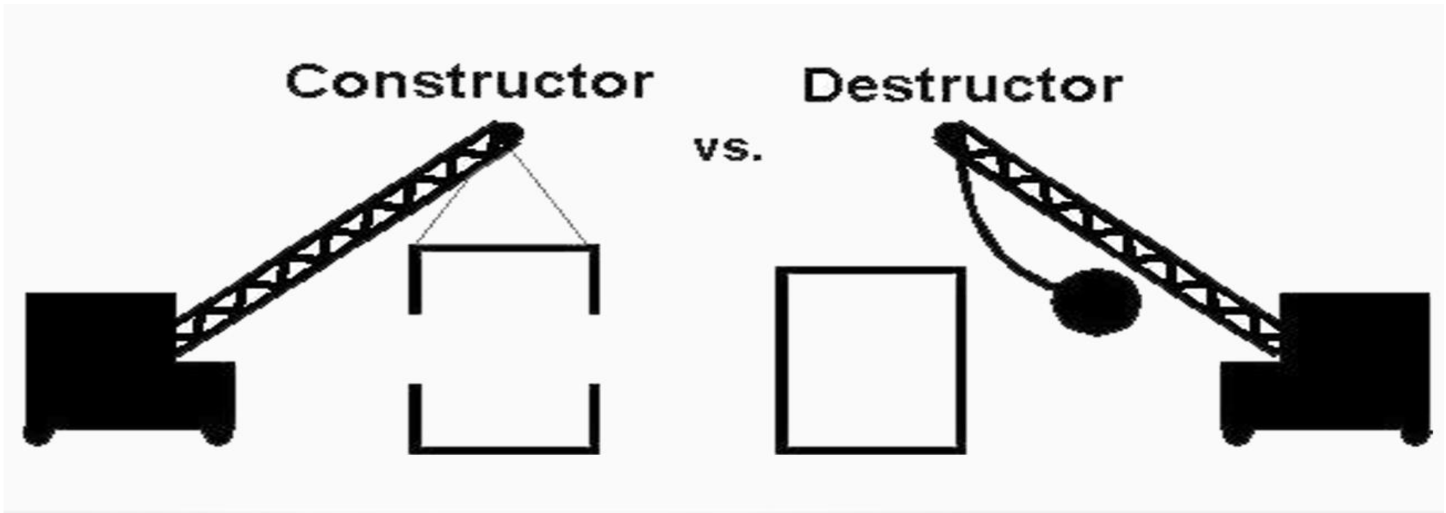
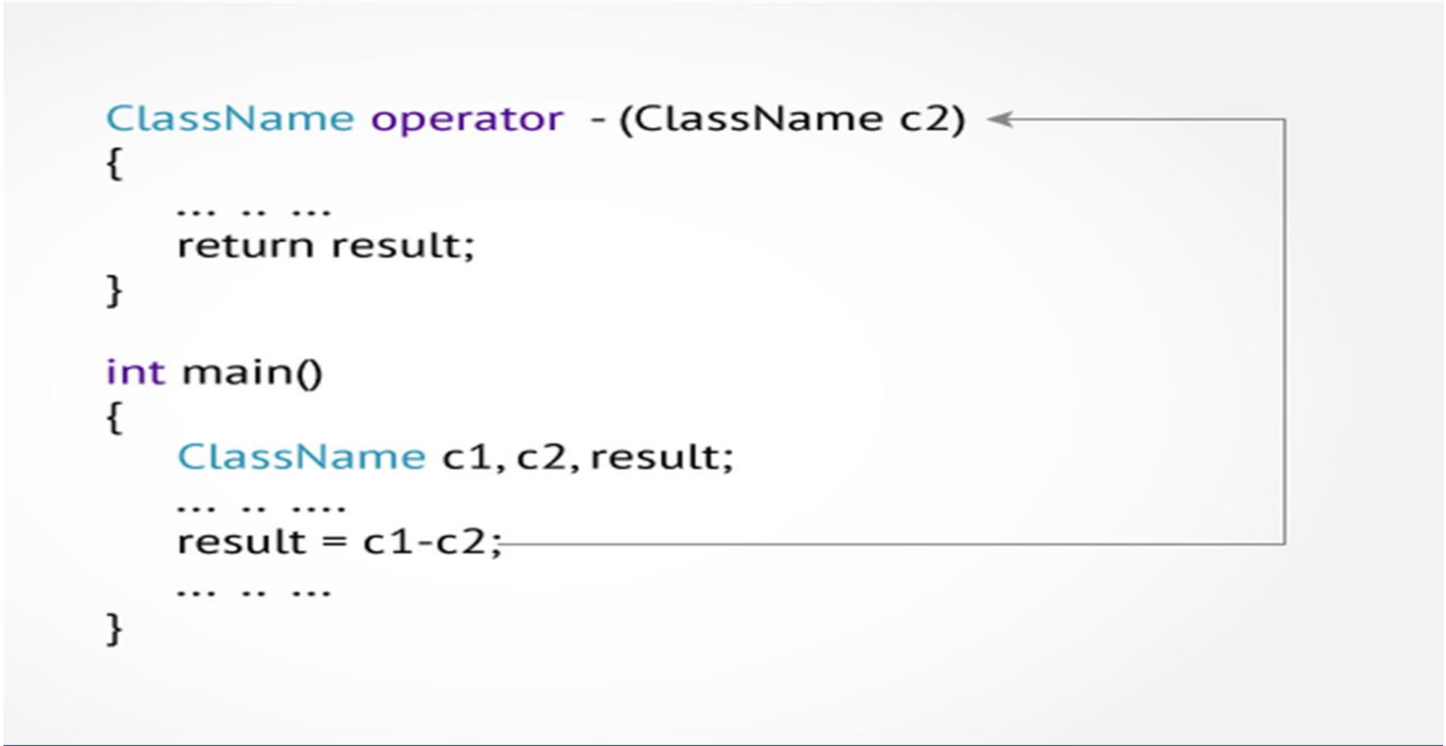



```
ClassName operator  - (ClassName c2)
{
    ... .. ...
    return result;
}

int main()
{
    ClassName c1, c2, result;
    ... .. ....
    result = c1-c2;
    ... .. ...
}
```

MODULE -3

Constructors, Destructors and Operator Overloading

GANESH Y
Dept. of ECE RNSIT

MODULE -3
Constructors, Destructors and Operator overloading

SYLLABUS

Constructors, Multiple constructors in a class, Copy constructor, Dynamic constructor, Destructors, Defining operator overloading, Overloading Unary and binary operators, Manipulation of strings using operators (Selected topics from Chap-6, 7 of Text)

Introduction

We have seen, so far, a few examples of classes being implemented. In all the cases, we have used member functions such as **putdata()** and **setvalue()** to provide initial values to the private member variables. For example, the following statement

```
A.input();
```

invokes the member function **input(),** which assigns the initial values to the data items of object **A.** Similarly, the statement

```
x.getdata(100,299.95);
```

passes the initial values as argument to the function getdata(), where these values are assigned to the private variables of object x.

All these 'function call' statements are used with the appropriate objects that have already been created. These functions cannot be used to initialize the member variables at the time *of* creation of their objects.

Providing the initial values as described above does not conform with the philosophy of C++ language. We stated earlier that one of the aims of C++ is to create user-defined data types such as **class**, that behave very similar to the built-in types.,

This means that we should be able to initialize a class type variable (object) when it is declared, much the same way as initialization of an ordinary variable. For example,

```
int m = 20;
float x = 5.75;
```

are valid initialization statements for basic data types.

Similarly, when a variable of built-in type goes out of scope, the compiler automatically destroys the variable. But it has not happened with the objects we have so far studied. It is therefore clear that some more features of classes need to be explored that would enable us to initialize the objects when they are created and destroy them when their presence is no longer necessary.

C++ provides a special member function called the ***constructor*** which enables 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 ***destroys the objects*** when they are no longer required.

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. 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.

A constructor is declared and defined as follows:

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

When a class contains a constructor like the one defined above, it is guaranteed that an object created by the class will be initialized automatically. For example, the declaration

```
integer int1; // object int1 created
```

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 (as we do with the normal member functions).

If a 'normal' member function is defined for zero initialization, we would need to invoke this function for each of the objects separately. This would be very inconvenient, if there are a large number of objects.

A constructor that accepts no parameters is called the ***default constructor***. The default constructor for **class A is A :: A().** If no such constructor is defined, then the compiler supplies a default constructor.

Therefore a statement such as

A a ;

invokes the default constructor of the compiler to create the object **a.**

The constructor functions have some special characteristics. These are:

• They should be declared in the public section.

• They are invoked automatically when the objects 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.** (Meaning of virtual will be discussed later)

• 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.

Remember, when a constructor is declared for a class, initialization of the class objects becomes mandatory.

PARAMETERIZED CONSTRUCTORS

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

The constructor **integer(),** defined above, initializes the data members of all the objects to zero. However, in practice it may be necessary 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 constructor **integer()** may be modified to take arguments as shown below:

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

When a constructor has been parameterized, the object declaration statement such as

```
	integer int1;
```

may not work. We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways:

• By calling the constructor explicitly.

• By calling the constructor implicitly.

The following declaration illustrates the first method:

```
integer int1 = integer (0,100); // explicit call
```

This statement creates an integer object int1 and passes the values 0 and 100 to it. The second is implemented as follows:

```
integer int1(0,100); // implicit call
```

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

Remember, when the constructor is parameterized, we must provide appropriate arguments for the constructor. Program below demonstrates the passing of arguments to the constructor functions.

