OOP(C++)

UNIT III

" CONSTRUCTORS & OPERATOR OVERLOADING"

KEY POINTS

- Concept of Constructor
- Types of constructors (Default, Parameterized, copy)
- Overloaded Constructors (Multiple Constructor)
- Constructor with default argument,
- Destructors
- Friend Function and Friend Class
- Operator overloading (overloading unary &binary operators)
- Rules for overloading operators.

INTRODUCTION

- C++ provides a special member function called constructor which enable an object to initialize itself when it is created. This is known as automatic initialization of objects.

- It also provides another member function called the destructor that destroy the objects when they are no longer required.

CONSTRUCTOR

- A special member function whose task is to initialize the objects of its class.
- Its name is the same as the class name.
- It is invoked whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.
- There is no need to write any statement to invoke the constructor function.
- Example

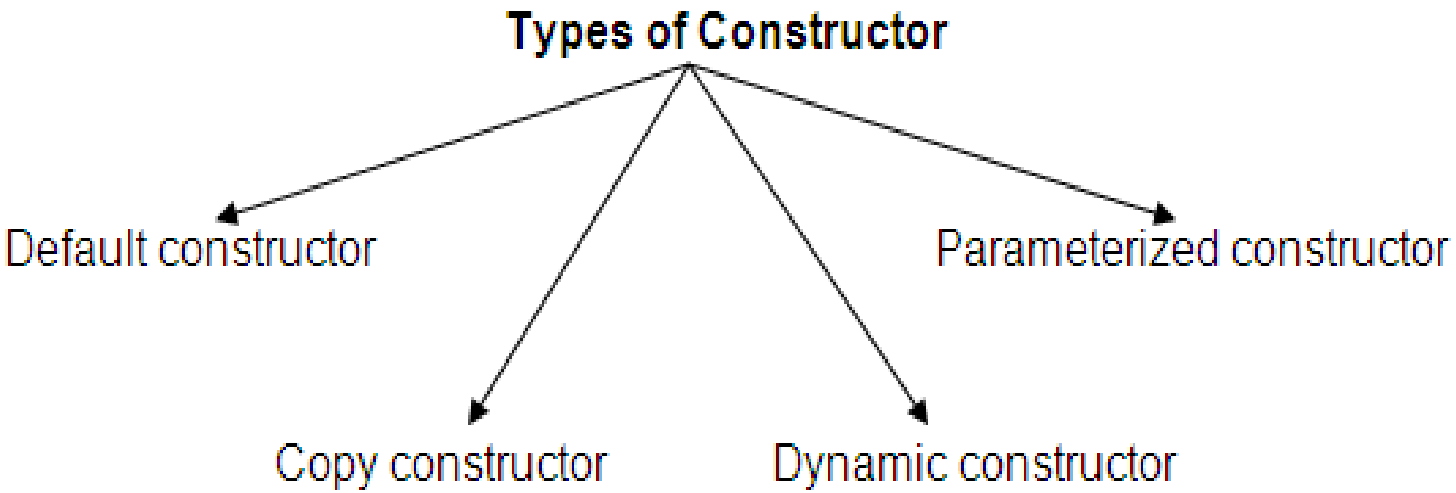
```
class time
{
        int hours, minutes, seconds;
        public:
                time ();   //Constructor Declared
};
```

```
time::time () //Constructor Defined
{
        hours=0;
        minutes=0;
        seconds=0;
}
```

CHARACTERISTICS OF CONSTRUCTOR

- Should be declared in public section
- Invoked automatically when the objects are created
- Do not have return types, **not even void**
- Cannot be inherited, through derived class can call the base class constructor
- Like function in C, they can have default arguments
- They make implicit calls to the operators new & delete when memory allocation is required

TYPES OF CONSTRUCTOR



DEFAULT CONSTRUCTOR

- It accepts no parameters is called default constructor
- For example,

  class is Emp

  Emp :: Emp ()

- If no such a constructor is defined, then the compiler supplies a default constructor

- Therefore a statement such as,

  Emp x;

  invokes the default constructor of the compiler to create the object X.

