Friend Function

Sometimes we need a non-member function that has full access to the private data of a class. Also, in other situations we would like many classes to share a particular function. In such situations, we define the function as a friend function with these classes. Such a function need not be a member of any of these classes.

To declare an outside function “friendly” to a class, the function declaration should be preceded by the keyword **friend** as shown below:

```
class  ABC
{
          .........
          .........
public:
          .........
          .........
       friend  return_type  xyz();
};
```

The friend function is defined elsewhere in the program like a normal C++ function. The definition of friend function does not use either the keyword **friend** or the scope operator :: .

Friend function has the following features:

- It is not in the scope of the class to which it has been declared as friend; therefore it cannot be called using the object of that class.
- It can be invoked like a normal function without the help of any object.
- Unlike member functions, it cannot access the member names directly and has to use an object name and dot operator with each member name.
- It can be declared either in the public or private sections.

- Usually, it has the objects as arguments.

Example 1: Using a friend function with a class

```
#include<iostream>
using   namespace  std;
class  sample
{
    private:
        int  x;
        int  y;
    public:
        void  setvalue(int a, int b)
            {  x = a;  y = b; }
        friend  float  mean(sample);
};

float  mean(sample  s)
{
    return  float(s.x + s.y)/2.0;
}

void   main()
{
    sample  X;
    X.setvalue(25,40);
    cout<< “Mean value = ” << mean(X) << “\n”;
}
```

The output of the above program is

```
Mean value = 32.5
```

Example 2: Using a friend function with two classes

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

```
class  ABC;          //Forward declaration
class  XYZ
{
    private:
        int   x;
    public:
        void  setvalue(int  t)  { x = t; }
        friend  void  max(XYZ, ABC);
};

class ABC
{
    private:
        int   a;
    public:
        void  setvalue(int  t) {  a = t;  }
        friend  void  max(XYZ, ABC);
};

void   max(XYZ  m,  ABC  n)
{
    if (m.x  >=  n.a)
        cout << m.x;
    else
        cout <<  n.a;
}

void  main()
{
    ABC  abc;
    abc.setvalue(10);
    XYZ   xyz;
    xyz.setvalue(20);
    max(xyz, abc);
}
```

The output of the above program is

```
    20
```

Member functions of one class can be friend functions of another class. In such cases, they are defined using the scope resolution operator as shown below:

```
    class  X
    {
         ……
         ……
         int  fun1();          // member function of  X
         ……
    };

    class  Y
    {
         ……
         ……
         friend int X :: fun1();   // fun1() of X
                                    // is friend of Y
         ……
    };
```

The function **fun1()** is a member of class X and a friend of class Y.

Friend Class

We can also declare **all** the member functions of one class as the friend functions of another class. In such cases, the class is called a **friend class**. This can be specified as follows:

```
class  Z
{
    ………
    friend  class  X;      // all member functions
                            // of X are friends to Z
};
```

Example

```
#include <iostream>
using namespace std;
class alpha
{
     private:
          int data;
     public:
          alpha() { data = 99; }
          friend class beta;   //beta is a friend class
};

class beta
{
     public:
          void func(alpha a)
               { cout << "\ndata=" << a.data; }
};

void main()
{
     alpha a;
     beta b;
     b.func(a);
     cout<< endl;
}
```

Memory management operators

an object can be created by using new and destroyed by using delete as and when required. a data object created inside a block with new will remain in existence untill it is explicitly destroyed by using delete.

the new operator can be used to create objects of any type . it takes the following general form :

pointer-variable = new data-type ;

the pointer-varible is pointer of type data-type. the new operator allocates sufficient memory to hold data object of type data-type and return the address of the object.

example

```
int *p;
float *q;
p= new int ;
q= new float;
```

- we can combine the declaration of pointers and their assignment as follows:

```
int *p = new int ;
float *q = new float ;
```

Subsequently, the statements

```
*p = 25;
*q = 7.5;
```

assign 25 to the newly created int object and 7.5 to the float object .

we can also initialize the memory using new operator.this is done as follows: `pointer-variable = new data-type(value);`

- new can be used to create a memory space for any data type including user defined types such as array,structure and class.

  the general form for one dimensional array is :

```
pointer-variable = new data-type [size];
```

for example ,the statement `int *p = new int[10];`

- when creating multi-dimensional arrays with new,the array size must be supplied .

```
array_ptr = new int[3][5][4];
array_ptr = new int[m][5][4];
```

- when data object is no longer needed,it is destroyed to release the memory space for reuse.

  the general form of its use is:

```
delete pointer-variable;
```

for example,will delete the entire array pointed to by p.

```
delete [ ] p;
```

Array of Object

we can also array of variables that are of the type class.Such variables are called *arrays of objects*

consider the following class definition:

```
class employee
{
    char name [30];
    float age ;
   public:
           void getdata(void);
           void putdata(void);
 };
```

```
employee  manager[3] ;  / /the array manager contains three
                          objects(manager)
employee  foreman[15];
employee   worker [75];
```

