## Constructors

→ A constructor is a special member function whose task is to initialize the objects of its class.

→ It is special because its name is the same as the class name. [but no return type]

→ The constructor 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.

constructor declaration & definition:

```
class integer
{   int m,n;
  public:
    integer(void); // constructor declared;
     ::::
};
integer :: integer( ) // constructor defination
{ m=0; n=0;
}
```

→ when a class contains a constructor, it is guaranteed that an object created by the class will be initialized automatically.

Ex: integer int1;

→ not only creates the object int1 of type integer but also initializes its data members 'm' and 'n' to zero

→ There is no need to write any statement to invoke the constructor function.

→ A constructor that accepts no parameter is called the default constructor.

→ ~~The default constructor for class A is A().~~

The constructor functions have some special characteristics

* They should be declared in the public section
* They are invoked automatically when object are created.
* They do not have return types not even void and therefore, and they cannot return values.
* They cannot be inherited, though a derived class can call the base class constructor.
* Like other C++ functions, they can have default arguments
* constructors cannot be Virtual.
* We cannot refer to their addresses.
* An object with a constructor (or destructor) cannot be used as a member of a union.
* They make "implicit calls" to the operators 'new' and 'delete' when memory allocation is required.

Note: When constructor is declared for a class, initialization of class object becomes mandatory.

# Parameterized Constructors

→ To initialize the various data elements of different objects with different values when they are created

→ C++ permits us to achieve this objective by passing arguments to the constructor function when the objects are created.

→ The constructors that can take arguments are called **parameterized constructors.**

→ The constructor integer() may be modified to take arguments as shown below:

```
class integer
{
    int m, n;
    public:
        integer (int x, int y); //parameterized constructor
        ....
        ....
};
integer :: integer (int x, int y)
{  m = x;  n = y;
}
```

→ where the constructor has been parameterized, the object declaration statement such as.

~~integer int1(20,100); // implicit call~~

~~integer int1 = integer(20,100); // explicit call~~

There are two ways of declaration of object for parameterized constructors:

* By calling the constructor explicitly

  Ex: integer int1 = integer(0, 100);

* By calling the constructor implicitly

  Ex: integer int1(0, 100);

⇒ both data members m & n are assigned with 0 & 100 respectively

Note: when constructor is parameterized, we must provide appropriate arguments for the constructor.

Ex: characteristics of parameterized constructors:

1) constructor functions can also be defined as inline functions.

```
class integer
{
    int m, n;
  public:
        integer ( int x, int y ) // inline constructor
        { m=x; y=n;
        }
};
```

**

2) parameterized constructor can be of any type except that of the class to which it belongs.

```
Ex: class A
    { ...
      ...
    public:
          A(A); //error.
    };
```

3) we have to pass the ~~objects~~ constructor argument with reference.

```
Eg: class A
    {  ....
       ....
     public:
          A(A&); // valid
    };
```

Example of parameterized constructor

A class called Point that stores x and y coordinates of a point. The class uses parameterized constructor for initializing the class objects.

```
#include<iostream>
using namespace std;
class point
{   int x,y;
    public:
        point(int a, int b) // inline parameterized constructor
        {  x=a;
           y=b;
        }
        void display()
        {  cout<<"x="<<x<<"y="<<y<<endl;
        }
};
int main()
{   point P1(1,3); // invokes parameterized constructor
    point P2(5,10);
    cout<<"point P1 P2="<<endl;
     P1.display();
     P2.display();
    return 0;
}
```

## Multiple constructors in a class

Ex: 1) integer ( ); // No arguments
2) integer ( int, int); // Two arguments

→ from ① the constructor itself supplies that data values and no values are passed by the calling program

→ from ② the function call passes the appropriate values from main().

→ C++ permits us to use both these constructors in the same class.

