1) Constructor ... name.
2) Why we need constructor :- ... class is declared ... me initial value
3) We cannot initilize any data member at the declaration because ... memory is allocated for the data member

CHAPTER THREE

Constructor and Destructor

TOPIC 1 Constructor

A constructor is a special member function of a class that is executed whenever we create new objects of that class. A constructor will have exactly same name as the class and it does not have any return type at all, not even void. Constructors are used to assign initial values for certain member variables.

Special Characteristics of Constructor

Special characteristics of constructor are given below:

(i) Constructors called automatically when the objects are created.
(ii) All objects of the class having a constructors are initialised before their use.
(iii) Constructor should be declared in the public section for availability to all the functions.
(iv) Constructors cannot be static.
(v) The address of the constructor cannot be taken.
(vi) A constructor can call member functions of its class.
(vii) A constructor can have default argument.

Declaration of Constructor

A constructor is declared without any return value excludes void. Therefore, when constructor is implemented, it does not return a value. A general syntax to declare a constructor is:

```
class_name(argument_list);
```

Like member function of a class, there are two ways to define a constructor. These are as follows:

(i) **Inside the class** If we define the constructor inside the class then we don't need to declare it first, we can directly define the constructor.

Syntax

```
class class_name
{
  :
  :
  class_name(argument_list)

  {
    //body of constructor
  }
```

Chapter Checklist

- Constructor
  - Special Characteristics of Constructor
  - Declaration of Constructor
  - Types of Constructor
  - Member Initialisation in Constructor
- Destructor
  - Special Characteristics of Destructor
  - Declaration of Destructor

```
    :
    :
};
```

e.g.

```
class X
{
   public:
      int a;
      X()
      {
         a=0;
      }
};
```

(ii) Outside the class (Using scope resolution (::) operator) If we define the constructor outside the class definition then we must declare the constructor inside the class definition and then define, outside the class using scope resolution (::) operator.

Syntax

```
class class_name
{
   :
   :
   class_name(argument_list);
   :
   :
};
class_name::class_name(argument_list)
{
   //body of constructor
}
```

e.g.

```
class X
{
   public:
      int a;
      X();
};
X::X()
{
   a=0;
}
```

Generally, constructors are in public section, but they can defined as private or protected as well.

Types of Constructor

There are five types of constructor that are given below:

1. Default Constructor

A constructor without arguments is known as a default constructor.

Program 1. *To illustrate the use of default constructor.*

```
#include<iostream.h>
#include<conio.h>
class Line
{
   public:
      void setLength(double len);
      double getLength(void);
      Line();
         //This is the default constructor
   private:
      double length;
};
Line :: Line()  //definition of constructor
                //outside the class
{
   cout<<"Object is being created";
   cout<<endl;
}
void Line :: setLength(double len)
{
   length = len;
}
double Line :: getLength()
{
   return length;
}
void main()
{
   Line line;  //automatically default
               //constructor is called
   line.setLength(6.0);
   cout<<"Length of line ";
   cout<<line.getLength()<<endl;
   getch();
}
```

When the above code is compiled and executed, it produces following result:

Object is being created
Length of line 6

Note *No object of a class is created without a constructor. If the programmer does not provide constructor, then this job is done by the compiler. The compiler automatically provides a default constructor for creating the objects with some dummy values.*

2. Parameterized Constructor

The constructor with arguments is known as parameterized constructor. Parameterized constructors are very helpful in a situation, where the programmer wants to initialise various data elements of different objects with different values, when they are created. In a parameterized constructor, the initial values must be passed at the time of object creation. This can be done in two manners:

(i) By calling the constructor explicitly (Explicit call)

It means that the name of constructor is explicitly provided to invoke it, so that the object has been initialised. Where, the initial value of the private data member can retrieve by the constructor explicitly.

Syntax

```
classname objectname = constructor_name
                       (arguments);
```

(ii) By calling the constructor implicitly (Implicit call)

It means that the constructor is called even its name has not been mentioned in the statement.

Syntax

```
classname objectname(arguments);
```

This method is sometimes called the shorthand method. It is used very often as it is shorter, looks better and easy to implement.

Program 2. *To illustrate the use of parameterized constructor using implict and explicit call.*

```
#include<iostream.h>
#include<conio.h>
class Line
{
   public:
     double getLength(void);
     Line(double len);
   private:
     double length;
};
Line :: Line(double len)
{
    cout<<"length="<<len<<endl;
    length = len;
}
double Line :: getLength(void)
{
    return length;
}
void main()
{
   Line line(10.0); //Implicit call
   cout<<"Length of line ";
   cout<<line.getLength()<<endl;
   Line line1=Line(6.0); //Explicit call
   cout<<"Length of line ";
   cout<<line.getLength()<<endl;
   getch();
}
```

When the above code is compiled and executed, it produces following results:

length = 10
Length of line 10
length = 6
Length of line 10

3. Overloaded Constructor

A constructor can also be overloaded that have the same name but different types of parameters. Constructor overloading is used to increase the flexibility of a class, by having more number of constructors for a single class.

