CHAPTER 4

OPERATOR OVERLOADING AND TYPE CONVERSIONS

Structure of the Unit

- Operator Overloading
- Defining Operator Overloading
- Overloading Binary Operator
- Overloading Unary Operator
- Overloading Prefix and Postfix Increment/Decrement Operators
- Operator Overloading Using Friend Functions
- Type Conversions
- Implicit Type Conversion
- Explicit Type Conversion
- Basic to Class Type
- Class to Basic Type
- One Class to another Class Type

4.1 Introduction

Operator overloading refers giving special meaning to an existing operator. Operator overloading is similar to function overloading. In fact, operator overloading is really just a type of function overloading. However, some additional rules apply. This chapter introduces you to operator overloading that deals with overloading of both unary and binary operators. The second part of the chapter provides necessary details regarding type conversions or type casting. Type conversion or typecasting refers to changing an entity of one data type into another. This chapter focuses on implicit and explicit type conversions.

4.2 Learning Objectives

- To introduce the concept of Operator overloading
- To define operator overloading function
- To understand the steps involved in overloading binary operators
- To understand the steps involved in overloading unary operators
- To show how to overload unary prefix and postfix operators

- To discuss overloading operators using friend functions
- To present the concept of Type casting
- To discuss implicit and explicit type casting.

4.3 Operator Overloading

As mentioned earlier operator overloading refers giving special meaning to an existing operator. When an operator is overloaded, that operator loses none of its original meaning. Instead, it gains additional meaning relative to the class for which it is defined. An operator is always overloaded relatively to a user defined type, such as a class. To overload an operator, you create an *operator function*. Most often an operator function is a member or a friend of the class for which it is defined.

Almost all operators in C++ can be overloaded, except the following few operators.

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

4.3.1 Defining Operator Overloading

The general form of a member operator function is shown here:

return-type class-name::**operator#**(*arg-list*)
{
    // *operation to be performed*
}

- The return type of an operator function is often the class for which it is defined (however, operator function is free to return any type).
- The operator being overloaded is substituted for **#.** For example, if the operator **+** is being overloaded, the operator function name would be **operator +.**
- The contents of ***arg-list*** vary depending upon how the operator function is implemented and the type of operator being overloaded.

There are two important restrictions to remember when you are overloading an operator:

- The precedence of the operator cannot be change.
- The number of operands that an operator takes cannot be altered.

4.3.2 Overloading Binary Operator

When a member operator function overloads a binary operator, the function will have only one parameter. This parameter will receive the object that is on the right side of the

operator. The object on the left side is the object that generates the call to the operator function and is passed implicitly by **this pointer**.

Example 1:

The following program overloads the "+" operator such that it adds two objects of the type complex.

```
#include <iostream.h>
class complex
{
        int real;
        float imag;
        public:
                complex()
                {
                        real=imag=0;
                 }
                complex (int r,float i)
                {
                          real = r; imag=i;
                }
                //overload + operator for complex class
                complex operator +(complex);
                void display()
                {
                        cout<<"The Result is ";
                        cout<<real << " +j " <<imag ;
                }
};

complex  complex :: operator + (complex c1)
{
        complex temp;
        temp.real=real +c1.real;
        temp.imag=imag + c1.imag;
        return temp;
}

void main()
{
        complex c1,c2,c3;
        int a; float b;
        cout<<"ENTER REAL AND IMAG PART OF COMPLEX NUMBER1"<<endl;
        cin>>a>>b;
        c1=complex(a,b);
```

```
        cout<<"ENTER REAL AND IMAG PART OF COMPLEX NUMBER2"<<endl;
        cin>>a>>b;
        c2=complex(a,b);
        //adds two complex objects invokes operator + (complex) function
        c3=c1 + c2 ;
        c3.display();
}
```

Output of the Program

```
ENTER REAL AND IMAG PART OF COMPLEX NUMBER1

    5       7.1
ENTER REAL AND IMAG PART OF COMPLEX NUMBER2

    3       8.3
The Result is 8 + j 15.3
```

Example 2:

The program given below is the modified version of the example 10 program in Chapter3. The member function add() present in that program is replaced by the operator overloading function.

```
#include<iostream.h>
class money
{
        int rs;
        int ps;
        money()
        {
                rs=0;
                ps=0;
        }
        money(int r,int p)
        {
                rs=r;
                ps=p;
        }
        money operator +(money);
        void display();
}
money money :: operator +(money m1)
{
        money temp;
```

```
        temp.ps=m1.ps+ps;
        if(temp.ps>99)
        {
                temp.rs++;
                temp.ps-=99
        }
        temp.rs+=m1.rs+rs;
        return temp;
}
void money :: display()
{
        cout<<"Total Rupees   "<<rs;
        cout<<"\n Total Paise   "<<ps;
}
void main()
{
        money m1(10,55);
        money m2(11,55);
        money m3;
        m3=m1+m2;
        m3.display();
}
```

Output of the Program

```
Total Rupees  22
Total Paise   10
```

4.3.3 Overloading Unary Operator

Overloading a unary operator is similar to overloading a binary operator except that there is one operand to deal with. When you overload a unary operator using a member function, the function has no parameters. Since, there is only one operand, it is this operand that generates the call to the operator function. There is no need for another parameter.

