LAB-MANUAL-09

Operator Overloading and Friend Functions

Overloading in C++

If we create two or more member of the same class having the same name but different in number or type of parameter, it is known as C++ overloading.

In C++, we can overload:

- methods
- constructors
- indexed properties

It is because these members have parameters only.

Moving on with this article on Operator Overloading in C++.

Types of overloading in C++

- Function overloading
- Operator overloading

Why is operator overloading used?

C++ programs can be written without the knowledge of operator overloading. Then too, operator operating are profoundly used by programmers to make the program intuitive. For example,

We can replace the code like:

```
1 calculation = add(divide(a, b),multiply(a, b));
```

Syntax of Operator Overloading

```
return_type class_name  : : operator op(argument_list)
{
// function body
}
```

Where the return type is the type of value returned by the function. class_name is the name of the class.

Implementing Operator Overloading in C++

Operator function must be either non-static (member function) or friend function to get overloaded. Operator overloading function can be applied on a member function if the left operand is an object of that class, but if the Left operand is different, then the Operator overloading function must be defined as a non-member function.

Operator overloading function can be made friend function if it requires access to the private and protected members of the class. For example, the operator op is an operator function where op is the operator being overloaded, and the operator is the keyword. Operator overloading can be achieved by implementing a function that can be either member function,non- member function or friend function.

What is the difference between operator functions and normal functions?

Operator functions are the same as normal functions. The only difference is, the name of an operator function is always operator keyword followed by the symbol of operator and operator functions are called when the corresponding operator is used.

Moving on with this article on Operator Overloading in C++.

Types of overloading approaches

Operator Overloading can be done by using three approaches, they are

- Overloading unary operator.
- Overloading binary operator.
- Overloading binary operator using a friend function.

Moving on with this article on Operator Overloading in C++.

Overloading Unary Operator

Let us consider the unary '– 'operator. A minus operator when used as a unary it requires only one operand. We know that this operator changes the sign of an operand when applied to a basic data variable. Let us see how to overload this operator so that it can be applied to an object in much the same way as it is applied to an int or float variable. The unary minus, when applied to an object, should decrement each of its data items.

Example:

```
1
2  #include <iostream>
3  using namespace std;
4
5
6  class Height {
7  public:
8  // Member Objects
9  int feet, inch;
10 // Constructor to initialize the object's value
11 Height(int f, int i)
12 {
13 feet = f;
14 inch = i;
15 }
16 // Overloading(-) operator to perform decrement
17 // operation of Height object
18 void operator-()
19 {
20 feet--;
21 inch--;
22 cout << "Feet & Inches after decrement: " << feet << " ' " << inch
23 <<endl;
24 }
25 };
26 int main()
27 {
28 //Declare and Initialize the constructor of class Height
29 Height h1(6, 2);
30 //Use (-) unary operator by single operand
31 -h1;
32 return 0;
33 }
34
```

Output:

```
Feet & Inches after decrement: 5'1
[Finished in 1.5s]
```

Explanation:
In the above example, we overload ' – ' unary operator to perform decrement in the two variables of Height class. We pass two parameters to the constructor and save their values in feet and inch variable. Then we define the operator overloading function (void operator-()) in which the two variables are decremented by one position. When we write -h1 it calls the operator overloading function and decrements the values passed to the constructor.

Overloading Binary Operator

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

        class Rectangle
        {

                int L,B;

                public:

                Rectangle()          //Default Constructor
                {
                        L = 0;
                        B = 0;
                }

                Rectangle(int x,int y)          //Parameterize Constructor
                {
                        L = x;
                        B = y;
```

```
                }

                Rectangle operator+(Rectangle Rec)      //Binary operator overloadin
g func.
                {
                        Rectangle R;

                                R.L = L + Rec.L;
                                R.B = B + Rec.B;

                        return R;
                }

                void Display()
                {
                        cout<<"\n\tLength : "<<L;
                        cout<<"\n\tBreadth : "<<B;
                }

        };

        void main()
        {

                Rectangle R1(2,5),R2(3,4),R3;
                //Creating Objects


                cout<<"\n\tRectangle 1 : ";
                R1.Display();
```

```
        cout<<"\n\n\tRectangle 2 : ";
        R2.Display();

        R3 = R1 + R2;       Statement 1

        cout<<"\n\n\tRectangle 3 : ";
        R3.Display();
}


Output :

  Rectangle 1 :
  L : 2
  B : 5

  Rectangle 2 :
  L : 3
  B : 4

  Rectangle 3 :
  L : 5
  B : 9
```

In statement 1, Left object R1 will invoke operator+() function and right object R2 is passing as argument.

Another way of calling binary operator overloading function is to call like a normal member function as follows,