Program 3. *To illustrate the working of overloaded constructor.*

```
#include<iostream.h>
#include<conio.h>
class Overclass
{
   public:
     int x;
     int y;
     Overclass()
     {
        x = y = 0;
     }
     Overclass(int a)
     {
        x = y = a;
     }
     Overclass(int a, int b)
     {
        x = a; y = b;
     }
};
void main()
{
   Overclass A;
   Overclass A1(4);
   Overclass A2(8, 12);
   cout<<"Overclass A's x,y value ";
   cout<<A.x<<","<<A.y<<"\n";
   cout<<"Overclass A1's x,y value ";
   cout<<A1.x<<","<<A1.y<<"\n";
   cout<<"Overclass A2's x,y value ";
   cout<<A2.x<<","<<A2.y<<"\n";
   getch();
}
```

When the above code is compiled and executed, it produces following results:

Overclass A's x,y value 0,0
Overclass A1's x,y value 4,4
Overclass A2's x,y value 8,12

In the above example, the constructor "Overclass" is overloaded thrice with different parameters.

4. Copy Constructor

The copy constructor is a constructor which creates an object by initialising it with an object of the same class, which has been created previously.

The copy constructor is used to:

(i) Initialise one object from another object of the same type.
(ii) Copy an object to pass it as an argument to a function.
(iii) Copy an object to return it from a function.

The copy constructor is defined in the class as a parameterized constructor receiving an object as argument passed-by-reference are given below:

```
class student
{
    ........
    ........
    public:
      student(student &obj);
      //copy constructor declaration
      ........
      ........
};
```

Note *The argument to a copy constructor is passed by reference, the reason being that when an argument is passed by value, a copy of it, is constructed. But the copy constructor is creating a copy of the object for itself, thus, it calls itself. Again, the called copy constructor requires another copy so again it is called. Infact, it calls itself again until the compiler runs out of the memory. So, in the copy constructor, the argument must be passed by reference.*

Syntax

```
Classname object1;
        //Creates first object
Classname object2(object1);
        //Copy constructor
                    or
Classname object2=object1;
        //Copy constructor
```

e.g.

```
student S2(S1);
        //Initialises S2 with contents of S1
                    or
student S2=S1;
        //Initialises S2 with contents of S1
```

Program 4. *To illustrate the working of copy constructor.*

```
#include<iostream.h>
#include<conio.h>
class student
{
    int roll_no;
    int marks;
public:
    student()    //Default constructor
    {
      roll_no=1;
      marks=80;
    }
    student(int x, int y)
        //Parameterized constructor
    {
      roll_no=x;
      marks=y;
    }
    student(student &s)
        //copy constructor
    {
      roll_no=s.roll_no;
      marks=s.marks;
    }
    void show()
    {
      cout<<"\nRoll No = "<<roll_no;
      cout<<"\nMarks = "<<marks;
    }
};
void main()
{
    clrscr();
    student s1;
      //Default constructor called
    cout<<"\nValues of default constructor";
    s1.show();
    student s2(s1);
      //Copy constructor called
    cout<<"\nCopied value of default
                                constructor";
    s2.show();
    student s3(10,90);
      //Parameterized constructor called
    cout<<"\nValues of parameterized
                                constructor";
    s3.show();
    student s4=s3; //Copy constructor called
    cout<<"\nCopied values of parameterized
                                constructor";
    s4.show();
    getch();
}
```

When the above code is compiled and executed, it produces following result:

Values of default constructor

Roll No = 1

```
Marks = 80
Copied value of default constructor
Roll No = 1
Marks = 80
Values of parameterized constructor
Roll No = 10
Marks = 90
Copied values of parameterized constructor
Roll No = 10
Marks = 90
```

5. Constructor with Default Arguments

We can define a constructor with default arguments.

Program 5. *To illustrate the working of default arguments. Calculate interest making use of default arguments.*

```
#include<iostream.h>
#include<conio.h>
class INTEREST
{
    long principal,rate,year;
    float interest;
  public:
    INTEREST(int p,int t,int r = 10);
    //Constructor with default arguments
    void compute();
};
INTEREST :: INTEREST(int p,int t,int r)
{
    principal = p;
    year = t;
    rate = r;
}
void INTEREST :: compute()
{
    cout<<"\nprincipal = "<<principal;
    cout<<"\nrate = "<<rate;
    cout<<"\nyear = "<<year;
    interest = (principal*year*rate)/100;
    cout<<"\nInterest = "<<interest;
}
void main()
{
    INTEREST obj1(2500,2);
    INTEREST obj2(2500,2,15);
    obj1.compute();
    obj2.compute();
    getch();
}
```

When the above code is compiled and executed, it produces the following result:

```
principal = 2500
rate = 10
year = 2
Interest = 500
principal = 2500
rate = 15
year = 2
Interest = 750
```

In this example, the data members principal and year of object obj1 are initialised to 2500 and 2 respectively at the time of creation. The data member rate takes the default value 10.

