Unit II *Classes and Objects & Constructors and Destructors*

Contents:

- **Classes and Objects:** Simple classes (Class specification, class members accessing), Defining member functions, passing object as an argument, Returning object from functions, friend classes, Pointer to object, Array of pointer to object.

- **Constructors and Destructors:** Introduction, Default Constructor, Parameterized Constructor and examples, Destructors

- Summary

- Question Bank

- MCQs

Recommended Books:

1. Object Oriented Analysis and Design , 3rd Edition 2012, Timothy Budd , TMH
2. Mastering C++ , 2nd Edition 2011, K R Venugopal, Rajkumar Buyya, T Ravishankar , Tata McGraw Hill
3. C++ for beginners, 2013, B. M. Hirwani, SPD
4. Effective Modern C++ , SPD, Scott Meyers
5. Object Oriented Programming with C++ , 4TH Edition , E. Balagurusamy , Tata McGraw Hill
6. Learning Python , 5th Edition 2013, Mark Lutz , O' Reilly
7. Mastering Object Oriented Python , 2014, Steven F. Lott , Pact Publishing

Prerequisites

Unit I	Pre-requisites	Sem. III	Sem. IV	Sem. V	Sem. VI
Object Oriented Methodology and Principle of OOPS	--	Java and Data Structures			

Unit II

Classes and Objects:

Simple classes (Class specification, class members accessing)

A class is a user-defined data type that binds data and functions that operate on the data together in a single unit. It allows the data to be hidden from external use. Like other user-defined data types, it also needs to be defined before using its objects in the program. A class definition specifies a new data type that can be treated as built-in data type.

The general form of a class declaration is:

```
class className
{
        private :
                variable declarations;
                function declarations;
        public:
                variable declarations;
                function declaration;
        protected:
                variable declarations;
                function declaration;
   };
```

Where
class, private, public and protected are C++ keywords
className is the name of the class
variable declarations are the variables (*data members*) of the class
function declarations are *member functions* of the class that defines the behaviour of the class

The variables and functions declared within the curly braces are collectively known as members of the class.

The keywords private, public and protected are known as access specifier or visibility mode. Each member of a class is associated with an access specifier that controls its accessibility. When a member is declared as private, it can be accessed only inside the class while a public member is accessible both inside and outside the class. Protected members are accessible both inside the class in which they are declared as well as inside the derived classes of the same class.

Once an access specifier has been used, it remains in effect until another access specifier is encountered or the end of the class declaration is reached. An access specifier is provided by writing the appropriate keyword (private, public or protected) followed by writing a colon ':'. The default access specifier of the members of a class is provate. That is, if no access specifier is provided in the class definition, the access specifier is considered to be private.
For example,

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

```
        };
```
Another example

```
class Student
{
        int rollno;
        char name[30];
        float percentage;
    public:
        void acceptData();
        void displayData();
};
```

In this example, a class named ‘Student’ with three data members ‘rollno’, ‘name’ and ‘percentage’ and two member functions acceptData() and displayData() is created. As no access specifier is provided for data members, they are private by default, whereas, the member functions are declared as public. It implies that the data members are accessible only through the member functions while the member functions can be accessed anywhere in the program.

Generally, data members are declared as private and member functions are declared as public. Declaring the data members as private hides them from the rest of the program. This safeguards the data members and prevents any accidental changes to them by other parts of the program, thereby, implementing the concept of data hiding of object oriented programming. Similarly, specifying the member functions as public provides an interface that is visible and accessible to the other parts of the program.

Note that it is useful to declare the member functions as private if they are to be accessed only within other member functions of the same class and not outside the class.

Defining member functions

Member functions of a class can be defined either inside or outside the class definition. In both the cases the function body remains the same, however, the function header is different.