Example

```
class integer
{
    int m, n;
    public:
                                    constructor 1
     integer ( )          // (default constructor)
     { m=0;
       n=0;
     }
                                        Constructor 2
     integer (int a, int b) //(parameterized constructor)
     {
         m=a;
         n=b;
     }
     integer(integer & i)  // constructor 3 with object
     {                        as parameter.
       m= i.m;
       n = i.n;
     }
};
```

→ The constructor 1 receives no arguments. called default constructor

→ The constructor 2 receives two arguments called parameterised constructor.

→ The constructor 3 receives object as an argument

Example; In declaration of object

1) integer I1;
   * automatically invoke the first constructor and set both 'm' and 'n' of I1 to zero.

2) integer I2(20,40);
   * call the second constructor and initialize m = 20 and n = 40

3) integer I3(I2);
   * Invokes 3rd constructor which copies the values of I2 into I3 [called copy constructor]

---

overloaded constructors

Example:

```
#include<iostream>
using namespace std;
class complex
{   float x, y;
}
```

→ The constructor 1 receives no arguments. called default constructor

→ The constructor 2 receives two arguments called parameterised constructor.

→ The constructor 3 receives object as an argument

Example; In declaration of object

1) integer I1;
   * automatically invoke the first constructor and set both 'm' and 'n' of I1 to zero.

2) integer I2 (20, 40);
   * call the second constructor and initialize m = 20 and n = 40

3) integer I3 (I2);
   * Invokes 3rd constructor which copies the values of I2 into I3 [called copy constructor]

---

Overloaded constructors

Example:

```
#include<iostream>
using namespace std;
class complex
{   float x, y;
}
```

## Overloaded Constructors

Example:

```
#include<iostream>
using namespace std;
class complex
{
    float x, y;
  public:
    complex(){ }
    complex(float a)
    { x=y=a;
    }
    complex(float real, float imag)
    { 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 <<endl;
}
```

```
int main()
{
   complex A (2.7, 3.5);
   complex B (1.6);
   complex C;

    C = Sum(A, B);
    cout << "A =" << endl;
    show(A);
    cout << " B=" << endl;
    show(B);
    cout << " C =" << endl;
    show(C);

// complex P, Q, R;
   P= complex(2.5, 3.9);
   Q = complex(1.6, 2.5);
    R = Sum(P, Q);

    show(P);
    show(Q);
    show(R);
    return 0;
}
```

not their in Syllabus

## * Dynamic initialization of object

→ class objects can be initialized dynamically

→ ~~Initial value of an object~~.

→ Initialization of values of an object is at runtime.

→ We can provide various initialization formats using overloaded constructors.

Example:

* Long term deposit schemes working in the commercial banks
* Bank provide different rates for different schemes as well as for different periods of investment

```
#include <iostream>
using namespace std;
class Fixed_deposit
{
    long int P; // principal amount
    int year; // period of investment
    float rate; // interest rate
    float R; // return value

  public:
    Fixed_deposit() { }
    Fixed_deposit(long int P, int Y, float r=0.12);
    Fixed_deposit(long int P, int y, int r);
    void display();
};
```

```
Fixed_deposit :: Fixed_deposit (long int P, int y, float r)
{
    P = P;
    year = y;
    rate = r;
    R = P;
  for (int i = 1 ; i <= y; i++)
  {
    R = R * (1.0 + r);
  }
}
Fixed_deposit :: Fixed_deposit (long int P, int y, int r)
{ P = P;
    year = y;
    rate = r;
    R = P;
  for (int i = 1 ; i <= y ; i++)
  { R = R * (1 + float(r)/100);
  }
}
void Fixed_deposit :: display ( )
{ cout << "principal amount = " << P << "Return value=" << R << endl;
}
int main ( )
{ Fixed_deposit FD1, FD2, FD3;
  long int P;
  int y;
  float r;
  int R;
```

```
cout<< "Enter the amount, period, intrest rate "<<endl;
cin >> P>> Y >>R;
FD1 = Fixed_deposit (P, Y, R);
cout<<"Enter the amount, period, intrest rate "<<endl;
Cin >>P>> Y>>R;
FD2 = Fixed_deposit ( P, Y, R);
cout<< "Enter amount and period"<<endl;
Cin>> P >>Y;
FD3 = Fixed_deposit (P, Y);
cout <<"display FD1"<<endl;
FD1. display();
cout <<"display FD2"<<endl;
FD2. display();
cout<<"display FD3"<<endl;
FD3. display();
return 0;
}
```

