Chapter 02. Classes & Objects

Class

Class is a way to bind data and its associated functions together. Class can also be defined as user defined data type that contains data members and member functions in it. So, class is a blueprint for object. That is, it defines what an object of the class will consist of and what operations can be performed on such an object.

Specifying a Class

Generally, a class specification has two parts:

1. Class declaration
2. Class function definitions

Class declaration syntax:

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

Objects

Objects are the basic unit of OOP. They are instances of class, in which data members and various member functions that operate on data are bundled as a unit called as object.

Creating Objects

Once a class is declared, we can create variables of that type by using the class name, like any other built-in type variable.

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

For example, let's say we have a class Car which has data members (variables) such as speed, weight, price and functions such as gearChange(), slowDown(), brake() etc. Now let's say I create a object of this class named FordFigo which uses these data members and functions and give them its own values. Similarly we can create as many objects as we want using the blueprint (class).

```
class Car
{
   //Data members
   char name[20];
```

```
    int speed;
    int weight;

public:
    //Functions
    void brake(){
    }
    void slowDown(){
    }
};

int main()
{
   //ford is an object
   Car ford;
}
```

Access Specifiers

Access specifiers in C++ class defines the access control rules. C++ has 3 new keywords introduced, namely,

1. public
2. private
3. protected

These access specifiers are used to set boundaries for availability of members of class be it data members or member functions. Access specifiers in the program, are followed by a colon. You can use one, two or all 3 specifiers in the same class to set different boundaries for different class members.

Public

Public, means all the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. That means, public members can be accessed from outside the class also.

```
class PublicAccess
{
 public:   // public access specifier
 int x;            // Data Member Declaration
```

```
 void display();   // Member Function decaration
}
```

Private

Private members can be accessed only from within the class. That means, the class members declared private cannot be accessed outside that class. If someone tries to access the private member, they will get a compile time error. By default class variables and member functions are private.

```
class PrivateAccess
{
 private:   // private access specifier
 int x;           // Data Member Declaration
 void display();   // Member Function decaration
}
```

Protected

Protected, is the last access specifier, and it is similar to private, it makes class member inaccessible outside the class. But they can be accessed by any subclass of that class. (If class A is inherited by class B, then class B is subclass of class A.)

```
class ProtectedAccess
{
 protected:   // protected access specifier
 int x;           // Data Member Declaration
 void display();   // Member Function decaration
}
```

Defining Member Functions

The definition of member functions can be:

1. Inside the definition of class.
2. Outside the definition of class.

Inside the class definition

If the member function is defined inside the class definition it can be defined directly. If we define the function inside class then we don't not need to declare it first, we can directly define the function. Example,

```
class Cube
{
        public:
        int side;
```

```
        int getVolume()
        {
                return side*side*side;       //returns volume of cube
        }
};
```

Outside the class definition

But if we plan to define the member function outside the class definition then we must declare the function inside class definition and then define it outside. And if its defined outside the class, then we have to use the scope resolution :: operator along with class name along with function name.

Example :

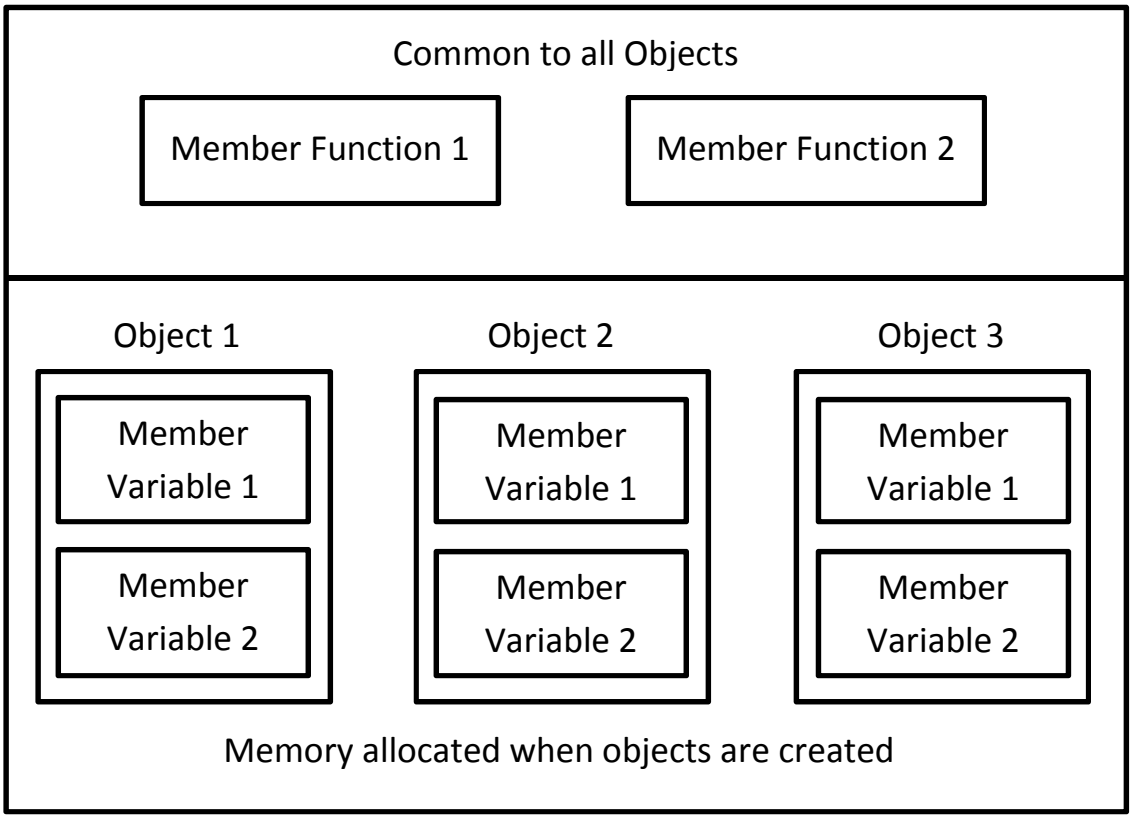
```
class Cube
{
        public:
        int side;
        int getVolume();
}

int Cube :: getVolume()     // defined outside class definition
{
        return side*side*side;
}
```