DEFAULT CONSTRUCTOR

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

class construct {
public:
    int a, b;

    // Default Constructor
    construct()
    {
        a = 10;
        b = 20;
    }
};

int main()
{
    // Default constructor called automatically
    // when the object is created
    construct c;
    cout << "a: " << c.a << endl
         << "b: " << c.b;
    return 1;
}
```

PARAMETERIZED CONSTRUCTOR

- Typically arguments help to initialize an object when it is created.
- To create parameterized constructor, simply add parameter to the constructor function as the same way you do with normal functions.
- Constructor that can take arguments are called parameterized constructor.
- We must pass the initial values as arguments to the constructor function when an objects is declared. This can be done in two ways,

PARAMETERIZED CONSTRUCTOR

i. By calling constructor explicitly:

```
Time T1 = Time (10, 45, 34);        // Explicit call
```

ii. By calling constructor implicitly:

```
Time T1 (10, 45, 34);        // Implicit call
```

EXAMPLE

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

class time
{
        int hrs, min, sec;
        public:
        time (int h, int m, int s)
        {
            hrs=h;
            min=m;
            sec=s;
        }
        void display ()
        {
            cout << "Hrs: "<<hrs<<endl;
            cout << "Min: "<<min<<endl;
            cout << "Sec: "<<sec<<endl;
        }
};
void main ()
{
        clrscr ();
        time t1 (2, 20, 40);
        time t2 (4, 40, 20);
        t1.display ();
        t2.display ();
        getch ();
}
```

EXAMPLE

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

class Point {
private:
   int x, y;
public:
   // Parameterized Constructor
   Point(int x1, int y1)
   {
      x = x1;
      y = y1;
   }
    int getX()
   {
      return x;
   }
   int getY()
   {
      return y;
   }
};
int main()
{
   // Constructor called
   Point p1(10, 15);

   // Access values assigned by constructor
   cout << "p1.x = " << p1.getX() << ", p1.y = "
     << p1.getY();

   return 0;
}
```

COPY CONSTRUCTOR

- A copy constructor is a member function which initializes an object using another object of the same class.

- A copy constructor has the following general function prototype:

  Syntax:

  ```
  ClassName (const ClassName &old_obj);
  ```

- It can be used to declare & initialize an object of another object.

- For ex. The statement **Time T2(T1);** would defined object T2 & at same time initialize it to the values of T1.

Copy Constructor

- Process of initializing through a copy constructor is known as copy initialization.

- But the statement **T2=T1;** will not invoke copy constructor. However if T1 & T2 are objects, this statement is legal & simply assigns the values of T1 & T2, member by member.

  It takes a reference to an object of same class as itself as an

  argument.

COPY CONSTRUCTOR (EXAMPLE)

```
#include<iostream>
using namespace std;
class Samplecopyconstructor
{
    private:
    int x, y;    //data members

    public:
    Samplecopyconstructor(int x1, int y1)
    {
        x = x1;
        y = y1;
    }

    /* Copy constructor */
    Samplecopyconstructor (Samplecopyconstructor &sam)
    {
        x = sam.x;
        y = sam.y;
    }

    void display()
    {
        cout<<x<<" "<<y<<endl;
    }
};
```

```
int main()
{
    Samplecopyconstructor obj1(10, 15);      // Normal constructor
    Samplecopyconstructor obj2 = obj1;       // Copy constructor
    cout<<"Normal constructor : ";
    obj1.display();
    cout<<"Copy constructor : ";
    obj2.display();
    return 0;
}
```

COPY CONSTRUCTOR (EXAMPLE)

- Program for copy constructor

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

class code
{
        int id;
        public:
        code () {}   //Constructor
        code (int a) //Parameter Constructor
        {id=a;}
        code (code &x) //Copy Constructor
        {
                id = x.id; //Copy in the value
        }
        void display ()
        {
        cout << id;
        }
};
```

```
void main ()
{
        code A(100); //Object A is created & initialized
        code B(A);  //Copy Constructor called
        code C = A; //Copy Constructor called again
        code D;    //D is created, not initialized
        D = A;    //Copy constructor not called
        clrscr();
        cout << "\n ID of A: ";
        A.display();
        cout << "\n ID of B: ";
        B.display();
        cout << "\n ID of C: ";
        C.display();
        cout << "\n ID of D: ";
        D.display();
        getch();
}
```

