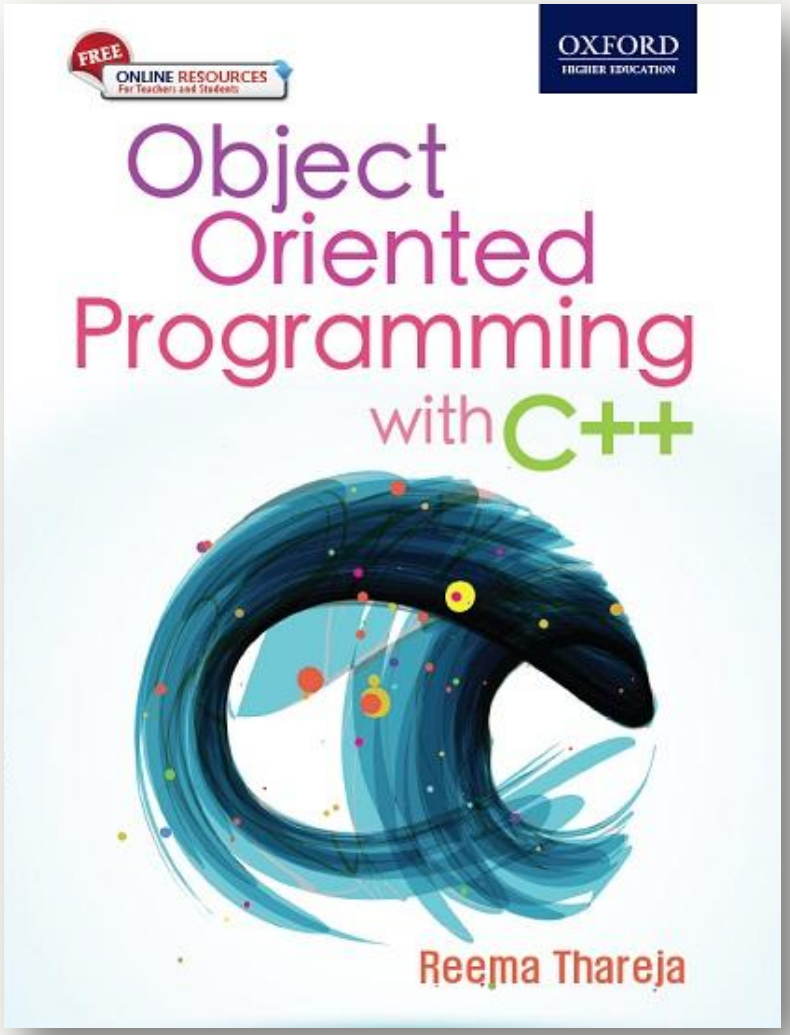

OXFORD
HIGHER EDUCATION

Object Oriented Programming with C++

Reema Thareja

Chapter Nine

Classes and Objects

INTRODUCTION

Classes form the building blocks of the object-oriented programming paradigm. They simplify the development of large and complex projects and help produce software which is easy to understand, modular, re-usable, and easily expandable Syntactically, classes are similar to structures, the only difference between a structure and a class is that by default, members of a class are private, while members of a structure are public.

Although there is no general rule as when to use a structure and when to use a class, generally, C++ programmers use structures for holding data and classes to hold data and functions.

SPECIFYING A CLASS

- A class is the basic mechanism to provide data encapsulation. Data encapsulation is an important feature of object-oriented programming paradigm.

- It binds data and member functions in a single entity in such a way that data can be manipulated only through the functions defined in that class.

- The process of specifying a class consists of two steps—class declaration and function definitions (Fig.).

Class Declaration

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

(a)

```
class Rectangle
{ private:
    float length;
    float breadth;
  public:
    void get_data();
    void show_data();
    float area();
};
```

(b)

The keyword class denotes that the class_name that follows is user-defined data type. The body of the class is enclosed within curly braces and terminated with a semicolon, as in structures. Data members and functions are declared within the body of the class.

Visibility Labels

- Data and functions are grouped under two sections :-Private and Public data.
- **Private Data**:- All data members and member functions declared private can be accessed only from within the class.
- They are strictly not accessible by any entity—function or class—outside the class in which they have been declared. In C++, data hiding is implemented through the private visibility label.
- **Public Data**:- and functions that are public can be accessed from outside the class.
- By default, members of the class, both data and function, are private. If any visibility label is missing, they are automatically treated as private members—private and public.