The main function for both the function definition will be same. Inside main() we will create object of class, and will call the member function using dot . operator.

```
int main()
{
        Cube C1;
        C1.side=4;   // setting side value
        cout<< "Volume of cube C1 ="<< C1.getVolume();
}
```

Memory allocations for objects

Once you define class it will not allocate memory space for the data member of the class. data member of the class can contain different value for the different object thus memory allocation is performed separately for each data member for different object at the time of creating an object. Member function remains common for all objects. Memory allocation is done only once for member function at the time of defining it. The memory allocation of class members is shown below:



Static data members

It is a variable which is declared with the static keyword, it is also known as class member, thus only single copy of the variable creates for all objects.

Characteristics of static member variable are:

1. It is initialized to zero when the first object of the class is created. No other initialization is permitted.
2. Static elements are allocated storage only once in a program lifetime in static storage area.
3. It is visible only within the class, but its scope is entire program lifetime.

Static members are normally used to maintain values common to the entire class.

Any changes in the static data member through one member function will reflect in all other object's member functions.

Declaration

```
static data_type member_name;
```

Defining the static data member

It should be defined outside of the class following this syntax:

```
data_type class_name :: member_name =value;
```

If you are calling a static data member within a member function, member function should be declared as static (i.e. a static member function can access the static data members)

Consider the example, here static data member is accessing through the static member function:

```
#include <iostream.h>
class Demo
{
	private:
		static int X;

	public:
		static void fun()
		{
			cout <<"Value of X: " << X << endl;
		}
};

//defining
int Demo :: X =10;


int main()
{
	Demo X;

	X.fun();

	return 0;
}
```

Output

Value of X: 10

Accessing static data member without static member function

A static data member can also be accessed through the class name without using the static member function (as it is a class member), here we need an Scope Resolution Operator (SRO) :: to access the static data member without static member function.

Syntax:

```
class_name :: static_data_member;
```

Consider the example:

```
#include <iostream.h>
class Demo
```

```
{
	public:
		static int ABC;
};

//defining
int Demo :: ABC =10;


int main()
{

	cout<<"\nValue of ABC: "<<Demo::ABC;
	return 0;
}
```

Output

Value of ABC: 10

In above program ABC is a class member (static data member), it can directly access with help on scope resolution operator.

static member function

A static member function is a special member function, which is used to access only static data members, any other normal data member cannot be accessed through static member function. Just like static data member, static member function is also a class function; it is not associated with any class object.

We can access a static member function with class name, by using following syntax:

```
	class_name:: function_name(perameter);
```

Consider the example:

```
#include <iostream.h>

class Demo
{
	private:
		//static data members
		static int X;
		static int Y;

	public:
	//static member function
	static void  Print()
	{
		cout <<"Value of X: " << X << endl;
```

```
            cout <<"Value of Y: " << Y << endl;
        }
};

//static data members initializations
int Demo :: X =10;
int Demo :: Y =20;


int main()
{
        Demo OB;
        //accessing class name with object name
        cout<<"Printing through object name:"<<endl;
        OB.Print();

        //accessing class name with class name
        cout<<"Printing through class name:"<<endl;
        Demo::Print();

        return 0;
}
```

Output

```
Printing through object name:
Value of X: 10
Value of Y: 20
Printing through class name:
Value of X: 10
Value of Y: 20
```

In above program X and Y are two static data members and print() is a static member function. According to the rule of static in C++, only static member function can access static data members. Non-static data member can never be accessed through static member functions.

friend Function

If a function is defined as a friend function in C++ then the protected and private data of a class can be accessed using the function.

