Lesson 6: Tools For Defining Abstract Data Types

Objectives

After completing this chapter you will understand about:

- Defining Abstract Data Type Operations on objects as non-member functions, i.e., friend functions
- Overloading different operators.
- How to compile the Abstract data types into libraries, so that it is used in the same way as predefined libraries.

Structure Of The Lesson

6.1 Defining ADT Operations

Till now ADT operations have been implemented as member functions of the class. For some operations it is better to implement it as ordinary functions. We define operations on objects as non-member functions. Here is a simple example.

Program to explain the accessor functions.

```
#include<iostream.h>
class dayofyear
      {
        public :
          dayofyear();              //constructor
          dayofyear(int m,int d); //constructor
          void read();
          void print();
int getmonth();          //accessor function
int getday();            //accessor function
       private :
         int month,day;
      };

dayofyear :: dayofyear()
 {
    month =1;
    day =1;
 }
dayofyear :: dayofyear(int m, int d)
 {
    month= m;
    day= d;
 }

void dayofyear ::read()
 {
   cout<< " Enter month and day";
   cin>>month>>day;
}
void dayofyear::print()
  {
     cout<<day<<'-'<<month;
}
int dayofyear::getmonth( )
```

```
{
    return month;
}

int dayofyear::getday( )
{
    return day;
}

//nonmember function
int equal (dayofyear d1, dayofyear d2)
 {
    if((d1.getmonth() == d2.getmonth()) && (d1.getday() ==
    d2.getday()))
    return 1;
else
   return 0;
}
int main( )
{
    dayofyear today, bach_birthday(12, 3);
    cout << "Enter today's date:\n";
    today.read();
    cout << "Today's date is ";
    today.print();
    cout << "Nirupama'sbirthday is ";
    bach_birthday.print( );
    if ( equal(today, bach_birthday))
        cout << "Happy Birthday \n";
    else
        cout << "Good day\n";
    return 0;
}
```

output:

```
Enter month and day8
3
Today's date is 3-8Nirupama'sbirthday is 3-12Good day
```

Here equal is not a member function. But, using the accessor functions, it compares the objects of type day of year. If “equal” is a member function, one of the dates are taken as calling object. To treat both the dates in the same way, it is better to make “equal “ an ordinary non-member function.

6.1.1 Friend Functions

A friend function of a class is a non-member function, which has access to private members of objects of that class. To make a function a friend of a class, the function prototype must be listed in the class definition. The prototype is preceded by the keyword "**friend**". Prototype may be placed in either private section or public section.

Syntax of a class definition with friend functions:

```
class class_name
{
 public :
          Member function declarations;
          friend prototype-for-friend-function1;
          friend prototype-for-friend-function2;
          ----------
          ----------
  private :
        Declaration of private members.
};

Eg :
        class dayofyear
        {
        public :
            friend int equal dayofyear d1, dayofyear d2;

private :
           int month, day;
        };

        int equal (dayofyear d1, dayofyear d2)
        {
          return( d1.month = = d2.month && d1.day = = d2.day);
        }
```

A friend function is not a member function. It is defined and called the same way as an ordinary function.

6.1.2 Difference Between Friend Functions And Member Functions

- A member function is defined with scope resolution operator and type qualifier whereas a friend function is defined like an ordinary function.
- Both function prototypes are written in the class definition but the keyword friend appears before the prototype declaration of friend function.
- A member function is always called with reference to an object whereas a friend function is called like a normal function.

  Note: Member functions are used if the function involves one argument.

  Eg: r3.add(r1,r2)
  r1.add(r2)

- Non-member functions are used if the function involves more than one argument of the same class type or different class type.