Defining a Function Inside the Class

- In this method, function declaration or prototype is replaced with function definition inside the class.
- Though it increases the execution speed of the program, it consumes more space.
- A function defined inside the class is treated as an inline function by default, provided they do not fall into the restricted category of inline functions.

```
class rectangle
{   private:
        float length;
float breadth;
    public:
        void get_data()
        {   cout<<"\n Enter the length and breadth of the rectangle: ";
            cin>>length>>breadth;
        }
        void show_data();
        float area();
};
```

Example:- Function inside the class

Defining a Function Outside the Class

- class_name:: and function_name tell the compiler that scope of the function is restricted to the class_name. The name of the :: operator is scope resolution operator.
- The importance of the :: is even more prominent in the following cases.
- When different classes in the same program have functions with the same name. In this case,it tells the compiler which function belongs to which class to restrict the non-member functions of the class to use its private members.
- It allow a member function of the class to call another member function directly without using the dot operator.

```
return_type function_name(list of arguments)
{
FUNCTION BODY
}
```

(a)

Scope resolution operator

```
return_type class_name :: function_name(list of arguments)
{
FUNCTION BODY
}
```

(b)

```
float Rectangle :: area(void)
{   return length * breadth;
}
```

Example:-Defining a function outside the class

CREATING OBJECTS

- To use a class, we must create variables of the class also known as objects.
- The process of creating objects of the class is called class instantiation.
- Memory is allocated only when we create object(s) of the class.
- The syntax of defining an object (variable) of a class is as follows:-class_name object_name; .

```
class Rectangle
{   private:
        float length;
float breadth;
    public:
        void get_data();
        void show_data();
        float area();
}rect1, rect2, rect3;
```

```
Rectangle rect;
```

Example:- Object Creation.

ACCESSING OBJECT MEMBERS

- There are two types of class members—private and public.
- While private members can be only accessed through public members, public members, on the other hand, can be called from main() or any function outside the class.
- The syntax of calling an object member is as follows:- object_name.function_name(arguments);

```
class Sample
{   private:
        int a;
void func1()
        {SOME CODE}
    public:
        int b;
void func2();
        {SOME CODE}
};
main()
{   Sample s;
    s.a = 1;              //Wrong, a is private, cannot be accessed in main()
    s.func1();            //Wrong, func1() is a private function
    s.b = 2;              //Right, b is a public data
    s.func2();            //Right, func2() is a public function
    a = 3;                //Wrong, a is private data
    b = 4;                //Wrong b cannot be accessed without object name
    func2();              //Wrong can't be accessed without object name and '.'
}
```

Example:- Calling of object.

Static Data Members

- When a data member is declared as static, the following must be noted:-
- Irrespective of the number of objects created, only a single copy of the static member is created in memory.
- All objects of a class share the static member.
- All static data members are initialized to zero when the first object of that class is created.
- Static data members are visible only within the class but their lifetime is the entire program.

Static Data Members

- **Relevance:-**Static data members are usually used to maintain values that are common for the entire class. For example, to keep a track of how many objects of a particular class has been created.
- **Place of Storage:-**Although static data members are declared inside a class, they are not considered to be a part of the objects. Consequently, their declaration in the class is not considered as their defination. A static data member is defined outside the class. This means that even though the static data member is declared in class scope, their defination persist in the entire file. A static member has a file scope.

Static Data Members contd.

- However, since a static data member is declared inside the class, they can be accessed only by using the class name and the scope resolution operator.