MULTIPLE CONSTRUCTORS/CONSTRUCTOR OVERLOADING

- Also called as constructor overloading.
- We have used no argument constructors, one argument constructor & even parameterized constructor.
- C++ permits to use all these constructors in the same class
- When more than one constructor function is defined in a class, then constructors are overloaded same as the function overloading.
- For example,

```
class integer
{
        int m, n;
        public:
        integer ()        // Constructor 1
        { m=0; n=0;}
        integer (int a, int b)   //Constructor 2
        {m=a; n=b;}
        integer (integer &i)  //Constructor 3
        {m=i.m; n=i.n;}
};
```

MULTIPLE CONSTRUCTORS/CONSTRUCTOR OVERLOADING

- Program for copy constructor

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

class code
{
        int id;
        public:
        code () {}   //Constructor
        code (int a) //Parameter Constructor
        {id=a;}
        code (code &x) //Copy Constructor
        {
                id = x.id; //Copy in the value
        }
        void display ()
        {
        cout << id;
        }
};
```

```
void main ()
{
        code A(100); //Object A is created & initialized
        code B(A);  //Copy Constructor called
        code C = A; //Copy Constructor called again
        code D;    //D is created, not initialized
        D = A;    //Copy constructor not called
        clrscr();
        cout << "\n ID of A: ";
        A.display();
        cout << "\n ID of B: ";
        B.display();
        cout << "\n ID of C: ";
        C.display();
        cout << "\n ID of D: ";
        D.display();
        getch();
}
```

MULTIPLE CONSTRUCTORS/CONSTRUCTOR OVERLOADING

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

class num
{
   private:
   int a;
   float b;
   char c;
   public:
   num(int m, float j , char k);
   num (int m, float j);
   num();
   void show()
   {
   cout<<"\n\ta="<<a<<"b="<<b<<"c="<<c;
   }
};
num:: num (int m, float j , char k)
{
   cout<<"\n Constructor with three arguments";
   a=m;
   b=j;
   c=k;
}
num:: num (int m, float j)
{
   cout<<"\n Constructor with two arguments";
   a=m;
   b=j;
   c=0;
}
num:: num()
{
   cout<<"\n Constructor without arguments";
   a=b=c=NULL;
}
main()
{
   clrscr();
   class num x(4,5.5,'A');
   x.show();
   class num y(1,2.2);
   y.show();
   class num z;
   z.show();
   return 0;
}
```

OUTPUT

Constructor with three arguments
a= 4 b= 5.5 c= A
Constructor with two arguments
a= 1 b= 2.2 c=
Constructor without arguments
a= 0 b= 0 c=

MULTIPLE CONSTRUCTORS IN A CLASS

- Program to use overloaded constructors

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

class complex
{
    float x,y;
    public:
    complex() // No arg. constructor
    {}
    complex(float a) //One arg. cosntructor
    {x=y=a;}
    complex(float real, float imag) //Two arg. Constr.
    { x=real; y=imag;}
    friend complex sum (complex, complex);
    friend void show (complex);
};
complex sum (complex c1, complex c2)
{
    complex c3;
    c3.x=c1.x+c2.x;
    c3.y=c1.y+c2.y;
    return(c3);
}
void show(complex c)
{
     cout<<c.x<<" + j"<<c.y<<"\n";
}
void main()
{
    complex A(2.7, 3.5);
    complex B(1.6);
    complex C;
    clrscr();
    C=sum (A,B);
    cout<<"A= "; show(A);
    cout<<"B= "; show(B);
    cout<<"C= "; show(C);
    //Another vay to give initial value.
    complex P,Q,R;
    P=complex(2.5,3.9);
    Q=complex(1.6,2.5);
    R=sum(P,Q);
    cout<<"\n";
    cout<<"P= "; show(P);
    cout<<"Q= "; show(Q);
    cout<<"R= "; show(R);
    getch();
}
```

CONSTRUCTORS WITH DEFAULT ARG.

- Possible to define constructors with default argument.
- For example, constructor **complex()** can be declared as follows,

  complex(float real, float imag=0);
- Default value of arg. imag is 0,then statement complex C(2.0,3.0); Assign 2.0 to real & 3.0 to imag.
- Actual parameter, when specified, overrides default value.
- 'Default constructor' A :: A() is totally different than 'Constructor with default argument' A :: A(int=0).
- Default argument constructor can be called with either one or no arguments. When called with no argument, it becomes a default constructor.

CONSTRUCTORS WITH DEFAULT ARG.

