
Lab No.13 Operator Overloading

13.1 Introduction

An operator in a programming language is a symbol that tells the compiler or interpreter to perform specific mathematical, relational or logical operation and produce final result. To perform different operations with ease, a programming language provides variety of operators. This includes basic assignment operator, arithmetic operators, relational operators, logical operators, unary and ternary operators, bitwise and bit shift operators, comparison operators, compound assignment operators, etc. These operators are used with built-in data types. In some programming languages, it is allowed to use these operators with user-defined types such as structures or objects via mechanism of operator overloading. This lab covers operator overloading in classes with examples. Also, lists few languages that do not support operator overloading.

This lab completes the topic of Operator Overloading taken from CISCO Network Academy. In this lab, students are asked to perform the CISCO Chapter 8 – Operators and Enumerated Types Module and cover it successfully.

13.2 Objectives of the lab:

- 1 Understand operator overloading of binary and unary operators.
- 2 Develop operator overloaded function for different operators in a class (C++/Python).
- 3 Use operators with class objects.
- 4 Complete all the tasks available in CISCO Chapter 8.

13.3 Pre-Lab

13.3.1 Operator overloading

- 1 Allows to use operators for ADTs
- 2 Most appropriate for math classes e.g. matrix, vector, etc.
- 3 Gives operators class-specific functionality
- 4 Analogous to function overloading -- operator@ is used as the function name
- 5 Operator functions are not usually called directly
- 6 They are automatically invoked to evaluate the operations they implement
- 7 Consider that + operator has been overloaded:

Actual C++ code becomes

`c1+ c2 + c3 +c4`

The resultant code is very easy to read, write, and maintain

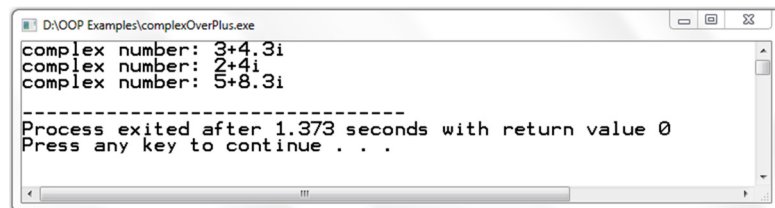
13.3.2 C++ Syntax of operator function for binary operators

```
TYPE operator OP (TYPE rhs) {  
    // body of function...  
}
```

13.3.3 Example: complexOverPlus.cpp

```
#include <iostream>  
using namespace std;  
class complex {  
    private:  
        double re, im;  
    public:  
        complex(): re(0),im(0) { }  
        complex(double r, double i): re(r), im(i) { }      // parameterized, overloaded ctor  
        void show(){  
            cout<<"Complex Number: "<<re<<"+"<<im<<"i"<<endl;  
        }  
        complex operator + (complex rhs){  
            complex temp;  
            temp.re=re + rhs.re;  
            temp.im=im + rhs.im;  
            return temp;  
        }  
};  
int main() {  
    complex    c1(3, 4.3), c2(2, 4), c3;  
    c1.show();    c2.show();  
    c3 = c1 + c2;          //Invocation of "+" Operator -- direct  
    //or  
    //c3 = c1.operator+ (c2);      //Invocation of "+" Operator -- Function  
    c3.show();  
    return 0;  
}
```

Output:



```
D:\OOP Examples\complexOverPlus.exe  
complex number: 3+4.3i  
complex number: 2+4i  
complex number: 5+8.3i  
-----  
Process exited after 1.373 seconds with return value 0  
Press any key to continue . . .
```

Figure 13.1: Output complexOverPlus.cpp

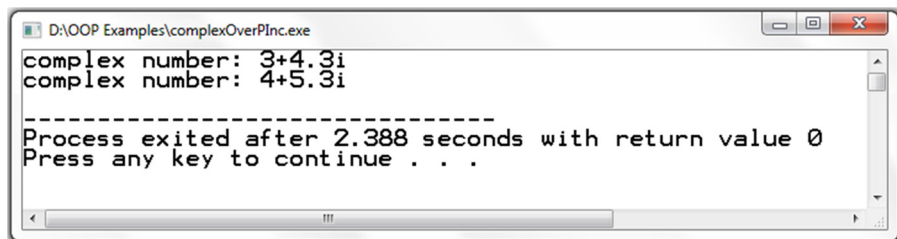
13.3.4 C++ Syntax of operator function for unary operators

```
TYPE operator OP () {  
    // body of function...  
}
```