Program to explain the friend functions.

```
#include<iostream.h>
class dayofyear
      {
        public :
          dayofyear();
          dayofyear(int m,int d);
          void read();
          void print();
          friend int equal (dayofyear d1, dayofyear d2);
        private :
          int month,day;
      };

dayofyear :: dayofyear()
 {
     month = 1;
     day = 1;
}
```

```
dayofyear :: dayofyear(int m,int d)
{
    month= m;
     day = d;
 }
void dayofyear ::read()
 {
    cout<< " Enter month and day";
    cin>>month>>day;
}
void dayofyear::print()
  {
      cout<<day<<'-'<<month;
}
int equal (dayofyear d1, dayofyear d2)
 {
   if((d1.month == d2.month) && (d1.day == d2.day))
    return 1;
else
   return 0;
}
void main()
{
  dayofyear d1(10,20),d2(10,12),d3(9,5),d4(9,5);
if(equal (d1,d2))
  cout << "d1 & d2 are equal\n";
else
 cout<<"d1 and d2 are not equal\n ";
if (equal(d3,d4))
  cout << "d3 or d4 are equal\n";
else
 cout << "d3 and d4  are not equal\n";
}
```

Output:

```
d1 and d2 are not equal
 d3 or d4 are equal
```

A simple rule to use the member function or non-member function is

- To use a member function if the involved task uses only one object.
- To use a non-member function if the involved task uses two or more objects.

6.1.3 Constant Parameter Modifier

A "**const**" modifier is used with call by reference parameter within the function header. A reference parameter can be modified by a function. By using constant modifier, the reference parameter can be prevented from accidental modifications.

Syntax:

```
Returntype memberfunctionname
(const datatype1& s1,const datatype2& s2,…);
```

e.g. :

```
int equal (const dayofyear& d1, const dayofyear& d2)
 {
if((d1.month = = d2.month) && (d1.day = = d2.day)
return1;
else
return 0; }
```

Thus any modifications to the data in the objects can be prevented.

6.1.4 Constant Member Functions

We can declare a member function of class as a constant member function. This is done by writing "**const**" keyword at the end of the prototype and function header in the function definition. A constant member function does not change an objects data. Such a function may be an accessor function or it may simply print function.

Syntax:

```
//Class definition
class  Class_name
        {
          public :
             return_type member_function_name (parameter list)
const;
                    --------
         };
```

```
//Member function definition
        return_type class_name ::Function_name
        (parameter_list)const

        {
        function body
        }

        eg: class dayofyear
        {
        public :
      void print() const;
       --------
       --------
        };
void dayofyear :: print() const
 {
     cout << day << ‘  ‘ << month;
 }
```

6.1.5 Operator Overloading

Operator overloading is a mechanism of redefining the meaning of C++ operators, i.e., making an operator to behave differently at different instances. They can be member functions or friend functions**.** An operator such as +,-,*,/ etc is used on predefined data. In one way these operators are also functions. As functions can be overloaded, operators can also be overloaded. They can be directly applied on different objects. An operator definition is written in the same way as a function definition, except that the operation definition includes the reserved word "**operator**", before the operator name. The predefined operators such as +, -, etc. can be overloaded by giving them a new definition for a class type.

Syntax:

```
Return type class name :: operator symbol (Parameters list)
{
        body of the function

}
```

Eg: rno rno::operator + (rno r2);

To overload a + operator with a member function one argument has to be passed into the function and to overload “+” with a friend function, two arguments must be passed. The arguments that are passed can be reference arguments or value arguments.

Note: During operator overloading, the arguments are listed before and after the operator in the function call.

Eg: v3 = v1+v2;

With a function the arguments are listed after the function name.

v3= add(v1,v2);

Thus, the predefined operators can be overloading by giving them a new definition for the class type or object.

6.1.6 Rules For Operator Overloading

1. When overloading an operator at least one argument of the resulting overloaded operator must be class type.

2. An overloaded operator can be a friend of the class, but it is not compulsory. It can be a member of the class.

3. A new operator cannot be created, but an existing operator can be overloaded.

4. The number of arguments that an operator takes cannot be changed. For example: % is a binary operator. It cannot be overloaded into a unary operator and vice versa.

