access level of a class member, access to it is allowed or denied.
- Within the body, the keywords *private:* and *public:* specify the access level of the members of the class.
  - the default is private.
- Usually, the data members of a class are declared in the *private:* section of the class and the member functions are in *public:* section.

Classes in C++

```
class class_name
{
    private:
            ...
            ...
            ...
    public:
            ...
            ...
            ...
};
```



Private:

only members of that class have accessibility

- can be accessed only through member functions of that class.
- Private members and methods are for internal use only.

Public:

- Accessible from outside the class
- can be accessed through member function of any class in the same program.

Protected:

- Stage between private and public access.
- They cannot be accessed from outside the class, but can be accessed from the derived class(inheritance)

Class Example

- This class example shows how we can encapsulate (gather) information into one package (unit or class)

```
class item
{
    private:
            int number;     // variables declaration
            float cost;
    public:
            void getdata(int a, float b);
            void putdata(void); // fun declaration
};  // end with semicolon
```

No need for others classes to access and retrieve its value directly. The class methods are responsible for that only.

They are accessible from outside the class, and they can access the member (radius)

Creating Objects

- In class declaration, it only specifies what class will contain.
- Once class has been declared, we can create variables of that type by using the class name.
- Ex: item x; // memory for x is created
- In C++, the class variables are known as **objects.**
- The necessary space is allocated to an object at this stage.

- Objects can also be created when a class is defined by placing their names immediately after the closing brace, as we do in structures. That is

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

Accessing class members

- Syntax for calling a member function:

Object_name.function_name(actual parameters);

Ex:

x.getdata(100,75.5); // value 100 to number & 75.5 to cost

Similarly

x.putdata();

Accessing class members

Note: a member function can be invoked only by using an object(of same class)

The statement like:

```
getdata(100,75.5);    // no meaning

x.number=100;         // illegal
```

because data member (number) declared private can be accessed only through a member function and not by object directly.

Accessing class members

- A variable declared as public can be accessed by the objects directly.

Ex:
```
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
```

Defining Member functions

It can be defined in two places:

- Outside the class definition
- Inside the class definition

Outside the Class Definition

- Member functions that are declared inside a class have to be defined separately outside the class
- 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 a normal function is that a member function incorporates a membership " identity label" in the header.
- This "label" tells the compiler which class the function belongs to

- The general form of a member function definition 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 i.e. The scope of the function is restricted to the class-name specified in the header line

Example

```
void item :: getdata (int a, float b)
{
number = a;
cost = b;
}
void item :: putdata (void)
{
cout<<“Number :”<<number;
cout<<“Cost”<<cost;
}
```

Member function have some special characteristics :

- 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 cannot do so.
- A member function can call another member function directly, without using dot operator.

Inside the class definition

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

```
{
int number;
float cost;
public:
void getdata(int a, float b);   //declaration
```

```
// inline function
Void putdata(void)      // definition inside the class
{
cout<<number<<"\n";
cout<<cost<<"\n";
}
};
```

When a function is defined inside the class , it is treated as inline function. Therefore all restrictions and limitations that apply to inline function are also applicable here.

Making an outside function inline

- Can declare member function inline by using qualifier inline in header line of function definition.
- Ex:

```
class item
{
public:
void getdata(int a, float b);       //declaration
};
inline void item:: getdata( int a, float b)        //definition
{
      number =a;
       cost=b;
}
```

Nesting of Member Functions

- Member function of a class can be called only by an object of that class using a dot operator.
- However, there is an exception to this.
- 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>
#include<conio.h>
using namespace std;
class binary
{
Int s;
public:
void read(void)
{
cout<<“Enter a binary number”;
Cin>>s;
}
void chk_bin(void)
{
read();       //calling member function
}
};
int main()
{
binary  b;
b.read();
b.chk_bin();
getch();
return 0;
}
```

Private member functions

- It can only be called by a another function i.e. Member of its class.
- Even an object cannot invoke a private function using the dot operator.
- Consider a class as defined below:

```
class sample
{
int  m;
Void read(void);   // private member function
Public:  void update(void);
                 void write(void);
};
```

If s1 is an object of sample, then

```
        s1.read();   //won't work; objects cannot access private members
```

- However, the function read() can be called by the function update() to update the value of m

```
void sample :: update(void)
{
read();      //simple call;no object used
}
```

Array within a class

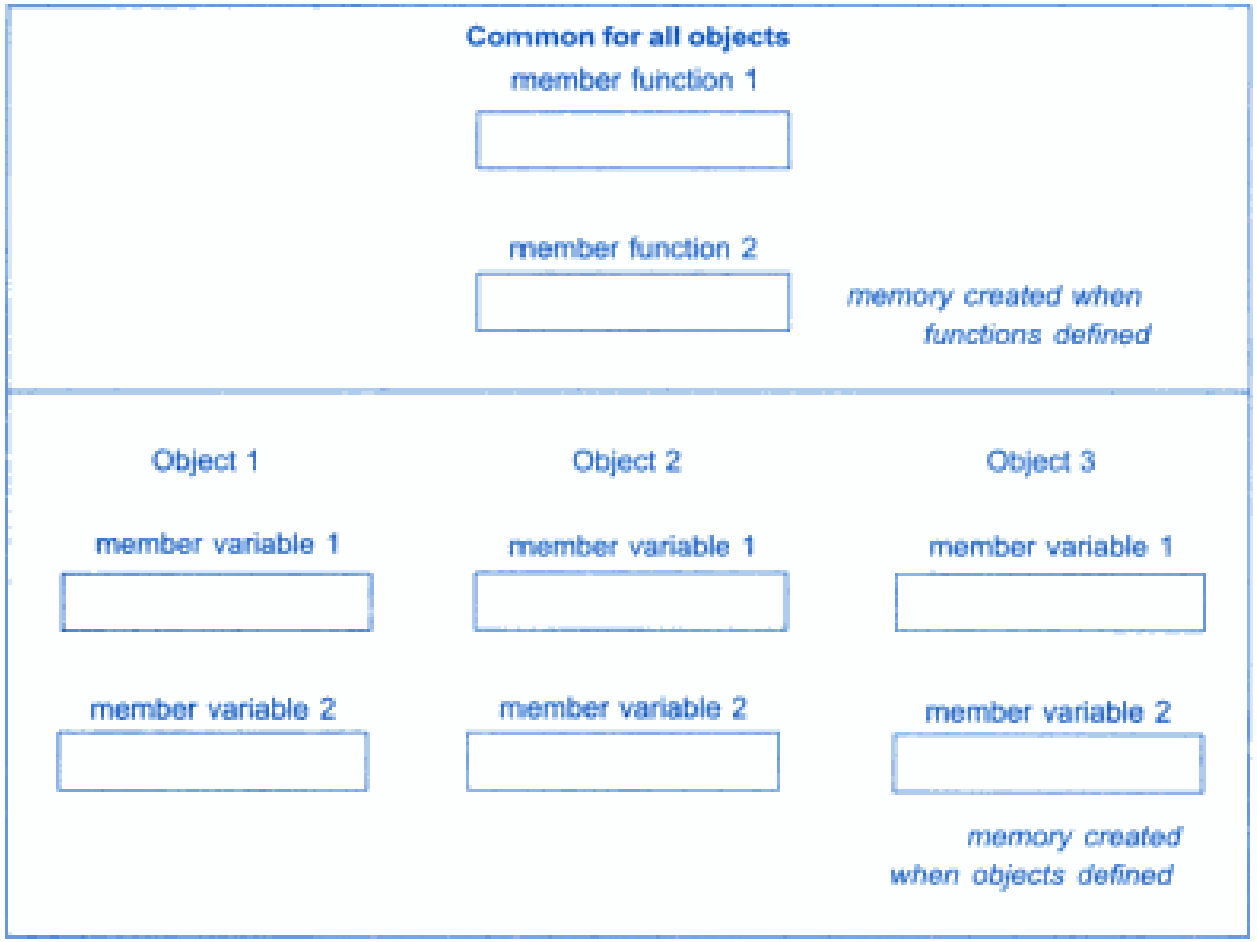
- The array can be used as a member variable in a class.
- Const int size=10; // provide value for array size