Inside the class definition

- This is the method of replacing the member function declaration by actual function definition inside the class.
- The member function defined inside a class definition are by default inline functions
- Only small functions can be defined inside the class definition

General form of member function definition:

```
return-type function-name ( )
{
        function body
}
```

For Example :

```
        #include<iostream>
        using namespace std;

        class Cube
        {
```

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

Another example:

```
#include<iostream>
using namespace std;

class Student
{
        int rollno;
        char name[30];
        float percentage;
    public:
        void acceptData()
        {
                cout<< "Enter roll no";
                cin>> rollno;
                cout<< "Enter name";
                cin>> name;
                cout<< "Enter percentage";
                cin>> percentage;
        }
        void displayData()
        {
                cout<< "Roll no: "<< rollno;
                cout<< "Name: "<< name;
                cout<< "Percentage"<< percentage;
        }
};
```

Outside the class definition:

- Defining a member function outside a class requires the function declaration to be provided inside the class definition
- The declaration informs the compiler that the function is a member of the class and that it has been defined outside the class
- The function definition header is different from function declaration, as it includes the class name and the scope resolution operator '::' – between the return-type of a function and function-name. The scope resolution operator informs the compiler what class the member belongs to
- General form of member function definition:

```
return-type class-name : : function-name (argument declaration)
{
        function body
}
```

For example,

```
#include<iostream>
using namespacestd;

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

Another example,

```
#include<iostream>
 using namespacestd;
  class Student
  {
        int rollno;
        char name[30];
        float percentage;
      public:
        void acceptData();
        void displayData();
  };
  void Student :: acceptData()
  {
        cout<< "Enter roll no";
        cin>> rollno;
        cout<< "Enter name";
        cin>> name;
        cout<< "Enter percentage";
        cin>> percentage;
 }
  void Student :: displayData()
  {
        cout<< "Roll no: "<< rollno;
        cout<< "Name: "<< name;
        cout<< "Percentage"<< percentage;
  }
```

In both cases function performs the same task therefore the function body will be identical, only the difference is in defining the header.

Note:

Namespace

Namespace is a container for identifiers. It puts the names of its members in a distinct space so that they don't conflict with the names in other namespaces or global namespace. **using** keyword allows you to import an entire namespace into your program with a global scope.

Creating Objects:
When class is defined, only the specification for the object is defined; no memory or storage is allocated.
To use the data and access functions defined in the class, you need to create objects.
The relation between an object and a class is the same as that of a variable and its data types.

Syntax to Define Object in C++

```
className objectVariableName;
```

For example,

```
class Student
{
    Body of the class
};
void main()
{
    Student s1, s2, s3;
}
```

Another example,

```
class Cube
{
    Body of the class
};
void main()
{
    Cube vol1, vol2, vol3;
}
```

Memory Allocation for Objects
The memory space is allocated to the data member of a class only when an object of the class is declared and not when the data members are declared inside the class. Every object declared for the class has an individual copy of all the data members.

On the other hand, the memory space for the member functions is allocated only once when the class is defined. For class Students, there are three objects as s1,s2,s3. Each object – s1, s2, s3 – has individual copies of the data members – rollno, name and percentage. However, there is only one copy of the member functions acceptData() and displayData() that is shared by all the three objects.

Accessing Data Members of Class

Accessing a data member depends solely on the access control of that data member. If it is public, then the data member can be easily accessed using the direct member access (.) operator with the object of that class.

If, the data member is defined as private or protected, then we cannot access the data variables directly. Then we will have to create special public member functions to access, use or initialize the private and protected data members. These member functions are also called **Accessors** and **Mutator** methods or **getter** and **setter** functions.

Accessing Public Data Members

Following is an example to show you how to initialize and use the public data members using the dot (.) operator and the respective object of class.

For example,