The constructor functions can also be defined as **inline** functions. Example:

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

The parameters of a constructor can be of any type except that of the class to which it belongs: For example,

```
class A
{
      ......
      .........
public :
      A (A) ;
};
```

is illegal.

However, a constructor can accept a *reference* to its own class as a parameter. Thus, the statement

```
                    class A
                    {
                         ……
                         ………
                    public :
                         A (A&) ;
                    };
```

is valid. In such cases, the constructor is called the ***copy constructor.***

```
//This program defines a class called Point that stores the x and y
//coordinates of a point.The class uses parameterized constructor for
//initializing the class objects
          #include <iostream>
          class Point
          {
               int x , y;
          public :
               Point (int a, int b) //inline parameterized constructor
          definition
               {
                    x=a;
                    y=b;
               }
               void display ()
               {
               Cout<<"("<<x<<","<<y<<")\n";
               }
          int main ()
          {
               point p1(1, 1); //invokes parameterized constructor
               point p2(5, 10);
               cout<<"Point pl =";
               p1.display();
               cout<<"Point p2 =";
               p2.display();
               return 0;
          }
```

The output of above Program would be:

Point p1= (1,1)
Point p2 = (5,10)

```
          #include <iostream>
          using namespace std;
          class str
          {
               int a_count,e_count,i_count,o_count,u_count;
               char user_str[100];
          public:
               str (char gg[]);
               void count_vowels();
```

```
};
str::str (char gg[])
{
	a_count=0;e_count=0;i_count=0;o_count=0;u_count=0;
	for (int i=0;i<100;i++)
	{
		user_str[i]=gg[i];
		if (user_str[i]=='\0')
			break;
	}
}
void str::count_vowels()
{
	int i=0;
	do {
		switch (user_str[i])
		{
			case 'A':
			case 'a': a_count++;
					break;
			case 'E':
			case 'e': e_count++;
					break;
			case 'I':
			case 'i': i_count++;
					break;
			case 'O':
			case 'o': o_count++;
					break;
			case 'U':
			case 'u': u_count++;
					break;
		}
		i++;
	} while(user_str[i]!='\0');
	cout<<"\n a or A Count "<<a_count;
	cout<<"\n e or E Count "<<e_count;
	cout<<"\n i or I Count "<<i_count;
	cout<<"\n o or O Count "<<o_count;
	cout<<"\n u or U Count "<<u_count;
}
int main ()
{
	char s[]={'G','a','n','e','s','h'};
	str obj1(s);
	obj1.count_vowels();
	str obj2("Ganesh Yernally");
	obj2.count_vowels();
	return 0;
}
```

Multiple Constructors in a Class / Overloaded Constructors

So far we have used two kinds of constructors. They are:

```
integer();// No arguments
integer (int, int);// Two arguments
```

In the first case, the constructor itself supplies the data values and no values are passed by the calling program. In the second case, the function call passes the appropriate values from **main().** C++ permits us to use both these constructors in the same class. For example,

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

This declares three constructors for an **integer** object. The first constructor receives no arguments, the second receives two **integer** arguments and the third receives one integer object as an argument. For example, the declaration

```
integer I1;
```

would automatically invoke the first constructor and set both **m** and **n** of **I1** to zero. The statement

```
integer I2(20,40);
```

would call the second constructor which will initialize the data members **m** and **n** of **I2** to 20 and 40 respectively. Finally, the statement

```
integer I3(I2);
```

would invoke the third constructor which copies the values of **I2** into **I3.** In other words, it sets the value of every data element of **I3** to the value of the corresponding data element of **I2.**

```cpp
#include <iostream>
using namespace std;
class complex
{
float x , y ;
public:
      complex() { } // constructor no arg
      complex (float a) {x=y=a;} // constructor-one arg
      complex (float real, float imag) {x=real; y=imag;}// constructor-two arg
      friend complex sum(complex, complex);
      friend void show( complex);
};
complex sum(complex c1, complex c2) //friend
{
      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";
}
int main ()
{
      complex A(2.7, 3.5); // define & initialize
      complex B(1.6);       // define & initialize
      complex C;            // define
      C = sum(A, B);                // sum() is a friend
      cout << "A = ";show (A); // show() is also friend
      cout << "B = ";show (B);
      cout << "C = ";show (C);
// Another way to give initial values
      complex P,Q,R;        // define
      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);
      return 0;
}
```

Output:

```
A = 2.7+j3.5        P = 2.5+j3.9
B = 1.6+j1.6        Q = 1.6+j2.5
C = 4.3+j5.1        R = 4.1+j6.4
```

Let us look at the first constructor again.

```
complex() { }
```

It contains the empty body and does not do anything. We just stated that this is used to create objects without any initial values. Remember, we have defined objects in the earlier examples without using such a constructor.

Why do we need this constructor now? As pointed out earlier, C++ compiler has an ***implicit constructor*** which creates objects, even though it was not defined in the class.

This works fine as long as we do not use any other constructors in the class. However, once we define a constructor, we must also define the "do-nothing" implicit constructor. This constructor will not do anything and is defined just to satisfy the compiler.

CONSTRUCTORS WITH DEFAULT ARGUMENTS

It is possible to define constructors with default arguments. For example, the constructor complex() can be declared as follows:

```
complex (float real, float imag=0);
```

The default value of the argument **imag** is zero. Then, the statement

```
complex C(5.0);
```

assigns the value 5.0 to the **real** variable and 0.0 to **imag** (by default). However, the statement

```
complex C(2.0, 3.0);
```

assigns 2.0 to **real** and 3.0 to **imag.** The actual parameter, when specified, overrides the default value.

As pointed out earlier, the missing arguments must be the trailing ones. It is important to distinguish between the default constructor **A :: A()** and the default argument constructor **A :: A(int = 0).**

The default argument constructor can be called with either one argument or no arguments. When called with no arguments, it becomes a default constructor. When both these forms are used in a class, it causes ambiguity for a statement such as

A a;

The ambiguity is whether to 'call' **A :: A() or A :: A(int = 0).**

DYNAMIC INITIALIZATION OF OBJECTS

Class objects can be initialized dynamically too. That is to say, the initial value of an object may be provided during run time.

One advantage of dynamic initialization is that we can provide various initialization formats, using overloaded constructors. This provides the flexibility of using different format of data at run time depending upon the situation.

Consider the long-term deposit schemes working in the commercial banks. The banks provide different interest rates for different schemes as well as for different periods of investment.

Program shown below illustrates how to use the class variables for holding account details and how to construct these variables at run time using dynamic initialization.

```
// Long-term fixed deposit system
#include <iostream>
using namespace std;
class Fixed_deposit
{
        long int P_amount;      // Principal amount
        int     Years;          // Period of investment
        float   Rate;           // Interest rate
        float    R_value;       // Return value of amount
    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(void);
};
Fixed_deposit :: Fixed_deposit(long int p, int y, float r)
{
        P_amount = p;
        Years = y;
        Rate = r;
        R_value = P_amount;
        for(int i = 1; i <= y; i++)
            R_value = R_value * (1.0 + r);
}
```

```
Fixed_deposit :: Fixed_deposit(long int p, int y, int r)
{
    P_amount = p;
    Years = y;
    Rate = r;
    R_value = P_amount;
    for(int i=1; i<=y; i++)
      R_value = R_value*(1.0+float(r)/100);
}
void Fixed_deposit :: display(void)
{
    cout << "\n"
         << "Principal Amount = " << P_amount << "\n"
         << "Return Value     = " << R_value  << "\n";
}
int main()
{
     Fixed_deposit FD1, FD2, FD3;            // deposits created
     long int p;                             // principal amount
int     y;                                   // investment period, years
float   r;                                   // interest rate, decimal form
int     R;                                   // interest rate, percent form
cout << "Enter amount, period,interest rate(in percent)"<<"\n";
cin >> p >> y >> R;
FD1 = Fixed_deposit(p,y,R);
cout << "Enter amount, period, interest rate(decimal form)" << "\n";
cin >> p >> y >> r;
FD2 = Fixed_deposit(p,y,r);
cout << "Enter amount and period" << "\n";
cin >> p >> y;
FD3 = Fixed_deposit(p,y);
cout << "\nDeposit 1";
FD1.display();
cout << "\nDeposit 2";
FD2.display();
cout << "\nDeposit 3";
FD3.display();
return 0;
}
```

The output of Program would be:

```
Enter amount,period,interest rate(in percent)
10000 3 18
Enter amount,period,interest (in decimal form)
10000 3 0.18
Enter amount and period
10000 3
Deposit 1
Principal Amount = 10000
Return Value = 16430.3
Deposit 2
Principal Amount = 10000
Return Value = 16430.3
Deposit 3
Principal 'Amount = 10000
Return Value = 14049.3
```

COPY CONSTRUCTOR

We have used the copy constructor

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

as one of the overloaded constructors.

As stated earlier, a copy constructor is used to declare.and initia.lize an object from another object.

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 I2 = I1;
```

The process of initializing through a copy constructor is known as ***copy initialization.*** Remember, the statement

```
I2 = I1;
```

will not invoke the copy constructor. However, if **I1** and **I2** are objects, this statement is legal and simply assigns the values of **I1** to **I2,** member-by-member. This is the task of the overloaded assignment operator(=).

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

{
    int id;
    public:
    code(){ }                  // constructor
    code(int a) { id = a;}     // constructor again
    code(code & x)             // copy constructor
    {
      id = x.id; // copy in the value
    }
    void display(void)
    {
      cout << id;
    }
};
int main()
{
    code A(100);               // object A is created and 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
    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();
    return 0;
}
```

The output of above Program would be:

```
id of A: 100
id of B: 100
id of C: 100
id of D: 100
```

When no copy constructor is defined, the compiler supplies its own copy constructor.

