C++ and Object Oriented Programming

Unit – 2 (Topic – 1)

Classes & Objects

Prepared By

Prof. Shah Brijesh

C Structure Revisited

-> We know that one of the unique features of the C language is structures.

-> A structure is a convenient tool for handling a group of logically related data items.

-> It is a user defined data type with a template that serves to define its data properties.

For Example:

```
struct Demo
{
        int a;
        float b;
        char c;
};
```

-> The keyword struct declares Demo as a new data type that can hold three fields of different types.

-> These fields are known as structure members or elements.

-> The identifier Demo, which is referred to as structure name or structure tag, can be used to create variables of type Demo.

C Structure Revisited

For Example:

struct Demo d;

-> Here d is a variable of type Demo and has three member variables as defined by the template.

-> Member variables can be accessed using dot or period operator as follows:

d.a = 100;

d.b = 20.54;

d.c = ‘x’;

Note : structures can have arrays, pointers or structures as memebrs.

Limitations of C Structures:

-> The strandatd C does not allow the struct data type to be treated like built-in types.

-> We can not perform any operations on struct type variable.

-> Another important limitation is that they do not permit data hiding . Structure members can be directly accessed by the structure variables by any function anywhere in their scope.

-> In other words, the structure members are public members.

Specifying A Class

Class:

-> 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.

-> When defining a class, we are creating a new abstract data type that can be treated like any other built-in data type..

-> Generally a class specification has two parts

1. Class Declaration
2. Class Function Definitions

Class Declaration:

-> The class declaration describes the type and scope of its members.

Syntax:

```
class Class_name
{
        private:
                var. declaration;
                func. Declaration;
        public:
                var. declaration;
                func. Declaration;
};
```

Specifying A Class

Class:

-> The **class** declaration is similar to a **struct** declaration.

-> The keyword class specifies that what follows is an abstract data of type class_name.

-> The body of a class is enclosed within braces and terminated by a semicolon.

-> The class body contains the declaration of variables and functions.

-> These functions and variables are collectively called class members.

-> They are usually grouped under two sections, namely private and public to denote which of the members are private and which of them are public.

-> The keywords private and public are known as visibility labels.

-> The class members that have been declared as private can be accessed only from within the class.

-> public members can be accessed from outside the class also.

-> The use of keyword private is optional.

-> By default the members of the class are private.

-> The variable declared inside the class are known as member variables(data members) and functions are known as member functions.

-> Only the member functions can have the access to the private data members and private functions.

-> However public members can be accessed from outside the class.

-> The binding of data and functions together into a single class type variable is referred to as encapsulation.

Specifying A Class

Class:

Defining Member Functions:

-> Member functions can be defined in two places.

1. Outside the class definition
2. 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 definition 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.

-> Syntax:

```
returntype class_name :: function_name(arg. list)
{
        function body;
}
```

-> The membership label **class_name::** tells the compiler that the function function_name belongs to the class class_name.

Specifying A Class

Class:

-> That is, the scope of the function is restricted to the class_name specified in the header line.

-> The symbol :: is called the scope resolution operator.

For Example:

-> Assume that we create a class A, which have get() member function which does not return any value.

```
void A :: get()
{
        ..............
        ..............
}
```

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.

Specifying A Class

Class:

Inside the class definition:

For Example:

```
class A
{
        int a;
        public:
                void get()
                {
                        .....................
                }
}
```

-> When a function is defined inside a class, it is treated as an inline function.

-> Therefore, all limitations and restrictions that apply to an inline function are also applicable here.

-> Normally, only small functions are defined inside the class definition.

Creating Objects

Creating Object:

-> We know that object is a variable of type class.

-> Once a class has been defined, we can create any number of objects using the class name (Like any other built-in type)

For Example:

-> Assume that we create a class A, then the statement

```
A a1, a2;
```

-> Create two objects a1 and a2 of type class A.

-> The declaration of an object is similar to that of a variable of any basic type.

-> The necessary memory space is allocated to an object at this stage.

Note: The class specification, like a structure, provides only a template and does not create any memory space for the objects.

