UNIT-IV

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 same as the class name.

-> The constructor is invoked automatically whenever an object of its class is created.

-> It is called constructor bec~~ose~~ause it constructs the values of data members of the class.

-> A constructor is declared and defined as follows.

Syntax:

```
class  <class name>
{
  Private:
        datamembers;
        member functions;
  Public:
        constructorname()
        {
          initialization;
        }
};
```

Eg:

```
class A
{
Private:
   int a,b,c;
Public:
   A()
   {
   a=0;
   b=1;
   }
   void getdata().
};
```

The constructor function have some characteristics:

1. They should be declared in public section only.
2. They are invoked automatically when the objects are created.
3. They do not have any return type. Not even void.
   ∴ They don't return any value.
4. They can't be inherited. Though a derived class can call the base class constructor.
5. They can have default arguments.

6. constructors can not be virtual.

7. They make implicit calls to the operators new & delete when memory allocation is required.

Example: Program for constructor (or) default constructor.

```
#include <iostream> using ~~na~~ namespace std;
class fibonacci
{
 private:
      int n1,n2,n3;
   public:
        fibonacci()
        {
        n1=0;
        n2=1;
        n3=n1+n2;
        }
        void process()
        {
        n1=n2;
        n2=n3;
        n3=n1+n2;
        }
      void output()
      {
      cout<<"n3 is"<<n3;
      }
 };
```

```
~~void~~
int main()
{
fibonacci f;
int x,i;
cout<<"\n how many
              numbers";
cin>>x;
cout<<0<<1<<"\n";
for(i=1; i<x; i++)
{
 f1.output();
 f1.process();
}
}
```

Types of Constructors:

→ They are different types of constructors. They are

1. Default constructor.
2. parameterized constructor.
3. copy constructor.
4. Dynamic constructor.

① Default constructor:

→ A constructor that accepts no parameters is called default constructor.

→ The default constructor for a class A is A::A().

→ If no constructor is defined, then the compiler supply the default constructor.

eg: The addition of 2 numbers program using default constructor.

```
#include<iostream>
using namespace std;
 class addition
 {
   private:
           int a,b,c;
   Public:
        addition()
        {
          a=0;
          b=0;
        }
     Void getdata()
     {
   cout<<"enter a and b values";
    Cin>>a>>b;
   }
                        void calculate()
                        {
                          c=a+b;
                        }
                        void ~~d~~display()
                        {
                       cout<<"a value is "<<a;
                       cout<<"b value is"<<b;
                       cout<<"sum is"<<c;
                         }
                       };
                             int main()
                             {
                              addition A;
                              A.getdata();
                              A.calculate();
                              A.display();
                             }
```

Parameterized Constructor:

→ A constructor that can take arguments are called parameterized constructor.

→ C++ Permits us to ~~create~~ achieve this objective by passing argument to the constructor function when the objects are created.

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

→ when a parameterized constructor have been declare the object - declaration statement (does) integer inte; does not work.

→ we must pass the initial values as arguments to the constru-ctor function when an object is declared. This can be done in 2 ways.

1. By calling the constructor explicitly.

Eg: Explicit call.

```
integer inte = integer(0, 100);
```

2. By calling the constructor implicitly.

Eg: implicit call

```
integer inte(0,100);
```

3. Eg: Program to demonstrate parameterized constructor.

```
#include<iostream>
  using namespace std;
  class integer
  {
   int m,n;
  Public:
       integer(int, int); //declared constructor
     void display();
  };
  integer::integer(int x, int y) // constructor defined.
  {
   m=x;
   n=y;
  }
  void integer::display()
  {
  cout<<"m="<<m<<endl;
  cout<<"n="<<n <<endl;
  }
```

```
int main()
{
integer s1(0,100);
//constructor called implicitly
integer s2 = integer(25,75);
//constructor called explictly
cout <<"object1"<<endl;
  s1.display();
cout<<"\n object2"<<endl;
s2.display();
}
```

output:

```
object 1
  m=0   n=100
object 2
  m=25   n=75
```

Copy constructor:

→ The copy constructor is a constructor that is used to declare and initialized an object from another object.