DYNAMIC CONSTRUCTORS

The constructors can also be 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 known as dynamic construction of objects. The memory is allocated with the help of the new operator.

Program shown below the use of new, in constructors that are used to construct strings in objects.

```
#include <iostream>
#include <string>
using namespace std;
class String
{
     char *name;
     int length;
     public:
      String()                        // constructor-1
      {
        length = 0;
        name = new char[length + 1];
      }
     String(char *s)  // constructor-2
      {
        length = strlen(s);
         name = new char[length + 1];  // one additional
                                       // character for \0
        strcpy(name, s);
      }
     void display(void)
     {cout << name << "\n";}
     void join(String &a, String &b);
};
void String :: join(String &a, String &b)
{
     length = a.length + b.length;
     delete name;
      name = new char[length+1];      // dynamic allocation
     strcpy(name, a.name);
     strcat(name, b.name);
};
int main()
{
     char *first = "Joseph ";
     String name1(first), name2("Louis "), name3 ("Lagrange"),
     s1,s2;
     s1.join(name1, name2);
     s2.join(s1, name3);
     name1.display();
     name2.display();
     name3.display();
     s1.display();
     s2.display();
     return 0;
}
```

Output

```
Joseph
Louis
Lagrange
Joseph Louis
Joseph Louis Lagrange
```

This Program uses two constructors. The first is an empty constructor that allows us to declare an array of strings. The second constructor initializes the ***length*** *of the string, allocates necessary space for the string to be stored and creates the string itself. Note that one additional character space is allocated to hold the end-of-string character* '\0'.

The member function **join( )** concatenates two strings. It estimates the combined length of the strings to be joined, allocates memory for the combined string and then creates the same using the string functions **strcpy( )** and **strcat( )**.

Note that in the function **join( )**, **length** and **name** are members of the object that calls the function, while **a.length** and **a.name** are members of the argument object **a.**

The **main( )** function program concatenates three strings into one string. The output is as shown below:

```
Joseph Louis Lagrange
```

CONSTRUCTING TWO-DIMENSIONAL ARRAYS

The following illustrates how to construct a matrix of size m X n.

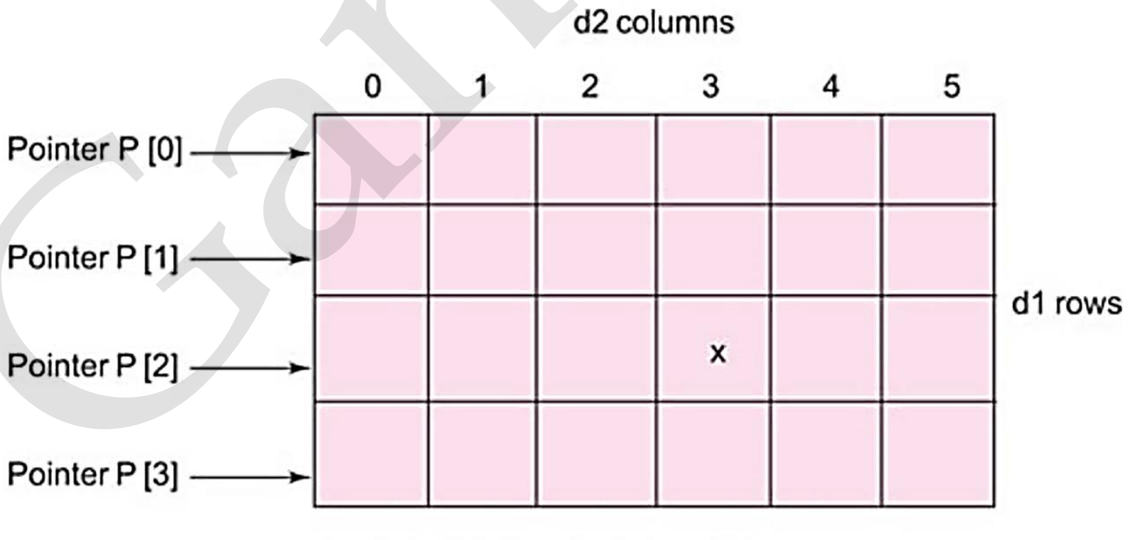
```
#include <iostream>
using namespace std;
class matrix
{
    int **p;                        // pointer to matrix
    int d1,d2;                      // dimensions
    public:
    matrix(int x, int y);
    void get_element(int i, int j, int value)
    {p[i][j]=value;}
    int & put_element(int i, int j)
    {return p[i][j];}
};
matrix :: matrix(int x, int y)
{

   d1 = x;
   d2 = y;
   p = new int *[d1];            // creates an array pointer
   for(int i = 0; i < d1; i++)
   p[i] = new int[d2];           // creates space for each row
}
```

```
int main()
{
     int m, n;
     cout << "Enter size of matrix: ";
     cin  >> m >> n;
     matrix A(m,n);                    // matrix object A constructed
     cout << "Enter matrix elements row by row \n";
     int i, j, value;
     for(i = 0; i < m; i++)
        for(j = 0; j < n; j++)
        {
          cin >> value;
          A.get_element(i,j,value);
        }
     cout << "\n";
     cout << A.put_element(1,2);
     return 0;
};
```

Output

```
Enter size of matrix: 3 4
Enter matrix elements row by row
11 12 13 14
15 16 17 18
19 20 21 22
17
17 is the value of the element (1, 2).
```

The constructor first creates a vector pointer to an **int** of size **d1 .** Then, it allocates, iteratively an **int** type vector of size **d2** pointed at by each element **p[i].**



x represents the element P[2] [3]

Thus, space for the elements of a d1 x d2 matrix is allocated from free store as shown above.

const OBJECTS

We may create and use constant objects using **const** keyword before object declaration. For example, we may create X as a constant object of the class **matrix** as follows:

```
const matrix X(m, n) ; // object X is constant
```

Any attempt to modify the values of **m** and **n** will generate compile-time error. Further, a constant object can call only **const** member functions.

As we know, **a const** member is a function prototype or function definition where the keyword const appears after the function's signature. Whenever **const** objects try to invoke **nonconst** member functions, the compiler generates errors.

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 integer can be defined as shown below:

```
~integer () { }
```

A destructor never takes any argument nor does it return any value. It will be invoked implicitly by the compiler upon exit from the program (or block or function as the case may be) 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. For example, the destructor for the **matrix** class discussed above may be defined as follows:

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

This is required because when the pointers to objects go out of scope, a destructor is not called implicitly.

The example below illustrates that the destructor has been invoked implicitly by the compiler.

```
#include<iostream>
using namespace std;
int count=0;
class test
{
    public:
      test()
      {
      count++;
      cout<<"\n\nConstructor Msg: Object number "<<count<<
      "created..";
      }
      ~test()
      {
      cout<<"\n\nDestructor Msg: Object number "<<count<<"
      destroyed..";
      count--;
      }
};
int main()
{
    cout<<"Inside the main block..";
    cout<<"\n\nCreating first object T1..";
    test T1;
    {   //Block 1
       cout<<"\n\nInside Block 1..";
       cout<<"\n\nCreating two more objects T2 and T3..";
       test T2,T3;
       cout<<"\n\nLeaving Block 1..";
    }
    cout<<"\n\nBack inside the main block..";
    return 0;
}
```

```
Inside the main block ..
Creating first object T1 ..
Constructor Msg: Object number 1 created ..

Inside Block 1 ..
Creating two more objects T2 and T3 . .
Constructor Msg: Object number 2 created ..
Constructor Msg: Object number 3 created ..
```

```
Leaving Block 1 ..

Destructor Msg: Object number 3 destroyed .
Destructor Msg : Object number 2 destroyed.
Back inside the main block . .
Destructor Msg: Object number 1 destroyed . .
```

A class constructor is called every time an object is created. Similarly, as the program control leaves the current block the objects in the block start getting destroyed and destructors are called for each one of them.

Note that the objects are destroyed in the reverse order of their creation. Finally, when the main block is exited, destructors are called corresponding to the remaining objects present inside main.

Similar functionality can be attained by **using static data members with constructors and destructors.** We can declare a static integer variable count inside a class to keep a track of the number of its object instantiations.

Being static, the variable will be initialized only once, i.e., when the first object instance is created. During all subsequent object creations, the constructor will increment the count variable by one. Similarly, the destructor will decrement the count variable by one as and when an object gets destroyed.

To realize this scenario, the code in following program will change slightly, as shown below:

```
#include <iost ream>
using namespace std;
class test
{
private:
     static int count=0;
public :
}
test ()
{
     count++;
}
~test ()
{
     count--;
}
```

The primary use of destructors is in freeing up the memory reserved by the object before it gets destroyed. Program shown below demonstrates *how a destructor releases the memory allocated to an object:*

```
#include<iostream>
#include<conio.h>
using namespace std;
class test
{
     int *a;
     public:
     test(int size)
     {
       a = new int[size];
       cout<<"\n\nConstructor Msg: Integer array of size "<<size<<"
       created..";
     }
     ~test()
     {
       delete a;
       cout<<"\n\nDestructor Msg: Freed up the memory allocated for
       integer array";
     }
};
int main()
{
     int s;
     cout<<"Enter the size of the array..";
     cin>>s;
     cout<<"\n\nCreating an object of test class..";
     test T(s);
     cout<<"\n\nPress any key to end the program..";
     getch();
     return 0;
}
```

Output

```
Enter the size of the array .. 5
Creating an object of test class. :
Constructor Msg: Integer array of sizes created ..
Press any key to end the program ..
Destructor Msg: Freed up the memory allocated for integer array
```

Operator Overloading

C++ tries to make the user-defined data types behave in much the same way as the built-in types. For instance, C++ permits us to add two variables of user-defined types with the same syntax that is applied to the basic types.