13.3.5 Example: complexOverPInc.cpp

```
#include <iostream>  
using namespace std;  
  
class complex{  
    private:  
        double re, im;  
    public:  
        complex(): re(0),im(0) { }  
        complex(double r, double i): re(r), im(i) { }  
        void show(){  
            cout<<"Complex Number: "<<re<<"+ "<<im<<"i"<<endl;  
        }  
        void operator ++(){          // pre-increment overloaded  
            ++re;  
            ++im;  
        }  
};  
  
int main() {  
    complex    c1(3, 4.3);          // original  
    c1.show();  
  
    ++c1;                          // overloaded  
    c1.show();  
  
    return 0;  
}
```

Output:



```
D:\OOP Examples\complexOverPInc.exe  
complex number: 3+4.3i  
complex number: 4+5.3i  
-----  
Process exited after 2.388 seconds with return value 0  
Press any key to continue . . .
```

Figure 13.2: Output complexOverPInc.cpp

13.3.6 How can the above program be modified to allow for a statement like `c2=++c1`;

13.3.7 Example in Java

Java doesn't support user-defined operator overloading. Other languages like Java are: C, JavaScript, Go, Objective-C, Pascal, and Visual Basic.

13.3.8 Example in Python

- 1 Python supports operator overloading of different operators. List and names of different operators are shown in Table 13.1.

Table 13.1a – Unary Operators

Unary Operator	Special Method
-	<code>__neg__</code>
+	<code>__pos__</code>
~	<code>__invert__</code>

Table 13.1b – Binary Operators

Binary Operator	Special Method
+	<code>__add__</code>
-	<code>__sub__</code>
*	<code>__mul__</code>
/	<code>__div__</code>
%	<code>__mod__</code>
**	<code>__pow__</code>
&	<code>__and__</code>
	<code>__or__</code>
^	<code>__xor__</code>
+=	<code>__iadd__</code>
-=	<code>__isub__</code>
*=	<code>__imul__</code>
/=	<code>__idiv__</code>
%=	<code>__imod__</code>
==	<code>__eq__</code>
!=, <>	<code>__ne__</code>
>	<code>__gt__</code>
<	<code>__lt__</code>
>=	<code>__ge__</code>
<=	<code>__le__</code>

- 2 It must be noted that there are no pre-increment/pre-decrement or post-increment/post-decrement operators available in Python. However, += and -= operators

are provided to perform the required task.

3 Python syntax for binary operator:

```
def op_specical_funtion_name (self, object) {  
    // body of function...  
}
```

4 Python syntax for unary operator:

```
def op_specical_funtion_name (self) {  
    // body of function...  
}
```

5 Next, an example use of both binary and unary operators are demonstrated in complexOverOps.py.

```
class complex:  
    def __init__(self, r=0, i=0):  
        self.re = r  
        self.im = i  
    def sum(self, c):  
        temp = complex()  
        temp.re = self.re + c.re  
        temp.im = self.im + c.im  
        return temp  
    def __add__(self, c):    # binary + is overloaded  
        temp = complex()  
        temp.re = self.re + c.re  
        temp.im = self.im + c.im  
        return temp  
    def __invert__(self):    # unary ~ is overloaded  
        temp = complex()  
        temp.re = -self.re  
        temp.im = -self.im  
        return temp  
    def show(self):  
        print self.re, "+", self.im, "i"
```

```
c1 = complex(2,3.5)  
c1.show()  
c2 = complex(3.5,2)  
c2.show()  
c3 = complex()  
c3 = c1.sum(c2)
```

```
c3.show()
c3 = c1 + c2
c3.show()
c1 = ~c1
c1.show()
```

Output:

```
In [1]: runfile('C:/Users/sumayya/complexOverOps.py', wdir='C:/Users/sumayya')
2 + 3.5 i
3.5 + 2 i
5.5 + 5.5 i
5.5 + 5.5 i
-2 + -3.5 i
```

Figure 13.3: complexOverOps.py

In this example, a complex class is written in python. It consists of a parameterized constructor, a simple sum() function, and a show() function. Two overloaded functions i.e. __add__() to perform binary addition of complex objects and __invert__() to perform unary inversion of complex object has been developed.