The data members principal, year and rate of object obj2 are initialised to 2500, 2 and 15 respectively at the time of creation.

Member Initialisation in Constructor

When a constructor is used to initialise other members, these other members can be initialised directly, without resorting to statements in its body. This is done by inserting, before the constructor's body, a colon(:) and a list of initialisations for class members.

e.g. consider a class with the following declaration.

```
class Square
{
   int side;
   public:
     Square(int);
     int area()
     {
        return side*side;
     }
}
```

The constructor for this class could be defined, as usual, as:

```
Square::Square(int x)
{
   side=x;
}
```

But it could also be defined using member initialisation as:

```
Square::Square(int x):side(x)
{
}
```

If for a class C, you have multiple members X, Y, Z, etc. to be initialised and then use the same syntax and separate the members by comma as follows:

```
C::C(double a, double b,double c):X(a), Y(b), Z(c)
{
    :
    :
}
```

① same as the class name (N).
② no return type.
③ Not take arguments.

TOPIC 2 Destructor

A destructor is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

A destructor will have exactly same name as the class, prefixed with a tilde (~) sign and it can neither return a value nor take any parameters. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories, etc.

Special Characteristics of Destructor

Special characteristics of destructor are:

(i) Destructors called automatically when the objects goes out of scope.

(ii) All objects of the class having a destructor, de-initialise each object before the object goes out of scope.

(iii) Destructors should be declared in the public section for availability to all the functions.

(iv) These cannot be static.

(v) The address of the destructors cannot be taken.

(vi) A destructor can call member functions of its class.

(vii) A destructor can have default argument.

Declaration of Destructor

The declaration of destructor is same as the constructor, the class name is used for the name of destructor, with a tilde(~) sign as prefix to it. A general syntax to declare a destructor is:

```
~class_name();
```

Like a constructor, destructor can also be define inside the class or outside the class/using scope resolution operator(::).

e.g.

(i) Destructor inside the class

```
class X
{
  public:
    X()
    {
      :
    }
    ~X
    {
      :
    }
};
```

(ii) Destructor outside the class

```
class X
{
  public:
    X()
    {
      :
    }
    ~X();
};
X::~X()
{
  :
}
```

A destructor being a member function obeys the usual access rules for a member function. That is, if it is defined as a private or protected member function, it becomes available only for member functions. However, if a class defines a public destructor, its object can be created, used and destroyed by any function.

Program: *To illustrate the working of destructor.*

```
#include<iostream.h>
#include<conio.h>
class Line
{
  private:
    double length;
  public:
    void setLength(double len);
    double getLength();
    Line();              //This is the constructor
                         //declaration
    ~Line();             //This is the destructor
                         //declaration
};
Line :: Line()           //Constructor definition
{
  cout<<"Object is being created"<<endl;
}
Line :: ~Line()     //Destructor definition
{
  cout<<"Object is being deleted"<<endl;
}
void Line :: setLength(double len)
{
  length = len;
}
```

```
double Line :: getLength()
{
   return length;
}
void main()
{
   Line line;
   line.setLength(6.0); //set line length
   cout<<"Length of line";
   cout<<line.getLength()<<endl;
   getch();
}
```

When the above code is compiled and executed, produces following result:

Object is being created
Length of line 6
Object is being deleted

Order of Constructor and Destructor Calling

Constructors are called in the order of their object definition while the destructors are called in the reverse order of the constructor called.

EXAM Practice

Short Answer Type Questions [2 Marks]

1. *Write any two differences between constructor and destructor. Write the function headers for constructor and destructor of a class Member.*

Delhi 2013

Or

Differentiate between constructor and destructor functions in a class. Give a suitable example in C++ to illustrate the difference.

Delhi 2012C

Ans A constructor is called when you want to create a new instance of a class. A destructor is called when you want to free up the memory of an object (when you delete it).

A constructor constructs the value of an object. A destructor destructs the value created by the constructor for the object.

e.g. Constructor

```
class Fraction
{
   int m_nNumerator;
   int m_nDenominator;
   public:
      Fraction()    //Default constructor
      {
         m_nNumerator = 0;
         m_nDenominator = 0;
      }
      int GetNumerator()
      {
         return m_nNumerator;
      }
      int GetDenominator()
      {
         return m_nDenominator;
      }
      double GetFraction()
      {
         return(m_nNumerator
                        /m_nDenominator);
      }
};
```

e.g. Destructor

```
class A
{
   public:
      A()
      {
         cout<<"A::A()"<<endl;
      }
      ~A()
      {
         cout<<"A::~A()"<<endl;
      }
};
void main()
{
   char *p = new char[sizeof(A)];
   A *ap = new A;
```