## Copy constructor

Ex:

```
class integer
{
    int rollno;
    public:
    integer (integer &c)
    {
        rollno = c.rollno;
    }
};
```

we use copy constructor as

```
integer (integer &i);
```

→ Copy constructor is used to declare and initialize an object from other object.

Example

integer I2(I1);

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

→ copy constructor takes a reference to an object of the same class as itself as an argument.

Example: (for copy constructor)

```
#include<iostream>
using namespace std;
class code
{  int id;
  public:
  code()          // default constructor
  { id=0;
  }
  code(int a) // parameterized constructor
  { id=a;
  }
  code(code &c) // copy constructor.
  { id=c.id;
  }
};
int main()
{   code A(100);
    code B(A);   // copy constructor
    B.display();   return 0;
}
```

## Dynamic Constructors

→ Constructors are used to allocate memory while creating objects.

→ This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in the saving of memory.

→ Allocation of memory to objects at the time of their construction is called dynamic constructor.

→ **new** operator is used to allocate memory for object.

→ use **delete** operator to deallocate memory to save memory.

Example

```
#include<iostream>
#include<string.h>
using namespace std;
class integer
{
    int m;
    int n;
  public:
    integer( )
    { m=0;
      n=0;
    }
```

```
integer (int a, int b)
 {   m=a;
     n=b;
  }
void display()
 {
   cout <<"m= "<< m <<"n= "<< n <<endl;
 }
void product()
 {  cout <<" proud of m & n = "<<(m*n)<<endl;
 };
};
int main()
{ integer *b;
     b= new integer(100, 200);
    b-> product();
    b-> display();
 return 0;
 }
```

Example

write a program to construct two dimensional array using constructors.

```
#include<iostream>
using namespace std;
class matrix
{
   int **p; // pointer to matrix
   int d1,d2; // dimensions (rows, columns)
  public:
   matrix(int x, int y);
   void get_element(int i, int j, int value)
    { p[i][j] = value;
    }
  int display(int i, int j)
  { return p[i][j];
   }
};
matrix :: matrix(int x, int y)
{  d1 = x;
   d2 = y;
   p= new int *[d1];  // creat an array pointer
   for(int i=0; i<d1; i++)
    p[i] = new int[d2]; // create space for each row
   }
```

```
int main()
{
    int m, n;
   cout<< "Enter size of matrix"<<endl;
   cin>>m>>n;
    matrix A(m,n) // matrix object A constructed
    cout<<"enter matrix elements"<<endl;
   int i, j, value;
   for(i=0; i<m; i++)
      for(j=0; j<m; j++)
       {  cin >> value;
          A.get_elements (i, j, value);
       }
   cout<<endl;
   cout<< A.display(m, n)<<endl;
   return 0;
}
```

**Destructors:**

→ used to destroy the objects that have been created by a constructor.

→ destructor is a member function whose name is the same as the class name but is preceded by a tilde.

→ It is defined as;

~integer();

where integer is class name

→ destructor never takes any argument nor does it return any value.

→ It will be invoked implicity by compiler upon exit from the program to clean up storage that is no longer accessible.

→ It releases memory space for future use

→ when "new" operator used to allocate memo in the constructors, we should use delete to free that memory.

