UNIT 9 : CONSTRUCTORS AND DESTRUCTORS

UNIT STRUCTURE

9.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- define and use constructors in programming
- learn about default constructor
- learn to use parameterized contructors
- define and use copy constructor
- learn about destructors
- describe the way of declaring a destrcutor
- initialize object dynamically

9.2 INTRODUCTION

In the previous unit we have studied data members and member functions of class. We have seen so far a few examples of classes being implemented. In all cases, we defined a separate member function for reading input values for data members. With the use of object, member functions are invoked and data members are initialized. These functions cannot be used to initialize the data members at the time of creation of object. C++ provides a pair of special member functions called *constructor* and *destructor*. Constructor enables an object to initialize itself when it is created and destructor destroys the object when it is no longer required.

In this unit we will discuss the constructor and destructor. Different types of contructor and their implementation will also be discussed in this unit.

9.3 CONSTRUCTORS

A constructor is a special member function in a class that is called when a new object of the class is created. It, therefore, provides the opportunity to initialize objects as they are created and to ensure that data members only contain valid values. ***It is a member function whose task is to initialize the object of its class*** and allocate the required resources such as memory. It is distinct from other members of the class because it has the same name as its class.

A constructor can be declared and defined with the following syntax:

```
class  classname
{                           Private members

      ........ ;
      public :                    Constructor declaration

            classname( );
};                                Constructor definition

classname :: classname( )
{
            //Body of constructor
}
```

Constructors have some special constraints or rules that must be followed while defining them. These are as follows:

- Constructor is executed automatically whenever the class is instantiated i.e., object of the class is created.
- It always has the same name as the class in which it is defined
- It cannot have any return type, not even *void*.
- It is normally used to initialize the data members of a class.
- It is also used to allocate resources like memory to the dynamic data members of a class.
- It is normally declared in the public access within the class.

For example, let us declare a class with class name 'circle'. If we use constructor, then the constructor name must also be 'circle'. The program can be written as follows:

/****Program 9.1:*** Program to demostrate constructor while calculating the area of a circle*/

```
#include<iostream.h>
#include<conio.h>
#define PI 3.1415
class circle
{
    private:
    float radius;
    public:
    circle();          //constructor declaration
    void area()        // member function
    {
        cout<<"\nArea of the circle is " ;
        cout<<PI*radius*radius<<" sq.units\n";
    }
};
circle :: circle()   //constructor definition
{
    cout<<"\nEnter radius of the circle:  ";
    cin>>radius;
}
```

```
    int main()
    {
        clrscr();
        circle c;//constructor invoked automatically
        c.area();
        getch();
        return 0;
    }
```

The output of the above program will be :

```
    Enter radius of the circle: 5
    Area of the circle is 78.5375 sq. units
```

In the above program, the constructor **circle()** takes the value of radius from the keyboard. Although the data member is private, the constructor could initialize them. The statement **circle c** declares a variable **c** as object of (type) class **circle**. It also calls the constructor implicitly.

The Default Constructors

The constructor which takes no arguments is called the ***default constructor***. The following code fragment shows the syntax of a default constructor.

```
class class_name
{
    private:
    data members;
    public:
    class_name( );       //default constructor
};
class_name : : class_name( )
{
    /* definition of constructor without any arguments and body */
}
```

If no constructor is defined for a class, then the compiler supplies the default constructor.

Instantiation of Object

Instantiating an object is what allows us to actually use objects in our program. We can write hundreds and hundreds of class declarations, but none of that code will be used until we create an instance of an object. A class declaration is merely a template for what an object should look like. When we instantiate an object, C++ follows the class declaration as if it were a blueprint for how to create an instance of that object.

CHECK YOUR PROGRESS

1. State whether the following statements are True (T) or False (F):

   (i) The constructor is not a member function.

   (ii) It is wrong to specify a return type for a constructor.

   (iii) It is possible to define a class which has no constructor at all.

   (iv) The name of a constructor need not be same as that of the class to which it belongs.

   (v) A class may have two default constructors.

2. Choose the appropriate option:

   (i) A function that is automatically called when an object is created is known as :

(a) constructor	(c) delete
(b) destructor	(d) free( ) function

   (ii) Constructor with no parameter is known as

(a) stand-alone constructor	(c) copy constructor
(b) default constructor	(d) none of these

   (iii) For a class namely *Shape*, the contructor will be like

(a) void Shape() {........}	(b) shape() {.......}
(c) Constructor Shape(){......}	(d) Shape() {.......}

3. Write a program in C++ that displays the factorial of a given number using a constructor member function.
4. Write a program in C++ to find the area of a rectangle of length 10.0 and breadth 6.0 by using a class “rectangle”. The program should also contain a constructor along with other member function.

9.3.1 Parameterized Constructors

The constructor that can take arguments are called ***parameterized constructor.*** The arguments can be separated by commas and they can be specified within braces similar to the argument list in function.