```
class array
 {
 int a[size];    //'a' is int type array
 Public :  void setval(void);
                 void display(void);
 };
```

Memory allocation for objects

- It stated that memory space for objects is allocated when they are declared and not when the class is specified. This statement is only partly true.
- Actually, the member functions are created and placed in the memory space only once when they are defined as a part of class.
- No separate space is allocated for member functions when the objects are created.
- Only space for member variables is allocated separately for each object. As this is essential, because the member variables will hold different data values for different objects.

Memory allocation for objects



Static data members

- A data member of a class can be declared as static.
- The properties of static member variable are similar to that of C.
- It has special characteristics:
  - It is initialized to 0 when the first object of its class is created. No other initialization is permitted.
  - Only one copy of that member is created for entire class and is shared by all objects of that class no matter how many objects are created.
  - It is visible only within the class, but its lifetime is the entire program

Static variables are normally used to maintain values common to the entire class

```
class item
{
static int count;
int number;
public:
 void getdata(int a)
{
number=a;
count++;
}
void getcount(void)
{
cout<<count;
}
};
int item :: count;     // definition of static data member
int main()
{
item a,b,c;            // count is initialized to zero
a.getcount();
b.getcount();
a.getdata(100);
b.getdata(200);
a.getcount()
b.getcount();
return 0;
}
```

Output:
count: 0
count: 0
After reading data
count: 2
count: 2

Static Member Functions

- A member function that is declared static has the following properties:
  - A static function can have access to only other static members(functions or variables) declared in the same class.
  - A static member function can be called using the class name(instead of its objects) as follows:

    Class name :: function-name;

```
#include<iostream.h>
using namespace std;
class test
{
int code;
static int count;  // static member variable
public:
void setcode(void)
{
code=++count;
}
void showcode(void)
{
cout<<“object number:”<<code<<“\n”;
}
static void showcount(void)  //static member
                                    function
{
cout<<“count:”<<count<<“\n”;
}
};
int test::count;
int main()
{
test t1, t2;
t1.setcode();
t2.setcode();
test::showcount();   //accessing
                       static function

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
```

Arrays of Objects

- We can have arrays of variables that are of the type **class.** Such variables are called arrays of objects.
- **Ex:** class employee

```
{       char name[30];
        float age;
public:
        void getdata(void);
        };
int main()
{
employee manager[3];
employee worker[3];
        for( int i=0;i<3;<i++)          {
        manager[i].getdata();
                }
```

Default Arguments

```
#include <iostream>
using namespace std;
void display(char = '*', int = 1);
int main()
{
  cout<<"No argument passed:\n";
 display();
  cout<<"\n\nFirst argument passed:\n";
 display('#');
 cout<<"\n\nBoth argument passed:\n";
 display('$', 5);
  return 0;
}
void display(char c, int n){
 for(int i = 1; i <=n; ++i) {
    cout<<c;
 }
  cout<<endl;
}
```

Objects as Function arguments

- This can be done in two ways:
  - A copy of the entire object is passed to the function(pass by value)
  - Only the address of the object is transferred to the function(pass by reference)

  In first way, a copy of the object is passed to function, any changes made to the object inside the function do not affect the object used to call the function.

  In second method, address of object is passed, the called function directly works on actual object used in call. So any changes made to object inside the function will reflect in the actual object