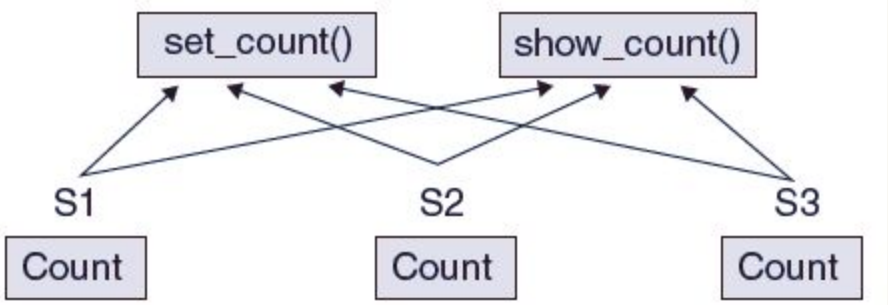

Figure 9.7 Memory allocation for the class sample having non-static data members

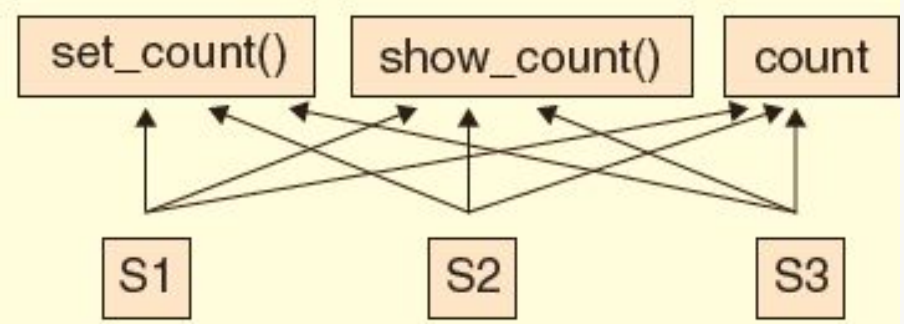

Figure 9.8 Memory allocation for the class Sample having static data member

Example:- Static data members

Static Member Functions

- It can access only the static members—data and/or functions—declared in the same class.
- It cannot access non-static members because they belong to an object but static functions have no object to work with.
- Since it is not a part of any object, it is called using the class name and the scope resolution operator.
- As static member functions are not attached to an object, the this pointer does not work on them.
- A static member function cannot be declared as virtual function.
- A static member function can be declared with const, volatile type qualifiers.

```
#include<iostream.h>
class ID_Generator
{   private:
       static int next_ID;          // next_ID is a static data member
    public:

        static int GenNextID()       // GenNextID is a static member function
    {     cout<<"\n NEXT ID = "<<next_ID++;     }
};
int ID_Generator :: next_ID = 1;
main()
{   for(int i=0;i<5;i++)
    ID_Generator :: GenNextID();
}
```

OUTPUT

```
NEXT ID = 1
NEXT ID = 2
NEXT ID = 3
NEXT ID = 4
NEXT ID = 5
```

Static Object

- To initialize all the variables of an object to zero, we have two techniques:-
- Make a constructor and explicitly set each variable to 0. We will read about it in the next chapter.
- To declare the object as static, when an object of a class is declared static, all its members are automatically initialized to zero.

ARRAY OF OBJECTS

- Just as we have arrays of basic data types, C++ allows programmers to create arrays of user-defined data types as well.
- We can declare an array of class student by simple writing student s[20]; // assuming there are 20 students in a class.
- When this statement gets executed, the compiler will set aside memory for storing details of 20 students.
- This means that in addition to the space required by member functions, 20 * sizeof(student) bytes of consecutive memory locations will be reserved for objects at the compile time.
- An individual object of the array of objects is referenced by using an index, and the particular member is accessed using the dot operator. Therefore, if we write, s[i].get_data() then the statement when executed will take the details of the *i th student.*

OBJECTS AS FUNCTION ARGUMENTS

- Pass-by-value In this technique, a copy of the actual object is created and passed to the called function.
- Therefore, the actual (object) and the formal (copy of object) arguments are stored at different memory locations.
- This means that any changes made in formal object will not be reflected in the actual object.
- Pass-by-reference In this method, the address of the object is implicitly passed to the called function.
- Pass-by-address In this technique, the address of the object is explicitly passed to the called function.

```
#include<iostream.h>
class myClass
{   private:
        int num;
    public:
        void get_data()
        {   cout<<"\n Enter a number : ";
            cin>>num;
        }
        void set_data(myClass o2)     // function accepts object
         {   num = o2.num;
         }
         // function accepts address of object
         void set_data(int a, myClass &o2)
         {   num = o2.num;
         }
         void set_data(myClass *o2) // function accepts pointer
         {   num = o2->num;
         }
         void show_data()
         {   cout<<"\n Num = "<<num;
         }
};
main ()
{   myClass o1, o2, o3, o4;
    int a=0;
    o2.get_data();
    o1.set_data(o2);              //call by value
    o3.set_data(&o2);          //call by pointer
    o4.set_data(a, o2);           //call by reference
    o1.show_data();
    o2.show_data();
    o3.show_data();
}
```