This means that C++ has the ability to provide the operators with a special meaning for a data type. The mechanism of giving such special meanings to an operator is known as ***operator overloading***.

Operator overloading provides a flexible option for the creation of new definitions for most of the C++ operators. We can overload (***give additional meaning to***) all the C++ operators <u>except the following:</u>

- Class member access operators (. , .*)
- Scope resolution operator (::)
- Size operator **(sizeof)**
- Conditional operator (?:)

The reason why we cannot overload these operators may be attributed to the fact that these operators take names (example class name) as their operand instead of values, as is the case with other normal operators.

Although the ***semantics*** of an operator can be extended, we cannot change its ***syntax***, the grammatical rules that govern its use such as the number of operands, precedence and associativity. For example, the multiplication operator will enjoy higher precedence than the addition operator.

Remember, when an operator is overloaded, its original meaning is not lost. For instance, the operator +, which has been overloaded to add two vectors, can still be used to add two integers.

DEFINING OPERATOR OVERLOADING

The general form of an operator function is:

```
return_type classname :: operator op (arglist)
{
     Function body // task defined
}
```

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

This is done with the help of a special function, called *operator function,* which describes the task.

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

Operator functions must be either member functions (**non-static**) or friend functions. A basic difference between them is that a friend function will have only one argument for unary operators and two for binary operators, while a member function has no arguments for unary operators and only one for binary operators.

This is because 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.

```
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);                   // subtraction
int operator==(vector);                        // comparison
friend int operator==(vector,vector);          // comparison
```

vector is a data type of **class** and may represent both magnitude and direction (as in physics and engineering) or a series of points called elements (as in mathematics).

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 function or a **friend** function.
3. Define the operator function to implement the required operations.

Overloaded operator functions can be invoked by expressions such as

```
op x or x op
```

for unary operators and

```
x op y
```

for binary operators. *op* x (or x op) would be interpreted as

```
operator op (x)
```

for **friend** functions.

Similarly, the expression **x op y** would be interpreted as either

```
x.operator op (y)
```

in case of member functions, or

```
operator op (x,y)
```

in case of **friend** functions. When both the forms are declared, standard argument matching is applied to resolve any ambiguity.

OVERLOADING UNARY OPERATORS

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 the sign of an operand when applied to a basic data item.

We will see here how to overload this operator so that it can be applied to an object in much the same way as is applied to an **int** or **float** variable. The unary minus when applied to an object should change the sign of each of its data items.

```
#include <iostream>

using namespace std;

class space
{
    int x;
    int y;
    int z;
  public:
    void getdata(int a, int b, int c);
    void display(void);
    void operator-();            // overload unary minus
};
void space :: getdata(int a, int b, int c)
{
    x = a;
    y = b;
    z = c;
}
void space :: display(void)
{
    cout<<"x = "<<x<<" ";
    cout<<"y = "<<y<<" ";
    cout<<"z = "<<z<<"\n";
}
void space :: operator-()
{
    x = -x;
    y = -y;
    z = -z;
}

int main()
```

```
{
    space S;
    S.getdata(10, -20, 30);
    cout << "S : ";
    S.display();

    -S;                    // activates operator-() function
    cout << "-S : ";
    S.display();

    return 0;
}
```

Output

```
S  : x = 10  y = -20  Z = 30
-S : x = - 10  y = 20  Z = - 30
```

The function ***operator - ( )*** *takes no argument. Then, what does this operator function do? It changes the sign of data members of the object* ***S.*** *Since this function is a member function of the same class, it can directly access the members of the object which activated it.*

Remember, a statement like

S2 = -S1 ;

will not work because, the function **operator-()** does not return any value. It can 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)// definition
{
     s.x = - s.x;
     s.y = - s.y;
     s.z = - s.z;
}
```

Note that the argument is passed by reference. It will not work if we pass argument by value because only a copy of the object that activated the call is passed to operator-( ). Therefore, the changes made inside the operator function will not reflect in the called object.

OVERLOADING BINARY OPERATORS

we illustrated, how to add two complex numbers using a friend function. A statement like

```
C = sum (A,B) ;// functional notation.
```

was used. The functional notation can be replaced by a natural

looking expression

```
C = A+B; //arithmetic notation
```

by overloading the + operator using an operator+( ) function. The Program shown below illustrates how this is accomplished.

```
#include <iostream>

using namespace std;

class complex
{
    float x;                          // real part
    float y;                          // imaginary part
  public:
    complex(){ }                      // constructor 1
    complex(float real, float imag)   // constructor 2
    { x = real; y = imag; }
    complex operator+(complex);
    void display(void);
};

complex complex :: operator+(complex c)
{
    complex temp;                     // temporary
    temp.x = x + c.x;                 // these are
    temp.y = y + c.y;                 // float additions
    return(temp);
}

void complex :: display(void)
{
    cout << x << " + j" << y << "\n";
}

int main()
{
    complex C1, C2, C3;               // invokes constructor 1
    C1 = complex(2.5, 3.5);           // invokes constructor 2
    C2 = complex(1.6, 2.7);
    C3 = C1 + C2;

    cout << "C1 = "; C1.display();
    cout << "C2 = "; C2.display();
    cout << "C3 = "; C3.display();

    return 0;
}
```

Output

```
C1= 2.5 + j3.5
C2=1.6 + j2.7
C3=4.1 + j6.2
```

Let us have a close look at the function **operator** +( ) and see how the operator overloading is implemented.

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

<u>We should note the following features of this function:</u>

1. It receives only one **complex** type argument explicitly.

2. It returns a **complex** type value.

3. It is a member function of **complex.**

The function is expected to add two complex values and return a complex value as the result but receives only one value as argument. Where does the other value come from? Now let us look at the statement that invokes this function:

```
C3 =C1+ C2; // invokes operator+() function
```

We know that a member function can .be invoked only by an object of the same class. Here, the object **C1** takes the responsibility of invoking the function and **C2** plays the role of an argument that is passed to the function. The above **invocation** statement is equivalent to

```
C3 = C1.operator+(C2); // usual function call syntax
```

Therefore, in the **operator+()** function, the data members of **C1** are accessed directly and the data members of **C2** (that is passed as an argument) are accessed using the dot operator. Thus, both the objects are available for the function. For example, in the statement

```
temp.x = x + c.x;
```

c.x refers to the object **C2** and x refers to the object **C1**. **temp.x** is the real part of **temp** that has been created specially to hold the results of addition of **C1** and C2. The function returns the complex temp to be assigned to C3. Figure below shows how this is implemented.

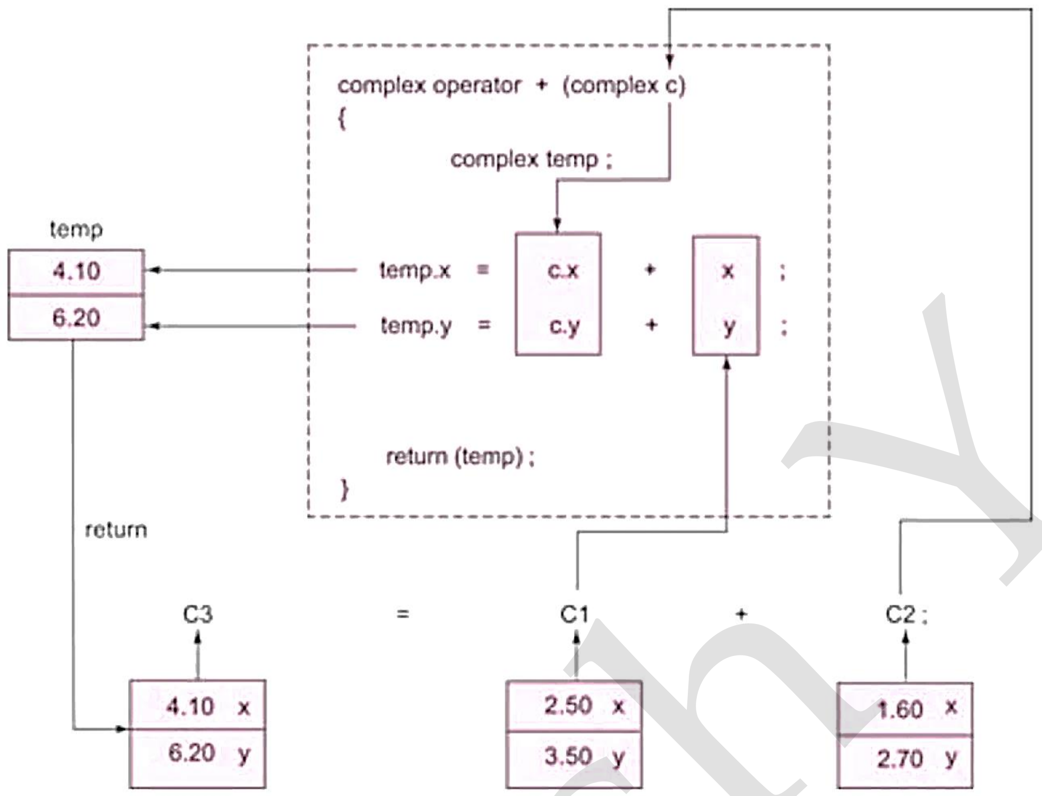


Fig. *Implementation of the overloded + operator*

As a rule, in 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.

We can avoid the creation of the **temp** object by replacing the entire function body by the following statement:

```
        return complex ((x+c.x),(y+c·y)); // invokes constructor 2
```

What does it mean when we use a class name with an argument list? When the compiler comes across a statement like this, it invokes an appropriate constructor, initializes an object with no name and returns the contents for copying into an object.

Such an object is called a temporary object and goes out of space as soon as the contents are assigned to another object. Using *temporary objects* can make the code shorter, more efficient and better to read.

OVERLOADING BINARY OPERATORS USING FRIENDS

As stated earlier, **friend** functions may be used in the place of member functions for overloading a binary operator, the only difference being that a **friend** function requires two arguments to be explicitly passed to it, while a member function requires only one.

The complex number program discussed in the previous section can be modified using a **friend** operator function as follows:

1. Replace the member function declaration by the **friend** function declaration.