```
#include<iostream>
using namespace std;
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”<< “\n”;
}
void sum(time, time);  // declaration with objects
                                as arguments
};
void time:: sum(time t1, time t2)    //t1,t2 are
                                                objects
{
minutes=t1.minutes + t2.minutes;
hours= minutes/60;
minutes= minutes%60;
hours= hours + t1.hours + t2.hours;
}
```

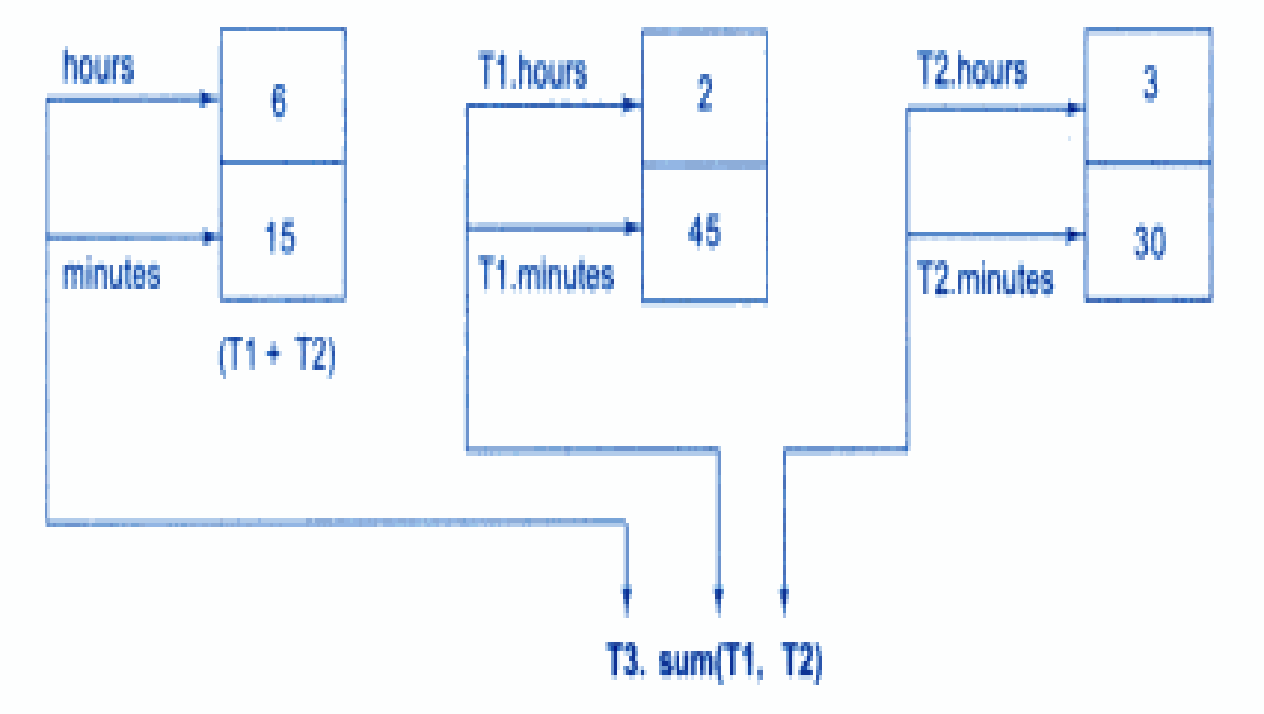
```
int main()
{
Time T1, T2, T3;
T1.gettime(2,45);        //get  T1
T1.gettime(3,30);       //get T2
T3. sum(T1, T2);         // T3= T1+T2
cout<<“T1=“; T1.puttime();      //display T1
cout<<“T2=“; T2.puttime();      //display T2
cout<<“T3=“; T3.puttime();      //display T3
return 0;
}
```

Output:

T1= 2 hours and 45 minutes

T2= 3 hours and 30 minutes

T3= 6 hours and 15 minutes


hours
6
15
minutes
(T1 + T2)
T1.hours
2
45
T1.minutes
T2.hours
3
30
T2.minutes
T3. sum(T1, T2)

Friend function

- To make outside function “friendly” to a class, declare this function as a friend of the class :

```
class ABC
{
        ..............
    public:
        ............
    friend void xyz(void);      //declaration
 };
```

Friend function

- Function definition does not use either the keyword friend or the scope operator ::
- A function can be declared as a friend in any number of classes.
- A friend function, although not a member function, has full access rights to the private members of the class.

A friend function possesses certain special characteristics:

- It is not in the scope of the class to which it has been declared as 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.
- Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name (e.g., A.x)
- It can be declared either in public or private part of a class without affecting its meaning.
- Usually, it has the objects as arguments.