```
class Student
{
 public:
    int rollno;
    char name[30];
};

int main()
{
 Student s1,s2;

 s1.rollno=1;
 s1.name="Adam";

 s2.rollno=2;
 s2.name="Bella";

 cout <<"Name and Roll no of s1 is :"<< s1.name << s1.rollno;
 cout <<"Name and Roll no of s2 is :"<< s2.name << s2.rollno;
}
```

Accessing Private Data Members

To access, use and initialize the private data member you need to create getter and setter functions, to get and set the value of the data member.

The setter function will set the value passed as argument to the private data member, and the getter function will return the value of the private data member to be used. Both getter and setter function must be defined public.

For example :

```
class Student
{
```

```
 private:    // private data member

    int rollno;

 public:     // public accessor and mutator functions

    int getRollno()
   {
     return rollno;
    }

    void setRollno(int i)
    {
       rollno=i;
    }

  };

int main()
{
 Student s;
 s.rollono=1;  //Compile time error
 cout<< s.rollno; //Compile time error
 s.setRollno(1);  //Rollno initialized to 1
 cout<< s.getRollno(); //Output will be 1
}
```

Link: **https://www.youtube.com/watch?v=JhbXZUief8w**

Passing object as an argument

The objects of a class can be passed as arguments to member functions either by value or by reference. When an object is passed by value, a copy of the actual object is created inside the function. This copy is destroyed when the function terminates. Moreover, any changes made to the copy of the object inside the function are not reflected in the actual object. On the other hand, in

pass by reference, only a reference to that object (not the entire object) is passed to the function. Thus, the changes made to the object within the function are also reflected in the actual object.

Whenever an object of a class is passed to a member function of the same class, its data members can be accessed inside the function using the object name and the dot operator. However, the data members of the calling object can be directly accessed inside the function without using the object name and the dot operator.

To understand how objects are passed and accessed within a member function, consider this example.

Example: A program to demonstrate passing objects by value to a member function of the same class

```
#include<iostream.h>
class weight
{
        int kilogram;
        int gram;
    public:
        void acceptdata ();
        void putdata ();
        void sum_weight (weight,weight) ;
} ;
void weight :: acceptdata()
{
        cout<<"\nKilograms:";
        cin>>kilogram;
        cout<<"Grams:";
        cin>>gram;
}
void weight :: putdata ()
{
        cout<<kilogram<<" Kgs. and "<<gram<<"gms. \n";
}
void weight :: sum_weight(weight wl, weight w2)
{
        gram = wl.gram + w2.gram;
        kilogram=gram/1000;
        gram=gram%1000;
        kilogram+=wl.kilogram+w2.kilogram;
}
int main ()
{
        weight wl,w2 ,w3;
        cout<<"Enter weight in kilograms and grams\n";
        cout<<"\n Enter weight #1" ;
        wl.acceptdata();
        cout<<" \n Enter weight #2" ;
        w2. acceptdata();
        cout<<"/n Weight #1 = ";
        wl.putdata();
```

```
        cout<<"Weight #2 = ";
        w2.putdata();
        cout<<"Total Weight = ";
        w3.sum_weight(wl,w2);
        w3.putdata();
        return 0;
}
```

The output of the program is

```
Enter weight in kilograms and grams
Enter weight #1
Kilograms: 12
Grams: 560
Enter weight #2
Kilograms: 24
Grams: 850
Weight #1 = 12 Kgs. and 560 gms.
Weight #2 = 24 Kgs. and 850 gms.
Total Weight = 37 Kgs. and 410 gms.
```

In this example, the sum_weight () function has direct access to the data members of calling object (w3 in this case). However, the members of the objects passed as arguments (w1 and w2) can be accessed within function using the object name and dot operator. Note that, the objects w1 and w2 are passed by value; however, they can re passed by reference also. For example, to pass w1 and w2 by reference to the function sum_weight the function will be declared and defined as

```
void sum_weight (weight &,weight &) ;
void weight :: sum_weight (weight & w1, weight &w2)
{
// body of function