Example 3:

The following program overloads the unary – operator that negates the object of point class

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

```
        int x, y;
public:
        point( )
        {
                x = 0; y = 0;
        }
        point(int i, int j)
        {
                x = i; y = j;
        }
        point operator-( );
        void display()
        {
                cout<<x <<"\t"<<y;
        }
};
// Overload - operator for point class
point point::operator-( )
{
        x=-x;
        y=-y;
}
int main( )
 {
        point o1(10, 10);
        int x, y;
        -o1; //Negate the object
        o1.display();
}
```

Output of the Program

```
-10      -10
```

4.3.4 Overloading Prefix and Postfix Increment/Decrement Operators

When overloading the prefix ++ operator you write the member function as

```
void operator ++( )
```

Similarly when overloading the postfix ++ operator we will write the member function as

```
void operator ++( )
```

Hence there is an ambiguity because both the member function takes the same form the same problem exist for the pre and post fix decrement operators. To resolve this you have to pass a dummy integer argument for the postfix operator.

```
void operator ++( );            //UNARY PREFIX INCREMENT
void operator -- ( ) ;          //UNARY PREFIX DECREMENT
void operator ++(int);          //UNARY POSTFIX INCREMENT
void operator -- ( int ) ;      //UNARY POSTFIX DECREMENT
```

Note:

The int parameter in the postfix operator is a dummy variable

Example 4:

The program given below overloads unary prefix ++ operator to increment the object of type point class

```
#include <iostream.h>
class point
{
        int x, y;
        public:
                point( )
                {
                        x = 0; y = 0;
                }
                point(int i, int j)
                 {
                        x = i; y = j;
                 }
                point operator++( );
                void display()
                {
                        cout<<"The Object is Incremented to \n";
                        cout<<x<<"\n";
                        cout<<y;
                }
};

point point::operator++( )  // Overload prefix ++ operator for point class
 {
        ++x;
```

```
        ++y;
}
int main( )
{
        point o1(10, 10);
        ++o1; //increment an object
        o1.display();
}
```

Output of the Program

The Object is Incremented to
11 11

Example 5:

The program given below overloads unary postfix -- operator to decrement the object of type point class

```
#include <iostream.h>
class point
{
        int x, y;
        public:
                point( )
                {
                        x = 0; y = 0;
                }
                point(int i, int j)
                {
                         x = i; y = j;
                }
                point operator--( int );
                void display()
                {
                        cout<<"The Object is Decremented to \n";
                        cout<<x<<"\t";
                        cout<<y;
                }
};
point point::operator--( int a)  // Overload postfix -- operator for point class
 {
        x--;
        y--;
}
```

```
int main( )
{
        point o1(10, 10);
        o1--; //decrement an object
        o1.display();
}
```

Output of the Program

```
The Object is Decremented to
9        9
```

4.3.5 Operator Overloading Using Friend Functions

It is possible to overload an operator relative to a class by using a friend rather than a member function. A friend function does not have a **this** pointer. In the case of a binary operator, this means that a friend operator function is passed both operands explicitly. For unary operators, the single operand is passed.

Note:

We cannot use a friend to overload the assignment operator. The assignment operator can be overloaded only by a member operator function.

Example 6:

The following program overloads + operator using friend function.

```
#include <iostream.h>
class point
{
        int x, y;
        public:
                point( )
                {
                        x = 0; y = 0;
                }
                point(int i, int j)
                {
                        x = i; y = j;
                }
```

```
                void display()
                {
                   cout << “X IS :  " << x << ", Y IS : " << y <<"\n";
                }
                friend point operator+(point ob1, point ob2);
};
// Overload + using a friend.
point operator+(point ob1, point ob2)
 {
        point temp;
        temp.x = ob1.x + ob2.x;
        temp.y = ob1.y + ob2.y;
        return temp;
}
int main( )
{
        point o1(10, 10), o2(5, 3), o3;
        o3 = o1 + o2; //add to objects
        // this calls operator+( )
        o3.display();
        return 0;
}
```

Output of the Program

X IS : 15 ,Y IS : 13

Note that the left operand is passed to the first parameter and the right operand is passed to the second parameter

Have you Understood Questions?

1. When we overload a operator will its meaning change ?
2. Give the general form of operator function
3. Write examples for unary and binary operators
4. Which operator cannot overloaded using friend function ?

4.4 TYPE CONVERSIONS

Definition

Type conversion or typecasting refers to changing an entity of one data type into another. This is done to take advantage of certain features of type hierarchies. For instance, values from a more limited set, such as integers, can be stored in a more compact format and later converted to a different format enabling operations not previously possible, such as division with several decimal places worth of accuracy. Type conversion allows programs to treat objects of one type as one of their ancestor types to simplify interacting with them.