→ The copy constructor is used to
* Initialize one object from another object of the same type.
* copy an object to pass it as an argument to a function.
* copy an object to return and it from a function.

for eg the statement Integer I2(I1); would define the object I2 and at the same time initilize it to the value of I1.
* Another from of these Statement is Integer I2 = I1;
* The process of Initilizing through a copy constructor is known as copy Initialization. Ex: I2 = I1;

eg:
```
#include<iostream>
using namespace std;
class code
{
Private:
        int id;
 Public:
   code() //default constructor
   {
   }
  Code(int a) // Parameterized constructor
  {
    id = a;
  }
   code(code &x) //copy constructor
   {
      id = x.id;
   }
   void display()
   {
   cout<<~~id~~ ~~id of~~ cout<<id;
   }
};
```

```
int main()
{
 code A(100);
 code B(100); // B(A);   code
 Code C = A;
  code D;
   D = A;
  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();
  ~~get~~
}
```

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

Dynamic constructor:

→ The constructor can also be used to allocate memory while creating objects.

→ This will enable the system to allocate the small amount of memory for each object, when the object are not of the same size. This result in saving of memory.

* Allocation of memory to objects at time of their construction is known as dynamic construction of objects.

* The memory is allocated with the help of new operator.

ex:

```
#include<iostream>
#include <string>
using namespace std;
class string
{
char *name;
int length;
public:
    string()
    {
     length=0;
    name=new char[length+1];
    }
   string (char *s)
   {length = strlen(s);
   name=new char[length+1];
   strcpy(name,s);
   }
   void display()
   {
   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
```

Overloaded constructors (or) Multiple constructors in a class:

→ C++ Permits us to use more than one constructor in a class.

→ In the below Program we have 3 constructors.

1. constructor with no arguments.
2. constructor with one argument
3. constructor with two arguments

→ The First constructor, which takes no arguments is used to create objects which are not initialized.

→ The Second constructor, which takes one argument is used to create objects and initialize them.

→ The 3rd constructor, which takes two arguments is also used to create objects and initialize them, to specific value.

Eg: complex ( ) { } — constructor-1
complex (float a) { x = y = a; } — 2
complex (float real, float image) — 3
{ x = real; y = imag; }

object creation:
Const-1 → complex c.
Cons-2 → complex A(1.5)
Const-3 → complex B(1.2, 1.3)

Example Program for multiple constructors in a class

```
#include <iostream>
 using namespace std;
 class complex
 {
 float x,y;
Public:
      complex( ){ }// no,arg
      complex(float a) ~~// one arg~~ {x = y = a; }   // one argument
      complex(float real float img)
      {
        x = real; y = img;
      }
      friend complex sum(complex, complex);
      friend void ~~swap~~ Show(complex);
};
```

```
complex sum(complex c1, complex c2) // friend
{
    complex c3;
    c3.x = c1.x + c2.x;
    c3.y = c1.4 + c2.4;
    return(c3);
}
  void show(complex c) // friend  {
  cout<< c.x <<"+j" <<c.y<<"\n";
  }
  int main()
  {
  complex A(2.7,3.5);
   complex B(1.6);
   complex c;
   c = sum(A,B);
  cout<<"A =", show(A);
  cout<<"B=", show(B);
  cout<<"c=", show(c);
    complex P, Q, R;
     P = complex(2.5,3.9);
     Q = complex(1.6,2.5);
     R = sum(P,Q);
     cout<<"\n";
    cout<<"P =", show(P);
     cout<<" Q =", show(Q);
      cout<<" R=";show(R);
      return 0;
```

output:

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

const objects:

→ we can create and use constant objects using const keywoard, before object declaration.

Eg: const matrix x ; // x is constant object.

→ Any Attempt to modifie the values of ~~a~~ constant object will generate compile time error.

→ These const objects are used to call the only const member functions.

→ when ever const objects try to invoke non const member function, the compiler will generate an error.

