CC-211L

Object Oriented Programming

Laboratory 07

Operator Overloading

Version: 1.0.0

Release Date: 02-03-2023

Department of Information Technology

University of the Punjab

Lahore, Pakistan

Contents:

- Learning Objectives
- Required Resources
- General Instructions
- Background and Overview
  - Operator Overloading
  - Unary Operators
  - Dynamic Memory Management
  - Type Conversion
- Activities
  - Pre-Lab Activity
    - Overloading Unary Operators
      - Overloading Increment and Decrement Operators
    - Task 01
  - In-Lab Activity
    - Dynamic Memory Management
      - new Operator
      - delete Operator
    - Conversion Between Types
    - Overloading the Function Call Operator
    - Task 01
    - Task 02
    - Task 03
  - Post-Lab Activity
    - Task 01
- Submissions
- Evaluations Metric
- References and Additional Material
- Lab Time and Activity Simulation Log

Learning Objectives:

- Overloading Unary Operators
- Overloading ++ Operator
- Overloading – Operator
- Dynamic Memory Management
- Operators as Members vs Non-Members
- Conversion between Types
- Overloading the Function call Operator

Resources Required:

- Desktop Computer or Laptop
- Microsoft ® Visual Studio 2022

General Instructions:

- In this Lab, you are **NOT** allowed to discuss your solution with your colleagues, even not allowed to ask how is s/he doing, this may result in negative marking. You can **ONLY** discuss with your Teaching Assistants (TAs) or Lab Instructor.
- Your TAs will be available in the Lab for your help. Alternatively, you can send your queries via email to one of the followings.

Teachers:		
Course Instructor	Prof. Dr. Syed Waqar ul Qounain	swjaffry@pucit.edu.pk
Lab Instructor	Azka Saddiqa	azka.saddiqa@pucit.edu.pk
Teacher Assistants	Saad Rahman	bsef19m021@pucit.edu.pk
	Zain Ali Shan	bcsf19a022@pucit.edu.pk

Background and Overview:

Operator Overloading:

In C++, we can make operators work for user-defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +.

Operator overloading is a compile-time polymorphism. It is an idea of giving special meaning to an existing operator in C++ without changing its original meaning.

© https://www.geeksforgeeks.org/operator-overloading-c/

Unary Operators:

Unary Operator is an operation that is performed on only one operand. Unary Operators contrast with the Binary operators, which use two operands to calculate the result. Unary operators can be used either in the prefix position or the postfix position of the operand. To perform operations using Unary Operators, we need to use one single operand. There are various types of unary operators, and all these types are of equal precedence, having right-to-left associativity.

Dynamic Memory Management:

Dynamic memory management in C++ refers to performing memory allocation manually by programmer. Dynamically allocated memory is allocated on Heap and non-static and local variables get memory allocated on Stack.

Type Conversion:

Type conversion is the process that converts the predefined data type of one variable into an appropriate data type. The main idea behind type conversion is to convert two different data type variables into a single data type to solve mathematical and logical expressions easily without any data loss.

Activities:

Pre-Lab Activities:

Overloading Unary Operators:

The unary operators operate on a single operand and following are the examples of Unary operators −

- The increment (++) and decrement (--) operators
- The unary minus (-) operator
- The logical not (!) operator

The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometimes they can be used as postfix as well like obj++ or obj--.

Following example explain how minus (-) operator can be overloaded for prefix as well as postfix usage.

Example:

```
#include <iostream>
using namespace std;
class Distance {
private:
    int feet;
    int inches;
public:
    Distance(int f, int i) {
        feet = f;
        inches = i;
    }
    // method to display distance
    void displayDistance() {
        cout << feet <<" feet " << inches <<" inches."<< endl;
    }
    // overloaded minus (-) operator
    Distance operator- () {
        feet = -feet;
        inches = -inches;
        return Distance(feet, inches);
    }
};
int main() {
    Distance D1(11, 10), D2(-5, 11);
    -D1;                    // apply negation
    D1.displayDistance();   // display D1
    -D2;                    // apply negation
    D2.displayDistance();   // display D2
    return 0;
}
```

Fig. 01 (Overloading (-) Operator)

Output:

```
Microsoft Visual Studio Debug Console
-11 feet -10 inches.
5 feet -11 inches.
```

Fig. 02 (Overloading (-) Operator)

Overloading Increment and Decrement Operators:

Following example explain how Increment (++) operator can be overloaded for prefix usage.

Example:

```
#include <iostream>
using namespace std;
class Count {
private:
    int value;
public:
    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;
}
```

Fig. 03 (Overloading ++ Operator)

Output:

Fig. 04 (Overloading ++ Operator)

In the above example, when we use ++count1, the void operator ++ () is called. This increases the value attribute for the object count1 by 1. It 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.