```
friend complex operator+(complex , complex );
```

2. Redefine the operator function as follows:

```
complex operator +(complex a , complex b)
{
     return complex (( a.x + b.x ), (a.y + b.y));
}
```

In this case, the statement

`C3 = C1+ C2 ;`

is equivalent to

`C3 = operator +(C1 , C2 );`

In most cases, we will get the same results by the use of either a **friend** function or a member function. Why then an alternative is made available? There are certain situations where we would like to use **a friend** function rather than a member function.

For instance, consider a situation where we need to use two different types of operands for a binary operator, say, one an object and another a built-in type data as shown below,

`A = B + 2; ( or A = B * 2;)`

where **A** and **B** are objects of the same class. This will work for a member function but the statement

`A = 2 + B; ( or A = 2 * B)`

will not work. This is because the left-hand operand which is responsible for invoking the member function should be an object of the same class. However, **friend** function allows both approaches. How?

It may be recalled that an object need not be used to invoke a **friend** function but can be passed as an argument. Thus, we can use a friend function with a built-in type data as the *left-hand* operand and an object as the *right-hand* operand.

Program shown below illustrates this, using scalar *multiplication* of a vector. It also shows how to overload the input and output operators >> and <<.

```
#include <iostream.h>

const size = 3;
class vector
{
    int v[size];
public:
    vector();          // constructs null vector
    vector(int *x);    // constructs vector from array
    friend vector operator *(int a, vector b); // friend 1
    friend vector operator *(vector b, int a); // friend 2
    friend istream & operator >> (istream &, vector &);
    friend ostream & 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++)
       v[i] = x[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 << "(" << b.v [0];
     for(int i=1; i<size; i++)
        dout << ", " << b.v[i];
     dout << ")";
     return(dout);
}

int x[size] = {2,4,6};

int main()
{
     vector m;                       // invokes constructor 1
     vector n = x;                   // invokes constructor 2

     cout << "Enter elements of vector m " << "\n";
     cin  >> m;                      // invokes operator>>() function
     cout << "\n";
     cout << "m = " << m << "\n";    // invokes operator <<()

     vector p, q;

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

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

     return 0;
}
```

Output

```
Enter elements of vector m
5 10 15
m = (5, 10 , 15)
p = (10 , 20, 30 )
q = ( 4, 8 , 12)
```

The program overloads the operator * two times, thus overloading the operator function operator*() itself. In both the cases, the functions are explicitly passed two arguments and they are invoked like any other overloaded function, based on the types of its arguments. This enables us to use both the forms of scalar multiplication such as

```
          p = 2 * m; // equivalent to p = operator*(2, m) ;
          q = n * 2; // equivalent to q = operator* (n ,2) ;
```

The program and its output are largely self-explanatory. The first constructor

`vector() ;`

constructs a vector whose elements are all zero. Thus

```
vector m;
```

creates a vector m and initializes all its elements to 0. The second constructor

```
vector (int &x);
```

creates a vector and copies the ele1nents pointed to by the pointer argument x into it. Therefore, the statements

```
int x [3] = {2,4,6};
```

```
vector n = x;
```

create n as a vector with co1nponents 2, 4, and 6.

We have used vector variables like m and n in input and output statements just like simple variables. This has been made possible by overloading the operators >> and << using the functions:

```
friend istream & operator >> (istream &, vector &) ;
friend ostream & operator << (ostream &, vector &) ;
```

istream *and* ***ostream*** *are classes defined in the* ***iostream.h*** *file which has been included in the program.*

MANIPULATION OF STRINGS USING OPERATORS

Although these limitations exist in C++ as well, it permits us to create our own definitions of operators that can be used to manipulate the strings very much similar to the decimal numbers. (Recently, ANSI C++ committee has added a new class called **string** to the C++ class library that supports all kinds of string manipulations.

For example, we shall be able to use statements like

```
string3 = string1 + string2;
if(string1 >= string2) string = string1;
```

Strings can be defined as class objects which can be then manipulated like the built-in types. Since the strings vary greatly in size, we use *new* to allocate memory for each string and a pointer variable to point to the string array.

Thus we must create string objects that can hold these two pieces of information, namely, length and location which are necessary for string manipulations. A typical string class will look as follows:

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

We shall consider an example to illustrate the application of overloaded operators to strings. The example shown below overloads two operators, + and <= just to show how they are implemented. This can be extended to cover other operators as well.

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

class string
{
    char *p;
    int len;
  public:
    string() {len = 0; p = 0;}       // create null string
    string(const char * s);         // create string from arrays
    string(const string & s);        // copy constructor
    ~ string(){delete p;}           // destructor

    // + operator
    friend string operator+(const string &s, const string &t);

    // <= operator
    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);
}
```

```
// overloading + operator
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);
}
// overloading <= operator
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 <<   "\nstring1 = "; show(string1);
     cout <<   "\nstring2 = "; show(string2);
     cout <<   "\n";
     cout <<   "\nstring3 = "; show(string3);
     cout <<   "\n\n";
     if(string1 <= string3)
     {
       show(string1);
       cout << " smaller than ";
       show(string3);
```

```
            cout << "\n";
          }
          else
          {
            show(string3);
            cout << " smaller than ";
            show(string1);
            cout << "\n";
          }

          return 0;
      }
```

output

```
string1 = New
string2 = York
string3 = New Delhi
New smaller than New Delhi
```

Chapter 6

Review Questions

1. They should be declared in the public section.
2. They are invoked automatically when the objects are created.
3. They do not have return type, not even void.
4. They cannot be inherited.
5. Like other C++ functions, they can have default arguments.
6. Constructors cannot be virtual.

6.4: **What is a parameterized constructor?**

Ans:The constructors that can take arguments are called parameterized constructors.

6.5: **Can we have more than one constructors in a class? If yes, explain the need for such a situation.**

Ans:Yes, we have when we need to overload the constructor, then we have to do this.

6.6: **What do you mean by dynamic initialization of objects? Why do we need to this?**

Ans:Initializing value of object during run time is called dynamic initialization of objects.
One advantage of dynamic initialization is that we can provide various initialization formats using overloaded constructors.

6.7: **How is dynamic initialization of objects achieved?**

Ans:Appropriate function of a object is invoked during run-time and thus dynamic initialization of object is achieved.
Consider following constructor:
santo (int p, int q, float r);
santo (int p, int q, int r);
It two int type value and one float type value are passed then sant (int p, int q, float r) is invoked.
It three int type value are passed then santo (int p, into q, int r) is invoked.

6.8: **Distinguish between the following two statements:**
time T2(T1);
time T2 = T1;
T1 and T2 are objects of time class.

Ans:
time T2 (T1); ==> explicitly called of copy constructor
time T2 = T1; ==> implicitly called of copy constructor.

6.9: **Describe the importance of destructors.**

Ans:Destructors are important to release memory space for future use.

6.10: **State whether the following statements are TRUE or FALSE.**
(a) Constructors, like other member functions, can be declared anywhere in the class.
(b) Constructors do not return any values.
(c) A constructor that accepts no parameter is known as the default constructor.
(d) A class should have at least one constructor.
(e) Destructors never take any argument.

Ans:
(a) FALSE
(b) TRUE
(c) TRUE
(d) TRUE
(e) TRUE

Debugging Exercises

6.1: **Identify the error in the following program.**

```
#include <iostream.h>
class Room
{

    int length;
    int width;
public:
    Room(int 1, int w=0):
        width(w),
        length(1)
    {
    }
};
void main()
{
Room objRooml;
Room objRoom2(12, 8);
}
1
</br>
<span class="a">Solution:</span>Here there is no default constructor, so object could not be
written without any argument.
```

```
<strong>Correction :</strong>
1
 Void main ( )
   {
               Room Objroom2(12,8);
     }.
```

6.2: **Identify the error in the following program.**

```
#include <iostream.h>
class Room
{

    int length;
    int width;
public:
  Room()
  {
       length=0;
       width=0;
  }
Room(int value=8)
{
      length = width =8;
}
void display()
{
     cout<<length<< ' ' <<width;
}
};
void main()
{
Room objRooml;
objRoom1.display();
```

Solution:Room() and Room(int value=8) Functions are same, so it show Ambiguity error.
Correction : Erase Room() function and then error will not show.

6.3: **Identify the error in the following program.**

```
#include <iostream.h>
class Room
{
    int width;
    int height;
public:
 void Room()
   {
          width=12;
          height=8;
  }
Room(Room& r)
 {
        width =r.width;
        height=r.height;
        copyConsCount++;
 }
void discopyConsCount()
 {
       cout<<copyConsCount;
 }
};
int Room::copyConsCount = 0;
void main()
{
Room objRooml;
Room objroom2(objRoom1);
Room objRoom3 = objRoom1;
Room objRoom4;
objRoom4 = objRoom3;
objRoom4.dicopyConsCount();
}
```

Solution: Just erase “objRoom4 = objRoom3; invalid to call copy constructor.” for successfully run.

