C++ Foundation with Data Structures

Topic : Object Oriented Programming - 1

What is Object Oriented Programming?

Object Oriented Programming could be best understood with help of an example. Consider a library management system. Using procedural programming, the problem will be viewed in terms of working happening in the library i.e., issuing of book, returning the book, adding new book etc. The OOP paradigm, however aims at the objects and their interface. Thus in OOP, library management problem will be viewed in terms of the objects involved. Objects are real world entities around which the system revolves. The objects involved are: librarian, book, member etc. Hence, the object-oriented approach views a problem in terms of objects involved rather than procedure for doing it.

Why use Object Oriented Programming?

In real life we deal with lot of objects such as people, car, account, etc. Hence, we need our software to be analogous to real-world objects. Real world objects have data-type properties such as name, age for people, model name for car and balance for account, etc. Moreover, real-world objects can also do certain things such as people talk, cars move, account accumulates, etc. We want our code to mimic the way these objects behave and interact. Hence, Object Oriented Programming allows the program to be closer to real world and thereby making it less complex. Also it makes the software reuse feasible and possible, for example, we don't want to define a student every time we use it, hence using OOP, we just create the blueprint for the student object and use it whenever required.

Classes and objects

The class is a single most important C++ feature that implements OOP concepts and ties them together. Classes are needed to represent real-world entities. A class is a way to bind the data describing an entity and its associated functions together. For instance, consider a ***user*** having characteristics ***username*** and ***password*** and some of its associated operations are ***sign up***, ***sign in*** and ***logout***.

Class is just a template, which declares and defines characteristics and behavior, hence we need to declare objects of the class for it to be usable. In other words class represents a group of similar objects.

Data members and member functions

Data members are the data-type properties that describe the characteristics of the class.

Member functions are the set of operations that may be applied to objects of that class, i.e., they represent the behavioral aspect of the object. They are usually referred as class interface.

Syntax for definition of a class :

```
class class_name {
        Access Modifier:
                Data members
                Member functions
};
```

The class body contains the declaration of members (data and functions).

Declaring objects of a class

We can declare objects of a class either statically or dynamically just as we declare variables of primitive data types.

Statically

Syntax for declaring objects of a class statically is:

```
class_name   object_name;
```

Example code :

```
class Student {
    public :
        int rollno;
        char name[20];
};
int main( ){
    Student s1;       // Declaration of object s1 of type Student
    Student s2;       // Declaration of object s2 of type Student
}
```

Dynamically

Syntax for declaring objects of a class dynamically is:

class_name *object_name = new class_name

Example code :

```
class Student {
    public :
        int rollno;
        char name[20];
};
int main( ){
    Student *s1 =new Student;   // Declaration of object of type Student
                                // dynamically
}
```

Access Modifiers

The class body contains the declaration of its members (data and functions). They are generally two types of members in a class: private and public, which are correspondingly grouped under two sections namely private: and public: .

The **private members** can be accessed only from within the class. These members are hidden from the outside world. Hence they can be used only by member functions of the class in which it is declared.

The **public members** can be accessed outside the class also. These can be directly accessed by any function, whether member function of the class or non-member function. These are basically the public interface of the class.
By default, members of a class are private if no access specifier is provided.

Example code :

```
class A {
    int x, y;
    int sqr(){
            return x * x;
    }
    public :
            int z;
            int twice(){
                    return 2 * y;
            }
```

```
        int test(int i){
                int q = sqr( );       // private function being
        invoked                                  // by member function
                return q + i;
        }
};
int main() {
    A obj;
    obj.z = 10;              // valid. z is a public member
    obj.x = 4;               // Invalid. x is a private member and hence can
                             // be   accessed only by member functions
                            // not  directly by using object
    int j = obj.twice( );   // valid. twice( ) is a public member function
    int k = obj.sqr( );     // Invalid. sqr() is a private member function
    int l = obj.test( );    // valid. test is a public member function
}
```

Getter and setters

The private members of the class are not accessible outside the class, although sometimes there is a necessity to provide access even to private members, in these cases we need to create functions called getters and setters. Getters are those functions that allow us to access data members of the object. However, these functions do not change the value of data members. These are also called accessor function.

Setters are the member functions that allow us to change the data members of an object. These are also called mutator function.

Example code :

```
class Student {
    int rollno;
    char name[20];
    float marks;
    char grade;
    public :
        int getRollno( ){
            return rollno;
        }
        int getMarks( ){
```

```
        return marks;
    }

    void setGrade( ){
        if (marks > 90) grade ='A';;
        else if (marks > 80) grade = 'B';
        else if (marks > 70) grade = 'C';
        else if (marks > 60) grade = 'D';
        else grade = 'E';
    }
};
```

getRollno( ) and getMarks( ) are getter functions and setGrade( ) is a setter function.