RETURNING OBJECTS

- You can return an object using the following three methods:-
- Returning by value which makes a duplicate copy of the local object.
- Returning by using a reference to the object. In this way, the address of the object is passed implicitly to the calling function.
- Returning by using the ‘this pointer’ which explicitly sends the address of the object to the calling function.

```
Complex Complex :: add(Complex &c2)
{   Complex c3;
    c3.real = real + c2.real;
    c3.imag = imag + c2.imag;
    return c3;
}
```

```
Complex &Complex :: compare(Complex &c2)
{   if(real < c2.real)
        return c2;
    else if(real == c2.real)
    {   if(imag < c2.imag)
            return c2;
        else
        return *this;
    }
}
```

Example:- Returning an object.

this POINTER

- C++ uses the this keyword to represent the object that invoked the member function of the class.
- **this pointer** is an implicit parameter to all member functions and can, therefore, be used inside a member function to refer to the invoking object.

```
#include<iostream.h>
class Sample
{   private:
        int a;
    public:
    voidset_data(int num)
    {   a = num;   }
    void show_data()
    {     cout<<a; }
};
main()
{   Sample s;
    int num = 10;

    s.set_data(num);
    s.show_data();
}
```

OUTPUT

10

CONSTANT MEMBER FUNCTION

- Constant member functions are used when a particular member function should strictly not attempt to change the value of any of the class's data member.

  return_type function_name(arguments) const

- Note that the keyword const should be suffixed in function header during declaration and defination.

```
#include<iostream.h>
class Employee
{   private:
        int emp_no;
        float sal;
    public:
        void set_data(int e, float s);
        void show_data() const; // constant member function, cannot modify data values
};
void Employee :: set_data(int e, float s)
{   emp_no = e;
    sal = s;
}
void Employee :: show_data() const
{   cout<<"\n EMP NO. = "<<emp_no<<" \n Salary = "<<sal;
}
main()
{   Employee e;
    e.set_data(1, 10000);
    e.show_data();
}
```

CONSTANT PARAMETERS

- When we pass constant objects as parameters, then members of the object—whether private or public—cannot be changed.
- This is even more important when we are passing an object to a non-member function as in,

```
void increment(Employee e, float amt)
{ e.sal += amt;
}
```

- Hence, to prevent any changes by non-member of the class, we must pass the object as a constant object. The function will then receive a read-only copy of the object and will not be allowed to make any changes to it. To pass a const object to the increment function, we must write:-

```
void increment(Employee const e, float amt);
void increment(Employee const &e, float amt);
```

```
#include<iostream.h>
class Array
{   private:
        int *arr; // arr is a pointer to integer
        int size;
    public:
        void get_data(int n);
        int get_sum();
        void display_data();
};
void Array :: get_data(int n)
{   size = n;
    arr = new int [size];      //dynamically allocate memory for array
    for(int i=0;i<size;i++)
        cin>>arr[i];
}
void Array :: display_data()
{   for(int i=0;i<size;i++)
        cout<<"\t"<<arr[i];
    cout<<"\n Sum of elements = "<<get_sum();
    cout<<"\n Average of numbers = "<<float(get_sum()/size);
}
int Array :: get_sum()
{   int sum = 0;
    for(int i=0;i<size;i++)
       sum += arr[i];
     return sum;
}
main()
{     Array a; //a is an object of class Array
     int n;
     cout<<"\n Enter the number of elements : ";
     cin>>n;
     a.get_data(n);
     a.display_data();
}
```

LOCAL CLASSES

- A local class is the class which is defined inside a function. Till now, we were creating global classes since these were defined above all the functions, including main().
- However, as the name implies, a local class is local to the function in which it is defined and therefore is known or accessible only in that function.
- There are certain guidelines that must be followed while using local classes, which are as follows:-
- Local classes can use any global variable but along with the scope resolution operator.

LOCAL CLASSES

- Local classes can use static variables declared inside the function.
- Local classes cannot use automatic variables.
- Local classes cannot have static data members.
- Member functions must be defined inside the class.
- Private members of the class cannot be accessed by the enclosing function

```
#include<iostream.h>
int NUM;
main()
{   class Sample
    {   private:
        int a;
        public:
        void get_data()
        {       cout<<"\n Enter the value of a : ";
                cin>>a;
                cout<<"\n Enter the value of global variable : ";
                cin>>::NUM;
        }
        void show_data()
        {       cout<<"\n Class Private Data Member = "<<a;
                cout<<"\n Global Variable = "<<::NUM;
        }
        };
        Sample s;
        s.get_data();
        s.show_data();
        }
```

Programming Tip: Making an object of a local class outside the function in which it is defined will result in a compilation error.