}
```

Returning object from functions

A function can also return objects either by value or by reference. When an object is returned by value from a function, a temporary object is created within the function, which holds the return value. This value is further assigned to another object in the calling function.

The syntax for defining a function that returns an object by value is

```
class_name function_name (parameter_list)
{
// body of the function
}
```

To understand the concept of returning an object by value from a function, consider this example.

Example: A program to demonstrate the concept of returning objects from a function

```
#include<iostream.h>
class weight
{
```

```
        int kilogram;
        int gram;
    public:
        void acceptdata ();
        void putdata ();
        void sum_weight (weight,weight) ;
        weight sum_weight (weight) ;
} ;
void weight :: acceptdata()
{
        cout<<"/nKilograms:";
        cin>>kilogram;
        cout<<"Grams:";
        cin>>gram;
}
void weight :: putdata ()
{
        cout<<kilogram<<" Kgs. and"<<gram<<" gms.\n";
}
weight weight :: sum_weight(weight w2)
{
        weight temp;
        temp.gram = gram + w2.gram;
        temp.kilogram=temp.gram/1000;
        temp.gram=temp.gram%1000;
        temp.kilogram+=kilogram+w2.kilogram;
        return(temp);
}
int main ()
{
        weight w1,w2 ,w3;
        cout<<"Enter weight in kilograms and grams\n";
        cout<<"\n Enter weight #1" ;
        w3=w1.sum_weight (w2);
        wl.acceptdata();
        cout<<" \n Enter weight #2" ;
        w2.acceptdata();
        cout<<"/n Weight #1 = ";
        wl.putdata();
        cout<<"Weight #2 = ";
        w2.putdata();
        cout<<"Total Weight = ";
        w3.sum_weight(wl,w2);
        w3.putdata();

        return 0;
}
```

As the return type of function is weight (an object of class weight), a temporary object temp is created within the function for holding return values. These values are accessed as temp kilogram and temp gram by the function. When the control returns to main (), the object temp is assigned to the object w3 in main( ).

Friend classes

Friend functions

In principle, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not apply to *"friends"*.

Friends are functions or classes declared with the friend keyword.

A non-member function can access the private and protected members of a class if it is declared a *friend* of that class. That is done by including a declaration of this external function within the class, and preceding it with the keyword friend:

```
// friend functions
// friend functions
#include <iostream.h>
#include<conio.h>

class FriendEx
{
   int a, b;
  public:
   void acceptdata()
   {
         cout << "\nEnter two numbers: ";
         cin>>a>>b;
   }
   void putdata()
   {
         cout << "\na= "<<a<<"\nb= "<<b;
   }
   friend void display (FriendEx);
};

void display(FriendEx e)
{
   cout << "\na= "<<e.a<<"\nb= "<<e.b;
}

void main ()
{
  clrscr();
  FriendEx f;
  f.acceptdata();
  cout<<"\nObject data displayed through member function";
  f.putdata();
  cout<<"\nObject data displayed through friend function";
  display(f);
  getch();
}
```

Friend functions are generally used in two situations – firstly, in operator overloading, and second, if a particular function requires to be shared between two or more classes. A function that needs to be shared by two or more classes can be declared as a friend function in all classes.

```
// program to demonstrate friend function in more than one class
// friend functions
#include <iostream.h>
#include<conio.h>

class FriendC2;
class FriendC1
{
   int a, b;
  public:
   void acceptdata()
   {
         cout << "\nEnter two numbers: ";
         cin>>a>>b;
   }
   void putdata()
   {
         cout << "\na= "<<a<<"\nb= "<<b;
   }
   friend void sum (FriendC1,FriendC2);
};
class FriendC2
{
   int a, b;
  public:
   void acceptdata()
   {
         cout << "\nEnter two numbers: ";
         cin>>a>>b;
   }
   void putdata()
   {
         cout << "\na= "<<a<<"\nb= "<<b;
   }
   friend void sum (FriendC1,FriendC2);
};

