Unit -3

COPY CONSTRUCTOR

A copy constructor is used to declare and initialize an object from another object. A copy constructor takes a reference to an object of the same class as itself as an argument.
For example, the statement:

```
integer I2(I1);
```

would define the object I2 and at the same time initialize it to the values of I1.
Another form of this statement is

```
integer I1 = I1;
```

The process of initializing through a copy constructor is known as copy initialization.

DYNAMIC CONSTRUCTOR

Allocation of memory to objects at the time of their construction is known as Dynamic construction of objects. The memory is allocated with the help of the new operator.

```
/* Example program for dynamic constructor and
   Destructor */
#include <iostream.h>
#include <string.h>
class String
{
        char *name;
        int length;
 public:
        String(char *s)
        {
                length = strlen(s);
                name = new char[length + 1];
                strcpy( name, s );
        }
        void display(void)
        {
                cout << name << "\n";
        }
        ~String( );
};
String :: ~String( )
{
        delete name;
}

void main( )
{
        String name1("Vinayaga ");
        String name2("College ");
        name1.display( );
        name2.display( );
}
```

DESTRUCTORS:

A ***destructor***, as the name implies, is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a **tilde(~).** For example , the destructor for the class String can be defined as shown below:

~String( ) { }

A destructor never takes any argument nor does it return any value. It will be invoked

implicitly be the compiler upon exit from the program to clean up storage that is no longer accessible. It is a good practice to declare destructors in a program since it releases memory space for future use.

Whenever **new** is used to allocate memory in the constructors, we should use **delete** to free that memory.

const OBJECTS

We may create and use constant objects using **const** keyword before object declaration.

Example: **const** Matrix X(m,n);

Any attempt to modify the values of m and n will generate compile-time error.

```
/* Program using class, object, member function, constructors and Destructors  for calculating area and perimeter of a circle */
#include <iostream.h>
#include <conio.h>
class circle
{
private:
        float radius, area, perimeter;
public:
        circle() { }          //Default constructor
        circle(float r)    //Parameterized Constructor
        {
                radius = r;
                area = 0;
                perimeter = 0;
        }
        circle(float a, float p, float r=25)
            // Constructor with Default Argument
        {
                radius = r;
                area = a;
                perimeter=p;
        }
        circle(circle & x) //Copy constructor
        {
                radius   = x.radius;
                area     = x.area;
                perimeter= x.perimeter;
        }

        void read(); //Member function1 Declaration
       // Member function 2 defined inside the class
        void display()
        {
        cout<<"\n\n Given radius is :";
        cout<<radius;
        cout<<"\n Area of the circle is :";
        cout<<3.14 * radius * radius;
        cout<<"\n Perimeter of the circle is :";
```

```
int main()
{
        clrscr();
        cout<<"\n Program for calculating area and
                perimeter of a circle";
        cout<<"\n-------------------------------------";
        float n;

        circle C1;

        circle C3(0,0);

        circle C4[10];

        C1.read();
        cout << "\n ** Default Constructor **\n";
        C1.display();       //

        cout<<"\n ** Parameterized Constructor and
                Dynamic Initialization of Objects **";
        cout<<"\n Enter radius value :";
        cin>>n;
        circle C2 = circle(n);
        C2.display();

      cout<<"\n ** Constructor with Default
                arguments **";
        C3.display();

        cout<<"\n ***** Array of Objects *****";
        for(int i=0;i<2;i++)
        {
          C4[i].read();
          C4[i].display();
        }

        circle C5 = C1;
        circle C6(C2);

        cout<<"\n ***** Copy Constructor *****";
        C5.display();
```

```
cout<<2 * 3.14 * radius << endl;
        }
        ~circle() { }
//Destructor
};
    //Member function1 defined outside the
class
void circle::read()
{
        cout<<"\n Enter the radius of circle :";
cin>>radius;
}
```

```
        C6.display();
        getch();
        return 0;
}
```

OPERATOR OVERLOADING:

The mechanism of giving such special meanings to an operator is known as *operator overloading*.

We can overload (give additional meaning to) all the C++ operators except the following:

The operators that cannot be overloaded are:

- class member access operators (. , .*)
- scope resolution operator(: :)
- sizeof operator(sizeof)
- conditional operator(?:)

To define an additional task to an operator, we must specify what it means in relation to the class to which operator is applied. This is done with the help of a special function, called *operator function*.

General form of operator function is:.

```
return type  classname :: operator op(argument list)
{
        function body
}
```

where return type is the type of value returned by the specified operation and *op* is the operator being overloaded. The *op* is preceded by the keyword **operator**. **Operator** *op* is the function name.

Operator functions must be either member functions (non-static) or friend functions.

The operator functions are declared in the class using prototypes as follows:

```
vector operator+(vector);          // vector addition using member function
vector operator-( );               // unary minus using member function
friend vector operator+(vector, vector);   // vector addition using friend function
friend vector operator-(vector &a);        // unary minus using frien function
```

The process of operator overloading involves the following steps:

1. Create a class that defines the data type that is to be used in the overloading operation.

2. Declare the operator function operator *op( )* in the public part of the class. It may be either a

member function or a **friend** function

3. Define the operator function to implement the required operations.

OVERLOADING UNARY OPERATOR: (An operator with only one operand is called Unary operator)

Unary operators are operators that work with only one operand. Example of unary operators include unary plus, unary minus operators(+,-), increment, decrement operators (++,--) etc.,

Unary Operator overloading à Using member function : **No argument**
à Using friend function : **One argument**

OVERLOADING BINARY OPERATOR:(An operator with two operands is called Binary operator)

Binary operators are operators that work with two operands. Example of binary operators include arithmetic operators(+,-,*,/,%), arithmetic assignment operators(+=, -=, *=,/=), and comparison operators (<, >, <=, >=,==, !=).