-> Objects can also be created when a class is defined by placing their names immediately after the closing brace.

For Example:

```
class A
{
        .....
        .....
}a1, a2;
```

-> would create the objects a1 and a2 of type class A.

Accessing Class Members

Accessing Class Members:

-> We know that private members of a class can be accessed only through the member functions of that class.

-> The main() can not contain statements that access private members.

-> We can access the public members of the class from outside the class using member access operator dot(.).

Syntax:

objectname.functionname(actual-argument);

For Example:

-> The statement **a1.get();**

is valid and call the get().

How Objects Occupy The Memory

How Objects Occupy the Memory:

-> We know that the 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 part of a class specification.

-> Since all the objects belonging to that class use the same member functions, no separate space is allocated for member functions when the objects are created.

-> Only space for member variables is allocated separately for each object.

-> Separate memory locations for the objects are essential because the member variables will hold different data values for different objects.

Nesting Of Member Functions

Nesting Of Member Functions:

-> We know that a 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

For Example:

```
class A
{
        public:
                void show()
                {
                        cout<<"\nShow";                }
                void display()
                {
                        cout<<"\nDisplay"
                        show();                         }
}
void main()
{
        A x;
        x.display();
}
```

Private Member Functions

Private Member Functions:

-> Generally we declare member variables in private section and member functions in public section.

-> We can also declare member functions in private section.

-> We know that we can not access private member of the class from outside the class.

-> Therefore if we declare member functions in private section then we must have to call the functions from any public member function as we call the function in nesting of member functions.

For Example:

```
class A
{
        void show()
        {
                cout<<"\nShow";                    }
        public:
                void display()
                {
                        cout<<"\nDisplay"
                        show();                              }
}
void main()
{
        A x;
        x.display();
}
```

Array Within A Class & Array Of Objects

Array Within A Class:

-> Arrays can be declared as the members of a class.

-> The arrays can be declared as private, public or protected members of the class.

-> To understand the concept of arrays as members of a class, consider this example.

```
#include<iostream.h>
#include<conio.h>
const int size=5;
class student
{
        int roll_no;
        int marks[size];
        public:
                void getdata ();
                void tot_marks ();
                void display()
                {
                        cout<<"\n Roll no="<<roll_no;
                        for(int i=0;i<size;i++)
                                cout<<"\n "<<marks[i]; }
} ;
```

Array Within A Class & Array Of Objects

Array Within A Class:

```
void student :: getdata ()
{
        cout<<"\nEnter roll no: ";
        cin>>roll_no;
        for(int i=0; i<size; i++)
        {
                cout<<"Enter marks in subject"<<(i+1)<<": ";
                cin>>marks[i] ;
        }
}
void student :: tot_marks() //calculating total marks
{
        int total=0;
        for(int i=0; i<size; i++)
                total+ = marks[i];
        cout<<"\n\nTotal marks "<<total;
}
void main()
{
        student stu;
        clrscr();
        stu.getdata() ;
        stud.display();
        stu.tot_marks() ;
        getch();
}
```

Array Within A Class & Array Of Objects

Array Of Objects:

-> An object of class represents a single record in memory.

-> If we want more than one record of class type, we have to create an array of class or object.

-> As we know, an array is a collection of similar type, therefore an array can be a collection of class type.

-> Declaration of array of class type variable is known as array of objects.

```
#include<iostream.h>
#include<conio.h>
class Employee
{
        int Id;
        char Name[25];
        public:
                void GetData()
                {
                        cout<<"\n\tEnter Employee Id : ";
                        cin>>Id;
                        cout<<"\n\tEnter Employee Name : ";
                        cin>>Name;
                }
```

Array Within A Class & Array Of Objects

Array Of Objects:

```
                    void PutData()
                    {
                            cout<<"\n"<<Id<<"\t"<<Name;
                    }
};
void main()
{
        int i; Employee E[3];
        clrscr();
        for(i=0;i<3;i++)
        {
                    cout<<"\nEnter details of "<<i+1<<" Employee";
                    E[i].GetData();
        }
        cout<<"\nDetails of Employees";
        for(i=0;i<3;i++)
                    E[i].PutData();
        getch();
}
```

Static Member Vars & Member Funcs.