```
#include<iostream.h>
#include<conio.h>
#include<math.h>
class power
{
   private:
   int num;
   int power;
   int ans;
   public:
   power(int n=9,int p=3);
   // declaration of constructor
   with default arguments
   void show()
   {

   cout<<"\n"<<num
   <<"raise to"<<power <<"is" <<ans;

   }
};
power:: power (int n,int p )
{
   num=n;
   power=p;
   ans=pow(n,p);
}
```

```
main()
{
   clrscr();
   class power p1,p2(5);
   p1.show();
   p2.show();
   return 0;
}
```

OUTPUT

9 raise to 3 is 729
5 raise to 3 is 125

DESTRUCTOR

- Used to destroy the objects that have been created by a constructor.
- Its also a special member function whose name is same as class name but is preceded by a tilde.For Example, ~integer () {}
- **Neither takes any argument not return any** value.
- It will be **invoked by compiler upon exit** from program to **clean up storage** that is no longer accessible.
- It is not possible to have more than one destructor.

Syntax of Destructor

```
~class_name()
{
   //Some code
}
```

DESTRUCTOR (EXAMPLE)

```
#include <iostream>
using namespace std;
class HelloWorld{
public:
  //Constructor
  HelloWorld(){
    cout<<"Constructor is called"<<endl;
  }
  //Destructor
  ~HelloWorld(){
    cout<<"Destructor is called"<<endl;
   }
   //Member function
   void display(){
     cout<<"Hello World!"<<endl;
   }
};
int main(){
   //Object created
   HelloWorld obj;
   //Member function called
   obj.display();
   return 0;
}
```

Destructor (Example)

```
Constructor is called
Hello World!
Destructor is called
```

DESTRUCTOR (EXAMPLE)

- Program for destructor

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

int count=0;
class alpha
{
    public:
    alpha ()
    {
        count++;
        cout<<"\n No. of Object Created: "<<count;
    }
    ~alpha ()
    {
        cout<<"\n No. of Object Destroyed: "<<count;
        count--;
    }
};
```

```
void main ()
{
    clrscr ();
    cout<<"\n\nEnter Main...\n";
    alpha a1, a2, a3, a4;
    {
        cout<<"\n\nEnter Block1\n";
        alpha a5;
    }
    {
        cout<<"\n\nEnter Block2\n";
        alpha a6;
    }
    cout<<"\n\nRe-Enter Main\n";
}
```

Friend Function and friend Classes

- One of the important concepts of OOP is data hiding, i.e., a nonmember function cannot access an object's private or protected data.
- In some cases it causes overhead.
- There is mechanism built in C++ programming to access private or protected data from non-member functions.
- Done using Friend function/friend class.
- However friendship is not mutual.

Friend Function

- If a function is defined as a friend function then, the private and protected data of a class can be accessed using the function.
- It is achieved by using the keyword Friend.
- For accessing the data, the declaration of a friend function should be made inside the body of the class.
- It can be declared anywhere inside class either in private or public section.
- Starting with keyword friend.