Example:

```cpp
#include <iostream>
using namespace std;
class Count {
private:
    int value;
public:
    // Constructor to initialize count to 5
    Count() : value(5) {}
    // Overload ++ when used as prefix
    void operator ++ () {
        ++value;
    }
    // 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();
    // Call the "void operator ++ ()" function
    ++count1;
    count1.display();
    return 0;
}
```

Fig. 05 (Overloading ++ Operator)

Output:

Fig. 06 (Overloading ++ Operator)

Task 01: Big Integer **[Estimated time 30 minutes / 20 marks]**

- Write a C++ class called BigInt that represents a large integer. The BigInt class should overload the unary ++ (pre-increment) and -- (pre-decrement) operators.
- The BigInt class should be able to represent integers of arbitrary size. You can do this by storing the digits of the integer as an array of integers, where each element in the array represents a single digit. For example, the integer 12345 would be represented as the array {5, 4, 3, 2, 1}.
- When the ++ operator is called, it should increment the BigInt object's integer value by 1 and return the updated BigInt object.
- When the -- operator is called, it should decrement the BigInt object's integer value by 1 and return the updated BigInt object.

Here's an example of how the BigInt class should be used:

```
BigInt b1("123456789");
BigInt b2 = ++b1; // Increment b1.
BigInt b3 = --b1; // Decrement b1.

cout << b1.display() << endl; // Should print "123456789".
cout << b2.display() << endl; // Should print "123456790".
cout << b3.display() << endl; // Should print "123456789".
```

Task 02: Index operator **[Estimated time 30 minutes / 20 marks]**

Create a class Array. It will have a D-array and a data member size in in as private data members.

Create default constructor, parameterized constructor and a display function.

Create a function Input to input values in D- array in class.

- Overload index operator [ ] for this class which will receive index number in it as parameter and will return the number stored at that index. If the index is out of bound, then throw exception runtime error and return -1.
- Overload + operator to add Two Array class objects. Two Array class objects will add in such a way that each respective index value of one array gets added to respective index of the second array. Return a new Object of Array class from this operator which will hold sum of respective indexes of first and second Array object whom with which the + operator is called.
- Similarly, create - operator as well.

In-Lab Activities:

Dynamic Memory Management:

C++ allows us to allocate the memory of a variable or an array in run time. This is known as dynamic memory allocation.
In other programming languages such as Java and Python, the compiler automatically manages the memories allocated to variables. But this is not the case in C++.
In C++, we need to deallocate the dynamically allocated memory manually after we have no use for the variable.
We can allocate and then deallocate memory dynamically using the new and delete operators respectively.

new Operator:
The new operator allocates memory to a variable.

```
// declare an int pointer
int* pointVar;
// dynamically allocate memory
// using the new keyword
pointVar = new int;
// assign value to allocated memory
*pointVar = 45;
```

Fig. 08 (new Operator)

Here, we have dynamically allocated memory for an int variable using the new operator.
Notice that we have used the pointer pointVar to allocate the memory dynamically. This is because the new operator returns the address of the memory location.

delete Operator:
The delete operator deallocates memory assigned to a variable.

```
// declare an int pointer
int* pointVar;
// dynamically allocate memory
// for an int variable
pointVar = new int;
// assign value to the variable memory
*pointVar = 45;
// print the value stored in memory
cout << *pointVar; // Output: 45
// deallocate the memory
delete pointVar;
```

Fig. 09 (delete Operator)

Here, we have dynamically allocated memory for an int variable using the pointer pointVar.
After printing the contents of pointVar, we deallocated the memory using delete.

Example:

```
#include <iostream>
using namespace std;
int main() {
    // declare an int pointer
    int* pointInt;
    // declare a float pointer
    float* pointFloat;
    // dynamically allocate memory
    pointInt = new int;
    pointFloat = new float;
    // assigning value to the memory
    *pointInt = 45;
    *pointFloat = 45.45f;
    cout << *pointInt << endl;
    cout << *pointFloat << endl;
    // deallocate the memory
    delete pointInt;
    delete pointFloat;
    return 0;
}
```

Fig. 10 (Dynamic Memory Allocation)

Output:

Fig. 11 (Dynamic Memory Allocation)

Conversion between types:

A conversion operator (also called a cast operator) also can be used to convert an object of one class to another type. Such a conversion operator must be a non-static member function. The function prototype:

MyClass::operator string() const;

declares an overloaded cast operator function for converting an object of class MyClass into a temporary string object. The operator function is declared const because it does not modify the original object. The return type of an overloaded cast operator function is implicitly the type to which the object is being converted.

Overloaded cast operator functions can be defined to convert objects of user-defined types into fundamental types or into objects of other user-defined types.