When a constructor has been parameterized, the object declaration without parameter may not work. In case of parameterized constructor, we must provide the appropriate arguments to the constructor when an object is declared. This can be done in two ways:

- By calling the constructor ***explicitly***.
- By calling the constructor ***implicitly***.

The implicit call method is sometimes known as shorthand method as it is shorter and is easy to implement.

Let us consider the following example to demonstrate how parameterized constructor works.

// **Program 9.2:** Program creating parameterized constructor

```
#include<iostream.h>
#include<conio.h>
class student
{
    private:
    int roll,age, marks;
    public:
    student(int r,int m,int a);//parameterized
constructor
```

```
     void display( )
     {
               cout<<“\nRoll number :” <<roll <<endl;
               cout<<“Total marks : <<marks<<endl;
               cout<<“Age:”<<age<<endl;
          }
};   //end of class declaration
student::student(int r, int m, int a) //constructor
definition
{
     roll = r;
     marks =m;                        Implicit
     age=a;                       constructor call
}
int main( )
{
     student manoj(5,430,16);   //object creation
     cout<<“\nData of student 1 : “<<endl;
     manoj.display();
     student  rahul  =  student(6,380,15);//object
creation
     cout<<“\n\nData of student 2 : “<<endl;
     rahul.display();
     getch();
     return 0;                            Explicit
}                                     constructor call
```

The output of the above program will be like this:

```
Data of student 1  :
Roll number        :  5
Total Marks        :  430
Age                :  16
Data of student 2  :
Roll number        :  6
Total Marks        :  380
Age                :  15
```

In the above program, the statement **student(int r, int m, int a,);** is a parameterized constructor with two arguments. In the main( ) function, we see that there are two objects, **manoj** and **rahul** of class type student. The object manoj is initialized with the values 5,430 and16 with the *implicit call* statement **student manoj(5, 430,16);**. The object rahul is initialized with the values 6,380 and 15 with the *explicit call* statement **student rahul = student(6, 380,15);** This statement creates a student object rahul and passes the values 6, 380 and 15 to it.

Constructors with Default Arguments

It is possible to have a constructor with arguments having default values. It means that if we have a parameterized constructor with n parameters, then we can invoke it with less than n parameters specified in the call.

It is useful when most of the objects to be created are likely to have the same value for some data members. We need not specify that in every invocation. For example, suppose we want to record the data of class x (ten) students. For this, we can consider the same class **student** as shown in *Program 8.2.* Most of the students are of age 16. Hence, their birth year is likely to be the same. Only a few of them may have a different year of birth. For this we can use the statement

student(int r, int m, int a=16);

where the third parameter is given the default value. When we write the following statement

student s(1,360);

it assigns the values **1** and **360** to the argument **r** and **m** respectively and **16** to the argument **a** by default. Again, if we write

student t(2,400,15);

it assigns **2**, **400** and **15** to **r**, **m** and **a** respectively. Thus, the actual arguments, when specified, overrides the default values.

EXERCISE

Q. Write a C++ program to create a class "EMPLOYEE" to initialize EMP_ID, DESIGNATION and BASIC_PAY using a constructor and display data for three employees.

9.3.2 Copy Constructors

Object of the same class cannot be passed as argument to constructor to that class by value method.

```
class student
{
    private:
        .......         This is not a valid
    public:             declaration
        student(student s);
};
```

But it is possible to pass an object of the same class as argument to a constructor by reference method. Such type of constructor, i.e., a constructor having parameter which is a reference to object of the same class is called a ***copy constructor***. Thus, the following declaration is a valid declaration.

```
    class student
    {
        private:
                            int roll, marks, age;
//data members
        public:
                            student(student &s);
//copy constructor

    };                          Reference to an object
                                of class student
```

Copy constructor is also called one argument constructor as it takes only one argument. The main use of copy constructor is to initialize the objects while in creation, also used to copy an object. This constructor allows the programmer to create a new object from an existing one by initialization. Let us demostrate copy constructor with the following program :.

//***Program 9.3:*** Program to demonstrate copy constructor

```
#include<iostream.h>
#include<conio.h>
class student
{
    private:
        int roll, marks, age;
    public:
        student(int r,int m,int a)
// parameterized constructor
        {
            rol l = r;
            marks = m;
            age = a;
        }
        student(student &s) //copy constructor
        {
            roll = s.roll;
            marks = s.marks;
            age=s.age;
        }
        void display( )
        {
            cout<<"Roll number :" <<roll <<endl;
            cout<<"Total marks :"<<marks<<endl;
            cout<<"Age:"<<age<<endl;
        }
};
```

```
int main( )
{
    clrscr();
    student t(3,350,17);// or student k(t);
    student k = t;// invokes copy constructor
    cout<<"\nData of student t:"<<endl;
    t.display();
    cout<<"\n\nData of student k:"<<endl;
    k.display();
    getch();
    return 0;
}
```

The outout of the above program will be like this:

```
Data of student t  :
  Roll number  :  3
  Total marks  :  350
          Age  :  17
Data of student k  :
  Roll number  :  3
  Total marks  :  350
          Age  :  17
```

The statements

student t(3,350,17); //invokes parameterized constructor

student k = t; //invokes copy constructor

initialize one object **k** with another object **t**. The data members of **t** are copied member by member into **k**. When we see the output of the program we observe that the data of both the objects t and k are same.