5. Precedence of operators cannot be changed.

The following operators can be overloaded.

.	class member operator
::	scope resolution operator
?:	conditional operator
.*	de-referenced member access
sizeof	size of operator

Program to apply arithmetic and relational operations by operator overloading

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

class ratio
{
    int p,q;
    public:
      ratio()
      {
        p=q=1;
      }
      ratio(int a,int b)
      {
        p=a;
        q=b;
        if(q<=0)
        {
            cout<<"improper argument";
            exit(0);
        }
        int m=p<q?p:q;
        for(int i=m;i>1;i--)
        {
            if((p%i==0) && (q%i==0))
            {
                p/=i;
                q/=i;
                break;
            }
        }
      }
      void get()
      {
        cout<<"enter input";
          cin>>p>>q;
        int m=p<q?p:q;

        for(int i=m;i>1;i--)
        if((p%i==0) && (q%i==0))
        {
            p/=i;
            q/=i;
```

```
            break;
          }
        }
        void put()
        {
          cout<<p<<"/"<<q<<"\n";
        }
        void put1()
        {
         cout<<p<<"/"<<q;
        }
friend ratio operator+(ratio &,ratio &);
friend ratio operator-(ratio &,ratio &);
friend ratio operator*(ratio &,ratio &);
friend ratio operator/(ratio &,ratio &);
void operator<(ratio &);
void operator<=(ratio &);
void operator>(ratio &);
void operator>=(ratio &);
void operator==(ratio &);
void operator!=(ratio &);
};
ratio operator+(ratio &r1,ratio &r2)
 {
        ratio temp;
        temp.p=(r1.p*r2.q)+(r2.p*r1.q);
        temp.q=r1.q*r2.q;
        int min=temp.p<temp.q?temp.p:temp.q;
        for(int i=min;i>1;i--)
        if((temp.p%i==0)&&(temp.q%i==0))
        {
            temp.p/=i;
            temp.q/=i;
            break;
        }
        return temp;
}
ratio operator-(ratio &r1,ratio &r2)
{
        ratio temp;
        temp.p=(r1.p*r2.q)-(r2.p*r1.q);
        temp.q=r1.q*r2.q;
        int min=temp.p<temp.q?temp.p:temp.q;
        for(int i=min;i>1;i--)
        if((temp.p%i==0)&&(temp.q%i==0))
```

```
        {
            temp.p/=i;
            temp.q/=i;
            break;
        }
        return temp;
}
ratio operator*(ratio &r1,ratio &r2)
{
         ratio temp;
        temp.p=r1.p*r2.p;
        temp.q=r1.q*r2.q;
        int min=temp.p<temp.q?temp.p:temp.q;
        for(int i=min;i>1;i--)
        {
          if((temp.p%i==0)&&(temp.q%i==0))
          {
            temp.p/=i;
            temp.q/=i;
            break;
          }
        }
        return temp;
}
ratio operator/(ratio &r1,ratio &r2)
{
        ratio temp;
        temp.p=r1.p*r2.q;
        temp.q=r2.p*r1.q;
        int min=temp.p<temp.q?temp.p:temp.q;
        for(int i=min;i>1;i--)
        {
          if((temp.p%i==0)&&(temp.q%i==0))
          {
            temp.p/=i;
            temp.q/=i;
            break;
          }
        }
        return temp;
}
void ratio::operator<(ratio &r)
{
        float f1,f2;
        f1=float(p)/float(q);
```