Friend Function

Declaration of friend function in C++

```
class class_name
{
      ... .. ...
      friend return_type function_name(argument/s);
      ... .. ...
}
```

Friend Function

- Define the friend function as a normal function to access the data of the class.
- No friend keyword is used in the definition.

```
return_type functionName(argument/s)
{
     ... .. ...
     // Private and protected data of className can be accessed from
     // this function because it is a friend function of className.
     ... .. ...
}
```

Friend Function Example

```
#include <iostream>
using namespace std;
class XYZ {
private:
   int num=100;
   char ch='Z';
public:
   friend void disp(XYZ obj);
};
//Global Function
void disp(XYZ obj){
   cout<<obj.num<<endl;
   cout<<obj.ch<<endl;
}
int main() {
   XYZ obj;
   disp(obj);
   return 0;
}
```

Function Overloading

- Function overloading allows you to use the same name for different functions, to perform, either same or different functions in the same class.

- Function overloading is usually used to enhance the readability of the program. If you have to perform one single operation but with different number or types of arguments, then you can simply overload the function.

Function overloading: changing number of Arguments.

```
int sum (int x, int y) // first definition
{ cout << x+y; }
int sum(int x, int y, int z) // second overloaded defintion
 { cout << x+y+z; }
int main()
{
sum (10, 20); // sum() with 2 parameter will be called
sum(10, 20, 30); //sum() with 3 parameter will be called
}
```

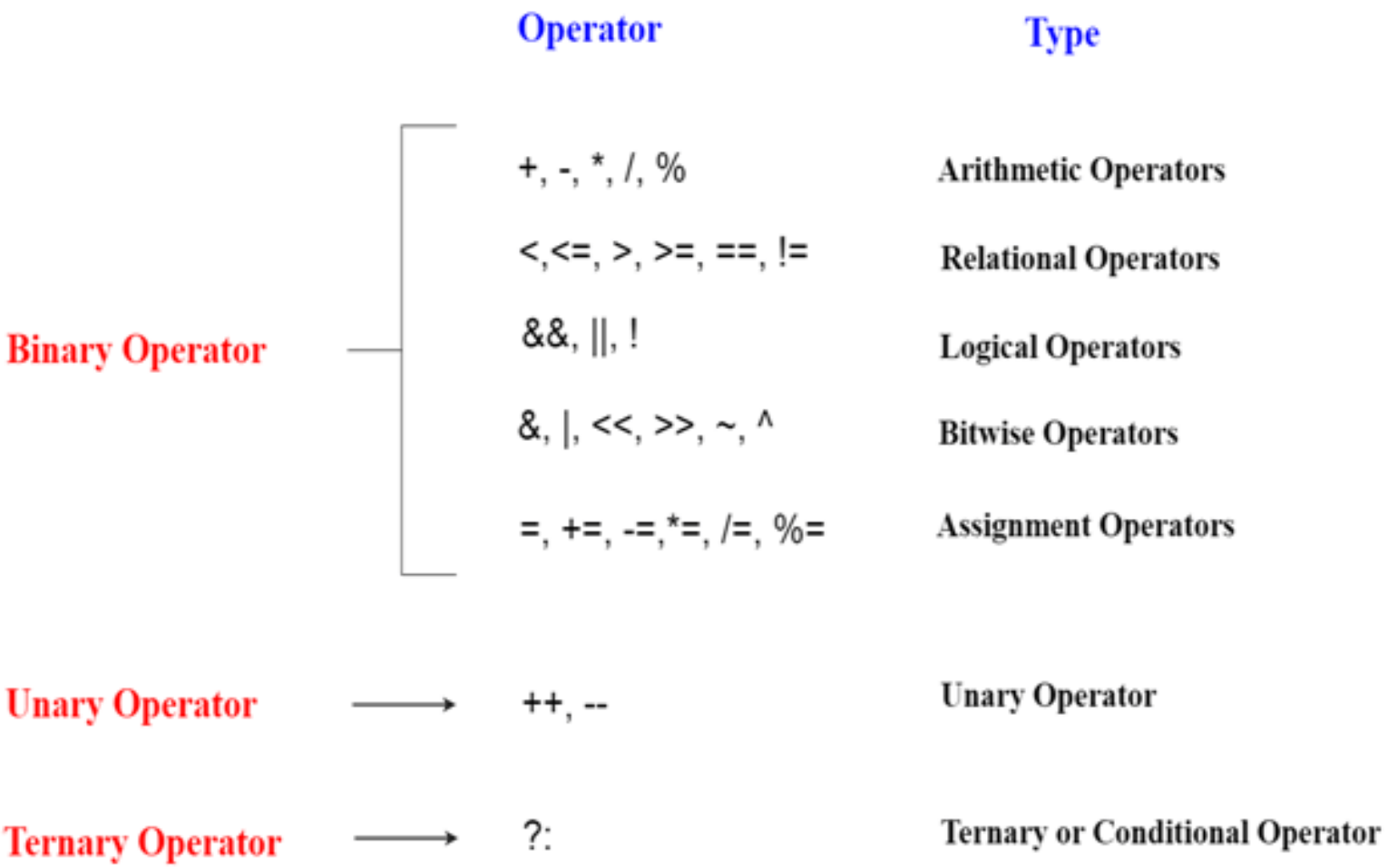
Function Overloading: Different Datatype of Arguments

```
int sum(int x, int y) // first overloaded definition
{
   cout<< x+y;
}
double sum(double x, double y) // second overloaded definition
{
   cout << x+y;
}
int main()
{
   sum (10,20);
   sum(10.5,20.5);
}
```