9.4 OVERLOADING OF CONTRUCTORS

A class may contain multiple constructors i.e., a class can have more than one constructor with the same name. The constructors are then recognized depending on the arguments passed. It may differ in terms of arguments, or data types of their arguments, or both. This is called ***overloading of constructors*** or ***constructor overloading***.

Program 9.3 is also an example of constructor overloading as it has two constructors: one is parameterized and the other is a copy constructor. Let us take a suitable example to demostrate overloading of constructors.

//**Program 9.4:** Demonstration of constructor overloading

```
#include<iostream.h>
#include<math.h>
#include<conio.h>
class complex
{
    private:
    float real,imag;
    public:
    complex( )//constructor with no argument
    {
        real = imag = 0.0;
    }
    complex(float r, float i)
    //constructor with two arguments
    {
        real = r;
        imag = i;
    }
    complex(complex &c)  //copy constructor
    {
        real = c.real;
        imag = c.imag;
    }
    complex addition(complex d);
   /*member function returning object and taking
               object as argument */
    void display( )  //Display member function
    {
        cout<<real;
        if(imag < 0)
            cout<<"-i";
        else
            cout<<"+i";
        cout<<fabs(imag);/*fabs calcute the absolute
value
                                    of a  floating point
number */
        }
    };
    complex complex::addition(complex d)
    {
        complex  temp;   //temporary  object  of  type
complex class
        temp.real=real+d.real;  //real parts added
        temp.imag=imag+d.imag;   //imaginary  parts
added
        return(temp);
    }
int main( )
{
    clrscr( );
    complex x1,x4;  //invokes default constructor
    cout<<"\nThe complex numbers in a+ib form :\n\n";
    cout<<"First complex number: ";
    x1.display();
    complex x2(1.5,5.3);//invokes parameterized construc-
tor
```

```
    cout<<"\nSecond complex number: ";
    x2.display();
    complex x3(2.4,1.9);
    cout<<"\nThird complex number: ";
    x3.display();
    cout<<"\nAddition of 2nd and 3rd complex number: ";
    x4=x2.addition(x3);  //function call
    x4.display( );
    cout<<"\nThe result is copied to another object:
";
        complex x5(x4); //invokes copy constructor
        x5.display();
        getch();
        return 0;
    }
```

The output of the above program will be:

```
    The complex numbers in a+ib form:
        First complex number: 0+i0
        Second complex number: 1.5+i5.3
        Third complesx number: 2.4+i1.9
        Addition of second and third complex number: 3.9+i7.2
        The result is copied to another object: 3.9+i7.2
```

We have the following three constructors

```
        complex( );
        complex(float r, float i) ;
    and complex(complex &c);
```

The above program indicates overloading of constructors. These constructors are invoked during the creation of an object depending on the number and types of arguments passed. The default constructor **complex( );** initializes the data members *real* and *imag* to 0.0. In main( ), the statement **complex x2 (1.5,5.3);** passes two parameters to the constructor explicitly with the help of the parameterized constructor **complex (float r, float i);** With the help of copy constructor **complex (complex &c);** data members of one object is copied member by member into another.

9.5 DESTRUCTORS

Like constructors, destructors are also special member functions used in C++ programming language. Destructors have the opposite function of a constructor. The main function of destructors is to free memory and to release resources. Destructors take the same name as that of the class name preceded by a ***tilde ~***. A destructor takes no arguments and has no return type.The general syntax of a destructor is as follows:

```
    class classname
    {
        private :           //Private members
            ........ ;
        public :
            ~classname( );      //Destructor declaration
    };
         classname :: ~classname( )//Destructor defini-
tion
    {
            //Body of destructor
    }
```

In the above, the symbol tilde ~ represents a destructor which precedes the name of the class. Like the default constructor, the compiler always creates a default destructor if we donot create one. Similar to constructors, a destructor must be declared in the public section of a class. Destructor cannot be *overloaded* i.e., a class cannot have more than one destructor.

/* **Program 9.5 :** Program demonstrating destructor */

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class student
{
    private:
        int age;
        char name[25];
    public:
        student(int a, char n[25])
        {
            age=a;
            strcpy(name,n);
        }
        void show()
        {
            cout<<"\nThe  name  of  the  student  is:
"<<name;
            cout<<"\nHis age is: "<<age;
        }
        ~student() //destructor defined
        {
            cout<<"\nObject Destroyed";
            }
    };
    int main()
    {
        student s1(21,"Rahul");
        student s2(23,"Dipankar");
        clrscr();
        s1.show();
        s2.show();
        return 0;
    }
```

To see the execution of destructor, we have to press Alt+F5 after compiling and running the program. The ouput will be like this:

```
The name of the student is :  Rahul
    His age is             :  21

The name of the student is :  Dipankar
    His age is             :  23
```

```
Object Destroyed
Object Destroyed
```

The following points should be kept in mind while defining and writing the syntax for the destructor

- A destructor must be declared with the same name as that of the class to which it belongs. But destructor name should be preceeded by a tilde (~).
- A destructor should be declared with no return type.
- A destructor must have public access in the class declaration.

9.6 DYNAMIC INITIALIZATION OF OBJECTS

We can dynamically initialize objects through constructors. Object's data members can be initialized dynamically at run time even after their creation.

//Program 9.6 Demonstration of dynamic initialization of object

```
#include<iostream.h>
#include<conio.h>
#include<math.h>
class number
{
    public:
    int num;
    number(int n)
    {
        num=n;
    }
    int sum( )
    {
        num=num+5;
        return(num);
    }
};
int main( )
{
    number obj1(1);  // parameterized constructor invoked
    number obj2(2);
    clrscr();
    cout<<"\nValue of object 1 and object 2 are : ";
    cout<<obj1.num<<"\t"<<obj2.num;
    number  obj3(obj1.sum(  ));//dynamic  initialization  of
object
    cout<<"\nValue of object 1 after calling sum() :
"<<obj1.num;
    cout<<"\nValue of object 3 is :"<<obj3.num;
    getch();
    return 0;
}
```

The output of the above program will be :

```
    Value of object 1 and object 2 are      :   1       2
    Value of object 1 after calling sum()   :   6
    Value of object 3 is                    :   6
```

The statements **number obj1(1);** and **number obj2(2);** will initialize obj1 and obj2 by 1 and 2 respectively by invoking the constructor number(int n). **obj1.sum( )** will return the value 6. This is passed as argument in the statement

```
            number obj3(obj1.sum( ));
```

where obj3 is initialized dynamically with the value returned by **obj1.sum( )**.

CHECK YOUR PROGRESS

5. State whether the following statements are true(T) or false(F) :
   (i) A class may have more than one destructor.
   (ii) When an object is destroyed, destructor is automatically called.
   (iii) Presence of many destructor in a class is called destructor overloading.
   (iv) A destructor can have a return type.
   (v) A destructor can have arguments like destructor.
   (vi) We can pass arguments to a constructor.
   (vii) Destructors cannot take arguments.
   (viii) Destructors can be overloaded.
6. Choose the appropriate option:
   (i) Return type of a destructor is
   (a) int (b) tilde
   (c) void (d) nothing, destructor has no return value
   (ii) A class may have ___________ destructor
   (a) two (b) only one
   (c) many (d) none of these
7. Distinguish between the following two statements:
   time t1(14, 10, 30); //statement1
   time t1 = time(14, 10, 30); //statement2

9.7 LET US SUM UP

Based on the discussion so far, the key points to be kept in mind from this unit are:

- A constructor is a special member function for automatic initialiation of an object. Whenever an object is created, the constructor will be called.

- Constructor should have the same name as that of class name to which it belong.
- Constructor has no return type.
- The constructor without argument is called a default constructor.
- Constructors may be overloaded to provide different ways of initializing an object.
- We may have more than one constructor with the same name, provided each has a different signature or arguments list.
- A constructor can accept a reference to its own class as a parameter. Such a constructor having a reference to an instance of its own class as an argument is known as copy constructor.
- C++ also provides another member function called destructor that releases memory by destroying objects when they are no longer required.
- A destructor never takes any argument nor does it return any value.

9.8 FURTHER READING

- Balagurusamy, E. (2011), *Object-oriented programming with C++*, 6e. Tata McGraw-Hill Education
- Venugopal, K.R. (2013), Rajkumar, *Mastering C++*. Tata McGraw-Hill Education
- Ravichandan D. (2002), *Programming with C++*, 2e. Tata McGraw-Hill Education

9.9 ANSWERS TO CHECK YOUR PROGRESS

1. (i) False (ii) True (iii) True (iv) False (v) False
2. (i) (a) constructor (ii) (b) default constructor (iii) (d) Shape() {......}
3. // Program for finding factorial of a number

```
#include<iostream.h>
#include<conio.h>
class factorial
{
        private:
            long n;
        public:
        factorial();
    };
    factorial::factorial()
    {
        cout<<"\nEnter the number to find factorial:
";
        cin>>n;
        long fact =1;
        while(n>1)
        {
            fact =fact *n;
            n=n-1;
        }
        cout<<"\n\nThe factorial is : "<<fact;
    }
    int main()
    {
        factorial f;
        getch();
        return 0;
    }
```

4. /*Program for finding the area of a rectangle of length 10.0 and breadth 6.0*/