Static Member Variables:

-> We can define class members static using **static** keyword.

-> When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.

-> A static member is shared by all objects of the class.

-> All static data is initialized to zero when the first object is created, if no other initialization is present.

-> We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator **::** to identify which class it belongs to.

datatype static-var;

-> Let us try the following example to understand the concept of static data members

Static Member Vars & Member Funcs.

Static Member Variables :

```
#include <iostream.h>
#include<conio.h>
class A
{
        static int a;
        int b;
        public:
                void display()
                {
                        cout<<“\n a=“<<a++;
                        cout<<“\t b=“<<b++;
                }
};
int A :: a;
int main()
{
        clrscr();
        A x,y,z;
        x.display();
        y.display();
        z.display();
        getch();
}
```

Static Member Vars & Member Funcs.

Static Member Functions:

-> By declaring a function member as static, you make it independent of any particular object of the class.

-> A static member function can be called even if no objects of the class exist and the **static** functions are accessed using only the class name and the scope resolution operator **::**.

For Example:

```
classname :: staticfunc();
```

-> A static member function can only access static data member, other static member functions and any other functions from outside the class.

-> Static member functions have a class scope and they do not have access to the **this** pointer of the class.

-> You could use a static member function to determine whether some objects of the class have been created or not.

-> Let us try the following example to understand the concept of static function members

Static Member Vars & Member Funcs.

Static Member Functions :

```
#include <iostream.h>
#include<conio.h>
class A
{
        static int a;
        int b;
        public:
                void display()
                {
                        cout<<“\n a=“<<a++;
                         cout<<“\t b=“<<b++;
                }
                static void show()
                {
                        cout<<“\n a=“<<a++;
                        // cout<<“\t b=“<<b++;
                }

};
int A :: a;
```

Static Member Vars & Member Funcs.

Static Member Functions :

```
int main()
{
        clrscr();
        A x,y,z;
        x.display();
        y.display();
        z.display();
        A :: show();
        x.show();
        getch();
        return 0;
}
```

Object As Function Argument & Returning Objects

Object As Function Arguments:

-> There are following ways of passing objects to functions.

-> Let's assume you have a class X and want to pass it to a function fun, then

1. Pass by Value

-> This creates a shallow local copy of the object in the function scope.

-> Things you modify here won't be reflected in the object passed to it.

-> For example,

Declaration : void fun(X x);

Calling : X x; fun(x);

2. Pass by Reference

-> This passes a reference to the object to the function.

-> Things you modify here will be reflected in the object passed to it.

-> No copy of the object is created.

-> For example,

Declaration : void fun(X &x);

Calling : X x; fun(x);

Object As Function Argument & Returning Objects

Object As Function Arguments:

3. **Pass by const Reference**

-> This passes a const reference to the object to the function.
-> You cannot modify/reassign the object here directly(you can use its methods that do so though).
-> This is useful if you want the function to have only a readonly copy of the object.
-> No copy of the object is created.
-> For example,

Declaration : void fun(X const &x);
Calling : X x; fun(x);

4. **Pass by const Pointer**

-> This passes a const pointer to the object to the function.
-> You cannot modify/reassign the pointer here.
-> This is useful if you want the function to have only the address of this object in the pointer.
-> No copy of object is created.
-> For example,

Declaration : void fun(X *x);
Calling : X x; fun(&x);

Object As Function Argument & Returning Objects

Object As Function Arguments:

5. **Pass by Pointer**

-> This passes a pointer to the object to the function.

-> This is similar to passing a reference to the object.

-> No copy of object is created.