Example:

```
3  class Power {
4      double b,val;
5      int e;
6  public:
7      Power(double base, int exp){
8          b = base, e = exp;
9          val = 1;
10         if (exp == 0)
11             return;
12         for (; exp > 0; exp--)
13             val = val * b;
14     }
15     Power operator+(Power o) {
16         double base;
17         int exp;
18         base = b + o.b;
19         exp = e + o.e;
20         Power temp(base, exp);
21         return temp;
22     }
23     operator double() { return val; } // convert to double
24 };
25 int main() {
26     Power x(4.0, 2);
27     double a;
28     a = x;              // convert to double
29     cout << x + 1.2<<"\n"; // convert x to double and add 100.2
30     Power y(3.3, 3), z(0, 0);
31     z = x + y;  // no conversion
32     a = z;      // convert to double
33     cout << a;
34 }
```

Fig. 12 (Conversion between types)

Output:

```
Microsoft Visual Studio Debug Console
17.2
20730.7
```

Fig. 13 (Conversion between types)

Overloading the Function Call Operator:

Overloading the function call operator () is powerful, because functions can take an arbitrary number of comma-separated parameters. In a customized String class, for example, you could overload this operator to select a substring from a String—the operator's two integer parameters could specify the start location and the length of the substring to be selected. The operator() function could check for such errors as a start location out of range or a negative substring length.

When you overload ( ), you are not creating a new way to call a function. Rather, you are creating an operator function that can be passed an arbitrary number of parameters.

Example:

```
#include <iostream>
using namespace std;
class Distance {
private:
    int feet, inches;
public:
    Distance() {
        feet = 0, inches = 0;
    }
    Distance(int f, int i) {
        feet = f;
        inches = i;
    }
    // overload function call
    Distance operator()(int a, int b, int c) {
        Distance D;
        D.feet = a + c + 10;
        D.inches = b + c + 100;
        return D;
    }
    void displayDistance() {
        cout << "F: " << feet << " I:" << inches << endl;
    }
};
int main() {
    Distance D1(11, 10), D2;
    cout << "First Distance : ";
    D1.displayDistance();
    D2 = D1(10, 10, 10); // invoke operator()
    cout << "Second Distance :";
    D2.displayDistance();
    return 0;
}
```

Fig. 14 (Overload Function call operator)

Output:

Fig. 15 (Overload Function call operator)

In Lab Task

Task 1: Complex Numbers [Estimated time 60 minutes / 60 marks]

Write a C++ program that defines a Complex class representing a complex number. It will have two data members:

- real_part
- imaginary_part

Create default constructor, parameterized constructor and a display function. Create setters and getters.

Output format will be real +imaginary i .

Overload following operators for the Complex class. Also you must show their use case in Main function, operator function without implementation in main function will not be considered:

1. **+** : This operator will add two Complex numbers. Two complex numbers are added in such a way that real part is added to real part and imaginary part is added to imaginary part. Return a new object of complex class from this operator which will hold the sum of the two complex numbers.
2. **-** : This operator will subtract two Complex numbers. Two complex numbers are subtracted in such a way that real part is subtracted from real part and imaginary part is subtracted from imaginary part. Return a new object of complex class from this operator which will hold the difference of the two complex numbers.
3. ***** : This operator will multiply two Complex numbers. Two complex numbers are multiplied in such a way that real part is multiplied to real part and imaginary part is multiplied to imaginary part. Return a new object of complex class from this operator which will hold the product of the two complex numbers.
4. / : This operator will divide two Complex numbers. Two complex numbers are divided in such a way that real part is divided by real part and imaginary part is divided by imaginary part. Return a new object of complex class from this operator which will hold the division of the two complex numbers.
5. **<< stream insertion operator:** Overload the stream insertion operator for output of the complex number. Output format will be real +imaginary i .
6. **>> stream extraction operator:** Overload the stream extraction operator for input of the complex number. Output format will be first real number then imaginary number.

7. **== operator:** This operator will return boolean value to check if two complex numbers are equal to each other or not. Return true if equal.Two complex numbers are equal to each other if their real and imaginary part is equal to each other.
8. **!= operator:** This operator will return true if two complex objects are not equal to each other, false otherwise.
9. **>= operator:** This operator will return true if first complex object has greater absolute sum than second complex object. Absolute sum is obtained by formula:

   ((real)^2 + (complex)^2) ^ 1/2

10. **<= operator:** This operator will return true if first complex object has lesser absolute sum than second complex object.
11. **= assignment operator:** This operator will be used to assign one complex number to another.
12. **-- operator:** This operator will decrement one times from both real part and imaginary part of the complex number. It is a unary operator.
13. **++ operator:** This operator will increment one times to both real part and imaginary part of the complex number. It is a unary operator.
14. **~ operator:** This operator will provide absolute sum of the complex number.
15. ! Operator: This operator will return a new object, which will have the real and imaginary part of calling object but with inverse signs.