```
#include<iostream.h>
#include<conio.h>
class rectangle
{
    private:
        float length,breadth;
    public:
        rectangle( )
        {
            length = 10.0;
            breadth=6.0;
        }
        float area( )
        {
            return( length*breadth);
        }
};
void main( )
{
    rectangle r;
    clrscr();
    cout<<"\n The area of the rectangle  is:
"<<r.area()
        <<" square units";
    getch( );
    return 0;
}
```

5. (i) False (ii)True (iii) False (iv) False
   (v) False (vi) False (vii) True (viii) False
6. (i) (d) nothing, destructor has no return value,
   (ii) (b) only one
7. The first statement creates the object *t1* by calling the *time* constructor implicitly. On the other hand, the second statement creates the object *t1* by calling the *time* constructor explicitly.

9.10 MODEL QUESTIONS

1. What are constructor and destructors? Expalin how they differ from normal member functions.
2. What are the characteristics of a constructor?
3. Write short notes on (i) Default Constructor
   (ii) Copy Constructor

4. Differentiate between constructor and destructors.
5. Give the rules governing the declaration of a constructor and destructor.
6. What is a parameterized constructor ?
7. Write a C++ program to show overloading of constructors?
8. Define a class 'complex_no' which has two data members, one for representing the real part and the other for complex part. Define a constructor to initialize the object.
9. Define suitable constructor(s) for the STRING class defined below;

```
STRING
{
            int length;
            char s[50];
        public:
            //define suitable constructors
};
```

****

UNIT 10 : OPERATOR OVERLOADING

UNIT STRUCTURE

10.1 LEARNING OBJECTIVES

After going through this unit, you will able to :

- describe the fundamental concept of overloading
- describe the use of the keyword *operator*
- illustrate the overloading of unary and binary operators
- describe manipulation of strings using operators

10.2 INTRODUCTION

So far, we have discussed the concept of class and objects and how to allocate required resource such as memory and initialize the objects of classes using constructors and how to deallocate the memories using destructors. C++ offers an another important feature namely *operator overloading*, through which operators like +,-,<=,>= etc. can be used with user defined data types, with some additional meaning.

In this unit, we will concentrates on the dicussion of overloading of operators (unary and binary) as well as the string manipulations using operators.

10.3 BASIC CONCEPT OF OVERLOADING

We know that operators (+,-,<=,>= etc.) are used to perform operation with the constants and variables. Without the use of the operators a programmer cannot write or built an expression. We have already used these operators with the basic data types such as *int* or *float* etc. The operator +(plus) can be used to perform addition of two variables but we cannot apply the + operator for addition of two objects. If we want to add two objects using the + operator then the compiler will show an error message. To avoid this error message you must have to make the compiler aware about the addition process of two objects. To perform operation with objects you need to redefine the definition of various operators. For example, for addition of objects X and Y, we need to define operator +(plus). Re-defining an operator does not change its original meaning. It can be used for both variables of built-in data types as well as objects of user defined data types.

Operator overloading in C++, permits to provide additional meaning to the operators such as +, *, >=, –, = etc., when they are applied to user defined data types. Hence, the operator overloading is one of the most valuable concepts introduced by C++ language. It is a type of polymorphism. We will discuss polymorphism in a later unit. C++ allows the following list of operators for overloading.

Table 10. 1:C++ Overlodable Operators

Operator Category	Operators
Arithmetic	+, -, *, /, %
Bit-Wise	&, |, ~, ^
Logical	&&, ||, !
Relational	<, >, ==, !=, <=, >=
Assignment	=
Arithmetic assignment	+=, -=, *=, /=, %=, &=, |=, ^=
Shift	>>, <<, >>=, <<=
Unary	++, --
Subscripting	[ ]
Function call	( )
Dereferencing	->
Unary sign prefix	+, -
Allocate and free	new, delete

10.4 *operator* KEYWORD

The keyword *operator* helps in overloading of the C++ operators. The general format of operator overloading is shown below :

```
ReturnType opeator OperatorSymbol ([arg1], [arg2])
{
    // body of the function
}
```

Here, the keyword **operator** indicates that the *OperatorSymbol* is the name of the operator to be overloaded. The operator overloaded in a class is known as *overloaded operator function.*

The following statements shows the use of the *operator* keywords.

```
class Index
{
    // class data and member function
    Index operator ++( )
    {
          index temp;
```

```
                        value = value+1;
                        temp.value = value;
                        return temp;
                    }
        };
```

Here, return type of the operator function is the name of a class within which it is declared. It can be defined as follows :

```
    class Index
      {
          // class data and member function
             Index operator ++( );
    };

Index Index :: operator ++( )
{
    index temp;
    value = value+1;
    temp.value = value;
    return temp;
}
```

The operator function should be either a member function or a friend function. When the operator function is declared as member function and takes no argument, it is known as *unary operator overloading* and when it takes one argument it is known as *binary operator overloading*.

10.5 OVERLOADING UNARY OPERATORS

When an operator function takes no argument, it is called as unary operator overloading. You are already familiar with the operators ++, --, and -, *which have only single operands* are called unary operators. The unary operators ++ and -- can be used as **prefix** or **suffix** with the functions. The following program demonstrates the overloading of unary '--' operators.

// Program 10.1