6.4: **Identify the error in the following program.**

```
#include <iostream.h>
class Room
{
     int width;
     int height;
     static int copyConsCount;
public:
 void Room()
   {
          width=12;
          height=8;
  }
Room(Room& r)
 {
        width =r.width;
        height=r.height;
        copyConsCount++;
 }
void discopyConsCount()
 {
       cout<<copyConsCount;
 }
};
int Room::copyConsCount = 0;
void main()
{
Room objRooml;
Room objroom2(objRoom1);
Room objRoom3 = objRoom1;
Room objRoom4;
objRoom4 = objRoom3;
objRoom4.dicopyConsCount();
}
```

Solution: Same as 6.3 problem solution.

Programming Exercises

6.1: **Design constructors for the classes designed in Programming Exercise 5.1 through 5.5 of Chapter 5.**

Solution: Study on Constructor and then see solution of chapter 5.

6.2: **Define a class String that could work as a user-defined string type. Include constructors that will enable us to create an uninitialized string:**

String s1; // string with length 0
And also initialize an object with a string constant at the time of creation like
String s2(“Well done!”);
Include a function that adds two strings to make a third string. Note that the statement
S2 = s1;
will be perfectly reasonable expression to copy one string to another.
Write a complete program to test your class to see that it does the following tasks:
(a) Creates uninitialized string objects.
(b) Creates objects with string constants.
(c) Concatenates two strings properly.
(d) Displays a desired string object.

Solution:

```
#include
#include
class string
{
char *str;
int length;

public:
string()
{
length = 0;
str = new char [length + 1] ;
}
string(char *s);
void concat(string &amp;m,string &amp;n);
string(string &amp;x);
void display();

};
string::string(string &amp;x)
{
length = x.length + strlen(x.str);
str = new char[length + 1];
strcpy(str, x.str);

}
void string:: concat(string &amp;m,string &amp;n)
{
length=m.length+n.length;
delete str;
str=new char[length+1];
strcpy(str,m.str);
strcat(str,n.str);
}
void string:: display()
{
cout&lt;&lt;str&lt;&lt;"\n";
}
string::string(char *s)
{
length = strlen(s);
str = new char[length + 1];
strcpy(str,s);
}

int main()
{
string s1;
string s2(" Well done ");
string s3(" Badly done ");
s2.display();
s1.concat(s2,s3);
s2=s3;
s2.display();
s1.display();
return 0;
}
```

output

```
Well done
Badly done
Well done Badly done
```

6.3: **A book shop maintains the inventory of books that are being sold at the shop. The list includes details such as author, title, price, publisher and stock position. Whenever a customer wants a book, the sales person inputs the title and author and the system searches the list and displays whether it is available or not. If it is not, an appropriate message is displayed. If it is, then the system displays the book details and requests for the number of copies required. If the requested copies are available, the total cost of the requested copies is displayed; otherwise "Required copies not in stock" is displayed.**
Design a system using a class called books with suitable member functions and constructors. Use new operator in constructors to allocate memory space required.

Solution:

```
#include
#include
#include
#include
#include

class book
{
char **author;
char **title;
float *price;
char **publisher;
int *stock_copy;
int size;

public:
book();
void book_detail(int i);
void buy(int i);
```

```
int search();
};

book :: book()
{
size=4;
author=new char*[80];
title=new char*[80];
publisher=new char*[80];

for(int i=0;i&lt;size;i++)
{
author[i]=new char[80];
title[i]=new char[80];
publisher[i]=new char[80];
}
stock_copy=new int[size];
price=new float[size];

title[0]="object oriented programming with c++";
title[1]="programming in ANCI";
title[2]="electronic circuit theory";
title[3]="computer algorithm";

author[0]="balagurusamy";
author[1]="balagurusamy";
author[2]="boyelstade";
author[3]="shahani";

stock_copy[0]=200;
stock_copy[1]=150;
stock_copy[2]=50;
stock_copy[3]=80;

price[0]=120.5;
price[1]=115.75;
price[2]=140;
price[3]=180.5;

}
void book::book_detail(int i)
{
cout&lt;&lt;" *********book detail **********\n";
cout&lt;&lt;setw(12)&lt;&lt;"Title"&lt;&lt;setw(25)&lt;&lt;"Author Name"
&lt;&lt;setw(18)&lt;&lt;"Stock copy\n";
cout&lt;&lt;setw(15)&lt;&lt;title[i]&lt;&lt;setw(16)&lt;&lt;author[i]&lt;&lt;setw(15)
&lt;&lt;stock_copy[i]&lt;&lt;"\n";

}
int book::search()
{
```

```
char name[80],t[80];
cout&lt;&lt;"Enter author name : ";

gets(name);
cout&lt;&lt;"and title of book in small letter : ";
gets(t);

int count=-1;
int a,b;
for(int i=0;i&lt;size;i++)
{

a=strcmp(name,author[i]);
b=strcmp(t,title[i]);
if(a==0 &amp;&amp; b==0)

count=i;

}

return count;
}

void book::buy(int i)
{
if(i&lt;0)
cout&lt;&lt;" This book is not available \n";

else
{
book_detail(i);
cout&lt;&lt;" How many copies of this book is required : ? "; int copy; cin&gt;&gt;copy;
int remaining_copy;
if(copy&lt;=stock_copy[i])
{
remaining_copy=stock_copy[i]-copy;
float total_price;
total_price=price[i]*copy;
cout&lt;&lt;"Total price = "&lt;&lt;total_price&lt;&lt;" TK\n";
}
else
cout&lt;&lt;" Sorry your required copies is not available \n";
}
}

int main()
{
book b1;
int result;

result=b1.search();
```

```
b1.buy(result);
return 0;
}
```

output

```
Enter author name : shahani

and title of book in small latter : computer algorithm

*********book detail *********

Title                          Author Name                 Stock copy

computer algorithm         shahani                        80

How many copies of this book is required : ?  78

Total price = 14079 TK
```

6.4: **Improve the system design in Exercise 6.3 to incorporate the following features:**
(a) The price of the books should be updated as and when required. Use a private member function to implement this.
(b) The stock value of each book should be automatically updated as soon as a transaction is completed.
(c) The number of successful and unsuccessful transactions should be recorded for the purpose of statistical analysis. Use static data members to keep count of transactions.

Solution:

```
#include
#include
#include
#include
#include

class book
{
static int successful,unsuccessful;
char **author;
char **title;
float *price;
char **publisher;
int *stock_copy;
int size;

public:
```

```
book();
void book_detail(int i);
void buy(int i);
int search();
void showtransaction();
void showdetail();
void edit_price();
};
int book::successful=0;
int book::unsuccessful=0;

book :: book()
{
size=5;
author=new char*[80];
title=new char*[80];
publisher=new char*[80];

for(int i=0;i&lt;size;i++)
{
author[i]=new char[80];
title[i]=new char[80];
publisher[i]=new char[80];
}
stock_copy=new int[size];

price=new float[size];

title[0]="object oriented programming with c++";
title[1]="programming in ANCI";
title[2]="electronic circuit theory";
title[3]="computer algorithm";
title[4]="complete solution of balagurusamy(c++)";

author[0]="balagurusamy";
author[1]="balagurusamy";
author[2]="boyelstade";
author[3]="shahani";
author[4]="abdus sattar";

stock_copy[0]=200;
stock_copy[1]=150;
stock_copy[2]=50;
stock_copy[3]=80;
stock_copy[4]=300;

price[0]=120.5;
price[1]=115.75;
price[2]=140;
price[3]=180.5;
price[4]=120;
```

```

}

void book::book_detail(int i)
{
cout&lt;&lt;" *********book detail **********\n";
cout&lt;&lt;setw(25)&lt;&lt;"Title"&lt;&lt;setw(30)&lt;&lt;"Author Name"
&lt;&lt;setw(18)&lt;&lt;"Stock copy\n";
cout&lt;&lt;setw(15)&lt;&lt;title[i]&lt;&lt;setw(16)&lt;&lt;author[i]&lt;&lt;setw(15)
&lt;&lt;stock_copy[i]&lt;&lt;"\n";

}

int book::search()
{
char name[80],t[80];
cout&lt;&lt;"Enter author name in small letter : ";
gets(name);
cout&lt;&lt;" title of book in small letter : ";
gets(t);

int count=-1;
int a,b;
for(int i=0;i&lt;size;i++)
{

a=strcmp(name,author[i]);
b=strcmp(t,title[i]);
if(a==0 &amp;&amp; b==0)

count=i;

}

return count;
}

void book::buy(int i)
{
if(i&lt;0)
{
cout&lt;&lt;" This book is not available \n";
unsuccessful++;
}

else
{
book_detail(i);
cout&lt;&lt;" How many copies of this book is required : ? "; int copy; cin&gt;&gt;copy;

if(copy&lt;=stock_copy[i])
{
stock_copy[i]=stock_copy[i]-copy;
float total_price;
total_price=price[i]*copy;
cout&lt;&lt;"Total price = "&lt;&lt;total_price&lt;&lt;" TK\n";
successful++;
}
else
{
cout&lt;&lt;" Sorry your required copies is not available \n";
unsuccessful++;
}
}
}

void book::edit_price()
{
cout&lt;&lt;" To edit price ";
int i;
i=search();
cout&lt;&lt;"Enter new price : "; float p; cin&gt;&gt;p;
price[i]=p;
}
void book::showdetail()
{
cout&lt;&lt;setw(22)&lt;&lt;"Title"&lt;&lt;setw(30)&lt;&lt;" stock copy "&lt;&lt;setw(20)
&lt;&lt;" Price per book "&lt;&lt;endl;
for(int i=0;i&lt;size;i++)
{
cout&lt;&lt;setw(35)&lt;&lt;title[i]&lt;&lt;setw(10)&lt;&lt;stock_copy[i]
&lt;&lt;setw(18)&lt;&lt;price[i]&lt;&lt;endl;
}
}
void book::showtransaction()
{
cout&lt;&lt;setw(22)&lt;&lt;"Successful transaction"&lt;&lt;setw(34)
&lt;&lt;"unsuccessful transaction "&lt;&lt;endl&lt;&lt;setw(10)
&lt;&lt;successful&lt;&lt;setw(32)&lt;&lt;unsuccessful&lt;&lt;endl;
}

int main()
{
book b1;
int result;

result=b1.search();
b1.buy(result);
b1.showdetail();
b1.showtransaction();
b1.edit_price();
cout&lt;&lt;"************details after edit price
```

```
*****************"&lt;&lt;endl;
b1.showdetail();

return 0;
}
```