```
# include<iostream.h>
using namespace std;
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;       //object X
X.setvalue();
cout<<“MeanValue=“<<mean(X)<<“\n”;
// passes the object X by value to the
friend function
return 0;
}
```

Output:

Mean value = 32.5

- Member functions of one class can be friend functions of another class. In such cases , they are defined using scope resolution operator as shown below:

```
class X
{
        .....
        int fun1();  // member function of X
 };
class Y
{
        friend int X:: fun1();     // fun1() of X  is friend of Y
};
```

- Also declare all the member functions of one class as the friend functions of another class. In such cases, the class is called a **friend class.**
- A friend class is a class whose members have access to the private or protected members of another class
- This can be specified as follows:

```
class Z
{
        .....
        friend class X;   //all member functions of X are
                          //  friends to Z
```

Using friend function to Add data objects of two different classes

```
#incude<iostream.h>
class ABC;     //forward declaration
class XYZ
{
int data;
public:
 void setvalue( int value)
  {
  data = value;
 }
 friend void add (XYZ, ABC);  // friend function
                                          declaration
 };
 class ABC
  {
  int data;
 public:
 void setvalue(int value)
 {
 data=value;
}
 friend void add ( XYZ, ABC);
};
void add (XYZ obj1, ABC obj2)
{
 cout<<" sum of data
value="<<obj1.data + obj2.data;
}
 int main()
 {
 XYZ x;
ABC a;
x.setvalue(5);
a.setvalue(50);
 add (x,a);
return 0;
{
```

Function friendly to two classes

```
# include<iostream>
using namespace std;
class ABC;   //forward declaration
class XYZ
{
int x;
public:
void setvalue(int i) { 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)  //Definition of
                        friend
{
if(m.x>= n.a)
cout<<m.x;
else
cout<<n.a;
}
```

```
int main()
{
ABC abc;
abc.setvalue(10);
XYZ xyz;
xyz.setvalue(20)
max(xyz,abc);
return 0;
}
```

OUTPUT:
20

Swapping Private data of classes

```
# include<iostream.h>
using namespace std;
class class_2;
class class_1
{
int value1;
public:
void intdata(int a) { value1=a;}
void display(void)  {  cout<<value1<<‘\n”;   }
friend void exchange(classes_1 &,
classes_2 &);
};
void exchange(class_1 & x, class_2 & y)
{
int temp = x.value1;
x.value1 = y.value2;
y.value2 = temp;
}
int main()
{
class_1 C1;
class_2 C2;
C1.indata(100);
C2.indata(200);

cout<<“Values before
exchange”<“\n”;
C1.display();
C2.display();
exchange(C1,C2); //swapping

cout<<“values after exchange”<<“\n”;
C1.display();
C2.display();
return 0;
}
```

OUTPUT:

```
Values before exchange
100
200
Values after exchange
200
100
```

Returning Objects

- A function cannot only receive objects as arguments but also can return them.

```
# include<iostream.h>
using namespace std;
class complex
{
float x,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<<"\n";
}
int main()
{
complex A,B,C;

A.input(3.1, 5.65);
B. input(2.75, 1.2);

C= sum(A,B);
cout<<"A="; show(A);      //sum() is a friend
cout<<"B="; show(B);
cout<<"C="; show(c);
return 0;
}
```

Pointer to Objects

- A pointer to a C++ class is done exactly the same way as a pointer to a structure and to access members of a pointer to a class you use the member access operator **->** operator, just as you do with pointers to structures. Also as with all pointers, you must initialize the pointer before using it.