void sum(FriendC1 c1,FriendC2 c2)
{
   cout << "\na= "<<c1.a+c2.a<<"\nb= "<<c1.b+c2.b;
}

void main ()
{
  clrscr();
  FriendC1 c1;
  FriendC2 c2;
  c1.acceptdata();
```

```
  c2.acceptdata();
  cout<<"\nObject c1 data through member function";
  c1.putdata();
  cout<<"\nObject c2 data through member function";
  c2.putdata();
  cout<<"\nSum of data of both class object";
  sum(c1,c2);
  getch();
}
```

In this example, the function sum() is declared as a friend function of both the classes. The sum() takes object from both the classes as arguments and calculates the sums of their data members.

Friend classes

Similar to friend functions, a friend class is a class whose members have access to the private or protected members of another class:

```
// friend class
// friend functions
#include <iostream.h>
#include<conio.h>

class FriendC1
{
    int a, b;
  public:
    void acceptdata()
    {
          cout << "\nEnter two numbers: ";
          cin>>a>>b;
    }
    void putdata()
    {
          cout << "\na= "<<a<<"\nb= "<<b;
    }
    friend class FriendC2;
};
class FriendC2
{
  public:
    void sum (FriendC1 c1)
    {
          cout << "\nsum= "<<c1.a+c1.b;
    }
};

void main ()
{
  clrscr();
  FriendC1 c1;
  c1.acceptdata();
  cout<<"\nObject c1 data";
```

```
  c1.putdata();
  cout<<"\nSum of data members of object1";
  FriendC2 c2;
  c2.sum(c1);
  getch();
}
```

Link: **https://www.youtube.com/watch?v=HqnVgvBR4pA**

Pointer to object

Just like other pointers, the object pointers are declared by placing in front of a object pointer's name. It takes the following general form :

class-name ∗ object-pointer ;

where class-name is the name of an already defined class and object-pointer is the pointer to an object of this class type. For example, to declare optr as an object pointer of Sample class type, we shall write

Sample ∗optr ;

where Sample is already defined class. When accessing members of a class using an object pointer, the arrow operator (->) is used instead of dot operator.

The following program illustrates how to access an object given a pointer to it. This C++ program illustrates the use of object pointer

```
#include<iostream.h>
#include<stdio.h>
class person
{
    char name[20];
    int age;
   public:
    void acceptData()
    {
       cout<<"Enter name";
       gets(name);
```

```
      cout<<"Enter age";
      cin>>age;
   }
   void disp()
   {
      cout<<"\nEntered name: "<<name<<" and age: "<<age;
   }
};
void main()
{
    person p;
    person *ptr;
    ptr=&p;
    ptr->acceptData();
    ptr->disp();
}
```

To access public members using an object the dot operator is used and to access public members using an object pointer the arrow operator is used.

As you know, when a pointer is incremented, it points to the next element of its type. The same is true of pointer to object.

Array of pointer to object

```
#include<iostream.h>
#include<stdio.h>
class person
{
   char name[20];
   int age;
  public:
   void acceptData()
   {
      cout<<"\nEnter name";
      gets(name);
      cout<<"Enter age";
      cin>>age;
   }
   void disp()
   {
      cout<<"\nEntered name: "<<name<<" and age: "<<age;
   }
};
void main()
{
    int i;
    person p[3];
    person *ptr;
    //ptr=&p;
    ptr=&p[0];
    for (i=0;i<3;i++)
    {
```