OPERATOR OVERLOADING

- Mechanism of giving special meaning to an operator for the data type.
- It provides flexible option for creation of new definitions for most of the C++ operators.
- All operator can be overloaded except ., .*, ::, ?:, sizeof etc

Operators


Operator
Type
Binary Operator
+, -, *, /, %
Arithmetic Operators
<,<=, >, >=, ==, !=
Relational Operators
&&, ||, !
Logical Operators
&, |, <<, >>, ~, ^
Bitwise Operators
=, +=, -=,*=, /=, %=
Assignment Operators
Unary Operator
++, --
Unary Operator
Ternary Operator
?:
Ternary or Conditional Operator

DEFINING OPERATOR OVERLOADING

- To define an additional task to an operator, specify its meaning to class to which operator is applied. The special function called as operator function describes the task.
- General form of an operator function is,

```
returntype classname :: operator op(argument List)
{  Function body // Task Defined  }
```

- Operator function must be either member function or friend function.
- A basic difference between them is that friend function will have only one arg for unary operator & two for binary operators, while member function has no arguments for unary operators & only one for binary operators.

DEFINING OPERATOR OVERLOADING

- **Steps** of overloading process are,
  1. Create a class that defines data type that is to be used in overloading operation.
  2. Declare operator fn operator op () in public part of class. It may be member fn or friend fn.
  3. Define operator function to implement operations.
- Overloaded fn can be invoked by **expression** such as,
  1. For unary operator : op x OR y op
  2. For Binary operator : x op y
  3. For friend fn op x or op y would be interpreted as, operator op (x)
  4. For member fn x op y would be interpreted as, operator op (Y)

RULES FOR OVERLOADING OPERATORS

- Only existing operator can be overloaded not new one.
- It must have at least one operand that is of user defined type.
- We can not change basic meaning of an operator.
- Follows syntax rules of original operators.
- Some operators can not be overloaded (., .*, ::, ?:)
- Cannot use friend functions to overload certain operators(=, (), [], ->).however, member fn can be used to overload them
- Unary operators overloaded through a member function take one explicit arg. But this overloaded by means of a friend function take one reference argument.
- Binary operators overloaded through member fn take one arg & those which are overloaded through a friend fn take 2 explicit argument.

OVERLOADING UNARY OPERATORS

- When an operator used as a unary, take just one operand.
- Let us consider the unary minus operator. A minus operator, when used as a unary, takes just one operand.
- We know that this operator changes sign of an operator when applied to basic data item.
- A unary minus when applied to an object should change sign of each of its data items.
- Function operator –() takes no arguments.
- It can change sign of data members of object S. Since this function is member fn of same class, it can directly access members of object which activated it.

OVERLOADING UNARY OPERATORS

- Program to use unary minus operator.