Example program:

```
#include<iostream>
 using namespace std;
 class test
  { private :
            int data;
       public :
             test () { data=2; }
     void  showdata() const
      {
       cout << "data value = "<<data<<endl;
      }
    };
     int main()
      { Const test obj;
          obj. showdata(); return 0;
      }
```

output: data value = 2.

const objects:

→ we can create and use constant objects using const keywoard, before Object declaration.

Eg: const matrix x ; // x is constant object.

→ Any Attempt to modifie the values of ~~no~~ constant object will generate compile time error.

→ These const objects are used to call the only const member functions.

→ when ever const objects try to invoke non const member function, the compiler will generate an error.

Example program:

```
#include<iostream>
 using namespace std;
 class test
  { private :
              int data;
         Public :
               test () { data=2; }
        Void  showdata() const
          {
           cout << "data value = "<<data<<endl;
          }
         };
           int main()
            { Const test obj;
                obj. showdata(); return0;
            }
```

output: data value = 2.

Destructors:

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

→ like a constructor a destructor also a special member function, whose name is same as the class but preceded by the tilde (~).

→ A destructor never takes any arguments.

* It does not return any value.

* It will be invoked implicitly by the compiler, up on exist from the program.

* A destructor is used to clean up the storage that is no longer accessible.

Syntax:

```
        class integer
        {
Public;   integer()  //constructor
          {
          }
          ~integer()  //destructor
           {
           }
        };
```

Ex:

```
class integer
{
   int p;
Public:
   integer()
   {  p=0;
   }
  ~integer()
   {
   delete p;
   }
};
```

* It is a good Practice to declare destructor in a Program since it releases memory space for future use.

Program:

```
#include<iostream>
using namespace std;
  class test
  {
    int i;
  Public:
      test()
      {  i=0;
      cout<<"Hello"<<endl;
      }
      void A()
      { cout<<"I="<<++i <<endl;
      }
```

```
~test()
{
 cout<<"Hi"<<endl;
}
};
 int main()
 {
    test t1,t2;
    t1.f();
 }
```

output:

```
Hello
Hello
I=1
Hi
```

operator overloading :

1. overloading refers to the use of same thing for different purposes.
2. operator overloading is one of the existing featuresof C++ languages.
3. C++ has the ability to provide the operation with the special meaning for a data type.
4. ~~C++ has the ability~~ The mechanism of giving such a special meaning to an operator is known as operator overloading.
5. operator overloading provides a flexible option for the creation of new definitions.

eg: 2+3 ; + is operator , 2 & 3 are operands.

* Defining operator overloading:

* The general form of an operator function is follows, return data_type classname :: operator op(argument list)

```
{
    // function body
}
```

* where return data type is the type of value returned by the special operation.

* op is the operator beig overloaded.

* operator ~~op~~ is the function name.

* The op is Precided by the keywoard operator.

* Arguments may be passed either by value or by reference.

eg:
① vector operator +(vector); // vector addition.
②. int operator == (vector); // ~~comarission~~ comparison

The process of overloading involves the following steps:-
③. * create a class that defines the data type, that is to be used in the overloading operations.

* Declare the operator function operator op() in the public section of the class.
* It may be either a member function or a friend function.
* Define the operator function to implement the required operations.

Overloaded operator function can be invoked by expression such as op x (or) x op, operator op(x) for unary operator.
Ex: ++a (or) a++ Ex: -(a)

for Binary operator:

x op y (or) → Ex a+b;

x . operator op(y). Ex x . operator+(a);

Overloading unary operators: * write Rules of operator overloading

* Let us consider the unary minus operator.
* A minus operator when used as unary operator, takes just one operand.
* The unary minus when applied to an object should change the sign of its data items.

example program for refere class notes.

Overloading Binary operator:

→ Like unary operator overloading we can also overload Binary operators. → write Rules of operator overloading.
→ An operator '+' can be used to add two integer values or two floating point members (or) two complex no's.
→ In this program the function is accepted to add two complex values and return a complex value as it result.
→ It receives only one complex type argument explicitly.
→ It returns a complex data type value.
→ It is a member function of complex.