NESTED CLASSES

- C++ allows programmers to create a class inside another class. This is called nesting of classes.
- When a class B is nested inside another class A, class B is not allowed to access class A's members. The following points must be noted about nested classes:-
- A nested class is declared and defined inside another class.
- The scope of the inner class is restricted by the outer class.
- While declaring an object of inner class, the name of the inner class must be preceded with the name of the outer class.

```
#include<iostream.h>
class A
{   public:
          class B
          {   private:
                 int num;
              public:
                 void large(int x, int y)
                 {   num = x;
                     if(num<y)
                         num = y;
                 }
                 void show_data()
                 {   cout<<"\n Large = "<<num;     }
                 };
};
main()
{   A::B b;
    b.large(100, 200);
    b.show_data();
}
```

OUTPUT

```
Large = 200
```

COMPLEX OBJECTS (OBJECT COMPOSITION)

- Complex objects are objects that are built from smaller or simpler objects
- In OOP, it is used for objects that have a has-a relationship to each other. For ex, a car has-a metal frame, has-an engine, etc.
- **Benefits**
- Each individual class can be simple and straightforward.
- A class can focus on performing one specific task.
- The class is easier to write, debug, understand, and usable by other programmers.

COMPLEX OBJECTS (OBJECT COMPOSITION)

- While simpler classes can perform all the operations, the complex class can be designed to coordinate the data flow between simpler classes.
- It lowers the overall complexity of the complex object because the main task of the complex object would then be to delegate tasks to the sub-objects, who already know how to do them.

```
#include<iostream.h>
class One
{   private:
        int num;
    public:
        void set(int i)
        {   num = i;
        }
        int get()
        {   return num;
        }
};
class Two
{   public:
        One O;
        void show()
        {   cout<<"\n Number = "<<O.get();
        }
};
void main()
{   Two T;
    T.O.set(100);
    T.show();
}
```

OUTPUT

```
Number = 100
```

EMPTY CLASSES

- An empty class is one in which there are no data members and no member functions. These type of classes are not frequently used. However, at times, they may be used for exception handling, discussed in a later chapter.
- The syntax of defining an empty class is as follows:- **class class_name{};** .

```
#include<iostream.h>
class Empty{};
main()
{   Empty e;
    cout<<"\n Size of empty class = "<<sizeof(Empty);
    cout<<"\n Size of empty class's object = "<<sizeof(e);
    cout<<"\n Address of empty class's object = "<<hex<<&e;
}
```

OUTPUT

```
Size of empty class = 1
Size of empty class's object = 1
Address of empty class's object = 0x844ffee4c
```

Example:- Empty Classes

FRIEND FUNCTION

- A friend function of a class is a non-member function of the class that can access its private and protected members.
- To declare an external function as a friend of the class, you must include function prototype in the class definition. The prototype must be preceded with keyword friend.
- The template to use a friend function can be given as follows:

```
class class_name
{ -------
  -------
friend return_type function_name(list of arguments);
```

FRIEND FUNCTION contd.

```
--------
    };
   return_type function_name(list of arguments)
  { ---------
    ---------
  }
```

- Friend function is a normal external function that is given special access privileges.
- It is defined outside that class' scope. Therefore, they cannot be called using the **'.' or '->'** operator. These operators are used only when they belong to some class.

FRIEND FUNCTION

- While the prototype for friend function is included in the class definition, it is not considered to be a member function of that class.
- The friend declaration can be placed either in the private or in the public section.
- A friend function of the class can be member and friend of some other class.
- Since friend functions are non-members of the class, they do not require this pointer. The keyword friend is placed only in the function declaration and not in the function definition.
- A function can be declared as friend in any number of classes.
- A friend function can access the class's members using directly using the object name and dot operator followed by the specific member.

```
#include<iostream.h>
class Distance
{   private:
        int meters;
        int kms;
        friend float convert_meters(Distance &d);
        public:
            void get_data()
        {   cout<<"\n Enter kms and metres : ";
            cin>>kms>>meters;
        }
};
float convert_meters(Distance &d)
{   return ((d.kms * 1000) + d.meters);
}
main()
{   Distance d;
    d.get_data();
    cout<<"\n Distance in meters = "<<convert_meters(d);
}
```

Example:- Friend function

FRIEND CLASS

- A friend class is one which can access the private and/or protected members of another class.
- To declare all members of class A as friends of class B, write the following

  declaration in class A.

  friend class B;

```
class B;
class A
{   friend class B;
    data members
    member functions
};
class B
{   data members
    member functions
};
```

FRIEND CLASS

- Similar to friend function, a class can be a friend to any number of classes.
- When we declare a class as friend, then all its members also become the friend of the other class.
- You must first declare the friend becoming class (forward declaration).
- A friendship must always be explicitly specified. If class A is a friend of class B, then class B can access private and protected members of class B. Since class B is not declared a friend of class A, class A cannot access private and protected members of class B.