Example:

```
matrix::~matrix()
{ for (int i=0; i<d1;;i++)
      delete P[i];
      delete P;
}
```

where matrix is class name

~~This is required because when the pointer~~

Example:

```
#include<iostream>
using namespace std;
int count=0
class test
{
  public:
```

```
class test
{ public:
    test()
    {
     count++;
     cout<< " constructor msg : object number "<<count<<endl;
    }
   ~test()
    { cout<< " Destructor msg "<< count<<endl<<"destroyed"<<
                                                    endl;
     count--;
    }
};
int main()
{  cout<<"create first object"<<endl;
   test T1;
   { //Block 1
    cout<<"Inside Block1 "<<endl;
    cout<<"create two more objects"<<endl;
    test T2, T3;
    cout<<"Leaving Block 1"<<endl;
   }
   cout<<" Inside main Block"<<endl;
return 0;
}
```

## Defining operator overloading

→ To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied.

→ This is done with the help of a special function called **operator function**.

General form of an operator function is:

```
return type classname :: operator op (argument list)
{
    Function body // task defined
}
```

where:

return type → is the type of value returned by specific-operation.

op → is the operator being overloaded.

operator → keyword.

operator op → function name.

→ Operator function must be either **member functions** or **friend functions**.

→ difference blw them is that a* friend function will have one argument for unary operators and two for binary operators.

* a member function has no arguments for unary operators and only one for binary operators.

* Because of this the object used to invoke the member function is passed implicitly and therefore is available for the member function.
* This is not the case with friend functions. Arguments may be passed either by value (or) by reference.

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

* Vector operator + (vector) // vector addition
* Vector operator - (); // unary minus
* friend vector operator + (vector, vector) // Vector addition
* friend vector operator - (vector); // unary minus
* Vector operator - (vector &a); // substraction
* int operator == (vector); // comparision
* friend int operator == (vector, vector) // comparision

vector is a data type & class and may represent both magnitude and direction (or) a series of points called elements.

The process of 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 (or) a friend function

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

⇒ over loaded operator functions can be invoked by expression such as friend function:

op x (or) x op // for unary operators.

x op y // for binary operators

* op x (or) x op can be interpreted as

operator op(x)

for member function: for member function:

⇒ expression x op y would be interpreted as either

- x.operator op(y) // member function

(or)

operator op (x, y) // for friend function

**overloading** unary operators:

⇒ Consider minus (−) operator when used as a unary operator, takes just one operand.

⇒ we know that this operator changes the sign of an operand when applied to a basic data item

⇒ The unary minus (−) when applied to an object should change the sign of each of its data item

Example:

```
#include <iostream>
using namespace std;
class space
{
    int x;
    int y;
    int z;
  public:
    void getdata (int a, int b, int c)
    {   x = a;
        y = b;
        z = c;
    }
    void display()
    { cout<<"x="<<x<<"y="<<y<<"z="<<z<<endl;
    }
    void operator-()
    {   x=-x;
        y=-y;
        z=-z;
    }
int main()
{   space s;
    s.getdata (10,-20,30);
    s.display();
    cout<<"s:"<<endl;
    -s;
    s.display();
    return 0;
}
```

Note:

for example S2 = -S1;

* This will not work, because the function operator-() does not have return value. It will work if the function is modified to return an object.

* It is possible to overload a unary minus operator using a friend function as follows:

```
friend void operator-(space &s); // declaration
void operator-(space &S)  // defination
{
    s.x = -s.x;
    s.y = -s.y;
    s.z = -s.z;
}
```

## Overloading Binary operators

→ It is same mechanism as like overloading the unary operator.

Example: How to add two complex numbers using friend function.

C = sum(A, B) // functional notation

where C, A & B are of objects. any data type

* we can replace this expression function as by a natural expression

C = A+B; // arithmetic notation.

by overloading the '+' operator using an operator+() function.