Example Program: Refere class notes.

Rules for overloading operators:

· There are certain restrictions and Limitation in overloading operators. Some of them are.

1. Only existing operators can be overloaded.
2. New operators can not be created.
3. There are some operators that can not be overloded. They are:

* Conditional operators (?:) size of operator, Scope resolution operator, class member access operators (->;.)

4. we can not use friend function to overload certain (operations) operators (Assignment operator =, class member access operator ->, function call operator (), subscripting operator [ ]).
5. The overloaded operator must have atleast one operand that is user defined type.
6. We can not change the basic meaning of an operator.
7. Unary operators overloaded by means of a member functions can take no explicit argument and return no explicit value.
8. Binary operators overloaded to an operator take on explicit argument.

Manipulation of Strings using operators:

1. They are no operators manipulating the strings in c.
2. Although these limitations exist in c++ as well its permits us to create our own definititions of operators that can be use to manipulate the strings.
3. C++ has added a new class called string to the c++ class library that supports all kinds of string manipulations

Overloading Assignment operator:-

→ we can overload the assignment operator (=) just as we can, do the other operators and create an object just like the copy constructor.

→ write Rules of operator overloading.

Program:

```
#include <iostream>
using namespace std;
class distance
{
private:
        int feet, inches;
public:
        distance()
        {
          feet=0;
          inches=0;
        }
      distance(int f, int i)
      {
        feet =f;
        inches=i;
      }
      void operator=(const distance &d)
      {
        feet =d.feet;
        inches =d.inches;
      }
      void display_distance()
      {
      cout<<"F:"<<feet<<"I:"<<inches<<endl;
      }
};    int main()
      {
      distance D1(11,10), D2(5,11);
      cout<<"first distance";
       D1.display_distance();
       cout<<"2nd distance";
       D2.display_distance();
         D1=D2;
      cout<<"first distance";
        D1.display_distance;
      }
```

Type conversion (or) data conversion:

1. Representing the same data in multiple forms is a common Pratice in computations.
2. It involves the convesion of data from one form to another for.eg : conversion to Radians to degrees.
   int to float.
3. An assignment operator causes the automatic type conversion. The type of data to the right of an assignment operator is automatically converted to the type of the variable on the left.

* for example the ~~student~~ statements

```
int m;
float x=3.14169;
   m=x;
```

* Here x is converted to an integer before its value is assigned to m.
* The type conversion is automatic as long as the data types are build in types.

~~Theree~~ Three types of data conversions:

1. conversion from basic type to class type.
2. conversion from class type to Basic type
3. conversion from one class type to another class type.

Basic to class type:

1. The conversion from basic type to class type is easy to accomplish.
   * for example a constructor was used to build a vector object from an in type array.
2. Similarly we use another constructor to build a string type object from a character type variable.

* These are all examples where constructors ~~perform~~ perform a default type conversion from the argument type to the constructor class type.

consider an example of converting a ~~[illegible]~~ int type to a class type

eg: class time

```
    {
        int hrs, min;
      public :
          ==
          time (int t)        // constructor
          {
            hrs = t/60;
            mins = t%60;
          }
    };
```

The following conversion statements can be used in a function

```
time T1; // object T1 created.
int duration = 85;
T1 = duration    // int to class type.
```

Conversion from Class to Basic type:

1. C++ allows us to define an overloaded casting operator that could be used to convert a class type data to a basic data type.

The general form of an overloaded casting operator function usually refered to as a Convention function is:

```
operator type name( )
{
    ==  function statements
}
```

* The casting operator function should satisfy the following conditions.
• It must be a class member.
• It must not satisfy a returntype
• It must not have any arguments.

These function convert a class type data to type name.

One class to another class:

They are ~~[illegible]~~ situations where we should like to convert one class type data to another class type.

Eg: Obj x = Obj y //objects of different types.