```
        f2=float(r.p)/float(r.q);
        if(f1<f2)
        {
            put1();
            cout<<"<";
            r.put1();
        }
        else
        {
            r.put1();
            cout<<"<";
            put1();
        }
}
void ratio::operator<=(ratio & r)
{
        float f1,f2;
        f1=(float)p/(float)q;
        f2=(float)r.p/(float)r.q;
        if(f1<=f2)
        {
            put1();
            cout<<"<=";
            r.put1();
        }
        else
   {
            r.put1();
            cout<<"<=";
            put1();
        }
}
void ratio::operator>(ratio & r)
{
        float f1,f2;
        f1=(float)p/(float)q;
        f2=(float)r.p/(float)r.q;
        if(f1>f2)
        {
            put1();
            cout<<">";
            r.put1();
        }
        else
        {
```

```
            r.put1();
            cout<<">";
            put1();
        }
}

void ratio::operator>=(ratio & r)
{
        float f1,f2;
        f1=(float)p/(float)q;
        f2=(float)r.p/(float)r.q;
        if(f1>=f2)
                    {
                                put1();
            cout<<">=";
            r.put1();
        }
        else
        {
            r.put1();
            cout<<">=";
            put1();
        }
}
void ratio::operator==(ratio & r)
{
        float f1,f2;
        f1=(float)p/(float)q;
        f2=(float)r.p/(float)r.q;
        if(f1==f2)
        {
            put1();
            cout<<"==";
            r.put1();
        }
        else
        {
            r.put1();
            cout<<"!=";
            put1();
        }
}
void ratio::operator!=(ratio & r)
{
```

```
        float f1,f2;
        f1=(float)p/(float)q;
        f2=(float)r.p/(float)r.q;
        if(f1!=f2)
        {
            put1();
            cout<<"!=";
            r.put1();
        }
        else
        {
            r.put1();
            cout<<"!=";
            put1();
        }
}
int main()
{
        ratio r1(25,15),r2,r3(1,1),r4(5,3);
                    r2.get();
        cout<<"r1=";
        r1.put();
        cout<<"r2=";
        r2.put();
        cout<<"r3=";
        r3.put();
        cout<<"r4=";
        r4.put();
        cout<<"addition of r1 r2 is ";
        r3=r1+r2;
        r3.put();
         cout<<"subtraction of r1 r2 is ";
        r3=r1-r2;
        r3.put();
        cout<<"multiplication of r1 r2 is ";
        r3=r1*r2;
        r3.put();
        cout<<"division of r1  r2 is ";
        r3=r1/r2;
        r3.put();
        r1<r2;
        cout<<"\n";
        r1>=r4;
        cout<<"\n";
        r2>r4;
```

```
        cout<<"\n";
        r1==r2;
        cout<<"\n";
        r1<=r2;
        cout<<"\n";
        r1==r4;
        return 0;
}
```

output:

```
        enter input2
        3
        r1=5/3
        r2=2/3
        r3=1/1
        r4=5/3
        addition of r1 r2 is 7/3
        subtraction of r1 r2 is 1/1
        multiplication of r1 r2 is 10/9
        division of r1  r2 is 5/2
        2/3<5/3
        5/3>=5/3
        5/3>2/3
        2/3!=5/3
        2/3<=5/3
        5/3==5/3
```

6.1.7 Overloading Unary Operators

Unary and Binary operators can be overloaded. We have seen the overloading of binary operators in the above example of overloading. Thus unary – (negation) operator, prefix ++,- - can be overloaded. There is a different method to overload the postfix ++ and --.

```
#include<iostream.h>
class space
{
int x;
int y;
int z;
public:
space(int a1,int b1, int c1)
{x = a1; y = b1; z = c1;}
```

```
void operator -()
{
x = -x;
y= -y;
z =-z;
} void operator --()
{
--x;
--y;
--z;
}  void operator -=(space& s)
{
x-= s.x;
y-= s.y;
z-= s.z;
}
void putdata()
{
cout<<"\nx:"<<x<<"\ty:"<<y<<"\tz:"<<z;
}  };
int main()
{
space s1(2,-1,3),s2(1,1,1);
cout<<"\nData in s1:";s1.putdata();
cout<<"\nData in s2:";s2.putdata();
-s1;
cout<< "\nNegation of s1:";s1.putdata();
--s1;
cout<<"\nPredecrement  s1:";s1.putdata();
s1-=s2;
cout<<"\ns1-=s2";s1.putdata();
return 0;
}
```

output:

```
Data in s1:
x:2     y:-1    z:3
Data in s2:
x:1     y:1     z:1
Negation of s1:
x:-2    y:1     z:-3
Predecrement  s1:
x:-3    y:0     z:-4
s1-=s2
x:-4    y:-1    z:-5
```

6.1.8 Overloading Of Insertion And Extraction Operators

The insertion, << and extraction >> operators can be overloaded as any other operators. The value returned must be stream. The type of value returned must have the & symbol added to the end of the type name. The prototype and function definition is as follows:

Prototypes:

```
class  classname
{
public:
        . . . . .
         . . . . .
friend istream& operator >>(istream& parameter1,
                                    classname&  parameter2);
friend ostream&operator<<(ostream& parameter3,
                          classname & parameter4);
. . . . .
private:
. . . . . .
 . . . . . .
}

#include<iostream.h>
class space
{
int x;
int y;
int z;
public:
space(int a1,int b1, int c1)
{x = a1; y = b1; z = c1;}
void operator -()
{
x = -x;
y= -y;
z =-z;
}
void operator --()
{
--x;
--y;
```

```
--z;
}
void operator -=(space& s)
{
x-= s.x;
y-= s.y;
z-= s.z;
}
friend istream& operator >>(istream &din, space&);
friend ostream& operator <<(ostream &dout,space&);
};
istream& operator >>(istream &din, space& s)
{
din>>s.x>>s.y>>s.z;
return din;
}
ostream& operator <<(ostream &dout, space& s)
{
dout<<"\nx:"<<s.x<<"\ty:"<<s.y<<"\tz:"<<s.z;
return dout;
}
int main()
{
space s1(2,-1,3),s2(1,1,1);
cout<<"\nData in s1:"<<s1;
cout<<"\nData in s2:"<<s2;
-s1;
cout<< "\nNegation of s1:"<<s1;
--s1;
cout<<"\nPredecrement of s1:"<<s1;
s1-=s2;
cout<<"\ns1-=s2"<<s1;
return 0;
}
```

output:

```
Data in s1:
x:2    y:-1    z:3
Data in s2:
x:1    y:1    z:1
Negation of s1:
x:-2    y:1    z:-3
Predecrement of s1:
x:-3    y:0    z:-4
s1-=s2
x:-4    y:-1    z:-5
```

6.2 Abstract Data Types

A **Data type** consists of a collection of values together with a set of basic operations defined on these values . If the programmer who uses the data type does not have access to the details of how the values and operations are implemented, such data type is called an abstract data type.

To define a class as an abstract data type, the specification details of the class type used by the programmer is separated from the details of implementation. In order to do this,

- All the member variables of the class are made private.
- All basic operations used by the programmer are made public member functions and their usage is completely specified.
- Any helping functions are made private.

The **interface** of an ADT tells how to use the ADT in the program. It consists of public member functions of the class along with comments how to use them. The interface is to be known to use an abstract data type.

The **implementation** of the ADT tells how this interface is realized as C++ code. It consists of private members of the class and the definitions of both public and private member functions.

6.2.1 Separate Compilation

An ADT is defined as a class and the definition and implementation of its member functions are put in separate files.

- The definition of the class is put in a header file called the interface file. The name of the file ends with .h. The interface file also contains the prototypes for the functions and overloaded operators that define basic ADT operations.

- The definitions of all functions and overloaded operators mentioned in the interface file are placed in another file called implementation file. This file must include the directive that names the interface file. The interface and implementation files have the same filename, but end with different suffixes.

- The main part of the program is placed in another file called an application file. This also must include the naming directive of the interface file. The application file must be compiled separately from the implementation file.

The object code produced by compiling the application file and object code produced by compiling the implementation file must be linked and executed.

Program using separate compilation