To demonstrate, three objects c1, c2, and c3 are created using parameterized constructor. Output of c1 and c2 is shown. c3 stores the addition of c1 and c2. This is done in two ways. Firstly using sum() function and then using + operator. Both results are shown. Finally, c1 is inverted using ~ operator and output is displayed.

13.4 Activities

Perform these activities in C++ and Python.

13.4.1 Activity

Create a class called **Distance** containing two members feet and inches. This class represents distance measured in feet and inches. For this class, provide the following functions:

- A **no-argument constructor** that initializes the data members to some fixed values.
- A **2-argument constructor** to initialize the values of feet and inches to the values sent from the calling function at the time of creation of an object of type Distance.
- A **showDistance()** to show the distance in feet and inches.
- Overloaded arithmetic operators
 - operator+** to add two distances: Feet and inches of both objects should add in their corresponding members. 12 inches constitute one feet. Make sure that the result of addition doesn't violate this rule.

- b. **operator+=** for addition of two distances.
- i) Overloaded relational operators
 - a. **operator >** should return a variable of type **bool** to indicate whether 1st distance is greater than 2nd or not.
 - b. **operator <** should return a variable of type **bool** to indicate whether 1st distance is smaller than 2nd or not.
 - c. **operator >=** should return a variable of type **bool** to indicate whether 1st distance is greater than or equal to 2nd or not.
 - d. **operator <=** should return a variable of type **bool** to indicate whether 1st distance is smaller than or equal to 2nd or not.
- j) Overloaded equality operators
 - a. **operator==** should return a variable of type **bool** to indicate whether 1st Distance is equal to the 2nd distance or not.
 - b. **Operator!=** should a **true** value if both the distances are not equal and return a **false** if both are equal.
- k) Overloaded operators for **pre- and post- increment**. These increment operators should add a 1 to the **inches**. Keep track that **inch** should not exceed 12.
- l) Overload operators for **pre- and post- decrement**. These decrement operators should subtract a 1 from **inches**. If number of inches goes below 0, take appropriate actions to make this value valid.

NOTE 1: Define all the member functions outside the class (Only in C++).

NOTE 2: Use appropriate special function name for respective overloaded operator as given in Table 6.1 (Only in Python).

13.4.2 Activity

Create a class **RationalNumber** that stores a fraction in its original form (i.e. without finding the equivalent floating pointing result). This class models a fraction by using two data members: an integer for numerator and an integer for denominator. For this class, provide the following functions:

- a) A **no-argument constructor** that initializes the numerator and denominator of a fraction to some fixed values.
- b) A **two-argument constructor** that initializes the numerator and denominator to the values sent from calling function. This constructor should prevent a 0 denominator in a fraction, reduce or simplify fractions that are not in reduced form, and avoid negative denominators.

- c) A function **AddFraction** for addition of two rational numbers.

Two fractions a/b and c/d are added together as:

$$\frac{a}{b} + \frac{c}{d} = \frac{(a * d) + (b * c)}{b * d}$$

- d) A function **SubtractFraction** for subtraction of two rational numbers.

Two fractions a/b and c/d are subtracted from each other as:

$$\frac{a}{b} - \frac{c}{d} = \frac{(a * d) - (b * c)}{b * d}$$

- e) A function **MultiplyFraction** for multiplication of two rational numbers.

Two fractions a/b and c/d are multiplied together as:

$$\frac{a}{b} * \frac{c}{d} = \frac{a * c}{b * d}$$

- f) A function **DivideFraction** for division of two rational numbers.