Program : Addition of two complex numbers

```
#include<iostream>
using namespace std;
class complex
{
    float x;
    float y;
public:
    complex() // default constructor
    { }
    complex(float real; float imag)
    { x=real;
      y=imag;
    }
    complex operator+ (complex);
    void display()
    {
      cout << x << "+j" << y << endl;
    }
};
complex complex :: operator+ (complex c)
{ complex temp;
  temp.x = x+c.x;
  temp.y = y+c.y;
  return (temp);
}
int main()
{ complex A, B, C;
  C=A+B;
```

P.T.O

```
int main()
{
    complex A(2.5, 3.5);
    complex B(1.6, 2.7);
    complex C;

     C = A + B;
    cout << "C1 = " << endl;
     A.display();
     B.display();
     C.display();
}
```

Note:

* Above operator+() function having following features

1. It receive only one complex type argument explicitly.
2. It returns a complex type value.
3. It is a member function of complex.

```
C = A + B;
```

→ where operator+() function expected to add two complex values and return a complex value as the result but receives only one value as argument.

* the member function can be invoked only by an object of the same class.
* other argument value will taken implicitly by a object.

→ where A (is of type complex) is object, takes the responsibility of invoking the function and 'B' plays the role of an argument that is passed to the function.

The above invocation statement is equivalent to

```
c = A.operator+(B); // usual function call syntax
```

→ In function operator+(), the data members of 'A' are accessed directly and the data member of 'B' is passed as an argument are accessed using dot operator.

where: temp.x = x + c.x;

temp.x → refers to the temprary object
c.x → refers to the object B
x → refeves to the object A

Rule: In overloading of binary operator, the left-hand operand is used to invoke the operator function. and the right-hand operand is passed as an argument.