```
#include <iostream>
using namespace std;
class myclass {
  int i;
public:
  myclass(int j) {
     i = j;
  }
  int getInt() {
     return i;
  }
};
int main()
{
  myclass ob(88), *objectPointer;
  objectPointer = &ob;      // get address of ob
  cout << objectPointer->getInt();
    // use -> to call getInt()
  return 0;
}
```

Pointers to Members

- It is possible to take 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.

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

- Define a pointer to the member m as follows:

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

  – ip pointer created thus acts like a class member in that it must be invoked with a class object.
  – A::* means “pointer-to-member” of A class.
  – &A::m means the “address of the m member of A class”.
  – ip now be used to access the member m inside member functions.

- The dereferencing operator ->* is used to access a member when we use pointers to both the object and the member.
- The dereferencing operator .* is used when object itself is used with the member pointer.

Example

```
#include <iostream>

using namespace std;

class X
 {
 public:
 int a;
void f(int b)
 {
 cout << "The value of b is "<< b << endl;
 }
 };
 int main()
{
// declare pointer to data member
 int X::*ptiptr = &X::a;

 // declare a pointer to member function
 void (X::* ptfptr) (int) = &X::f;

// create an object of class type X

 X xobject;

 // initialize data member
 xobject.*ptiptr = 10;

cout << "The value of a is " << xobject.*ptiptr<<
endl;

// call member function
(xobject.*ptfptr) (20);
 }
```

```
#include<iostream>
using namespace std;
class M
{
int x;
int y;
public:
void set_xy(int a, int b)
{   x=a;   y=b;    }
friend int sum(M m);
};
int sum(M m)
{
// declare pointer to data member
int M :: * px =&M :: x;
int M :: * py =&M :: y;
M *pm=&m;
int S = m.*px + pm->*py;
return S;
}
```

```
int main()
{
M n;
//declare pointer to member function
void (M :: *pf)(int,int) = &M :: set_xy;

// call member function
(n.*pf)(10,20);
cout<<“SUM=“<<sum(n)  ;

M *op = &n;
(op->*pf)(30,40);
cout<<“SUM=“<<sum(n);

return 0;
}
```

this pointer

- Every object in C++ has access to its own address through an important pointer called **this** pointer.
- The **this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.
- Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.
- We cannot declare this pointer or make assignments to it.
- A static member function does not have a this pointer.

Example

```
#include <iostream>
using namespace std;
 struct X
{
private:
 int a;
 public:
void Set_a(int a)
 {
 // The 'this' pointer is used to retrieve 'xobj.a'
// hidden by the automatic variable 'a'
this->a = a;
 }
void Print_a()
{
cout << "a = " << a << endl;
 }
 };
int main()
 {
X xobj;
 int a = 5;
xobj.Set_a(a);
xobj.Print_a();
}
```

```
#include <iostream>
 using namespace std;
class Box
 {
 public:
 // Constructor definition
 Box(double l=2.0, double b=2.0, double
h=2.0)
 {
 cout <<"Constructor called." << endl;
length = l; breadth = b; height = h;
}
double Volume()
{
return length * breadth * height;
 }
int compare(Box box)
{
 return this->Volume() > box.Volume();
}
private:
double length;          // Length of a box
double breadth;         // Breadth of a box
double height;          // Height of a box
 };
```

```
int main(void)
{

Box Box1(3.3, 1.2, 1.5);     // Declare box1
Box Box2(8.5, 6.0, 2.0);      // Declare box2

if(Box1.compare(Box2))
{
cout << "Box2 is smaller than Box1" <<endl;
}
else
{
cout << "Box2 is equal to or larger than Box1"
<<endl;
 }
return 0;
 }
```

OUTPUT:

```
Constructor called.
 Constructor called.
 Box2 is equal to or larger than Box1
```