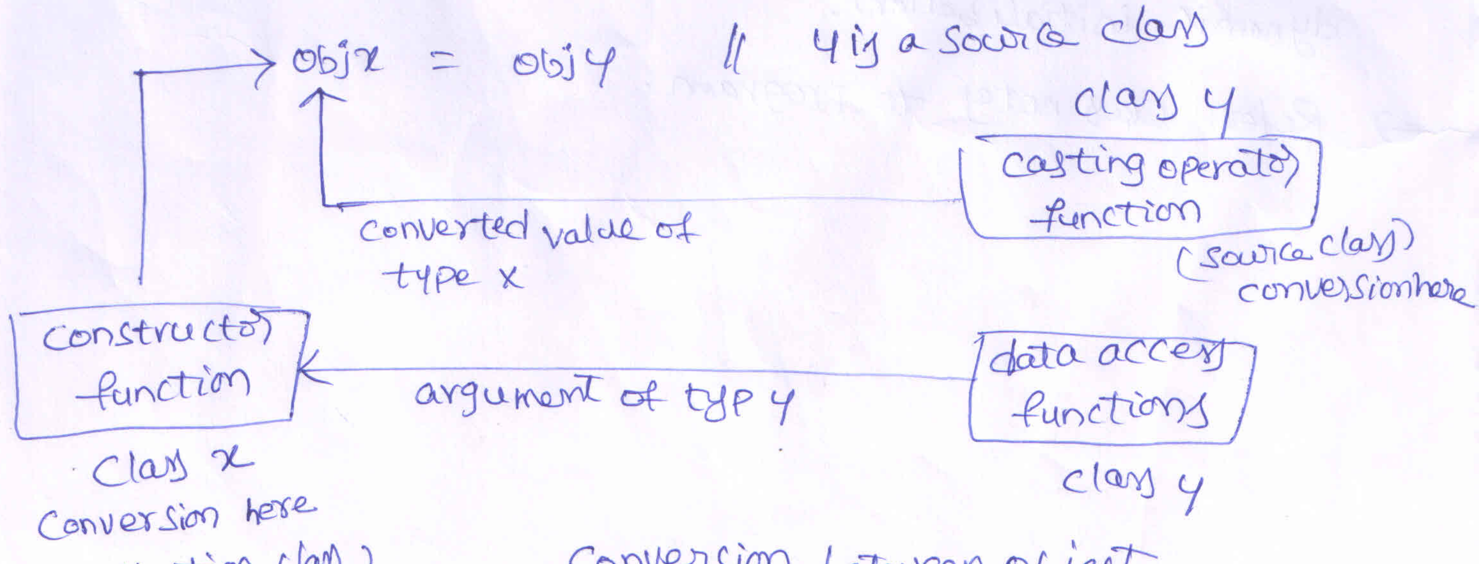
Here obj x is an object of class x and obj y is an object of class y.

The clas y type data is converted to the clas x type data
* Since the conversion takes place from clas y to clas x y is called source clas and x is called destination clas.
* such conversion between objects of different classes can be carried out by a constructor or a conversion function.

we know that the casting operator function

operator typename( )

* converts the clas object of which it is a member to typename. The type name may be a built-in type or a user defined one.
* In the case of conversions between objects, typename refers to the destination clas.
* Therefore, when a clas needs to be converted, a casting operator function can be used.
* The conversion takes place in the source clas and the result is given to the destination clas object.
* The below Table provides a summary of all 3 conversions. It shows that the conversion from a clas to any other type should make use of a casting operator in the source clas.
* on the other hand, to perform the conversion from any other type|clas to a clas type, a constructor should be used in the destination clas.



Conversion between object

Type conversions: Example program Refere C++ class notes.

	Conversion takes place in	
Conversion required	Source class	Destination class
Basic → class	Not applicable	constructor
class → basic	casting operator	Not applicable
class → class	casting operator	constructor

Dynamic initilization using constructors (or) Dynamic Initilization of objects:

→ class objects can be initialized dynamically too. i.e to say, the ~~init~~ 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 commer-cial banks.