output

```
Enter author name in small letter  :  abdus sattar

title of book in small letter  :  complete solution of balagurusamy(c++)

*********book detail **********

Title                                              Author Name             Stock copy

complete solution of balagurusamy(c++) abdus sattar              300

How many copies of this book is required : ? 100

Total price = 12000  TK

Title                                    stock copy            Price per book

object oriented programming with c++ 200            120.5

programming in ANCI              150                         115.75

electronic circuit theory           50                            140

computer algorithm                  80                            180.5

complete solution of balagurusamy(c++)  200          120

Successful transaction                         unsuccessful transaction

1                                                           0

To edit price Enter author name in small letter : shahani

title of book in small letter : computer algorithm

Enter new price : 200

************details after edit price*****************

Title                                    stock copy              Price per book
```

object oriented programming with c++	200	120.5
programming in ANCI	150	115.75
electronic circuit theory	50	140
computer algorithm	80	200
complete solution of balagurusamy(c++)	200	120

Chapter 7

Review Questions

7.1: **What is operator overloading?**

Ans: The mechanism of giving special meaning to an operator is known as operator overloading.

7.2: **Why is it necessary to overload an operator?**

Ans: We can almost create a new language of our own by the creative use of the function and operator overloading techniques.

7.3: **What is an operator function? Describe the syntax of an operator function.**

Ans: 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. By which function this is done, is called operator function. Syntax of operator function:

```
return type class name : : operator OP (argument list)
  {
 function body // task defined
   }
```

7.4: **How many arguments are required in the definition of an overloaded unary operator?**

Ans: No arguments are required.

7.5: **A class alpha has a constructor as follows:**
alpha(int a, double b);
Can we use this constructor to convert types?

Ans: No. The constructors used for the type conversion take a single argument whose type is to be converted.

7.6: **What is a conversion function How is it created Explain its syntax.**

Ans: C++ allows us to define an overloaded casting operator that could be used to convert a class type data to a basic type. This is referred to conversion function.
Syntax:

```
Operator type name ( )
 {
    (Function Statements)
  }
```

7.7: **A friend function cannot be used to overload the assignment operator =. Explain why?**

Ans: A friend function is a non-member function of the class to which it has been defined as friend. Therefore it just uses the functionality (functions and data) of the class. So it does not consist the implementation for that class. That's why it cannot be used to overload the assignment operator.

7.8: **When is a friend function compulsory? Give an example.**

Ans: When we need to use two different types of operands for a binary operator, then we must use friend function.
Example:
A = B + 2;
or
A = B * 2;
is valid
But A = 2 + B
or
A = 2 * B will not work.
Because the left hand operand is responsible for invoking the member function. In this case friend function allows both approaches.

7.9: **We have two classes X and Y. If a is an object of X and b is an object of Y and we want to say a = b; What type of conversion routine should be used and where?**

Ans: We have to use one class to another class type conversion. The type-conversion function to be located in the source class or in the destination class.

7.10: **State whether the following statements are TRUE or FALSE.**
(a) Using the operator overloading concept, we can change the meaning of an operator.
(b) Operator overloading works when applied to class objects only.
(c) Friend functions cannot be used to overload operators.
(d) When using an overloaded binary operator, the left operand is implicitly passed to the member function.
(e) The overloaded operator must have at least one operand that is user-defined type.
(f)Operator functions never return a value.
(g) Through operator overloading, a class type data can be converted to a basic type data.
(h) A constructor can be used to convert a basic type to a class type data.

Ans:
(a) FALSE
(b) TRUE
(c) FALSE
(d) FALSE
(e) TRUE
(f) FALSE
(g) TRUE
(h) TRUE

Debugging Exercises

7.1: **Identify the error in the following program.**

```
#include <iostream.h>
class Space
{
    int mCount;
public:
    Space()
    {
        mCount = 0;
    }

     Space operator ++()
     {
         mCount++;
         return Space(mCount);
    }
};
void main()
{
    Space objSpace;
    objSpace++;
}
```

Solution: The argument of Space() function is void type, so when this function is called there are no argument can send to it. But ‘mCount’ argument is sending to Space() function through return space(mCount); Statement.
Here return space (mCount); replaced by return space();

7.2: **Identify the error in the following program.**

```
#include <iostream.h>
enum WeekDays
{
    mSunday'
    mMonday,
    mtuesday,
    mWednesday,
    mThursday,
    mFriday,
    mSaturday
};
bool op==(WeekDays& w1, WeekDays& w2)
{
    if(w1== mSunday && w2==mSunday)
        return 1;
    else if(w1==mSunday && w2==mSunday)
        return 1;
    else if(w1==mSunday && w2==mSunday)
        return 1;
    else if(w1==mSunday && w2==mSunday)
        return 1;
    else if(w1==mSunday && w2==mSunday)
        return 1;
    else if(w1==mSunday && w2==mSunday)
        return 1;
    else if(w1==mSunday && w2==mSunday)
        return 1;
   else
        return 0;
}
void main()
{
    WeekDays w1 = mSunday, w2 = mSunday;
    if(w1==w2)
    cout<<"Same day";
    else
    cout<<"Different day";
}
```

Solution: bool OP = = (WeekDays & w1, WeekDays & w2) replaced by bool operator = = (Weekdays & w1, WeekDays & w2). All other code will remain same.

7.3: **Identify the error in the following program.**

```
#include <iostream.h>
class Room
{
    float mWidth;
    float mLength;
public:
    Room()
    {
    }
    Room(float w, float h)
            :mWidth(w), mLength(h)
   {
   }
   operator float ()
   {
      return (float)mWidth * mLength;
   }

   float getWidth()
   {
   }
   float getLength()
   {
       return mLength;
   }
};

void main()
{
    Room objRoom1(2.5, 2.5)
    float fTotalArea;
    fTotalArea = objRoom1;
    cout<< fTotalArea;
}
```

Solution: The float getWidth() function return float type data, but there is no return statement in getWidth() function. So it should write as follows.

```
 float getWidth()
{
   return mWidth;
}
```

All other code will remain unchanged.

Programming Exercises

7.1: **Crate a class FLOAT that contains one float data member. Overload all the four arithmetic operators so that they operate on the objects of FLOAT.**

Solution:

```
#include<iostream.h>

class FLOAT
{
        float data;
         public:
         FLOAT(){};
         FLOAT(float d)
         { data=d;}
         FLOAT operator+(FLOAT f1);
         FLOAT operator-(FLOAT f2);
         FLOAT operator*(FLOAT f3);
         FLOAT operator/(FLOAT f4);
        void display();
};
FLOAT FLOAT::operator+(FLOAT f1)
{
         FLOAT temp;
         temp.data=data+f1.data;
        return (temp);
}
FLOAT FLOAT::operator-(FLOAT f2)
{
        FLOAT temp;
        temp.data=data-f2.data;
        return temp;
}
FLOAT FLOAT::operator*(FLOAT f3)
```

```
{
        FLOAT temp;
    temp.data=data*f3.data;
       return temp;
}
FLOAT FLOAT::operator/(FLOAT f4)
{
          FLOAT temp;
          temp.data=data/f4.data;
          return (temp);
}
void FLOAT:: display()
{
          cout<<data<<"\n";
}
int main()
{
               FLOAT F1,F2,F3,F4,F5,F6;
               F1=FLOAT(2.5);
               F2=FLOAT(3.1);
               F3=F1+F2;
               F4=F1-F2;
               F5=F1*F2;
               F6=F1/F2;
               cout<<" F1 = ";
               F1.display();
              cout<<" F2 = ";
              F2.display();
              cout<<" F1+F2 = ";
              F3.display();
             cout<<" F1-F2 = ";
             F4.display();
             cout<<" F1*F2 = ";
             F5.display();
             cout<<" F1/F2= ";
             F6.display();
   return 0;
}
```

output

```
F1 = 2.5
F2 = 3.1
F1+F2 = 5.6
F1-F2 = -0.6
F1*F2 = 7.75
F1/F2= 0.806452
```

7.2: **Design a class Polar which describes a point in the plane using polar coordinates radius and angle. A point in polar coordinates is shown below figure 7.3**
Use the overload + operator to add two objects of Polar.
Note that we cannot add polar values of two points directly. This requires first the conversion of points into rectangular coordinates, then adding the respective rectangular coordinates and finally converting the result back into polar coordinates. You need to use the following trigonometric formula:
x = r *
cos(a);