Remember: Showcase all operators in main function.

Task 02: Array with Operators: **[Estimated time 30 minutes / 30 marks]**

- Create a class Array. It will have a D-array in it as data member and a size variable representing the size of array.
- Create default, parameterized and Copy constructor.
- Create a display function for array.
- Parameterized constructor will contain size of array. Allocate this much memory to array and fill it with zeros.
- Create an other function input which will input elements from user for the array.
- Create Destructor for the class to efficiently deallocate memory.

Overload following operators:

- ***Index operator [ ] :*** This operator just like index operator with a regular array. It will receive index number as parameter and will return the address of that index of array. If out of bound then throw out of bound exception and return address of first index.
- **+ operator:** This operator will sum the respective indexes of array of two objects and return a new object whose array will contain sum if the arrays of other two objects.
- **- operator:** This operator will sum the respective indexes of array of two objects and return a new object whose array will contain sum if the arrays of other two objects.
- **<< stream insertion operator:** Overload the stream insertion operator for output of the whole array.
- **>> stream extraction operator:** Overload the stream extraction operator for input of array from user of size given by user in constructor.

Task 03 (a): Debug the program

write the following code on your editor and take the screen shot of every variable value in memory whenever value of ans variable changes. Take images of call stack as well.

```
#include <iostream>
using namespace std;
int main()
{
    int x = 0;
    int ans = 0;
    int product = 1;
    while (x <= 2)
    {
        if (ans % 2 == 0)
        {
            cout << "Answer: " << ans << endl;
        }
        product *= ans;
        ans += 3;
        x++;
        cout << "Iteration complete" << endl;
    }
    return 0;
}
```

(b)- Take Screenshots of variables in functions in different call stacks.

```
#include <iostream>
using namespace std;
Codiumate: Options | Test this function
int factorial(int n)
{
    if (n <= 1)
        return 1;
    else
        return n * factorial(n - 1);
}

Codiumate: Options | Test this function
int main()
{
    int num = 5;
    cout << "Factorial of " << num << " is: " << factorial(num) << std::endl
    return 0;
}
```

Post-Lab Activities:

Task 01: Overloading Fractions [Estimated time 30 minutes / 30 marks]

Create a class Fraction which will have two Data members in it:

- numerator
- Denominator

➢ Create Default, Parameterized and Copy Constructor. Setters and getters as well.

➢ Create a display function which will display the Fraction in form:

numerator/denominator

Overload following operators:

+ operator: To add two Fraction numbers

- operator: To subtract two Fraction numbers

* operator: To multiply two Fraction numbers

/ operator: To divide two fraction numbers

<< : stream insertion operator

>>: stream extraction operator

! Operator: To change signs of both numerator and denominator

~ operator: This operator will invert the Fraction number.

++ operator: increment both numerator and denominator

-- operator: decrement both numerator and denominator.

Submissions:

- For In-Lab Activity:
  - Save the files on your PC.
  - TA's will evaluate the tasks offline.
- For Pre-Lab & Post-Lab Activity:
  - Submit the .cpp file on Google Classroom and name it to your roll no.

Evaluations Metric:

- All the lab tasks will be evaluated offline by TA's
- **Division of Pre-Lab marks:** **[20 marks]**
  - Task 01: Big Integer [20 marks]
- **Division of In-Lab marks:** **[80 marks]**
  - Task 01: Students GPA [20 marks]
  - Task 02: Cast Complex Numbers [30 marks]
  - Task 03: Overloading Polynomials [30 marks]
- **Division of Post-Lab marks:** **[30 marks]**
  - Task 01: Convert 2D to 1D [30 marks]

References and Additional Material:

- **Unary Operator Overloading**
  https://www.tutorialspoint.com/cplusplus/unary_operators_overloading.htm
- **Dynamic Memory Management**
  https://www.programiz.com/cpp-programming/memory-management
- **Overloading Function Call Operator**
  https://www.ibm.com/docs/en/i/7.2?topic=only-overloading-function-calls-c

Lab Time Activity Simulation Log:

- Slot – 01 – 00:00 – 00:15: Class Settlement
- Slot – 02 – 00:15 – 00:40: In-Lab Task
- Slot – 03 – 00:40 – 01:20: In-Lab Task
- Slot – 04 – 01:20 – 02:20: In-Lab Task
- Slot – 05 – 02:20 – 02:45: Evaluation of Lab Tasks
- Slot – 06 – 02:45 – 03:00: Discussion on Post-Lab Task