```
        ptr->acceptData();
        ptr++;
    }
    //ptr=&p;
    ptr=&p[0];

    for (i=0;i<3;i++)
    {
        ptr->disp();
        ptr++;
    }
}
```

Constructors and Destructors:

In C++, memory is allocated to the data members of a class when an object of the class is declared. Once the object of a class is declared, it needs to be initialized with valid values before it is used in the program. To initialize the objects of a class, a member function of the same class is to be defined that assigns valid values to the data members of the class.

Initialization of objects at the time of declaration is done with the help of a special member function of a class called the constructor. Similarly when an object is no longer needed, the memory allocated to the object needs to be released. The de-allocation is done by another special member function called the destructor.

Constructor

- A constructor is a special member function that constructs storage area for the data members of an object by allocating and initializing memory for them.
- It is special as it must have the same name as the class itself, doesn't have return type and is automatically invoked whenever an object of the class is created.

They have some special characteristics. These are given below:

- They should be declared in the public section for availability to all the functions
- They are called automatically when the objects are created
- They can't return any value as they don't have any return type
- They cannot be inherited, but derived class can call base class constructor
- They cannot be virtual
- They cannot be inherited, but a derived class can call the base class constructor

C++ Constructor Types

1. Default Constructor
2. Parametrized Constructor
3. Copy Constructor
4. Default argument Constructor

1: Default Constructor

If the programmer does not specify the constructor in the program then compiler provides the default constructor.

In C++ we can overload the default compiler generated constructor

In both cases (user created default constructor or default constructor generated by compiler), the default constructor is always parameter less.

Syntax

```
class_name() {
-----
-----
}
```

Example of Default Constructor

Let us take the example of class Marks which contains the marks of two subjects Maths and Science.

```
#include<iostream>
using namespace std;

class Marks
{
public:
  int maths;
  int science;

  //Default Constructor
  Marks() {
    maths=0;
    science=0;
  }

  display() {
    cout << "Maths :  " << maths <<endl;
    cout << "Science :" << science << endl;
  }
};

int main()
{
 //invoke Default Constructor
 Marks m;
 m.display();
 return 0;
}
```

Output :

```
Maths :  0
Science : 0
```

2: Parametrized Constructor

This type of constructor takes the parameters.

Syntax

```
class_name(Argument_List) {
-----
-----
}
```

Example of Parametrized Constructor

Let us take the example of class ‘Marks’ which contains the marks of two subjects Maths and Science.

```
#include<iostream>
using namespace std;

class Marks
{
public:
   int maths;
   int science;

   //Parametrized Constructor
   Marks(int mark1,int mark2) {
      maths = mark1;
      science = mark2;
   }

   display() {
      cout << "Maths :  " << maths <<endl;
      cout << "Science :" << science << endl;
   }
};

int main() {
  //invoke Parametrized Constructor
  Marks m(90,85);
  m.display();
  return 0;
}
```

Output

```
Maths :  90
Science : 85
```

3: Copy Constructor

All member values of one object can be assigned to the other object using copy constructor. For copying the object values, both objects must belong to same class.

Syntax

```
class_Name (const class_Name &obj) {
  // body of constructor
}
```

Example of Copy Constructor

Let us take the example of class ‘Marks’ which contains the marks of two subjects Maths and Science.

```
#include<iostream>
using namespace std;

class marks
{
public:
   int maths;
   int science;

   //Default Constructor
   marks()
   {
     maths=0;
     science=0;
   }

   //Copy Constructor
   marks(const marks &obj)
  {
     maths=obj.maths;
     science=obj.science;
   }

   display(){
     cout<<"Maths :   " << maths
     cout<<"Science : " << science;
   }
};

int main(){
   marks m1;

   /*default constructor gets called
        for initialization of m1  */
   marks m2(const marks &m1);

   //invoke Copy Constructor
   m2.display();

   return 0;
}
```

Output

Maths : 0
Science : 0

4. Default argument Constructor

The parameterized constructor that has all default arguments can be considered as a default constructor since it can be invoked without arguments. This implies that if an object is declared without parentheses, all the data members are initialised with the default values.

A problem arises when both the forms of a constructor are used in a same program.

For example,