```
/*This program uses operator overloading of =,--(pre decrement and post decrement), < , <<,>> operators*/
//dist.h(header file) interface file
#ifndef DIST_H
#define DIST_H
enum bool{false,true};
class distance
{
        int feet;
        float inch;
        public:
        distance();
        distance( int f, float i);
        distance operator + (distance &d);
        void operator --();
        void operator --(int i);
        friend bool operator <(distance d1,distance d2);
        friend istream &operator>>(istream &ip, distance &d);
        friend ostream &operator<<(ostream &op,distance &d);
};

#endif
//dist.cpp implementation file
#include<iostream.h>
#include "dist.h"
distance::distance()
{
        feet = 0;
        inch =0;
```

```
}
distance::distance( int f, float i)
{
        feet = f;
        inch = i;
}
distance distance::operator + (distance &d)
{
        distance temp;
        temp .inch = inch + d.inch;
        temp.feet = feet + d.feet;
        while (temp.inch >= 12.0)
        {
        temp.inch -=12.0;
        temp.feet +=1;
        }
        return temp;
}
void distance::operator --()
{
        --feet;
}
void distance::operator --(int )
{
        feet--;
}

bool operator <(distance d1,distance d2)
{
        if (d1.feet < d2.feet)
        return true;
        else
        if (d1.feet > d2.feet)
        return false;
        else
        if(d1.inch < d2.inch)
        return true;
        else
        return false;
}
istream &operator>>(istream &ip, distance &d)
{
        ip>>d.feet>>d.inch;
        return ip;
}
```

```
ostream &operator<<(ostream &op,distance &d)
{
        op<<d.feet<<"\'-"<<d.inch<<"\"";
        return op;
}

//distapp.cpp   application file

#include<iostream.h>
#include"dist.h"
#include "dist.cpp"
int main()
{
        distance d1(3,9),d2,d3;
        cout<<"\nEnter distance 2:";
        cin>>d2;
        cout<<"\nFirst distance:"<<d1;
        cout<<"\nSecond distance:"<<d2;
        d3 = d1 + d2;
        cout<<"\nSum of two distances:"<<d3<<endl;
        return 0;
}
```

Output:

```
Enter distance 2:3
4

First distance:3'-9"
Second distance:3'-4"
Sum of two distances:7'-1"
```

6.3 Summary

- We have covered different tools for defining the class as an Abstract Data Types. In this, we have studied how to write friend functions and also the operator overloading of unary binary and extraction and insertion operators .
- We have also studied about the constant parameter modifiers in functions.
- We have also reviewed an Abstract data type and how to convert a class as an Abstract data type. We have also covered the separate compilation. The definition of the class and implementation of its

functions are placed in separate files. This is compiled separately and can be used in any application.

6.4 Technical Terms

Accessory Function: Member functions that give access to the private members of the class.

Friend Function: A function although not a member of a class is able to access the private members of that class.

Overloading Operator: : A language feature that allows an operator to be given more than one definition. The types of arguments with which the operator is called, determines which definition is used.

6.5 Model Questions

1. Explain the different tools for defining Abstract data types with Examples.
2. What is a friend function? How is it different from the member function?
3. What is operator overloading? Explain with an example the overloading of a binary operator.
4. Explain the overloading of << and >> operators with an example?
5. Define an Abstract Data type class in separate files.

6.6 References

Object-oriented programming with C++,
by E. Bala Gurusamy.

Problem solving with C++
by Walter Savitch

Mastering C++
by K.R.Venugopal, RajkumarBuyya, T.RaviShankar

AUTHOR

M. NIRUPAMA BHAT, MCA., M.Phil.,
Lecturer,
Dept. Of Computer Science,
JKC College,
GUNTUR.