CC-211L

Object Oriented Programming

Laboratory 06

Copy Constructor and Operator Overloading

Version: 1.0.0

Release Date: 16-02-2023

Department of Information Technology

University of the Punjab

Lahore, Pakistan

Contents:

Learning Objectives:

- Copy Constructor
- Using Overloaded Operators of Standard Library
- Fundamentals of Operator Overloading
- Overloading Binary Operators

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:

Copy Constructor:

A copy constructor is a type of constructor that creates a copy of another object. If we want one object to resemble another object, we can use a copy constructor. If no copy constructor is written in the program compiler will supply its own copy constructor.

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/

Binary Operators:

A binary operator is an operator that operates on two operands and manipulates them to return a result. Operators are represented by special characters or by keywords and provide an easy way to compare numerical values or character strings.

Activities:

Pre-Lab Activities:

Copy Constructor:

Copy constructors are the member functions of a class that initialize the data members of the class using another object of the same class. It copies the values of the data variables of one object of a class to the data members of another object of the same class. A copy constructor can be defined as follows:

class A { A(A &old_object){ //copy constructor

variable_name = old_object.variable_name; …. …. } };

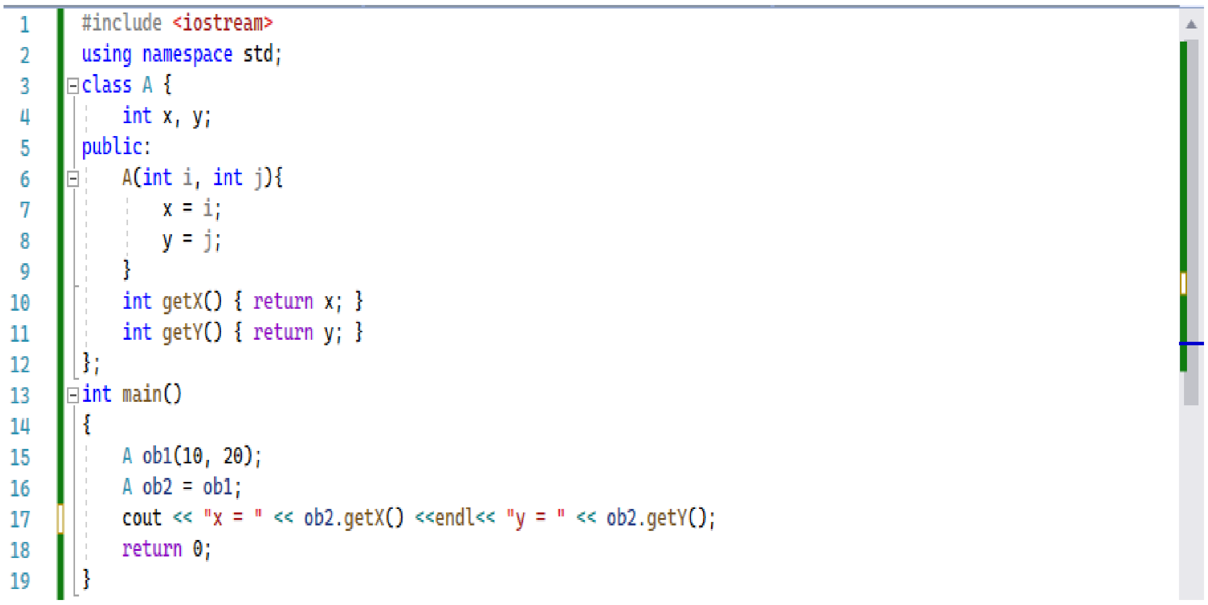
A copy constructor is further categorized into two types:

- Default Copy Constructor
- User-defined Copy Constructor

Default Copy Constructor:

When a copy constructor is not defined, the C++ compiler automatically supplies with its self-generated constructor that copies the values of the object to the new object.