```
#include<iostream.h>
#include<conio.h>
class unary
{
    int x, y, z;
    public :
    unary (int i, int j, int k) // parameterized construc-
tor
    {
        x=i; y=j; z=k;
    }
    void display();//displays contents of member variables
    void operator --();//overloads the unary operator --
};
void unary :: operator --()
{
    --x; --y; --z;// values of variables will decrease by 1
}
void unary :: display()
{
      cout<<"X="<<x<<"\n";
      cout<<"Y="<<y<<"\n";
      cout<<"Z="<<z<<"\n";
}
void main()
{
    clrscr();
    unary A(31, 41, 51);
    cout<<"\n Before Decrement of A :\n";
    A.display();
    --A;// calls the function operator --()
    cout<<"\n After Decrement of A :\n";
    A.display();
    getch();
}
```

```
OUTPUT :   Before Decrement of A   :   X=31
                                       Y=41
                                       Z=51
           After Decrement of A    :   X=30
                                       Y=40
                                       Z=50
```

10.6 OPERATOR RETURN TYPE

In the above example, we have declared the operator function of type *void* i.e. it will not return any value. But it is possible to return value and assign it to other object of same type. The return value of the operator is always of the class type, it means that class name will be in the place of the return type specification because we are applying the operator overloading properties only for the objects. Always remember that an operator cannot be overloaded for basic data types, so the return value of operator function will be of class type. The following program demonstrates the operator return types :

// Program 10.2

```
#include<iostream.h>
#include<conio.h>
class unary
{
    int x;
    public :
    unary () { x=0; }
    int getx() // returns the current value of variable x
    { return x; }
    unary operator ++();
};
unary unary :: operator ++()
{
    unary temp;
    x=x+1;
      temp.x=x;
      return temp;
}
void main()
{
    clrscr();
    unary A1,A2;
    cout<<"\n A1="<<A1.getx();
    cout<<"\n A2="<<A2.getx();
    A1=A2++;//first increment the value of A2 and assigns it to
A1
    cout<<"\n A1="<<A1.getx();
    cout<<"\n A2="<<A2.getx();
    A1++; // object A1 is incresed
    cout<<"\n A1="<<A1.getx();
    cout<<"\n A2="<<A2.getx();
    getch();
}
```

RUN :

```
A1 = 0
A2 = 0
A1 = 1
A2 = 1
A1 = 2
A2 = 1
```

10.7 OVERLOADING BINARY OPERATORS

Binary operators are overloaded by using member functions and friend functions. The difference is in the number of arguments used by the function. In the case of binary operator overloading, when the function is a member function then the number of arguments used by the operator member function is one. When the function defined for the binary operator overloading is a friend function, then it uses two arguments. Here, we will discuss the overloading of binary operator when the operator function is a member function.

Binary operator overloading, as in unary operator overloading, is performed using a keyword *operator*. The following program demonstrates the overloading of binary operators.

```
    // Program 10.3
#include <iostream.h>
#include<conio.h>
class Binary
{
    private:
      int x;
      int y;
    public:
      Binary()  //Constructor
      { x=0; y=0; }
    void getvalue() //Member Function for Inputting values
    {
      cout <<"\n Enter value for x:";
      cin >> x;
      cout << "\n Enter value for y:";
      cin>> y;
    }
    void displayvalue( )//Member Function for Outputting Values
    {
      cout<<"\n\nThe resultant value : \n";
      cout <<" x =" << x <<"; y ="<<y;
    }
    Binary operator +(Binary);//Binary is class name
};
Binary Binary :: operator +(Binary e2)
//Binary operator overloading for + operator defined
{
    Binary temp;
    temp.x = x+ e2.x;
    temp.y = y+e2.y;
    return (temp);
}
void main( )
{
    Binary e1,e2,e3;  //Objects e1, e2, e3 created
    clrscr();
    cout<<"\nEnter value for Object e1:";
    e1.getvalue( );
    cout<<"\nEnter value for Object e2:";
    e2.getvalue( );
    e3= e1+ e2;  //Binary Overloaded operator used
    e3.displayvalue( );
    getch();
}
```

OUTPUT :
```
Enter value for Object e1  :
         Enter value for x  :  10
         Enter value for y  :  20
  Enter value for Object e2  :
         Enter value for x  :  30
         Enter value for y  :  40
       The Resultant Value  :  x = 40; y = 60
```

In the above example, the class Binary has created three objects e1, e2, e3. The values are entered for objects e1 and e2. The binary operator overloading for the operator ‘+’ is declared as a member function inside the class Binary. The definition is performed outside the class Binary by using the scope resolution operator and the keyword *operator*.

The important aspect is that the following two statements are equivalent.