Example:

```
       #include <iostream>
        using namespace std;
        class employee
           {
               char name [3];
               float age ;
          public:
             void getdata(void);
             void putdata(void);
               };
         void  employee :: getdata(void)
           {
               cout<< " Enter  name :" ;
               cin>>  name ;
                cout<< " Enter age :" ;
                 cin >> age ;
            }
      void  employee :: putdata (void)
     {
           cout<< " Name : "  << name  << " \n" ;
             cout << " Age : "  <<  age  << "\n" ;
      }
     const  int size = 3  ;
      void main( )
     {
       employee manager[size] ; // Array of  managers
       for( int i = 0 ; i < size ; i++)
             {
```

```
cout << " \nDetails of manager "  << i+1 << "\n" ;
      manager[i].getdata ( );
           }
      cout<<  " \n" ;
      for( i = 0 ; i <  size ; i++)
         {
          cout << "\nManager "  << i+1 << " \n "  ;
           manager[i].putdata() ;
         }
    }
```

Objects as function arguments

example : the addition of time in the hour and minutes format

```
     #include <iostream>
      using namespace std;
      class time
       {
          int hours;
          int minutes;
          public:
              void  gettime(int h ,int m)
              { hours = h ; minutes = m ; }
              void puttime (void)
          { cout << hours << " hours and " ;
            cout << minutes << " minutes"<< " \n " ;
              }
         void sum(time , time) ;//object are arguments
     };
     void time::sum(time t1, time  t2)//t1,t2 are objects
     {   minutes = t1.minutes + t2.minutes;
```

```
        hours = minutes / 60 ;
        minutes = minutes % 60 ;
        hours = hours + t1.hours + t2.hours;
    }
void main( )
{
 time T1,T2,T3;
 T1.gettime(2,45); // get T1
 T2.gettime(3,30); // get T2
 T3.sum(T1,T2);   // T3 = T1+T2
 cout  << "T1 =  "   ; T1.puttime( ) ;// display T1
 cout  << "T2 =  "  ; T2.puttime( ) ;// display T2
 cout  << "T3 =  "   ; T3.puttime( ) ;// display T3
 }
```

Returning Objects

A function can not only receive objects as arguments but also can return them .this example illustrates how an object can be created(within a function) and returened to another function.

Example: add two complex numbers A and B to produce a third complex number C and display all the three numbers.

```
#include <iostream>
using namespace std;
class complex     // x+iy form
{
     float x ;   //real part
     float y;   // imaginary part
  public :
```

```
void input (float real , float imag)
{ x= real ; y= imag ;}
friend complex sum( complex , complex);
void show(complex);
};
complex sum ( complex c1,complex c2)
{  complex c3;             // object c3 is
created
     c3.x= c1.x + c2.x ;
     c3.y = c1.y + c2.y ;
     return(c3) ;        // return object c3
}
      void complex::show(complex  c)
      {  cout<< c.x <<"+j"<< c.y <<"\n";}

void main()
{
 complex  A,B,C ;
 A.input( 3.1, 5.65);
 B.input(2.75, 1.2) ;
 C = sum( A,B) ;                   // c = A+ B
 cout << " A = " ;  A.show(A) ;
 cout << " B = " ;  B.show(B) ;
 cout << " C = " ;  C.show(C) ;
 }
```

Operator Overloading

Operator overloading is one of the most exciting features of object-oriented programming. It can transform complex, obscure program listings into intuitively obvious ones. For example, statements like

```
d3.add_distances(d1, d2);
```

or the similar

```
d3 = d1.add_distances(d2);
```

can be changed to the much more readable

```
d3 = d1 + d2;
```

The term *operator overloading* refers to giving the normal C++ operators, such as +, *, <=, and +=, additional meanings when they are applied to user-defined data types like classes.

Defining Operator Overloading

To define an additional task to an operator, we must specify its means in relation to the class to which the operator is applied. This is done by using the ***operator** function*. The general form of an operator function is:

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

where :

return_type : is the type of value returned by the specified operation

op : is the operator being overloaded

"operator *op***"** : is the function name

Overloading Unary Operators

Unary operators act on only one operand. Examples of unary operators are the increment and decrement operators ++ and --, and the unary minus, as in -33. The unary minus when applied to an object should change the sign of each of its data items. The following example will show how to overload a unary operator.

Example :

```
#include <iostream>
using  namespace  std;
class  space
{
     private:
          int  x;
          int  y;
          int  z;
     public:
          void  getdata(int a, int b, int c);
          void  display()
           { cout << x <<" "<< y <<" "<< z <<"\n"; }

          void  operator++();   //overload increment
          void  operator-();   //overload unary minus
};

void  space :: getdata(int a, int b, int c)
{
     x = a;
     y = b;
     z = c;
}

void  space :: operator++()
{
     x++;
     y++;
     z++;
}

void space :: operator-()
```

```
{
     x = -x;
     y = -y;
     z = -z;
}
void main()
{
     space  S;
     S.getdata(10, -20, 30);
     cout<<"S : ";
     S.display();

     ++S;          // activates operator++() function

     cout<<"S : ";
     S.display();

     -S;           // activates operator-() function

     cout<<"S : ";
     S.display();
}
```

The output of the above program is

```
S : 10 -20 30
S : 11 -19 31
S : -11 19 -31
```

Note that 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.