By using the keyword **friend** compiler knows the given function is a friend function.

For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword **friend.**

Declaration of friend function in C++

```
class class_name
{
   friend data_type function_name(argument/s);
```

```
};
```

Example,

```
#include <iostream.h>
class Box
{
   private:
      int length;
   public:
      Box(): length(0) { }
      friend int printLength(Box); //friend function
};
int printLength(Box b)
{
   b.length += 10;
   return b.length;
}
int main()
{
   Box b;
   cout<<"Length of box: "<< printLength(b)<<endl;
   return 0;
}
```

Output:

```
Length of box: 10
```

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.

Syntax for Array of object

```
class class-name
{
      datatype var1;
      datatype var2;
      - - - - - - - - - -
      datatype varN;
```

```
        method1();
        method2();
        - - - - - - - - - -
        methodN();
    };

    class-name obj[ size ];
```

Example for Array of object

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

class Employee
{
    int Id;
    char Name[25];
    int Age;
    long Salary;

    public:
    void GetData()          //Statement 1 : Defining GetData()
    {
        cout<<"\n\tEnter Employee Id : ";
        cin>>Id;

        cout<<"\n\tEnter Employee Name : ";
        cin>>Name;

        cout<<"\n\tEnter Employee Age : ";
        cin>>Age;

        cout<<"\n\tEnter Employee Salary : ";
        cin>>Salary;
    }

    void PutData()          //Statement 2 : Defining PutData()
    {

      cout<<"\n"<<Id<<"\t"<<Name<<"\t"<<Age<<"\t"<<Salary
      ;
    }

};
```

```
void main()
{

    int i;

    Employee E[3];   //Statement 3 : Creating Array of 3 Employees

    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();

}
```

Output :

```
    Enter details of 1 Employee
            Enter Employee Id : 101
            Enter Employee Name : Suresh
            Enter Employee Age : 29
            Enter Employee Salary : 45000

    Enter details of 2 Employee
            Enter Employee Id : 102
            Enter Employee Name : Mukesh
            Enter Employee Age : 31
            Enter Employee Salary : 51000

    Enter details of 3 Employee
            Enter Employee Id : 103
            Enter Employee Name : Ramesh
            Enter Employee Age : 28
            Enter Employee Salary : 47000

    Details of Employees
            101     Suresh     29     45000
            102     Mukesh     31     51000
            103     Ramesh     28     47000
```

Object as function arguments

Like any other data type, Objects of a class can be passed as Function Arguments.

```
#include <iostream.h>
#include <conio.h>
class Demo
{
	private:
		int a;

	public:
		void set(int x)
		{
			a = x;
		}

		void sum(Demo ob1, Demo ob2)
		{
			a  = ob1.a + ob2.a;
		}

		void print()
		{
			cout<<"Value of A :  "<<a<<endl;
		}
};

int main()
{
	//object declarations
	Demo d1;
	Demo d2;
	Demo d3;
	clrscr();

	//assigning values to the data member of objects
	d1.set(10);
	d2.set(20);

	//passing object d1 and d2
	d3.sum(d1,d2);

	//printing the values
	d1.print();
	d2.print();
	d3.print();

	getch();
```

```
    return 0;
}
```

Output

```
Value of A :  10
Value of A :  20
Value of A :  30
```

Concepts of Constructors

Constructor is a special function used to initialize class data members or we can say constructor is used to initialize the object of class.

Characteristics constructor.

- Constructor name class name must be same.
- Constructor doesn't return value.
- Constructor is invoked automatically, when the object of class is created.

Types of Constructor

- Default Constructor
- Parameterize Constructor

Default Constructor

Construct without parameter is called default constructor.

Example of C++ default constructor

```
#include<iostream.h>
#include<string.h>
class Student
{
        int Roll;
        char Name[25];
        float Marks;
        public:
                Student()          //Default Constructor
                {
                        Roll = 1;
                        strcpy(Name,"Kumar");
                        Marks = 78.42;
                }
                void Display()
                {
                        cout<<"\n\tRoll : "<<Roll;
                        cout<<"\n\tName : "<<Name;
                        cout<<"\n\tMarks : "<<Marks;
                }
};
```

```
void main()
{

        Student S;          //Creating Object
        S.Display();        //Displaying Student Details

}
```

Output :

```
Roll : 1
Name : Kumar
Marks : 78.42
```

Parameterize Constructor

Constructor with parameter is called parameterize constructor.

Example of C++ parameterize constructor

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

class Student
{

        int Roll;
        char Name[25];
        float Marks;

        public:

        Student(int r,char nm[],float m)    //Parameterize Constructor
        {
                Roll = r;
                strcpy(Name,nm);
                Marks = m;
        }

        void Display()
        {
                cout<<"\n\tRoll : "<<Roll;
                cout<<"\n\tName : "<<Name;
                cout<<"\n\tMarks : "<<Marks;
        }
};
```

```
void main()
{

	Student S(2,"Sumit",89.63);
	//Creating Object and passing values to Constructor

	S.Display();
	//Displaying Student Details

}
```

Output :

```
Roll : 2
Name : Sumit
Marks : 89.63
```

In parameterize constructor, we have to pass values to the constructor through object.