```
R3 = R1.operator+ ( R2 );
```

Friend function :

A **friend function** will be friendly with a class even though it is not a member of that class and can access the private members of the class.

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

class Rectangle {
	int width, height;

public:
	Rectangle(int w = 1, int h = 1):width(w),height(h){}
	friend void display(Rectangle &);
};

void display(Rectangle &r) {
	cout << r.width * r.height << endl;
}

int main () {
	Rectangle rect(5,10);
	display(rect);
	return 0;
}
```

We make a function a friend to a class by declaring a prototype of this external function within the class, and preceding it with the keyword **friend**

```
        friend void display(Rectangle &);
```

The friend function **display(rect)** has an access to the private member of the Rectangle class object though it's not a member function. It gets the width and height using dot: **r.width** and **r.height**. If we do this inside main, we get an error because they are private members and we can't access them outside of the class. But friend function to the class can access the private members.

But what's the point of friend functions. In the above example, we could have made "display" as a member function of the class instead of declaring it as a friend function to the class.

Why do we need friend functions?

A friend function can be friendly to 2 or more classes. The friend function does not belong to any class, so it can be used to access private data of two or more classes as in the following example.

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

class Square;  // forward declaration

class Rectangle {
        int width, height;

public:
        Rectangle(int w = 1, int h = 1):width(w),height(h){}
        friend void display(Rectangle &, Square &);
```

```
};

class Square {
	int side;

public:
	Square(int s = 1):side(s){}
	friend void display(Rectangle &, Square &);
};

void display(Rectangle &r, Square &s) {
	cout << "Rectangle: " << r.width * r.height << endl;
	cout << "Square: " << s.side * s.side << endl;
}

int main () {
	Rectangle rec(5,10);
	Square sq(5);
	display(rec,sq);
	return 0;
}
```

Output is:

```
Rectangle: 50
Square: 25
```

The friend functions can serve, for example, to conduct operations between two different classes. Generally, the use of friend functions is out of an object-oriented programming methodology, so whenever possible it is better to use members of the

same class to perform operations with them as in the following example getting exactly same output.

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

class Rectangle {
	int width, height;

public:
	Rectangle(int w = 1, int h = 1):width(w),height(h){}
	void display() {
		cout << "Rectangle: " << width * height << endl;
	};
};

class Square {
	int side;

public:
	Square(int s = 1):side(s){}
	void display() {
		cout << "Square: " << side * side << endl;
	};
};

int main () {
	Rectangle rec(5,10);
	Square sq(5);
	rec.display();
	sq.display();
```

```
        return 0;
}
```

Summary:

1. Friend functions are not members of any class but they can access private data of the class to which they are a friend.
2. Because they are not members of any class, you should not call them using the dot operator.

EXERCISES:

- A students wants to add two complex number whose one part real number and the another part is an imaginary number, you are asked to write a program to add these numbers with the help of operator overloading technique.

- A vector namely V1 and its x and y components is v1_x and v1_y,the second vector is V2 and it has also two component namely v2_x and v2_y,perform vector addition by using its respective components into third V3 and its components is v3_x and v3_y with the help of operator overloading.

- Calculate Area of Triangle the one class is acute angle(less than 90 degree) and another one is obtuse angle (greater than 90 degree) both class wants to access only one function which is "Area of Triangle" which is independent of all classes.

Practice Task:

a. Define the class **bankAccount** to store a bank customer's account number and balance. Suppose that account number is of type int, and balance is of type double. Your class should, at least, provide the following operations: set the account number, retrieve the account number, retrieve the balance, deposit and withdraw money, and print account information. Add appropriate constructors.

b. Every bank offers a checking account. Derive the class **checkingAccount** from the class **bankAccount** (designed in part (a)). This class inherits members to store the account number and the balance from the base class. A customer with a checking account typically receives interest, maintains a minimum balance, and pays service charges if the balance falls below the minimum balance. Add member variables to store this additional information. In addition to the operations inherited from the base class, this class should provide the following operations: set interest rate,

retrieve interest rate, set minimum balance, retrieve minimum balance, set service charges, retrieve service charges, post interest, verify if the balance is less than the minimum balance, write a check, withdraw (override the method of the base class), and print account information. Add appropriate constructors.

c. Every bank offers a savings account. Derive the class **savingsAccount** from the class **bankAccount** (designed in part (a)). This class inherits members to store the account number and the balance from the base class. A customer with a savings account typically receives interest, makes deposits, and withdraws money. In addition to the operations inherited from the base class, this class should provide the following operations: set interest rate, retrieve interest rate, post interest, withdraw (override the method of the base class), and print account information. Add appropriate constructors.

d. Write a program to test your classes designed in parts (b) and (c).