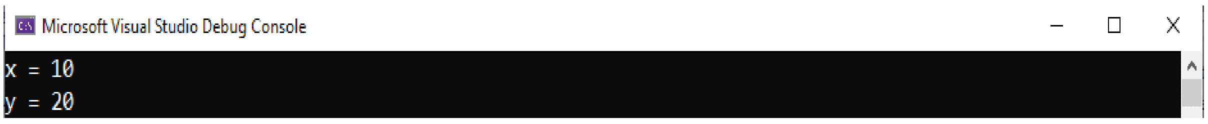
Example:



```cpp
#include <iostream>
using namespace std;
class A {
    int x, y;
public:
    A(int i, int j){
        x = i;
        y = j;
    }
    int getX() { return x; }
    int getY() { return y; }
};
int main()
{
    A ob1(10, 20);
    A ob2 = ob1;
    cout << "x = " << ob2.getX() <<endl<< "y = " << ob2.getY();
    return 0;
}
```

Fig. 01 (Default Copy Constructor)

Output:



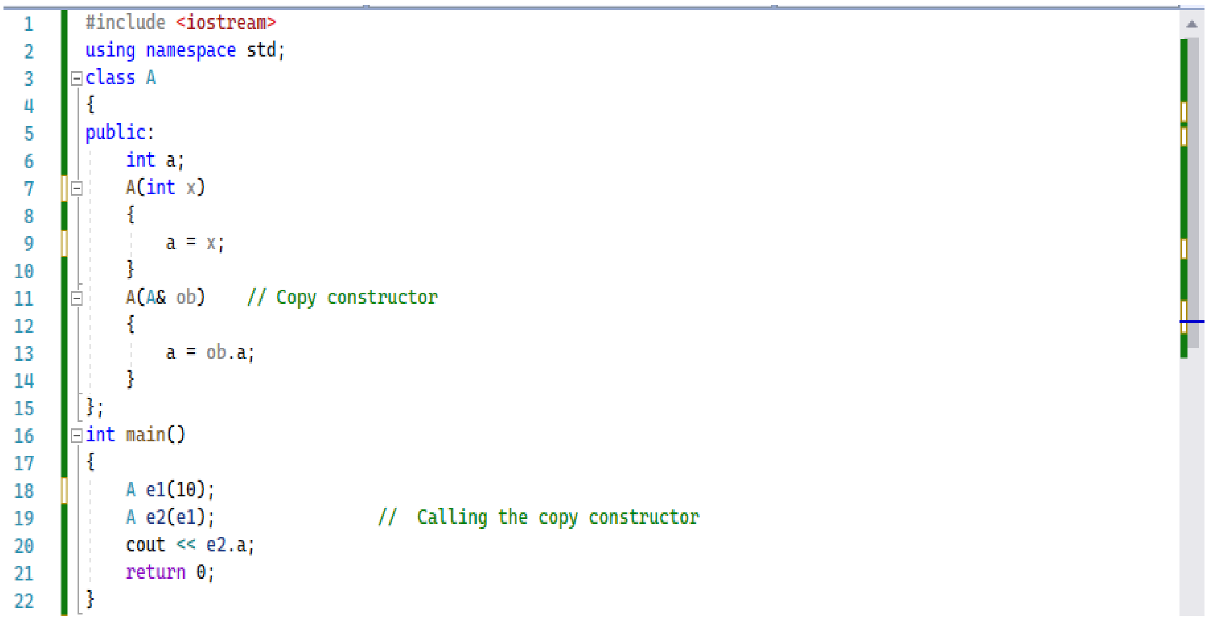
```
x = 10
y = 20
```

Fig. 02 (Default Copy Constructor)

User Defined Copy Constructor:

In user-defined copy constructor, the values of the parameterized object of a class are copied to the member variables of the newly created class object. The initialization or copying of the values to the member variables is done as per the definition of the copy constructor.