→ The banks provide different interest rates for different schemes as well as for different periods of investment.

→ Example program to use the class variables for holding account details and how to construct those variables at run time using dynamic initialization.

→ Refer class notes to program.

Private constructor/ destructor:

→ The constructor ~~can~~ may be Private.

→ The destructors must be public.

→ A class with only non public constructor, can not be constructed directly by a user.

→ A class with only non public destructor can not be destroyed directly by a user.

Ex:
```
#include <iostream>
  using namespace std;
    class constTest
    {
    private:
          constTest ()
            {
            }
        };

      int main()
      {
       constTest t1;
      }
```

Recursive constructors:

Similar to normal and member functions constructors also support recursion.

Ex: program:
```
#include<iostream>
 using namespace std;
     class trinum
      {
        int f;
public :
        trinum()
         {
           f=0;
         }
         void  sum(int j)
           {
             f=f+j;
           }
           trinum(int m)
            {
             if (m==0)
             cout<<" Triangular num:"<<f;
             exit(1);
             }
             sum(m);
             trinum::trinum(--m);
              }
           };

                 int main()
                 {
                 trinum a;
                 a.trinum::trinum(5);
                 }
```

O/P   Triangular num : 15.

constructor and destructor with Static Member:

→ Every object has its own set of data members.

→ when a member function is invoked, only copy of data data member of calling object is available to the function

→ If the member variable is declared as static, only one copy of such member is created for entire class.

→ All objects access the same copy of static variable.

→ The static member variable can be used to count the number of objects declared for a particular class.

→ The following program helps you to count the number of objects declared for a class.

Ex: Program:

```
#include<iostream>
 using namespace std;
 class max
 {
   static int no;
   char name;
   int age;
   public:
        max()
        {
         no++;
 cout<<"\n number of objects exists:"<<no;
 }
   ~max()
    {
     --no;
  cout<<"\n number of objects exists:"<<no;
 };  }
```

```
int max::no=0;
 int main()
 {
 max A,B,C;
cout<<"\n press any key to destroy object";
 }
```

output:

```
   number of objects exists : 1
   number of objects exists : 2
   number of objects exists : 3
   press any key to destroy object
   number of objects exists : 2
   number of objects exists : 1
   number of objects exists : 0
```

Local Versus Global objects:

→ The object declared outside all function bodies is known as global object.

→ All functions can access the global object.

→ The object declared inside a function body is known as local object.

→ The scope of local object is limited to its current block.

→ when global and local variables are declared with the same name, the scope access operator is used to access the global variable in the current scope of local variable.

→ we know that the local variable gets first precedence than the global variable.

→ The same is applicable for objects.

→ when the program contains global and local objects with the same name, the local object gets first preference in its own block.

→ In this scope operator is used with global object.

Ex: Program:

```
#include<iostream>
using namespace std.
Class Sample
{
public:
    void show(char *c)
    {
    cout<<"\n"<<c;
    }
};

Sample s.   // global object declaration

int main()
{
Sample s ; // local object declaration
::s.show("Global"). // call using
                       global object
t.show( "Local"). //call using local object
}
```

output: Global
        Local.

Dynamic operator and Constructor:

→ when constructor and destructor are executed they internally use new and delete operator to alloca-te and de-allocate memory.

→ Dynamic construction means ~~releasing~~ allocation memory ~~using~~ by constructor for objects.

→ Dynamic destruction means releasing memory using the destructor.

→ Following Program that Explains the use of new and delete operator with constructor and destructor.

```
#include<iostream>
 using namespace std;
  int number();
  class num
  {
     int *x;
     int s;
 Public:
         num()
         {
         s=number();
         x = new ~~int[s]~~ int [s];
         }
         ~num()
          {
          delete x;
          }
          Void input();
          Void sum();
        };
```

```
void num:: input()
{
 for(int h=0; h<s; h++)
 {
 cout<<" Enter number["<<h+1<<"]:";
 cin>> x[h];
 }
}

void num:: sum()
{
   int adi=0;
   for(int h=0; h<s; h++)
   adi += x[h];
}  cout<< "sum of elements ="<<adi;
number()
 {
   int n;
 cout<<" How many numbers:";
  cin>>n;
 return n;
}

 int main()
 {
  num n1;
  n1. input();
  n1. sum();
}
```

output:

```
How many numbers :3
 Enternumber [1] : 1
 Enter number[2]: 4
 Enter number[3]: 5
 Sum of elements = 10
```

Overloading stream operators:

→ C++ is able to input and output the built-in data types using the stream extraction operator >> and the stream insertion operator <<.