```
#include<iostream.h>
#include<conio.h>
Class constructor1
{
        int a,b;
        public:
                Constructor1(intn,int m=11)
                {
                        a=n;
                        b=m;
                }
                void display()
                {
                        cout<<"\nA="<<a<<"\tB"<<b;
                }
};
void main()
{
        clrscr();
        Constructor1 c1(10);
        Constructor1 c2=constructor1(20,30);
        cout<<"\n object 1 data with B as default argument";
        c1.display();
        cout<<"\n object 2 data with B's default value overwritten";
        c2.display();
        getch();
}
```

Link: **https://www.youtube.com/watch?v=CyTSm0sUgi4**

Destructors

C++ Destructor Basics

Destructors are special member functions of the class required to free the memory of the object whenever it goes out of scope.
Destructors are parameterless functions.
Name of the Destructor should be exactly same as that of name of the class. But preceded by '~' (tilde).
Destructors does not have any return type. Not even void.
The Destructor of class is automatically called when object goes out of scope.

For Example

```
#include<iostream>
using namespace std;

class Marks
{
public:
  int maths;
  int science;

  //constructor
  Marks() {
    cout << "Inside Constructor"<<endl;
    cout << "C++ Object created"<<endl;
  }

  //Destructor
  ~Marks() {
    cout << "Inside Destructor"<<endl;
    cout << "C++ Object destructed"<<endl;
  }
};

int main( )
```

```
{
  Marks m1;
  Marks m2;
  return 0;
}
```

Output

```
Inside Constructor
C++ Object created
Inside Constructor
C++ Object created

Inside Destructor
C++ Object destructed
Inside Destructor
C++ Object destructed
```

Explanation :

You can see destructor gets called just before the return statement of main function. You can see destructor code below –

```
~Marks() {
  cout << "Inside Destructor"<<endl;
  cout << "C++ Object destructed"<<endl;
}
```

C++ Destructor always have same name as that of constructor but it just identified by tilde (~) symbol before constructor name.

Below example will show you how C++ destructor gets called just before C++ object goes out of scope.

Another example: Out of scope

```
#include<iostream>
using namespace std;

class Marks
{
public:
  int maths;
  int science;

  //constructor
  Marks() {
    cout << "Inside Constructor"<<endl;
    cout << "C++ Object created"<<endl;
  }

  //Destructor
```

```
  ~Marks() {
     cout << "Inside Destructor"<<endl;
     cout << "C++ Object destructed"<<endl;
  }
};

int main( )
{
  {
  Marks m1;
  }

  cout<<"Hello World !!" <<endl;
  cout<<"Hello World !!" <<endl;
  cout<<"Hello World !!" <<endl;
  cout<<"Hello World !!" <<endl;

  return 0;
}
```

Output :

```
Inside Constructor
C++ Object created
Inside Destructor
C++ Object destructed

Hello World !!
Hello World !!
Hello World !!
Hello World !!
```

Explanation :

In this example we have defined the scope of the object as we declared object inside the curly braces i.e inside the inner block.

```
int main( )
{
  {
  Marks m1;
  }
}
```

So object will be accessible only inside the curly block. Outside the curly block we cannot access the object so destructor gets called when inner curly block ends

In this example Hello World !! gets printed after the destructor. however for below example Hello World !! gets printed before constructor
C++ Destructor Program #3 : before destructor

```
#include<iostream>
using namespace std;
```

```
class Marks
{
public:
  int maths;
  int science;

  //constructor
  Marks() {
    cout << "Inside Constructor"<<endl;
    cout << "C++ Object created"<<endl;
  }

  //Destructor
  ~Marks() {
    cout << "Inside Destructor"<<endl;
    cout << "C++ Object destructed"<<endl;
  }
};

int main( )
{
  Marks m1;

  cout<<"Hello World !!" <<endl;
  cout<<"Hello World !!" <<endl;
  cout<<"Hello World !!" <<endl;
  cout<<"Hello World !!" <<endl;

  return 0;
}
```

Output :

```
Inside Constructor
C++ Object created

Hello World !!
Hello World !!
Hello World !!
Hello World !!