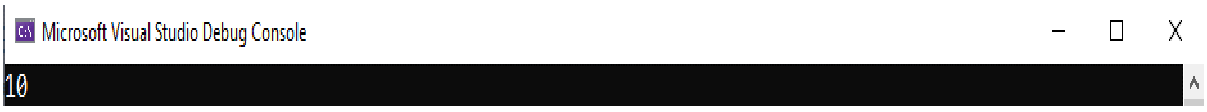
Example:



```cpp
#include <iostream>
using namespace std;
class A
{
public:
    int a;
    A(int x)
    {
        a = x;
    }
    A(A& ob)    // Copy constructor
    {
        a = ob.a;
    }
};
int main()
{
    A e1(10);
    A e2(e1);               //  Calling the copy constructor
    cout << e2.a;
    return 0;
}
```

Fig. 03 (User Defined Copy Constructor)

Output:



```
10
```

Fig. 04 (User Defined Copy Constructor)

Constructor Copies:

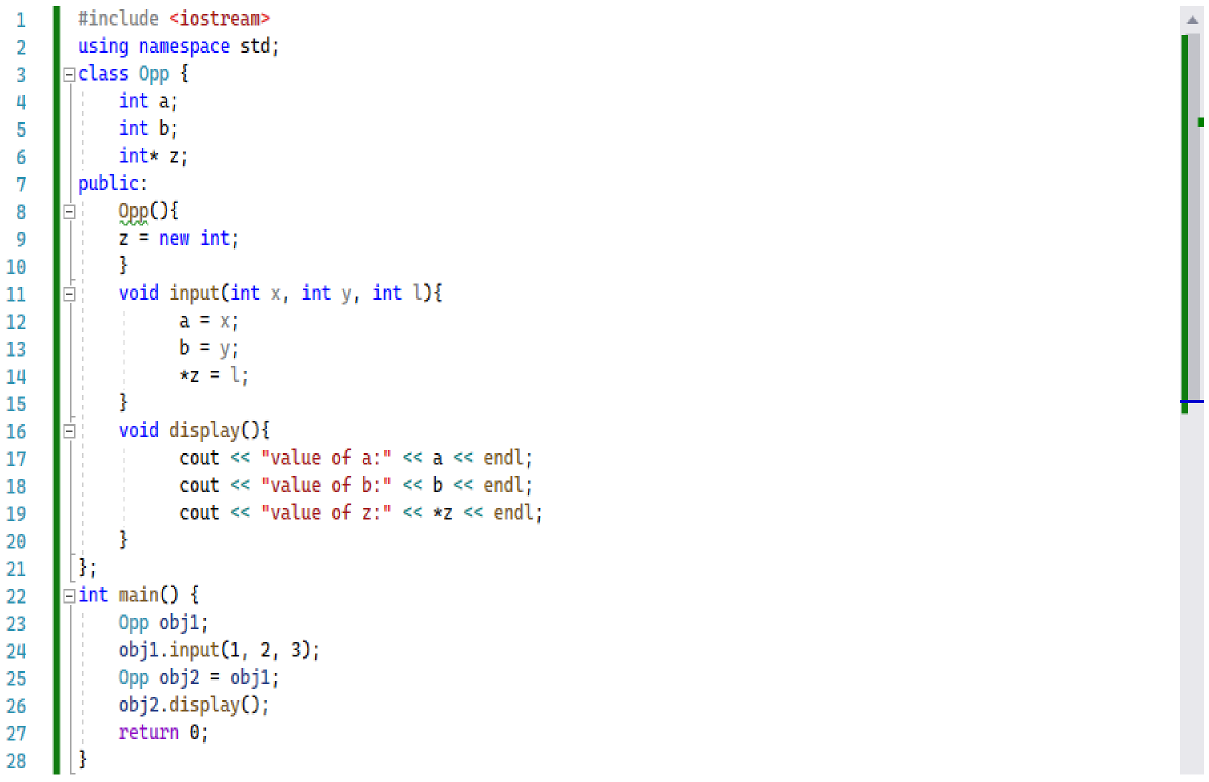
There are two ways in which copy constructor copies, and they are:

- Shallow copy
- Deep copy

Shallow Copy:

It is the process of creating a copy of an object by copying data of all the member variables as it is. Only a default Constructor produces a shallow copy (Constructor which has no arguments).