4.5 Implicit Type Conversion

Implicit type conversion is an automatic type conversion by the compiler. The type conversions are automatic as long as the data types involved are built-in types.

Example

```
int y;
float x=123.45;
y = x;
```

In this example the float variable x is automatically gets converted to int. Thus the fractional part of y gets truncated.

4.6 Explicit Type Conversion

Automatic type conversion for user defined data types is not supported by the compiler hence the conversion routines are ought to be specified explicitly. Three types of situations might arise in the data conversion between incompatible types.

- Conversion from basic type to class type.
- Conversion from class type to basic type
- Conversion from one class type to another class type.

4.6.1 Basic to Class Type

This is done by using constructors in the respective classes. In these types the '=' operator is overloaded.

Example 7:

The following program converts integer to the class type "length"

```
class length
{
        int cm,m;
 public:
         length(int n)          //Constructor
         {
                m=n/100;
                cm=m%100;
         }
}
void main()
{
        int n;
        cin >> n;

        length obj = n;         //Integer to class type
}
```

The constructor used for type conversions take a single argument whose type is to be converted.

4.6.2 Class to Basic Type

Overloaded casting operator is used to convert data from a class type to a basic data type.

Syntax:

```
operator typename()
{
        statements.....
}
```

This function converts a class type to specified type name. For example **operator int()** converts the class object to integer data type.

Conditions for a casting operator function

- It must be a class member.
- It must not specify a return type.
- It must not have any arguments.

Example 8:

The following example cast the object of type “time” to basic data types (int and float)

```
class time
{
        int min,sec;
 public:

         time(int n)
         {
                 min = n/60;
                 sec=n%60;
         }


         operator int()  //Casting Operator
         {
                 int x;
                 x= min * 60 + sec;
                 return sec;
         }
         operator float() //Casting Operator
         {
                 float y;
                 y = min + sec/60;
                 return y;
         }
};
void main()
{
         time t1(160);
         int n = t1;       //Conversion
         float x = t1;     //Conversion
}
```

4.6.3 One Class to another Class Type

The Constructors helped us to define the conversion from a basic data type to class type and the overloaded casting operator helped to define the conversion from a class type to basic data type. Now to convert from one class type to another class type these both help us to do.

Example 9:

The following program converts the class type length1 to type length2

```
class length2; //forward declaration
class length1
{
```

```
        int m,cm;
        public:
                length1(int n)
                {
                        m=n/100;
                        cm=n%100;
                }
                operator length2() //from length1 type to length2 type
                {
                        int x= m*100 + cm;
                        length2 tempobj(x);
                        return tempobj;
                }
}
class length2
{
        int cm;
        public:
                length2(int n)
                {
                        cm=n;
                }
                operator length1() //from length2 type to length1 type
                {
                        length1 tempobj(cm);
                        return tempobj;
                }
        }
void main()
{
        int x= 125;
        length2 obj1(x);
        length1 obj2= obj1;
}
```

Have you Understood Questions?

1. What is type casting?
2. Which operator you have to overload to convert a basic data type to user defined data type

Summary

- Operator overloading refers giving special meaning to an existing operator. When an operator is overloaded, that operator loses none of its original meaning. Instead, it gains additional meaning relative to the class for which it is defined.
- We cannot overload Class member access operators, Scope Resolution operator, Size operator(sizeof) , Conditional operator (? :).
- When a member operator function overloads a binary operator, the function will have only one parameter.
- When you overload a unary operator using a member function, the function has no parameters.
- To overload unary postfix operators a dummy integer is passed as a parameter to the operator function
- It is possible to overload an operator relative to a class by using a friend rather than a member function
- We cannot use a friend to overload the assignment operator.
- Type conversion or typecasting refers to changing an entity of one data type into another
- Implicit type conversion is an automatic type conversion by the compiler. The type conversions are automatic as long as the data types involved are built-in types.
- Explicit type conversion is necessary for user defined data types.

Exercises

Short Questions

1. List out the operators that cannot be overloaded in C++>
2. Write the various rules for overloading operators.
3. Write the advantage of overloading operators using friend functions.
4. What do you mean by implicit type conversion?

Long Questions

1. Explain how prefix and postfix operators are overloaded
2. Explain Explicit type conversion methods with example.

Programming Exercises

1. Write a program to overload binary operator + such that it is capable of adding two objects of “distance” class .The distance is expressed as foot and inches.
2. Write a program to overload ++ operator to increment time object. Time is expressed in hour, minute and second.

Answers to Have you Understood Questions

Section 4.3

1. No
2. *return-type class-name*::**operator#**(*arg-list*)
   {
   // *operation to be performed*
   }
3. Unary operator Eg !. Binary operator Eg. +
4. =(assignment operator)

Section 4.4

1. Type conversion or typecasting refers to changing an entity of one data type into another.
2. =(assignment operator)