Overloading Binary Operators

Binary operators act on two operands. Examples include +, -, *, and /. The mechanism of overloading binary operators is the same as that of unary operators. In the program below we will see how to add two complex numbers using binary operator overloading.

Example:

```
#include <iostream>
using  namespace  std;
class  complex
{
    private:
        float  x;
        float  y;
    public:
        complex(){ }
        complex(float real, float imag)
            { x = real;  y = imag; }
        complex  operator+(complex);
        void  display();
};


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

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

void main()
{
    complex  C1(2.5, 3.5), C2(1.6, 2.7), C3;

    C3 = C1 + C2;

    cout<<”C1 = ”;    C1.display();
    cout<<”C2 = ”;    C2.display();
    cout<<”C3 = ”;    C3.display();
}
```

The output of the program above is

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

Note that, 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.

Pointers to Objects

Sometimes, when we are writing the program we don't know how many objects we want to create. In such case, we can use **new** operator to create objects at run time. The **new** operator returns a pointer to an unnamed object. Thus, object pointers are useful in creating objects at run time. We can also use an object pointer to access the public members of an object.

Example

```
#include <iostream>
using  namespace  std;
class  item
{
    private:
        int  code;
        float  price;
    public:
        void  getdata(int a, float  b)
          {
            code = a;
            price = b;
          }
        void  show()
          {
            cout<< "Code  : " << code << "\n";
            cout<< "Price : " << price <<"\n";
            }
};


void main()
{
     int x;
     float  y;
     item *p = new item;
     cout<< “Input code and price for item: ”;
     cin >> x >> y;
     p->getdata(x,y);    // or (*p).getdata(x,y);
     p->show();          // or (*p).show();
}
```

this Pointer

C++ uses a unique keyword called **`this`** to represent an object that invokes a member function. **`this`** is a pointer that points to the object for which *this* function was called. The pointer **`this`** is automatically passed to a member function when it is called. It acts as an implicit argument to all the member functions. Consider the following example:

```
     class  ABC
     {
          private:
               int  a;
          public:
               void  setdata() { a=123; }
          ……
     };
```

The statement

```
     a = 123;
```

can also be expressed as

```
     this->a = 123;
```

In operator overloading, we have implicitly used **`this`** pointer. When a binary operator is overloaded using a member function, we pass only one argument to the function. The other argument is implicitly passed using the pointer **`this`**. In the previous complex number program, we had the binary **`operator+`**:

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

Note that the left-hand operands **x** and **y** are passed implicitly using this pointer. In fact these operands can be written as follows:

```
         temp.x = this->x + c.x;
         temp.y = this->y + c.y;
```

One important application of the pointer **`this`** is to return the object it points to. The statement

```
         return *this;
```

inside a member function will return the object that invoked the function. This statement is very useful when we want to compare two or more objects inside a member function and return the invoking object as a result as shown the program below.

Example:

```
#include <iostream>
#include <cstring>
using  namespace  std;
class  person
{
    private:
         char  name[20];
         float  age;
    public:
         person(char *s, float a)
         {
              strcpy(name, s);
              age = a;
         }
         person& greater(person&);
         void   display()
         {
            cout<< ”Name: ” << name << ”\n”
               << ”Age : ” << age << “\n”;
         }
};

person&  person :: greater(person& x)
     {
         if(x.age >= age)
              return  x;
         else
              return *this;
     }

void main()
{
    person P1(“John”, 37.50),
           P2(“Ahmed”, 29.0),
           P3(“Hebber”, 40.25);

    person P = P1.greater(P3);
    cout<<”Older person is: \n”;
    P.display();

    P = P1.greater(P2);
    cout<<”Older person is: \n”;
    P.display();
}
```

The output of the program above is

```
Older person is:
Name: Hebber
Age : 40.25
Older person is:
Name: John
Age : 37.5
```

Copy constructor

- It is a member function which initializes an object using another object of the same class.
- A copy constructor has the following general function prototype:
  class_name (*const* class_name&);

  A copy constructor takes a reference to an object of the same class as it self as an argument.we can not pass the argument by value to copy constructor.
- In the absence of a copy constructor, the C++ compiler builds a default copy constructor for each class which is doing a member-wise copy between objects.
- The default copy constructor will not work well if the class contains pointer data members ...

Example:

```
#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 ;
  }
 };

void main( )
{
      code A(100);
      code B(A) ;
```

```
            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( );
```

✓ What is the main difference between the copy constructor and the assignment operator?

- The copy constructor creates a new object.
- The assignment operator works on an already valid object.