Binary Operator overloading à Using member function : **One argument**
à Using friend function : **Two arguments**

Rules for Overloading operators:

- Only existing operators can be overloaded. New operators cannot be created.
- The overload operator must have at least one operand that is of user-defined type.
- We cannot change the basic meaning of an operator.
- Overloaded operators follow the syntax rules of the original operator.
- Unary Operator overloading à Using member function : No argument
  à Using friend function : One argument
- Binary Operator overloading à Using member function : One argument
  à Using friend function : Two arguments
- When using binary operator overloading, the left hand operand must be an object of the relevant class.

Example Program for Unary operator overloading

```
/*Program for unary operator overloading [++, minus(-)] with member & friend functions. */
#include<iostream.h>
#include<conio.h>
class space
{          int a,b;
public:
           void input(int,int);
           void display();
           void operator++();
           friend void operator-(space &s);
};
void space::input(int x,int y)
{        a=x;
            b=y;
}
void space::display()
{      cout<<"a value is " <<a<<endl;
           cout<<"b value is: "<<b<<endl<<"\n";
}
void space::operator ++()
{          a = ++a;
           b = ++b;
}
void operator-(space &s)
{          s.a = -s.a;
           s.b = -s.b;
}
void main()
{          space S;
           clrscr();
           S.input(10,-20);
           S.display();
           ++S;
           S.display();
           -S;
           S.display();
           getch();
}
```

Example program for binary operator overloading

```
#include <iostream.h>
#include <conio.h>
const s=2;
class matrix
{
        int m[s][s];
public:
        matrix(){ }
        matrix(int x[][s]);
        matrix operator +(matrix B);
        friend matrix operator -(matrix A,
matrix B);
        void display(matrix M);
};
matrix::matrix(int x[][s])
{       for(int i=0;i<s;i++)
                for(int j=0;j<s;j++)
                        m[i][j]=x[i][j];
}
matrix matrix::operator +(matrix B)
{       matrix C;
        for(int i=0;i<s;i++)
                for(int j=0;j<s;j++)
                        C.m[i][j] = m[i][j] +
B.m[i][j];
        return C;
}
matrix operator -(matrix A, matrix B)
{       matrix C;
        for(int i=0;i<s;i++)
            for(int j=0;j<s;j++)
                C.m[i][j] = A.m[i][j] - B.m[i][j];
        return C;
}
```

```
void matrix::display(matrix M)
{       for(int i=0;i<s;i++)
        {
                for(int j=0;j<s;j++)
                {
                   cout<<M.m[i][j]<<"\t";
                }
                cout<<"\n";
        }
}

void main()
{
        int X[][s]={11,12,13,14};
        int Y[][s]={1,2,3,4};
        matrix M1(X);
        matrix M2(Y);
        matrix M3, M4;
        M3 = M1+M2;
        M4 = M1-M2;
        clrscr();
        cout<<"\n\n Matrix A \n";
        M1.display(M1);
        cout<<"\n\n Matrix B \n";
        M2.display(M2);
        cout<<"\n\n Matrix Addition \n";
        M3.display(M3);
        cout<<"\n\n Matrix Subtraction \n";
        M4.display(M4);
        getch();
}
```

TYPE CONVERSIONS:

Every expression has a type that is determined by the components of the expression. Consider the following statement:

```
int x = 5.5 / 2 ;          // x contains 2, the fraction part is lost.
```

Data can be lost when it is converted from a higher data type to a lower data type.

Casts: We can force an expression to be of a specific type by using a type cast operator. The general form:

```
type-name (expression).        //C++ notation
```

where type is a valid data type.

For example, to make sure that the expression x/2 evaluates to type float, write

```
float y = 5.5 / float (2)
```

Since the user-defined data types are designed by us to suit our requirements, the compiler does not support automatic type conversions for such data types.

Three types of situations might arise in the data conversion between incompatible types:

1. Conversion from basic type to class type.
2. Conversion from class type to basic type.
3. Conversion from one class type to another class type.

Type Conversions

Conversion required	Conversion takes place in	
	Source class	Destination class
Basic à Class	Not Applicable	Constructor
Class à Basic	Casting Operator	Not Applicable
Class à Class	Casting Operator	Constructor

Conversion function / Casting operator

```
operator typename( )
{       ……
        …… (Function statements
}
```

Important Note :

1. The constructors used for the type conversions take a single argument whose type is to be converted.
2. The casting operator should satisfy the following conditions : a. It must be a class member
   b. It must not specify a return type
   c. It must not have any arguments

```
/*Example program for Type conversion & String
 manipulation using binary operator overloading
*/
class String
{       char *name;
        int length;
  public:
        String() { }
        String(char *s)
        {       length = strlen(s);
                name = new char[length + 1];
                strcpy( name, s );
        }
        void display(void)
        {       cout << name << "\n";
        }
        String operator +(const String &t);
        operator char*()
        {       return(name);
        }
        ~String();
};
String :: ~String()
        {       delete name;
        }
```

```
String String::operator +(const String &t)
{   String temp;
    temp.length = length + t.length;
    temp.name = new char[temp.length+1];
    strcpy(temp.name, name);
    strcat(temp.name, t.name);
    return (temp.name);
}
void main( )
{
        char* name1="Vinayaga ";
        char* name2="College";
        clrscr();
        String S1(name1);
        String S2 = name2;
        cout<<"\nGiven Strings are : \n";
        S1.display( );
        S2.display( );
        String S3;
        S3 = S1 + S2;
        char* N = S3;
        cout<<"The Joined string is : "<<N<<"\n";
        getch();
}
```