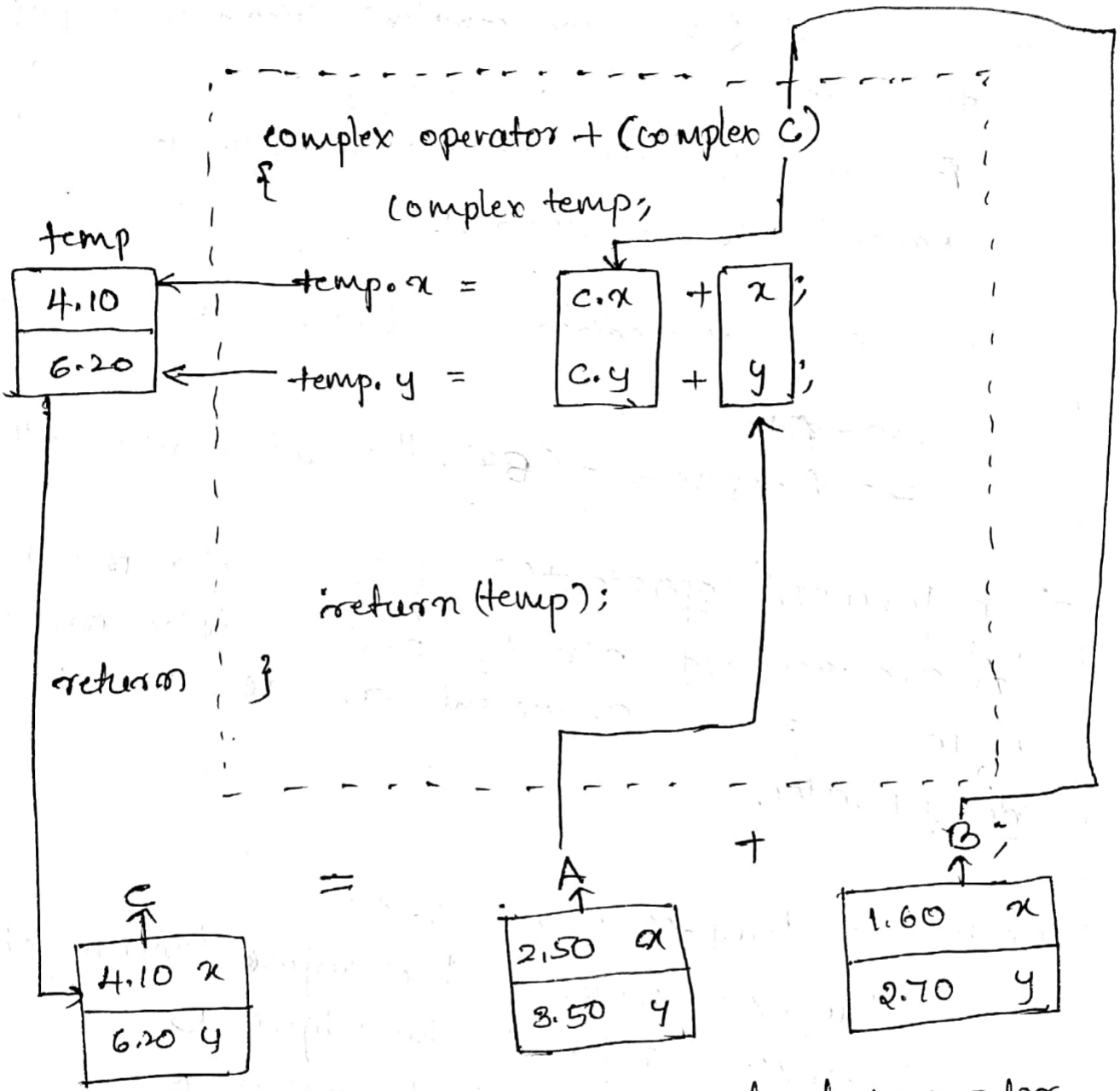


fig: Implementation of the overload + operator.

## Overloading Binary operators using friend function

→ friend function used in place of member functions for overloading binary operator.

→ difference being that a friend function requires two arguments to be explicitly passed to it, while member function requires only one.

~~The complex number program discussed in previous section~~ can be modified using friend

Example: Addition of two complex numbers

```
#include<iostream>
using namespace std;
class complex
{   float x;
    float y;
  public:
    complex ()
    { x=y=0;
    }
    complex (float real, float imag)
    { x=real;
      y=imag;
    }
   void display()
   { cout << x <<"+j" << y <<endl;
   }
   friend complex operator+ ( complex ;complex);
                    // operator friend function  operator
};
complex operator+ (complex c, complex d) // friend function
{   complex temp;                          operator
    temp.x= c.x+ d.x;
    temp.y= c.y+ d.y.
   return (temp);
}
```

```
int main()
{
    complex A(2.5, 3.5), B(1.6, 2.7), C;
    C = A+B; // function call;
    A.display();
    B.display();
    C.display();
}
```

→ In above program the friend function is declared as

friend complex operator+ (complex, complex);

→ taking two arguments for Binary operator

Explicitly:

→ Both the values are directly passed from main function.

definition is:

```
complex operator+ (complex c, complex d)
{   complex temp;
    temp.x = c.x + d.x;
    temp.y = c.y + d.y;
  return temp;
}
```

and function call is for operator friend function is

```
C = A+B;
```

(or)

```
C = operator+ (A,B);
```

~~Addition~~

Binary operators with Scalar:

Scalar addition  Scalar Multiplication

Ex: B = A + 2; (or) B = A * 2

(or) (or) B = 2 * A

B = 2 + A;

where '2' is a scalar quantity (ADD) (MUL) +/* with 'A' and store the result in 'B'. Here we are overloading '+' or '*' operator using friend function.