If fraction a/b is divided by the fraction c/d, the result is

$$\frac{a}{b} \bigg/ \frac{c}{d} = \frac{a * d}{b * c}$$

- g) Provide the following functions for comparison of two fractions

- isGreater**: should return a variable of type **bool** to indicate whether 1st fraction is greater than 2nd or not.
- isSmaller**: should return a variable of type **bool** to indicate whether 1st fraction is smaller than 2nd or not.
- isGreaterEqual**: should return a variable of type **bool** to indicate whether 1st fraction is greater than or equal to 2nd or not.
- isSmallerEqual**: should return a variable of type **bool** to indicate whether 1st fraction is smaller than or equal to 2nd or not.

- h) Provide the following functions to check the equality of two fractions

- isEqual**: should return a variable of type **bool** to indicate whether 1st fraction is equal to the 2nd fraction or not.

- f. **isNotEqual:** should a **true** value if both the fractions are not equal and return a **false** if both are equal.
- i) Provide the **showRN()** to display the respective rational number.

NOTE 1: Define all the member functions outside the class (Only in C++).

NOTE 2: Use appropriate special function name for respective overloaded operator as given in Table 13.1 (Only in Python).

13.4.3 Activity

Complete all the tasks available in CISCO Chapter 8 – Operators& Enumerated Types.

13.5 Testing

Test Cases for Activity 13.4.1

Sample Inputs	Sample Outputs
<p>Declare two Distance Objects d1 and d2. Take both object values from user.</p> <p>Display the content of d1 and d2 using showDistance()</p> <p>Add d1 and d2 using + and store in d3. Display the content of d3 using showDistance()</p> <p>Add d3 and d1 using += and store in d3. Display the content of d3 using showDistance()</p> <p>Check distance d1 is greater than d2 using > and display the respective message.</p> <p>Check distance d1 is less than d2 using < and display the respective message.</p> <p>Check distance d1 is greater than or equal to d2 using >= and display the respective message.</p> <p>Check distance d1 is less than or equal to d2 using <= and display the respective message.</p> <p>Check distance d1 is equal to d2 using == and display the respective message.</p>	<p>Enter First Distance (feet, in): 3 12 Enter Second Distance (feet, in): 12 0</p> <p>Input distances: feet: 3, inches: 12 feet: 12, inches: 0</p>

<p>Check distance d1 is not equal to d2 using != and display the respective message.</p> <p>Pre-increment d2 and display the content using showDistance().</p> <p>Post-decrement d1 and display the content using showDistance().</p>	
---	--

Test Cases for Activity 13.4.2

Sample Inputs	Sample Outputs
<p>Declare two RationalNumber Objects r1 and r2. Take both object values from user.</p> <p>Display the content of r1 and r2 using showRN()</p> <p>Add r1 and r2 using + and store in r3. Display the content of r3 using showRN()</p> <p>Subtract r1 and r2 using - and store in r3. Display the content of r3 using showRN()</p> <p>Multiply r1 and r2 using * and store in r3. Display the content of r3 using showRN()</p> <p>Divide r1 and r2 using / and store in r3. Display the content of r3 using showRN()</p> <p>Check rational number r1 is greater than r2 using > and display the respective message.</p> <p>Check rational number r1 is less than r2 using < and display the respective message.</p> <p>Check rational number r1 is greater than or equal to r2 using >= and display the respective message.</p> <p>Check rational number r1 is less than or equal to r2 using <= and display the respective message.</p> <p>Check rational number r1 is equal to r2 using == and display the respective message.</p> <p>Check rational number r1 is not equal to r2 using != and display the respective message.</p>	<p>Enter First Rational Number: 1 1 Enter Second Rational Number: 1 3</p> <p>Input Rational Numbers: R1 = 1 R2 = 1/3</p>

13.6 References:

1. Class notes
2. Object-Oriented Programming in C++ by *Robert Lafore*
3. How to Program C++ by *Deitel & Deitel*
4. Programming and Problem Solving with Java by *Nell Dale & Chip Weems*
5. Murach's Python Programming by *Micheal Urban & Joel Murach*