```
e3= e1 + e2; // e3= e1.operator +(e2)
```

The binary overloaded operator ‘+’ is used. When the compiler encounters such expressions, it examines the argument type of the operator. In this statement, the argument on the left side of the operator ‘+’, e1, is the object of the class Binary in which the binary overloaded operator '+' is a member function. The right side of the operator '+' is e2. This is passed as an argument to the operator '+' i.e. the expression means **e3 = e1.operator +(e2).** The operator returns a value (binary object temp in this case), which can be assigned to another object (e3 in this case).

Since the object e2 is passed as argument to the operator'+' inside the function defined for binary operator overloading, the values are accessed as e2.x and e2.y. This is added with e1.x and e1.y, which are accessed directly as x and y.

Always remember that, in the overloading of binary operators, the left-hand operand is used to invoke the operator function and the right-hand operand is passed as an argument to the operator function.

10.8 STRINGS AND OPERATOR OVERLOADING

We are already familiar with the *strcat()* function which is used for concatenation of strings. Consider the following two strings

```
char str1[50]   = "Bachelor of Computer";
char str2[20]   = "Application";
```

The string str1 and str2 are combined, and the result is stored in str1 by invoking the function *strcat()* as follows :

```
strcat(str1, str2);
```

The same operation can be done by defining a string class and overloading the + operator. The following program demonstrates the concatenation of two string using the overloading concept.

// Program 10.4

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
class String
{
      private:
      char str[100];
      public:
      String()   //Constructor
      { strcpy(str," "); }

      String(char *msg)  //Constructor
      { strcpy(str, msg); }

      void display() //Member Function for Display strings
      {
          cout <<str;
      }
      String operator +(String s);
    };

    String String :: operator +(String s)
    //Binary operator overloading for + operator defined
    {
          String temp = str;
          strcat(temp.str, s.str);
          return temp;
    }
void main( )
{
    clrscr();
    String str1 = "Bachelor of Computer";
    String str2 = "Application";
    String str3;
    str3= str1+str2;
    cout<<"\n str1 =";
    str1.display();
    cout<<"\n str2 =";
    str2.display();
    cout<<"\n The String after str3=str1+str2 \n \n";
    str3.display();
    getch();
    }
```

In this program, the concatenation is performed by creating a temporary string object *temp* and initializing it with the first string. The second string is added to first string in the object temp using the *strcat()* and finally the resultant temporary string object temp is returned. Here, in the program, the length of str1+str2 should not exceed the array size 100 (i.e. char str[100]).

Thus, we have seen that, in C++ programming language, operator overloading adds new functionality to its existing operators. The programmer must add proper comments concerning the new functionality of the overloaded operator. The program will be efficient and readable only if operator overloading is used only when necessary. The operators that cannot be overloaded are - **.**, **?:**, **sizeof**, **::**, **.*** , #, ##.

10.9 TYPE CONVERSION

We cannot convert between user-defined data types(classes) just as we can convert between basic types. This is beacuse the compiler does not know anything about the user-defined type.

Now, let us look into how C++ handles conversions for its **built-in types (int, float, char, double etc.)**. When you make a statement assigning a value of one standard type to a variable of another standard type, C++ automatically will convert the value to the same type as the receiving variable, provided the two types are compatible.

For example, the following statements all generate numeric type conversions:

```
long count = 8;    // int value 8 converted to type long
double time = 11;// int value 11 converted to type double
int side = 3.33;  // double value 3.33 converted to type int 3
```

These assignments work because C++ recognizes that the diverse numeric types all represent the same basic thing, a number, and because C++ incorporates built-in rules for making the conversions. However, you can lose some precision in these conversions. For example, assigning

3.33 to the int variable results in only getting the value 3, losing the 0.33 part.

The C++ language does not automatically convert types that are not compatible.

For example, the statement

```
        int * p = 10;  // type clash
```

fails because the left-hand side is a pointer-type, whereas the right-hand side is a number. And even though a computer may represent an address internally with an integer, integers and pointers conceptually are quite different. For example, you wouldn't square a pointer. However, when automatic conversions fail, you may use a type cast:

```
        int * p = (int *) 10;  // ok, p and (int *) 10 both pointers
```

This sets a pointer to the address 10 by type casting 10 to type pointer-to-int (that is, type int *).

Now, let us look into how C++ handles conversions from basic type to user-defined types vice-versa.

Basic type to user defined type :

This type of conversion can be easily carried out. It is automatically done by the compiler with the help of in-built routines or by type casting. In this type the left hand operand of = sign is always class type or user defined type and the right hand side operand is always basic type. The following program explains this type of conversion.

```
    //Program 10.5
    #include <iostream.h>
    #include<conio.h>

    class Test
{
      private:
        int x;
        float y;
        public:
        Test()   //Constructor
        { x=0; y=0; }
        Test(float z)  //Constructor with one argument
        { x=2; y=z; }
        void display()  // Function for displaying values
        {
            cout <<"\n  x =" << x <<" y ="<<y;
            cout <<"\n  x =" << x <<" y ="<<y;
        }
};
void main( )
{
        Test a;
        clrscr();
        a=9;
        a.display();
        a=9.5;
        a.display();
        getch();
}
```

OUTPUT :

```
x=2     y=9
x=2     y=9
x=2     y=9.5
x=2     y=9.5
```

In the above program, the class Test has two data member of type integer and float. It also has two constructors one with no arguments and the second with one argument. In *main()* function, **a** is an object of class Test. When a is created the constructor with no argument is called and data memberms are initialize to zero. When a is initialized to 9 the constructor with float argument i.e. ***Test(float z)*** is invoked. The integer value is converted to float type and assigned to data member y. Again, when a is assigned to 9.5, same process repeated. Thus, the conversion from basic to class type is carried out.