Example:

```
#include<iostream>
const int size = 3;
class vector
{   int v[size];
public:
    vector();                    // default constructor
    vector(int *x); // parameterized constructor
    friend vector operator * (int a, vector b); // friend1
    friend vector operator * (vector b, int a); // friend2
    friend istream & operator >> (istream &, vector &);
    friend istream & operator << (ostream &, vector &);
};
vector :: vector()
{   for(int i=0; i<size; i++)
        v[i] = 0;
}
vector :: vector(int *x)
{   for(int i=0; i<size; i++)
```

```
Vector operator*(int a, Vector b)
{
    Vector c;
    for(int i=0; i<size; i++)
        c.v[i] = a * b.v[i];

    return c;
}
Vector operator*(Vector b, int a)
{
    Vector c;
    for(int i=0; i<size; i++)
        c.v[i] = b.v[i] * a;
    return c;
}
Istream & operator>>(istream &din, Vector &b)
{
    for(int i=0; i<size; i++)
        din >> b.v[i];
    return(din);
}
Ostream & operator<<(ostream &dout, Vector &b)
{
    dout<<"Vector is:";
    for(int i=1; i<size; i++)
        dout<< b.v[i];
    return (dout);
}
```

```
int main()
{
    int x[size] = {2, 4, 6};
    Vector m;  // invokes default constructor
    Vector n(x);  (or) Vector n = x; // invokes parameterized constructor
    cout << "Enter vector elements" << endl;
    cin >> m;  // invokes operator>>()
    cout << "m=" << m << "\n";  // invokes operator<<()

    Vector p, q;
    p = 2 * m;  // invokes friend function 1
    q = n * 2;  // invokes friend function 2

    cout << "p=" << p << endl;  // invokes operator<<()
    cout << "q=" << q << endl;

    return 0;
}
```

From above program we can have operator function * in other way as follows

p = 2 * m; (or) p = operator*(2, m);

q = n * 2; (or) q = operator*(n, 2);

where vector() is a constructor to initialize zero to all elements

* vector(int *x); is the parameterized constructor to assign value to vector elements.

## Manipulation OF STRINGS using operators :

→ ~~There are many operators for manipulating the strings.~~

→ In C++ permits us to create our own definitions of operators that can be used to manipulate the strings very much similar to decimal numbers

→ In C++ committee has added a new class called **string** to the C++ class library that supports all kind of string manipulations.

→ Strings can be defined as class objects which can be then manipulated like the built-in types.

→ we use **new** operator to allocate memory for each string and pointer variable to point to the string array

→ we must create a object of string, that can hold length and - location which are necessary for string manipulation.

typical class of String is :

```
class string
{
    char *p;        // pointer to string
    int len;        // length of string
  public:
    ....            // member function to initialize and
    ....            // manipulate string.
};
```

Example to manipulate the strings and to overload operators + and <=

```
#include <iostream>
#include <string.h>
using namespace std;
class string
{
    char *p;
    int len;
  public:
     string()      // create null string
     { length = 0;
       p = 0;
     }
    string (const char *s);
    string (const string &s); // copy constructor
    ~string ()
    { delete P;    // destructor
    }
  friend string operator+(const string &s, const string &t);
  friend int operator<= (const string &s, const string &t);
  friend void show (const string s);
};
string :: string (const char *s)
 {  len = strlen(s);
    p= new char [len+1];
    strcpy (P, S);
 }
```

```
String :: String (const String &s)
{
    len = s.len;
    p= new char [len+1];
    Strcpy (p, s.p);
}
String operator + (const string &s, const string &t)
{
    String temp;
    temp.len = s.len + t.len;
    temp.p = new char [temp.len +1];
    Strcpy ( temp.p, s.p);
    Strcat (temp.p, t.p);
    return (temp);
}
int operator<= (const String &s, const String &t)
{
    int m = strlen (s.p);
    int n = strlen (t.p);
    if( m<=n)
     return 1;
   else
     return 0;
}
void show (const string s)
{
    cout << s.p;
}
```

```
int main()
{
    string S1 = "New";
    string S2 = "York";
    string S3 = "Delhi";
    string string1, string2, string3;
    String1 = S1;
    string2 = S2;
    String3 = S1+S3;
    cout << "String1 =" << endl;
    show (string1);
    cout << "string2) =" << endl;
    show (string2);
    cout << "String 3 =" << endl;
    show(string3);
    if (string1 <= string3)
    { show (string1);
      cout << " smaller than";
      show (string3);
      cout << endl;
    }
  else
    { show (string3);
      cout << " smaller than";
      show (string1);
      cout << endl;
    }
  return 0;
}
```