Inside Destructor
C++ Object destructed
```

Summary

- A class is a user-defined data type that binds data and functions that operate on the data together in a single unit. It allows the data to be hidden from external use. Like other user-defined data types, it also needs to be defined before using its objects in the program.

- Member functions of a class can be defined either inside or outside the class definition. In both the cases the function body remains the same, however, the function header is different.

- Namespace is a container for identifiers. It puts the names of its members in a distinct space so that they don't conflict with the names in other namespaces or global namespace. **using** keyword allows you to import an entire namespace into your program with a global scope.

- Accessing a data member depends solely on the access control of that data member. If it is public, then the data member can be easily accessed using the direct member access (.) operator with the object of that class.
- If, the data member is defined as private or protected, then we cannot access the data variables directly. Then we will have to create special public member functions to access, use or initialize the private and protected data members

- When an object is passed by value, a copy of the actual object is created inside the function. This copy is destroyed when the function terminates. Moreover, any changes made to the copy of the object inside the function are not reflected in the actual object.

- When an object is returned by value from a function, a temporary object is created within the function, which holds the return value. This value is further assigned to another object in the calling function.

- In principle, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not apply to *"friends"*.

- Similar to friend functions, a friend class is a class whose members have access to the private or protected members of another class.

- Initialization of objects at the time of declaration is done with the help of a special member function of a class called the constructor. Similarly when an object is no longer needed, the memory allocated to the object needs to be released. The de-allocation is done by another special member function called the destructor.

Question Bank

1. Write a short note on class declaration and objects in C++.
2. What is a class? How does it accomplish data hiding?
3. Describe the different methods of defining member functions in C++
4. How to pass an object to a function? Also explain how to return an object from a function with example.
5. Explain the importance of friend classes with example.
6. a) Explain pointer to objects with example.

   b) Write a program for Array of pointers to object.
7. List characteristics of the constructor. Also explain Default constructor with an Example.

8. Explain Parameterised constructor with an Example.
9. Define destructor. Also write an example and its characteristics.
10. Write a C++ program to create a class for student with the following data members (RollNo, Name, Marks in Five Subjects, Total-Marks, Percentage and Grade.)

    Include three methods as:-

    - For accepting the basic details like RollNo, Name and Marks in Five subjects
    - For calculating the Total-Marks, Percentage and Grade.
    - For displaying all the details of a student

    Using the class find the Grade of TEN students.

MCQs

1. C++ provides a special …… called the constructor, which enables an object to initialize itself when it is created.
   A) friend function
   B) member function
   C) public function
   D) private function
2. A constructor has the same …….. as that of class.
   A) variable
   B) object
   C) function
   D) name
3. Constructors are normally used to …….. and to allocate memory.
   A) define variables
   B) allocate variables
   C) initialize variables
   D) initialize object
4. A constructor that accepts no parameters is called the …….
   A) default constructor
   B) parameterized constructor
   C) implicit constructor
   D) null constructor
5. State whether the following statements about the constructor are True or False.
   i) constructors should be declared in the private section.
   ii) constructors are invoked automatically when the objects are created.
   A) True, True
   B) True, False
   C) False, True
   D) False, False
6. The constructors that can take arguments are called …… constructors.
   A) default constructor
   B) parameterized constructor
   C) implicit constructor
   D) argument constructor

7. When an object is created and initialized at the same time, a ……. gets called.
A) default constructor
B) parameterized constructor
C) implicit constructor
D) copy constructor

8. The …… constructor can be called with either one argument or no arguments.
A) default
B) default argument
C) implicit
D) copy

9. A …….. is used to declare and initialize an object from another object.
A) default constructor
B) default argument constructor
C) implicit constructor
D) copy constructor

10. A ……. takes a reference to an object of the same class as itself as an argument.
A) default constructor
B) default argument constructor
C) implicit constructor
D) copy constructor