-> For example,

Declaration : void fun(X *x);

Calling : X x; fun(&x);

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

class X
{
        public:
                void disp()
                {
                        cout<<"\ Disp";
                }
};
```

Object As Function Argument & Returning Objects

Object As Function Arguments:

```
void func1(X &x)            //  Pass By reference
{
        x.disp();           }

void func2(X *x)            // Pass By Pointer
{
        x->disp();          }

void func3(X const *x)      // Pass By  Const Pointer
{
        x->disp();          }

void func4(X const &x)      // Pass By Const Reference
{
        x.disp();
}
```

Object As Function Argument & Returning Objects

Object As Function Arguments:

```
void main()
{
        X x;
        clrscr();
        func1(x);
        func2(&x);
        func3(&x);
        func4(x);
        getch();
}
```

Returning Objects :

-> An object is an instance of a class.

-> Memory is only allocated when an object is created and not when a class is defined.

-> As we return any type of value same way we can return objects.

-> An object can be returned by a function using the return keyword.

-> A program that demonstrates this is given as follows

Object As Function Argument & Returning Objects

Returning Objects :

-> An object is an instance of a class.

-> Memory is only allocated when an object is created and not when a class is defined.

-> As we return any type of value same way we can return objects.

-> An object can be returned by a function using the return keyword.

-> A program that demonstrates this is given as follows

Object As Function Argument & Returning Objects

Returning Objects:

```
class A
{
        int a;
        public:
                void get()
                {
                        cout<<“Enter a:”;
                        cin>>a;
                }
                void put()
                {
                        cout<<”\n a=“<<a;
                }
                A sum(A x, A y)
                {
                        A z;
                        z.a = x.a + y.a;
                        return z;
                }
};
```

Object As Function Argument & Returning Objects

Returning Objects:

```
int main()
{
        A x,y,z;
        clrscr();
        x.get();
        y.get();
        z= x.sum(x,y);
        cout<<“\n Object x”;
        x.put();
        cout<<“\n Object y”;
        y.put();
        cout<<“\n Object z”;
        z.put();
        getch();
        return 0;
}
```

Friend Function

Friend Function:

-> We know that the private members cannot accessed fro outside the class.

-> That is, a non member function cannot have an access to the private data of a class.

-> However, there could be a situation where we would like two classes to share a particular function.

-> To make an outside function “friendly” to a class, we have to simply declare this function as a friend of the class.

-> A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class.

-> Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

-> A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

-> To declare a function as a friend of a class, precede the function prototype in the class definition with keyword **friend** as follows

```
class A
{
        int a;
        public:
                friend void print( A x );
                void get(int n);
};
```

Friend Function

Friend Function:

-> The function is defined elsewhere in the program like a normal c++ function.

-> The Function definition does not ise either the keyword friend or the scope operator (::)

-> The function is defined elsewhere in the program like a normal c++ function.

-> A friend function possesses certain special characteristics

1. It is not in the scope of the class to which it has been declared as friend.
2. Since it is not in the scope of the class, it cannot be called using the object of the class.
3. It can be invoked like a normal function without the help of any object.
4. Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator.
5. It can be declared either in the public or the private part of a class without affecting its meaning.
6. Usually, it has the objects as arguments.

Note : The friend functions are often used in operator overloading.

Friend Function

Friend Function:

```
#include<iostream.h>
#include<conio.h>
class A
{
        int a;
        public:
                void get()
                {
                        cout<<"Enter a:";
                        cin>>a;
                }
                void put()
                {
                        cout<<"\n a ="<<a;
                }
                friend A sum(A x, A y);
};
A sum(A x, A y)
{
        A z;
        z.a = x.a + y.a;
        return z;
}
```

Friend Function

Friend Function:

```
int main()
{
        A x,y,z;
        clrscr();
        x.get();
        y.get();
        z= sum(x,y);
        cout<<“\n Object x”;
        x.put();
        cout<<“\n Object y”;
        y.put();
        cout<<“\n Object z”;
        z.put();
        getch();
        return 0;
}
```

Friend Function

Friend Function:

```
#include<iostream.h>
#include<conio.h>
class B;            // Forward Declaration
class A
{
        int a;
        public:
                void get()
                {
                        cout<<"Enter a:";
                        cin>>a;
                }
                void put()
                {
                        cout<<"\n a =<<a;
                }
                friend int sum(A x, B y);
};
```

Friend Function

Friend Function:

```
#include<iostream.h>
#include<conio.h>
class B
{
        int b;
        public:
                void get()
                {
                        cout<<"Enter b:";
                        cin>>b;
                }
                void put()
                {
                        cout<<"\n b ="<<b;
                }
                friend int sum(A x, B y);
};

int sum(A x, B y)
{
        return (x.a+y.b);
}
```

Friend Function

Friend Function:

```
int main()
{
        A x;
        B y;
        clrscr();
        x.get();
        y.get();
        int z= sum(x,y);
        cout<<“\n Object Of Class A”;
        x.put();
        cout<<“\n Object Of Class B ”;
        y.put();
        cout<<“\n Sum = “<<z;
        getch();
        return 0;
}
```

Constant Member Functions

Constant Member Functions:

-> If a member function does not alter any data in the class, then we may declare it as a const member function.

-> The qualifier const is appended to the function prototypes (in both declaration and definition).

-> The compiler will generate an error message if such functions try to alter the data values.

-> For Example:

```
void mul(int, int) const;
double get_balance() const;
```

Pointers To Members

Pointers To Members:

-> It is possible to take the address of a member of a class and assign it to a pointer.

-> The address of a member can be obtained by applying the operator & to a “fully qualified” class member name.

-> A class member pointer can be declared using the opertor ::* with the class name.

-> For Example:

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

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

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

-> The ip pointer created thus acts like a class member in that it must be invoked with a class object.

-> In the statement above the phrase A::* means “pointer to member of A class”.

-> The phrase &A::m means the “address of the m member of A class”.

```
int *ip = &m          // won’t  wprk
```

Pointers To Members

Pointers To Members:

-> This is because m is not simply an int type data.

-> It has meaning only when it is associated with the class to which it belongs.

-> The scope operator must be applied to both the pointer and the member.

-> The pointer ip can now be used to access the member m inside member functions (or friend functions).

-> Let us assume that a is an object of A declared in a member functions.

-> We can access m using the pointer ip as follows:

```
cout<< a.*ip;        same as        cout<<a.m;
```

-> Now, look at the following code:

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

-> 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 the object itself is used with the member pointer.

-> Note that *ip is used like a member name.

-> We can also design pointers to member functions which, then can be invoked using the dereferencing operators in the main as shown below:

```
(object-name .* pointer-to-member-function) (10);
(pointer-to-object ->* pointer-to-member-function) (10);
```

Pointers To Members

Pointers To Members:

```
#include<iostream.h>
#include<conio.h>
class A
{
        int x,y;
        public:
                void set_xy(int a, int b)
                {
                        x = a;
                        y = b;
                }
                friend int sum(A a);
};
int sum(A a)
{
        int A ::* px = &A::x;
        int A ::* py = &A::y;
        A *pm = &m;
        int s = a.*px + pm->*py;
        return s;
}
```

Pointers To Members

Pointers To Members:

```
int main()
{
        A n;
        clrscr();
        void (A::*pf)(int, int) = &A::set_xy;
        (n.*pf)(10,20);
        cout<<“\n Sum =“<<sum(n);
        A *op = &n;
        (op->8pf)(30,40);
        cout<<“\n Sum =“<<sum(n);
        getch();
        return 0;
}
```

Local Classes

Local Classes:

-> Classes can be defined and used inside a function or a block.

-> Such classes are called local classes.

For Example:

```
void test(int a)
{
        ..............
        ..............
        class Student
        {
                ..............
                ..............
        };
        ..................
        ..................
        student s1(a);
        ..................
}
```

Local Classes

Local Classes:

-> Local classes can use global variables (declared above the function) and static variables (declared inside the function) but cannot use automatic local variables.

-> The global variables should be used with the scope resolution operator.

-> There are some restrictions in constructing local classes.

-> They cannot have static data members and member functions must be defined inside the local classes.

-> Enclosing function cannot access the private members of a local class.

-> However, we can achieve this by declaring enclosing function as friend.