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

class space
{
        int x1,y1,z1;
        public:
        void getdata(int a, int b, int c);
        void display(void);
        void operator -();
};
void space :: getdata(int a, int b, int c)
{
        x1=a;
        y1=b;
        z1=c;
}
void space :: display (void)
{
        cout<<x1<<" ";
        cout<<y1<<" ";
        cout<<z1<<"\n";
}
void space :: operator -()
{
        x1=-x1; y1=-y1; z1=-z1;
}
void main()
{
        space S;
        clrscr();
        S.getdata(10, -20, 30);
        cout<<"S : ";
        S.display();
        -S;
        cout<< "S : ";
        S.display();
        getch();
}
```

OVERLOADING UNARY OPERATORS

```
// unary_operator_overloading.cpp
#include <iostream>
using namespace std;

class check_count
 {
   public:
     int count_plus;
     int count_minus;

     check_count()
     {
       count_plus = 0;
       count_minus = 2;
     };
     void operator ++() { ++count_plus; } // count increment
     void operator --() { --count_minus; } // count increment
 };
```

OVERLOADING UNARY OPERATORS

```
int main()
 {
   check_count x, y; //creating objects

   //before increment/decrement
   cout << "x =" << x.count_plus<<"\n";
   cout <<"y =" << y.count_minus<<"\n";

   ++x;
   --y;

   //after increment/decrement
   cout<<"\nAfter increment/decrement\n";
   cout<<"x ="<<x.count_plus<<"\n";
   cout<<"y ="<<y.count_minus<<"\n";
 return 0;
}
```

OVERLOADING BINARY OPERATORS

- Operators which require two operands to perform operations.
- Same mechanism of overloading unary operators can be used overload binary operators.
- Program to use overloading binary operators.

OVERLOADING BINARY OPERATORS

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

class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0)  {real = r;   imag = i;}

    // This is automatically called when '+' is used with
    // between two Complex objects
    Complex operator + (Complex const &obj) {
         Complex res;
         res.real = real + obj.real;
         res.imag = imag + obj.imag;
         return res;
    }
    void print() { cout << real << " + i" << imag << endl; }
};

int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
}
```

OVERLOADING BINARY OPERATORS

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

class complex
{
        float x, y;
        public:
        complex() { }
        complex (float real, float imag)
        { x=real; y=imag;}
        complex operator +(complex);
        void display(void);
};
complex complex :: operator +(complex c)
{
        complex temp;
        temp.x=x+c.x;
        temp.y=y+c.y;
        return(temp);
}
void complex :: display(void)
{
        cout<<x<<" + j"<<y<<"\n";
}
void main()
{
        complex C1, C2, C3;
        C1=complex (2.5, 3.5);
        C2=complex (1.6, 2.7);
        clrscr();
        C3=C1+C2; //Invokes operator +()
        cout<<"C1= "; C1.display();
        cout<<"C2= "; C2.display();
        cout<<"C3= "; C3.display();
        getch();
}
```

OVERLOADING BINARY OPERATORS

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

class complex
{
 private:
  int real,imag;
 public:
  void getvalue()
  {
   cout<<"Enter the value of real number:";
   cin>>real;
   cout<<"Enter the value of imaginary number:";
   cin>>imag;
  }
  complex operator+(complex obj)
  {
   complex temp;
   temp.real=real+obj.real;
   temp.imag=imag+obj.imag;
   return(temp);
  }
```

OVERLOADING BINARY OPERATORS

```
  complex operator-(complex obj)
  {
   complex temp;
   temp.real=real-obj.real;
   temp.imag=imag-obj.imag;
   return(temp);
  }
  void display()
  {
   cout<<real<<"+"<<"("<<imag<<")"<<"i"<<"\n";
  }
};

int main()
{
  complex c1,c2,c3,c4;

  c1.getvalue();
  c2.getvalue();
```

OVERLOADING BINARY OPERATORS

```
   c3 = c1+c2;
   c4 = c1-c2;



   cout<<"Result is:\n";
   c3.display();
   c4.display();

  return 0;
}
```