Copy Constructor

Initialization of an object through another object is called copy constructor. In other words, copying the values of one object into another object is called copy constructor.

Example of C++ copy constructor

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class Student
{
	int Roll;
	char Name[25];
	float Marks;
	public:
		Student(int r,char nm[],float m)//Constructor 1: Parameterize
		{
			Roll = r;
			strcpy(Name,nm);
			Marks = m;
		}

		Student(Student &S)   //Constructor 2 : Copy Constructor
		{
			Roll = S.Roll;
			strcpy(Name,S.Name);
			Marks = S.Marks;
		}
```

```
        void Display()
        {
                cout<<"\n\tRoll : "<<Roll;
                cout<<"\n\tName : "<<Name;
                cout<<"\n\tMarks : "<<Marks;
        }
};


void main()
{

        Student S1(2,"Suman",89.63);

        Student S2(S1);    //Statement 1

        cout<<"\n\tValues in object S1";
        S1.Display();

        cout<<"\n\tValues in object S2";
        S2.Display();

}
```

Output :

```
Values in object S1
Roll : 2
Name : Suman
Marks : 89.63

Values in object S2
Roll : 2
Name : Suman
Marks : 89.63
```

Multiple Constructors in a Class / Constructor Overloading

When we create more than one constructor in a class with different number of parameters or different types of parameters or different order of parameters, it is called as constructor overloading. In sample words, writing multiple constructors in a class is known as constructor overloading in C++.

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

class Example
{
```

```
   int a, b;
public:

   Example()
   {
      a = 50;
      b = 100;
      cout << "\nIn Constructor";
   }


   Example(int x, int y)
   {
      a = x;
      b = y;
      cout << "\nIn Constructor";
   }

   void Display()
   {
      cout << "\nValues :" << a << "\t" << b;
   }
};

int main()
{
   Example Object(10, 20);
   Example Object2;
   Object.Display();
   Object2.Display();
   getch();
   return 0;
}
```

Sample Output

```
Im Constructor
Im Constructor
Values :10      20
Values :50      100
```

Constructors with default arguments

```
#include <iostream.h>

class Demo
{
   private:
      int X,Y;
   public

      Demo()
```

```
	{
		X = 0;
		Y = 0;

		cout << endl << "Constructor Called";
	}
	Demo(int X, int Y=20)
	{
		this->X = X;
		this->Y = Y;

		cout << endl << "Constructor Called";
	}

	~Demo()
	{
		cout << endl << "Destructor Called" << endl;
	}
	void putValues()
	{
		cout << endl << "Value of X : " << X;
		cout << endl << "Value of Y : " << Y << endl;
	}
};

int main()
{
   Demo d1= Demo(10);

   cout << endl <<"D1 Value Are : ";
   d1.putValues();

   Demo d2= Demo(30,40);

   cout << endl <<"D2 Value Are : ";
   d2.putValues();

   return 0;
}
```

Output

```
Constructor Called
D1 Value Are :
Value of X : 10
Value of Y : 20

Constructor Called
D2 Value Are :
Value of X : 30
Value of Y : 40
```

Destructor Called

Destructor Called

Destructors

Constructor allocates the memory for an object.
Destructor deallocates the memory occupied by an object. Like constructor, destructor name and class name must be same, preceded by a tilde(~) sign. Destructor take no argument and have no return value.

Constructor is invoked automatically when the object created.
Destructor is invoked when the object goes out of scope. In other words, Destructor is invoked, when compiler comes out form the function where an object is created.

Example of C++ destructor

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

class Student
{

        int Roll;
        char Name[25];
        float Marks;

        public:

        Student()   //Default Constructor
        {
                Roll = 4;
                strcpy(Name,"Sumit");
                Marks = 84.56;
        }

        void Display()
        {
                cout<<"\n\tRoll : "<<Roll;
                cout<<"\n\tName : "<<Name;
                cout<<"\n\tMarks : "<<Marks;
        }
```

```
        ~Student()   //Destructor
        {
                cout<<"\n\tEnd of program.";
        }

};


void main()
{

        Student S;

        S.Display();

}
```

Output :

```
Roll : 4
Name : Sumit
Marks : 89.56

End of program.
```