Defining Member functions outside the class

We can also define member functions outside the class using scope resolution operator :: .
For example lets move the definition of the two functions defined in student class above outside the class.

```
class Student {
    int rollno;
    char name[20];
    float marks;
    char grade;
    public :
        int getRollno( );

        int getMarks( );

        void setGrade( ){
            if (marks > 90) grade ='A';;
            else if (marks > 80) grade = 'B';
            else if (marks > 70) grade = 'C';
            else if (marks > 60) grade = 'D';
            else grade = 'E';
        }
};
```

```
int  Student::getMarks(){
  return marks;
}

int  Student::getRollNo(){
  return rollNo;
}
```

We can access member functions in similar manner via an object of class Student and using dot operator.

Constructors

A constructor in a class is means of initializing or creating objects of a class. A constructor allocates memory to the data members when an object is created. It may also initialize the object with legal initial value.
A constructor has following characteristics:

- Constructor is a member function of a class and has same name as that of the class.
- Constructor functions are invoked automatically when the objects are created.
- Constructor functions obey the usual access rules. That is, private constructors are available only for member functions, however, public constructors are available for all functions
- Constructor has no return type, not even void.

Default constructor

A constructor that accepts no parameter is called default constructor. The compiler automatically supplies a default constructor implicitly. This constructor is the public member of the class. It allocates memory to the data members of the class and is invoked automatically when an object of that class is created. Having a default constructor simply means that a program can declare instances of the class.

Example code :

```
class Sum {
    int a, b;
    public :
            int getSum(){
```

```
            return a + b;
        }
};
int main() {
    Sum obj;       //implicit default constructor invoked
}
```

Whenever an object of person class is created implicit default constructor is invoked automatically that assigns memory to its data members, i.e., ***name*** and ***age.***

- **Creating your own default constructor**

One can define their own default constructor. When a user-defined default constructor is created, the compiler's implicit default constructor is overshadowed.

Example code :

```
class Sum {
    int a, b;
    public :
        Sum( ) {              // user-defined default constructor
            cout << "constructor invoked";
            a = 10;
            b = 20;
        }
        int getSum(){
            return a + b;
        }
};
int main() {
    Sum obj;                  //explicitly defined default constructor invoked
}
```

Output :

constructor invoked

In this case, user-defined default constructor will be invoked when an object **obj** of person class is created and the data members of that object, ***a*** and ***b***, are initialized to default values 10 and 20.

Parameterized constructor

The constructors that can take arguments are called parameterized constructor.

Example code :

```
class Sum {
    int a, b;
    public :
            Sum(int num1, int num2 ) {          // parameterized constructor
                    a = num1;
                    b = num2;
            }
            int getSum(){
                    return a + b;
            }
 } ;
 int main() {
     Sum obj(4, 2);                  //parametrized constructor invoked
  }
```

Declaring a constructor with arguments hides the default constructor. Hence, the object declaration statement such as

Person obj;

may not work. It is necessary to pass the initial value arguments to the constructor function when an object is declared. This can be done in two ways :

1. By calling constructor explicitly.
   Sum obj = Sum(4 ,2) ;
2. By calling constructor implicitly.(as illustrated in example code)
   Sum obj(4, 2) ;

Destructors

Just as the objects are created, so are they destroyed. If a class can have constructor to set things up, it should have a destructor to destruct the objects. A destructor as the name itself suggests, is used to destroy the objects that have been created by a constructor. A destructor is also a member function whose name is the same as the class name but preceded by tilde ('~'). For instance, destructor for class Sum will be ~Sum( ).

A destructor takes no arguments, and no return types can be specified for it, not even void. It is automatically called by the compiler when an object is destroyed. A destructor frees up the memory area of the object that is no longer accessible.

Example code:

```
class Sum {
    int a, b;
    public :
            Sum(int num1, int num2 ) {              // parameterized constructor
                    cout << "Constructor at work" << endl;
                    a = num1;
                    b = num2;
            }
            ~Sum( ){                                //destructor
                    cout<< "Destructor at work" << endl;
            }
            int getSum(){
                    return a + b;
            }
 }
 int main() {
     Sum obj(4, 6);
  }
```

Output :

Constructor at work
Destructor at work

As soon as obj goes out of scope, destructor is called and obj is destroyed releasing its occupied memory.

NOTE:

- If we fail to define a destructor for a class, the compiler automatically generates one for static allocations.
- Destructors are invoked in the reverse order in which the constructors were called.
- Only the function having access to the constructor and destructor of a class, can define objects of this class types otherwise compiler reports an error.

this keyword

C++ uses a unique keyword called **this** to represent an object that invokes a member function. **this** is a pointer that points to the object for which this function was called. For example, the function call for obj.getSum() will set the

pointer **this** to the address of the object obj. This unique pointer is automatically passed to a member function when it is called. The pointer this acts as an implicit argument to all the member functions.

Example code:

```
class Sum {
    int a, b;
    public :
            Sum(int a, int b ) {
                    this->a = a;
                    this->b = b;
            }
            int getSum(){
                    return a + b;
            }
 }
```

In the constructor of the Sum class, since the data members and data members have the same name, this keyword is used to differentiate between the two. Here, ***this->a*** refers to the data member ***a*** of the object obj.