→ The stream insertion and stream extraction operators also can be overloaded to perform input and output for user defined types like an object.

→ It is important to make operator overloading function a friend of the class because it would be called without creating an object.

Ex:

```
#include<iostream>
 using namespace std;
 class Distance
 {
  private:
          int feet;
          int inches;
    public:
          Distance()
          {
            feet=0;
            inches=0;
          }
          ~~Distance(){~~ Distance (int f, int i)
          ~~int~~      {
                       feet=f;
                       inches=i;
                     }

          friend ostream &operator<<(ostream &output, const Distance &D)
          {
        output<<"F:"<<D.feet<<"I:"<<D.inches;
        } return output;
friend istream &operator>>(istream &input, Distance &D)
  {
    input >> D.feet >> D.inches.
     return input.
   }
 }.
    int main()
    {
    Distance D1(11,10), D2(5,11). D3.
  Cout << "Enter the value of Object:" <<endl.
   Cin >> D3.
  Cout << "first Distance :" << D1 << endl.
 Cout << " Second Distance:" << D2 << endl.
 Cout << " Third Distance! " << D3 << endl.
  return 0.
 }
```

output:
```
   enter the value of object: 70
                                10
   First Distance! F : 11 I : 10
   second Distance! F : 5 I : 11
  Third Distance : F : 70 I : 10
```

One argument constructor and operator Function:

→ A single argument constructor or an operator function could be used for conversion of objects of different classes.

→ The user cannot change the in built classes. The problem occurs when a programmer attempts conversion from object of class declared by him/her to typeof inbuilt class.

→ This problem can be avoided by defining conversion routine in the userdefined class.

→ The conversion routine may be single-argument con--structor or an operator function.

→ It depends on whether the object is a source or destination.

→ Table describes conversion type and place of routine to be defined, followed by description.

Table: Conversion Types.

S.No	Conversion Type	Routine in Destination class	Routine in Source code.
A	class to class	constructor	conversion function
B	class to basic	Not applicable	conversion function
C	Basic to class	constructor	Not applicable.

a. In case both the source and destination objects are of user-defined type, the conversion routine can be carried out using operator function in source class or using constructor in destination class.

b. If the user defined object is a destination object, the conversion routine should be carried out using single argument constructor in the destination object's class.

c. In case the user defined object is a source object, the conversion routine should be carried out using an operator function in the source object's class.

→ It is used to avoid the ambiguity, if one argument constructor is present in destination class and operator function in source class, the compiler can't select appropriate routines. Hence while defining conversion routines, follow the conditions given in Table.

Overloading Binary operator using Friends:

→ Friend functions may be used in the place of member functions for overloading a binary operator.

→ The difference between the friend function and member function is friend function requires two arguments to be explicitly passed to it.

→ A member function requires only one arguments.

→ The complex ~~member function declare~~ 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

$C_3 = C_1 + C_2;$

$C_3 = \text{operator} + (C_1, C_2);$

→ 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 for the same class. This will work for a member function but the statement will not work. Because

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

The left hand operand which is responsible for invoking the member function should be an object of the same class. However friend function is used it allows both approaches.

Program:

```
#include<iostream.h>
 using namespace std;
   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).  // friend2
     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";
                 // invokes operator >> () function
 cin>>m;
 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 ~~the~~ function operator *() itself.
the operator function are explicitly passed two arguments and they are
→ The function are invoked like any other overloaded function, based on the types of its arguments.

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

-.-