Lab Manual

CS112L – Object Oriented Programming Lab
Lab No: **09**
Topic: **Operator Overloading**

Class: **BSGM**
Semester: **II**
Session: **Spring, 2022**
Instructor: **Ms. Saira Qamar**

Lab Date: **April 19h, 2022**
Lab Time: **10:40hrs – 12:50hrs**

Air University Islamabad
FACULTY OF COMPUTING & ARTIFICIAL INTELLIGENCE

Faculty of Computing and AI

Instructions

Submission: Use proper naming convention for your submission file. Name the submission file as **LabNO_ROLLNUM (e.g. Lab01_00000).** Submit the file on Google Classroom within the deadline. Failure to submit according to the above format would result in deduction of 10% marks. Submissions on the email will not be accepted.

Plagiarism: Plagiarism cases will be dealt with strictly. If found plagiarized, both the involved parties will be awarded zero marks in the assignment, all of the remaining assignments, or even an F grade in the course. Copying from the internet is the easiest way to get caught!

Deadline: The deadlines to submit the assignment are hard. Late submission with marks deduction will be accepted according to the course policy shared by the instructor. Correct and timely submission of the assignment is the responsibility of every student; hence no relaxation will be given to anyone.

Comments: Comment your code properly. Bonus marks (maximum 10%) will be awarded to well comment code. Write your name and roll number (as a block comment) at the beginning of the solution to each problem.

Tip: For timely completion of the assignment, start as early as possible. Furthermore, work smartly - as some of the problems can be solved using smarter logic.

1. Note: Follow the given instructions to the letter, failing to do so will result in a zero.

Objectives

In this lab, you will learn:

- Operator Overloading
- Unary Operator Overloading
- Binary Operator Overloading
- Stream Insertion and Stream Extraction Operator

Concepts

1. Operator Overloading:

We are all familiar with the use of different operators with primitive data types. But **when it comes to abstract data types** we have to **define each operator individually**. The greatest advantage of defining each operator is that we can adapt/ modify an operator according to our class and the objects. As the name indicates, **operator overloading is a basically an operator function whose functionality has been overloaded**. In this lab we will explore the concept of operator adaptation and their overloading. Later on we will overload some operators for problems related to real world. This lab will explain how friend functions for overloading stream operators.

Operator Overloading – General Rules

- ✓ Only those operators can be overloaded that have been defined by C++. This means you cannot create your own operators
- ✓ Two operators = and & are already overloaded by default in C++. For example, to **copy objects of the same class**, we can directly use the = operator. We do not need to create an operator function
- ✓ Operator overloading cannot change the **precedence and associativity of operators**. However, if we want to change the order of evaluation, parentheses should be used.
- ✓ There are 4 operators that cannot be overloaded in C++. They are:
  - (::) (scope resolution)
  - (.) (member selection)
  - (.*) (member selection through pointer to function)
  - (?:) (ternary operator)

Operator Overloading – General Syntax

As explained previously, operator overloading is basically a function that has an operator symbolic name. The operator function is created inside the class and it is used basically in the main.

```
class className {
    ... .. ...
    public
       returnType operator symbol (arguments) {
           ... .. ...
       }
    ... .. ...
};
```

Here,

- ✓ **returnType** is the return type of the function.
- ✓ **operator** is a keyword.
- ✓ **symbol** is the operator we want to overload. Like: +, <, -, ++, etc.
- ✓ **arguments** is the arguments passed to the function.

2. Unary Operator Overloading:

Unary operators **operate on only one operand.** The increment operator ++ and decrement operator -- are examples of unary operators. The following is an **example**:

```
class Count {
   private:
    int value;

   public:

    // Constructor to initialize count to 5
    Count() : value(5) {}

    // Overload ++ when used as prefix
    void operator ++ () {
        ++value;
    }

    void display() {
        cout << "Count: " << value << endl;
    }
};

int main() {
    Count count1;

    // Call the "void operator ++ ()" function
    ++count1;

    count1.display();
    return 0;
}
```

Output:

```
Count: 6

--------------------------------
```

Here, when we use ++count1;, the void operator ++ () is called. This increases the value attribute for the object count1 by 1.

The above example works only when ++ is used as a prefix. To make ++ work as a postfix we use this syntax.

```
void operator ++ (int) {
    // code
}
```

Notice the int inside the parentheses. It's the **syntax used for using unary** operators as **postfix**; it's not a function parameter. The following is an **example**
:

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

class Count {
   private:
    int value;

   public:

    // Constructor to initialize count to 5
    Count() : value(3) {}

    // Overload ++ when used as postfix
    void operator ++ (int) {
        value++;
    }

    void display() {
        cout << "Count: " << value << endl;
    }
};

int main() {
    Count count1;
    // Call the "void operator ++ (int)" function
    count1++;
    count1.display();
    return 0;
}
```

Output:

```
Count: 4

--------------------------------
```

The Example 2 works when ++ is used as both prefix and postfix. However, it doesn't work if we try to do something like this:

```
// The previous two work for prefix and postfix but can't
Count count1, result;

// Error: because overloaded function return type is void
result = ++count1;
result.display();
```

To solve this refer to this:

```
class Count {
   private:
    int value;

   public:
    // Constructor to initialize count to 5
    Count() : value(3) {}

    // Overload ++ when used as postfix
    Count operator ++ (int) {
        Count temp;
        // Here, value is the value attribute of the calling object
        temp.value = value++;
        return temp;
    }

    void display() {
        cout << "Count: " << value << endl;
    }
};

int main() {
    Count count1, result;

    // Call the "Count operator ++ (int)" function
    result = count1++;
    result.display();

    return 0;
}
```

The variable value belongs to the count1 object in main() because count1 is calling the function, while temp.value belongs to the temp object.

3. Binary Operator Overloading:

Binary operators work on two operands. For example,

result = a + b;

Here, a and b are operands and + is operator applied on a and b operators.

When we overload the binary operator for user-defined types by using the code:

Obj1 = obj2 + obj3;

Here, obj2 and obj3 are user-defined data-types. The operator function is called using the **obj2** object and obj3 is passed as an argument to the function. The following is an **example**:

```
class Complex {
   private:
    int real;
    int imag;
   public:
    // default constructor
    Complex()
    {
        real = 0;
        imag = 0;
    }
    // Constructor to initialize real and imag to 0
    Complex(int r, int i)
    {
        real = r;
        imag = i;
    }
    // Overload the + operator
    Complex operator + (const Complex& obj) {
        Complex temp;
        temp.real = real + obj.real;
        temp.imag = imag + obj.imag;
        return temp;
    }
    void display() {
       cout << "Output Complex number: " << real << "+" << imag << "i";
    }
};
int main() {
    Complex complex1(2, 3);
    Complex complex2(3, 4);
    Complex result;

   // complex1 calls the operator function
   // complex2 is passed as an argument to the function
    result = complex1 + complex2;

    result.display();

    return 0;
}
```

Output:

```
Output Complex number: 5+7i
--------------------------------
```

4. Stream Insertion and Extraction Overloading:

C++ allows its programmers to overload the stream insertion (<<) and stream extraction (>>) operators so that they work with abstract types. Stream insertion and extraction operators already

work with conventional data types but when it comes to abstract data types the compiler is not aware of what to do if we use a class object with cout or cin. The stream insertion and extraction cannot be overloaded without the help of friend functions. Friend function provides input or output support with direct access to public and private members of the class.

```
class DoB
{
private:
      int month, day, year;

public:
        DoB( )
        {
             cout <<"Nullary constructor";
           month = day = year = 0;
        }
      ~DoB ( )
        {
           cout << "Destructor called";
        }
friend ostream & operator << ( ostream & os, DoB &d );
friend istream & operator >> ( istream & is, DoB &d );
};

ostream & operator << ( ostream & os, DoB &d )
{
        os << d.day << "." << d.month << "." << d.year;
        return os;
}
istream & operator >> ( istream & is, DoB & d )
{
            cout << "\n\n Enter day of birth: ";
            is >> d.day;
            cout << "Enter month of birth: ";
            is >> d.month;
            cout << " Enter year of the birth: ";
            is >> d.year;
            return is;
}
void main(){
            DoB date;
            cout << "\n\n Enter your date of birth";
            cin >> date;
            cout << "\n Entered date is: " << date;
}
```

Lab Tasks

1. Ali Raza is a frequent long distance driver. He wants to compute how much distance he covers over a number of days. Ali also wants to know the average distance travelled.
To perform this task you will need to create a class called distance. The class will contain the fuel price in liter and also the distance he has travelled. To find the average and the distance travelled you will need to overload two operators as follows:

- **+ operator** to compute the total distance travelled.
- **/ operator** to compute the average distance travelled.

In the main( ) you will create six objects and then you will write a single expression that will demonstrate the use of + and the / operator. Your expression should resemble the following:

obj1+obj2+obj3+obj4+obj5+obj6 / 6

Sample Inputs	Sample Outputs
obj1= 10 obj2= 20 obj3= 40 obj4= 20 obj5= 30 obj6= 20 Provide any value for fuel Obj1+obj2+obj3+obj4+obj5+obj6/6	The average distance covered is 23.33

2. Create a class "toll", that represents the toll tax of a vehicle. According to government policy the toll tax of a vehicle is the number of wheels of vehicle multiplied by 6. Your class should have data members as follows: wheels, tax, fine. Suppose the fine of a vehicle is 1 rupee. The + operator should work with multiple objects in main.

- **Overload the stream insertion and extraction to input and output data of the objects**
- Use the + operator to find the total tax collected in a day.
- Use the – operator to deduct 5 from the total tax.
- Create the ++ operator to fine a particular car.