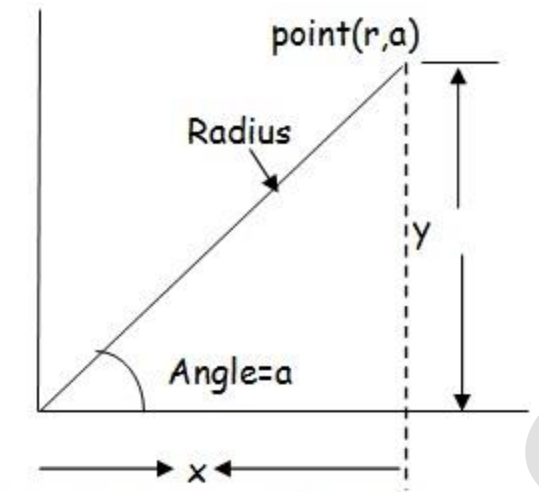


fig: polar coordinates of a point

y = r * sin(a);
a = atan(y/x); //arc tangent
r = sqrt(x*x + y*y);

Solution:

```
#include<iostream.h>
#include<math.h>
#define pi 3.1416
class polar
{
            float r,a,x,y;
            public:
            polar(){};
            polar(float r1,float a1);
            polar operator+(polar r1);
            void display(void);
};

polar :: polar(float r1,float a1)
{
          r=r1;
         a=a1*(pi/180);
```

```
        x=r*cos(a);
        y=r*sin(a);
}

polar polar :: operator+(polar r1)
{
        polar R;

        R.x=x+r1.x;
        R.y=y+r1.y;
        R.r=sqrt(R.x*R.x+R.y*R.y);
        R.a=atan(R.y/R.x);

  return R;
}

void polar::display()
{
       cout<<"radius = "<<r<<"\n angle = "<<a*(180/pi)<<"\n";
}

int main()
{
 polar p1,p2,p3;
 float r,a;
 cout<<" Enter radius and angle : ";
 cin>>r>>a;
 p1=polar(r,a);
 p2=polar(8,45);
 p3=p1+p2;
 cout<<" p1 : \n";
 p1.display();
 cout<<" p2 : \n ";
 p2.display();
 cout<<" p3 : \n ";
 p3.display();
 return 0;
}
```

output

```
Enter radius and angle : 10 45
P1:
radius = 10
angle = 44.999998
P2 :
radius = 8
angle = 44.999998
P3 :
radius = 18
angle = 44.999998
```

7.3: **Create a class MAT of size m * n. Define all possible matrix operations for MAT type objects.**

Solution:

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

class mat
{
            float **m;
            int rs,cs;
            public:
             mat(){}
             void creat(int r,int c);
             friend istream & operator >>(istream &,mat &);
             friend ostream & operator <<(ostream &,mat &);
             mat operator+(mat m2);
             mat operator-(mat m2);
             mat operator*(mat m2);
};

void mat::creat(int r,int c)
{
         rs=r;
         cs=c;
         m=new float *[r];
         for(int i=0;i<r;i++)
        m[i]=new float1;
}

istream &  operator>>(istream &din, mat &a)
{
       int r,c;
       r=a.rs;
       c=a.cs;
       for(int i=0;i<r;i++)
        {
                 for(int j=0;j<c;j++)
                    {
                        din>>a.m[i][j];
                    }
        }
    return (din);
}
ostream & operator<<(ostream &dout,mat &a)
{
             int r,c;
```

```
                r=a.rs;
                c=a.cs;
                        for(int i=0;i<r;i++)
                {
                     for(int j=0;j<c;j++)
                       {
                                          dout<<setw(5)<<a.m[i][j];
                                 }
                                 dout<<"\n";
                }
 return (dout);
}
mat mat::operator+(mat m2)
{
               mat mt;
               mt.creat(rs,cs);
              for(int i=0;i<rs;i++)
               {
                   for(int j=0;j<cs;j++)
                               {
                              mt.m[i][j]=m[i][j]+m2.m[i][j];
                      }
                }
      return mt;
}

mat mat::operator-(mat m2)
{
          mat mt;
          mt.creat(rs,cs);
           for(int i=0;i<rs;i++)
            {
                 for(int j=0;j<cs;j++)
                  {
                        mt.m[i][j]=m[i][j]-m2.m[i][j];
                  }
            }
     return mt;
}

mat mat::operator*(mat m2)
{
           mat mt;
                  mt.creat(rs,m2.cs);

     for(int i=0;i<rs;i++)
          {
                for(int j=0;j<m2.cs;j++)
                 {
                        mt.m[i][j]=0;
                        for(int k=0;k<m2.rs;k++)
```

```
                    mt.m[i][j]+=m[i][k]*m2.m[k][j];
               }
          }

      return mt;
 }
int main()
{
         mat m1,m2,m3,m4,m5;
         int r1,c1,r2,c2;
         cout<<" Enter first matrix size : ";
         cin>>r1>>c1;
         m1.creat(r1,c1);
         cout<<"m1 = ";
         cin>>m1;
         cout<<" Enter second matrix size : ";
         cin>>r2>>c2;
         m2.creat(r2,c2);
         cout<<"m2 = ";
         cin>>m2;
         cout<<" m1:"<<endl;
         cout<<m1;
         cout<<" m2: "<<endl;
         cout<<m2;
         cout<<endl<<endl;
        if(r1==r2 && c1==c2)
          {
               m3.creat(r1,c1);
                      m3=m1+m2;
               cout<<" m1 + m2: "<<endl;
               cout<<m3<<endl;
              m4.creat(r1,c1);

               m4=m1-m2;
              cout<<" m1 - m2:"<<endl;
              cout<<m4<<endl<<endl;

        }
   else
    cout<<" Summation & substraction are not possible n"<<endl
        <<"Two matrices must be same size for summation &   substraction "<<endl<<endl;
if(c1==r2)
{
           m5=m1*m2;
           cout<<" m1 x m2: "<<endl;
           cout<<m5;
}
else
cout<<" Multiplication is not possible "<<endl
<<" column of first matrix must be equal to the row of second matrix ";
return 0;
```

146}

output

```
Enter first matrix size : 2  2

m1 =

1     2

3     4

Enter second matrix size : 2    2

m2 =

5     6

7     8

m1 =

1     2

3     4

m2 =

5     6

7     8

m1+m2:

6       8

10     12

m1-m2:

-4       -4

-4      -4

m1 x m2:

19      22
```

```
43    50
```

7.4: **Define a class String. Use overload == operator to compare two strings.**

Solution:

```
#include<iostream.h>
#include<string.h>
#include<stdio.h>

class string
{
            char str[1000];
            public:
            void input(){gets(str);}
                 int operator==(string s2);
};
int string::operator==(string s2)
{
               int t= strcmp(str,s2.str);
       if(t==0)
                 t=1;
       else
       t=0;
    return t;
}

int main()
{

    char st1[1000],st2[1000];
    string s1,s2;
     cout<<" Enter 1st string : ";
     s1.input();
     cout<<" enter 2nd string : ";
     s2.input();

    if(s1==s2)
    cout<<" Two strings are equal ";
    else
    cout<<" Two string are not equal ";
  return 0;
}
```

output

```
Enter 1st string : our sweetest songs tel our saddest thought
enter 2nd string : a burning desire lead to success.
Two string are not equal
```

7.5: **Define two classes Polar and Rectangle to represent points in the polar and rectangle systems. Use conversion routines to convert from one system to the other.**

Solution:

```
#include<iostream.h>
#include<math.h>
#define pi 3.1416
class conversion_point
{
            float x,y,r,theta;
             public:
              void set_xy();
              void set_r_theta();
              void show_xy();
              void show_r_theta();
              void conversion(int t);
};
      void conversion_point::set_xy()
{
       cout<<"Enter the value of x & y : ";
       cin>>x>>y;
}
       void conversion_point::set_r_theta()
{
         cout<<"Enter the value of r & theta :";
         cin>>r>>theta;
         theta=(pi/180)*theta;
}

       void conversion_point::show_xy()
{
          cout<<" CERTECIAN FORM :\n"
               <<" x = "<<x<<"\n"
               <<" y = "<<y<<"\n";
}
void conversion_point::show_r_theta()
{
            cout<<" POLAR FORM :\n"
                 <<" r = "<<r<<"\n"
                 <<" theta = "<<(180/pi)*theta<<" degree \n";
}

void conversion_point::conversion(int t)
```

```
{
        if(t==1)
         {
                 r=sqrt(x*x+y*y);

                 if(x!=0)
                  {
                        theta=atan(y/x);
                        show_r_theta();
                  }

                else
                  {
                         cout<<" POLAR FORM :\n"
                                <<" r = "<<r<<"\n"
                               <<" theta = 90 degree\n";
                  }

          }
     else if(t==2)
       {
                x=r*cos(theta);
                y=r*sin(theta);
               show_xy();
       }
}

int main()
{
             conversion_point santo;
              int test;
             cout<<" press 1 to input certecian point \n"
                   <<" press 2 to input polar point \n "
                   <<" what is your input ? : ";
                  cin>>test;
                  if(test==1)
                  santo.set_xy();
                  else if(test==2)
                  santo.set_r_theta();
                  santo.conversion(test);

     return 0;
}
```

output

```
Press 1 to input certecian point
Press 2 to input polar point
what is your input ? 1                                   r = 6.403124
Enter the value of x & y : 4 5                           theta = 51.340073 degree
POLAR FORM :
```