FRIEND CLASS contd.

- A friendship is not transitive. This means that friend of a friend is not considered a friend unless explicitly specified. For example, if class A is a friend of class B and class B is a friend of class C, then it does not imply—unless explicitly specified—that class A is a friend of class C.

BIT-FIELDS IN CLASSES

- It allow users to reserve the exact amount of bits required for storage of values.
- C++ facilitates users to store integer members into memory spaces smaller than the compiler would ordinarily allow. These space-saving members are called bit fields. In addition to this, C++ permits users to explicitly declare width in bits.
- The syntax for specifying a bit field can be given as follows:

```
class Test
{   unsigned short a: 2;
    unsigned short b: 1;
    int c: 7;
    unsigned short d: 4;
};
```

type:-specifier declarator: constant-expression

```
#include<iostream.h>
#define MATHS 0
#define COMPUTER 1
#define ENGLISH 2
#define IP 1
#define DU 2
#define JNU 3
#define DTU 4
#define AU 5
class Choice
{   private:
        unsigned course : 2;
        unsigned university : 3;
    public:
        void set_data()
        {   course = COMPUTER;
            university = DTU;
        }
        void show_data()
        {   cout<<"\n COURSE = "<<course;
            cout<<"\n UNIVERSITY = "<<university;
        }
};
main()
{   Choice c;
    cout<<"\n Size of Choice's object = "<<sizeof(c);
    c.set_data();
    c.show_data();
}
```

Declaring and Assigning Pointer to Data Members of a Class

Recollect that the syntax for declaring a pointer to normal variable is as follows:

```
data_type *ptr_var;
```

The syntax for declaring a pointer to a class data member is as follows:

```
data_type class_name :: *ptr_var;
```

The syntax for assigning an address to a normal pointer variable is

```
ptr_var = &variable;
```

Similarly, the syntax for assigning the address of a class's data member to a pointer pointing to it is

```
ptr_var = &class_name :: data_member_name;
```

We can declare and assign pointer variables in the same line by writing

```
data_type *ptr_var = &variable;
```

Similarly, we can declare and assign a pointer to a class data member by writing

```
data_type class_name :: *ptr_var = &class_name :: data_member_name;
```

Accessing Data Members Using Pointers

To access a normal variable of a class we write

```
obj.data_member_name; or obj_ptr->data_member_name, if we have a pointer to object.
```

Similarly, to access the same data member using a pointer to the data member, we have to dereference the pointer and write

```
obj.*ptr_var; or obj_ptr->*ptr_var;
```

```
#include<iostream.h>
class Test
{   public:
        int x;
        void show_data();
};
void Test :: show_data()
{   cout<<"\n x = "<<x;     }
main()
{   Test t; // create object
    int Test :: *ptr = &Test :: x; //create pointer to member
    t.*ptr = 20;      // access member through pointer
    t.show_data();
    Test *tp = new Test; // dynamically allocate memory for object
    tp->*ptr = 80; //using pointer to member
    tp->show_data();
}
```

Pointer to Member Functions

Pointers can also be used to point to member functions of a class. The syntax for declaring and assigning a pointer to member function can be given as follows:

```
return_type(class_name :: *ptr_name)(arg_type) :: & clas_name :: function_name;
```

We can also separate declaration and assignment by writing,

```
return_type(class_name :: *ptr_name)(arg_type);
ptr_name = & clas_name :: function_name;
```

The general format to call a function using a pointer to member function is

```
(object_name.*ptr_name)(args);
```

```
#include<iostream.h>
class Test
{   public:
        void show_msg();
};
void Test :: show_msg()
{       cout<<"\n Hello World !!!";     }
main()
{       void (Test :: *fp)(void);
        fp = &Test :: show_msg;
        Test t;
        (t.*fp)();
}
```