User defined type to basic type :

As we know, the compiler does not have any prior information about user defined data type using class, so in this type of conversion it needs to inform the compiler how to perform conversion from class to basic type. For this purpose, a conversion function should be defined in the class in the form of the operator function. The operator function is defined as an overloaded basic data type which takes no arguments. The syntax of such a conversion function is shown below-

```
operator Basic type( )
        {
                // steps for converting
        }
```

In the above syntax, you have noticed that, the conversion function has no return type. While declaring the operator function the following condition should always remember :

i) the operator function should not have any argument.
ii) it has no any return type.
iii) it should be a class member.

The following program demonstrates this conversion mechanism :

```
// Program 10.6
#include <iostream.h>
#include<conio.h>
class Time
{
    private:
    int hour;
      int minute;
      public:
        Time(int a)
        {
            hour=a/60;
            minute=a%60;
        }
        operator int()
        {
            int a;
            a=hour*60+minute;
            return a;
        }
};
void main( )
{
    clrscr();
    Time t1(500);
    int i=t1;  //  operator int() is invoked
    cout<<"\n"<<"The value of i:"<<i;
    getch();
}
```

OUTPUT : The value of i : 500

In the above program, the statement ***int i=t1***, invokes the operator function which finally converts a time object to corresponding magnitude (of type int).

CHECK YOUR PROGRESS

1. Choose the correct answer from the following :
   a) Operator overloading is
      i) making C++ operators work with objects
      ii) giving C++ operators more than they can handle
      iii) giving new meaning to existing C++ operators
      iv) making new C++ operators

b) To convert from a user-defined class to basic type, you would use
   i) a built-in conversion function
   ii) a one argument constructor
   iii) an overloaded = operator
   iv) a conversion function that is a member of the class

c) To convert from a basic type to user-defined class, you would use
   i) a built-in conversion function
   ii) a one argument constructor
   iii) an overloaded = operator
   iv) a conversion function that is a member of the class

d) _______operator must have one class object.

i) +	ii) new
iii) all	iv) none of these

e) Binary overload operators are passed _______ arguments.

i) one	ii) two
iii) no	iv) none of the above

2. Fill in the blanks :
   i) The statement x=y will cause __________ if the objects are of different classes.
   ii) __________ is making operators to woork with user defined data types.
   iii) Single argument constructor is usually defined in the __________ class.
   iv) __________ function must not have a return type.
   v) __________ are operators that act on only one operand.

10.10 LET US SUM UP

- Operator overloading is one of the important concepts in C++ which allows to provide additional meaning to operators +, -, >=, <= etc. when they are applied to user defined data types.

- Overloaded operators are redefined within a class using the keyword ***operator*** followed by an operator symbol. When an operator is overloaded, the produced symbol is called the operator function name.
- Overloading of operator cannnot change the basic meaning of an operator. When an operator is overloaded, its prooperties like systax, precedence and associativity remain constant.
- Operators ++, --, and -, *which have only single operands* are called unary operators. The unary operators ++ and -- can be used as **prefix** or **suffix** with the functions.
- The binary operators require two operand. Binary operators are overloaded by using member functions and friend functions.
- The operators which cannot be overloaded are - **.**, **?:**, **sizeof**, **::**, **.*** , #, ##.
- The concept of operator overloading can also be applied to data conversion. C++ offers automatic conversion of primitive data types.
- Actually there are three possibilities of data conversion :
  a) Basic type to user defined type(class type)
  b) User defined type(class type) to basic type
  c) Class type to another class type (we have not discussed here)

10.11 FURTHER READINGS

- Balagurusamy, E. (2011), *Object-oriented programming with C++*, 6e. Tata McGraw-Hill Education
- Venugopal, K.R. (2013), Rajkumar, *Mastering C++*. Tata McGraw-Hill Education
- Ravichandan D. (2002), *Programming with C++*, 2e. Tata McGraw-Hill Education

10.12 ANSWERS TO CHECK YOUR PROGRESS

1. a) iii,	b) iv,	c) iii,
d) ii,	e) i.	

2. i) Compiler error,
   ii) operator overloading,
   iii) destination,
   iv) casting operator,
   v) unary operator

10.13 MODEL QUESTIONS

1. What is operator overloading? Give the advantage of operator overloading.
2. What is operator function? Describe operator function with syntax and examples.
3. What is the difference between overloading of binary operators and of unary operators?
4. Explain the conversion from basic type to user defined type(class type) with examples.
5. Explain the conversion from user defined type(class type) to basic type with examples.
6. Write a program to overload the -- operator.
7. Write a program to overload the binary operator + in order to perform addition of complex numbers.
8. Write a program to overload the relational operator (>, <, ==) in order to perform the comparision of two strings.

****