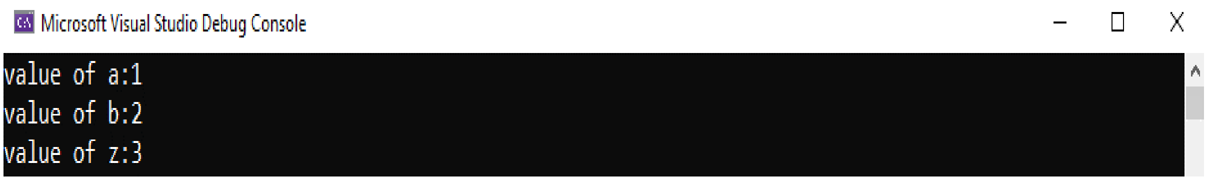
Example:



```
#include <iostream>
using namespace std;
class Opp {
    int a;
    int b;
    int* z;
public:
    Opp(){
    z = new int;
    }
    void input(int x, int y, int l){
        a = x;
        b = y;
        *z = l;
    }
    void display(){
        cout << "value of a:" << a << endl;
        cout << "value of b:" << b << endl;
        cout << "value of z:" << *z << endl;
    }
};
int main() {
    Opp obj1;
    obj1.input(1, 2, 3);
    Opp obj2 = obj1;
    obj2.display();
    return 0;
}
```

Fig. 05 (Shallow Copy)

Output:



```
value of a:1
value of b:2
value of z:3
```

Fig. 06 (Shallow Copy)

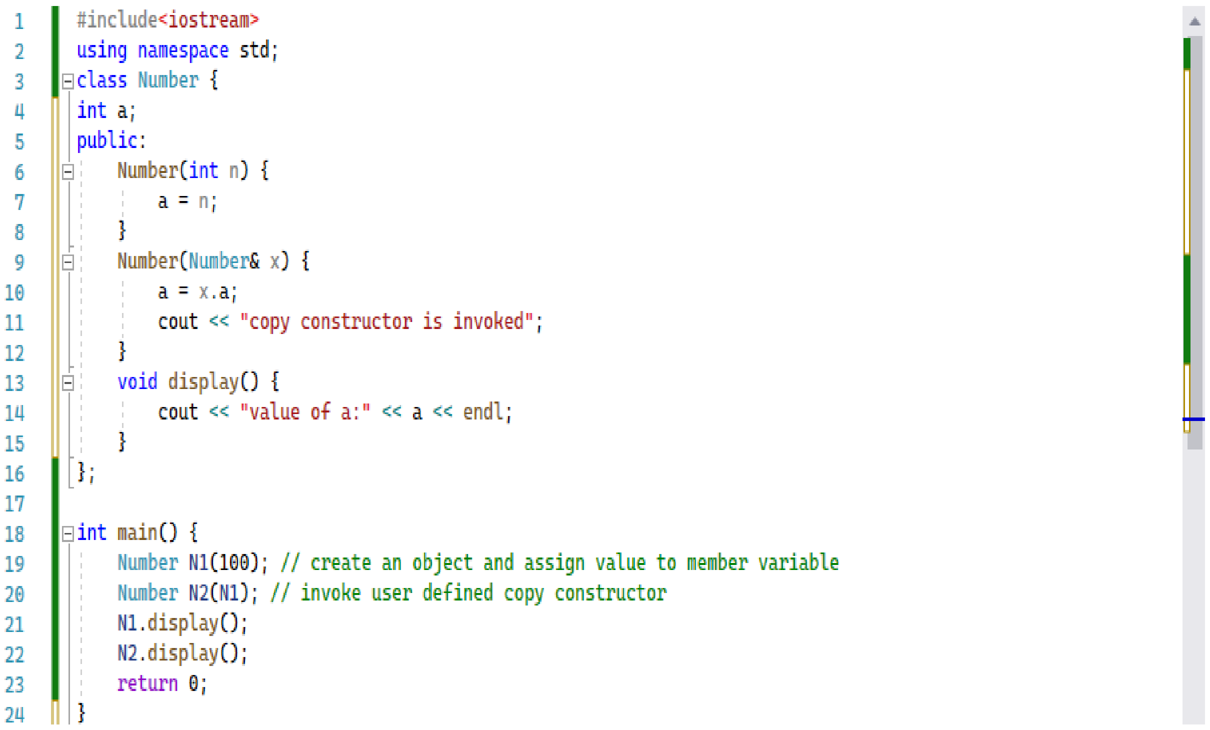
Explanation:

In the above example, both ‘obj1’ and ‘obj2’ will be having the same input and both the object variables will be pointing to the same memory locations. The changes made to one object will affect another one. This can be solved by using deep copy.

Deep Copy:

It dynamically allocates memory for the copy first and then copies the actual value. In a deep copy, both objects which have to copy and another which has to be copied will be having different memory locations. So, the changes made to one will not affect another. This is used by a user-defined copy constructor.

Example:

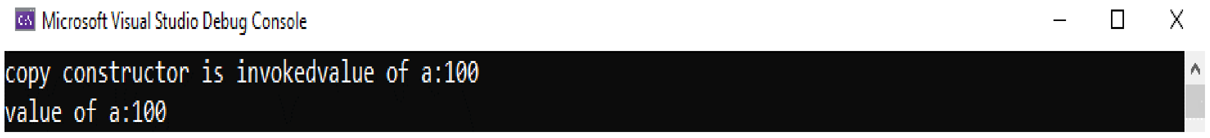


```
#include<iostream>
using namespace std;
class Number {
int a;
public:
    Number(int n) {
        a = n;
    }
    Number(Number& x) {
        a = x.a;
        cout << "copy constructor is invoked";
    }
    void display() {
        cout << "value of a:" << a << endl;
    }
};

int main() {
    Number N1(100); // create an object and assign value to member variable
    Number N2(N1); // invoke user defined copy constructor
    N1.display();
    N2.display();
    return 0;
}
```

Fig. 07 (Deep Copy)

Output:



```
copy constructor is invokedvalue of a:100
value of a:100
```

Fig. 08 (Deep Copy)

Explanation:

N1 and N2 are the two objects. ‘N2’ is the object which stores the value of objec’N1’. ‘N1’ takes 100 as input and will initialize to ‘N2’. Both N1 and N2 will have different locations. Changes made to one will not affect the other.

Task 01 Part(a): Shallow Copy **[Estimated time 20 minutes / 10 marks]**

- Create a class "Rectangle" with the following attributes:
  - Length (int)
  - Width (int)
- Implement a copy constructor for the "Rectangle" class. The copy constructor should create a new object that is a copy of an existing "Rectangle" object.
- In the main function, create a "Rectangle" object and initialize it with a length and width of your choice.
- Create a second "Rectangle" object that is a copy of the first object using the copy constructor.
- Verify that the copy constructor correctly created a new object with the same length and width as the original object.
- Modify the length or width of the second rectangle and verify that the changes affect both the rectangles.

Task 01 Part(b): Deep Copy **[Estimated time 20 minutes / 10 marks]**

- Create a class "Rectangle" with the following attributes:
  - Length (int)
  - Width (int)
- Implement a copy constructor for the "Rectangle" class. The copy constructor should create a new object that is a copy of an existing "Rectangle" object
- In the main function, create a "Rectangle" object and initialize it with a length and width of your choice
- Create a second "Rectangle" object that is a copy of the first object using the copy constructor
- Verify that the copy constructor correctly created a new object with the same length and width as the original object
- Modify the length or width of the second rectangle and verify that the changes only affect the second rectangle not the original one

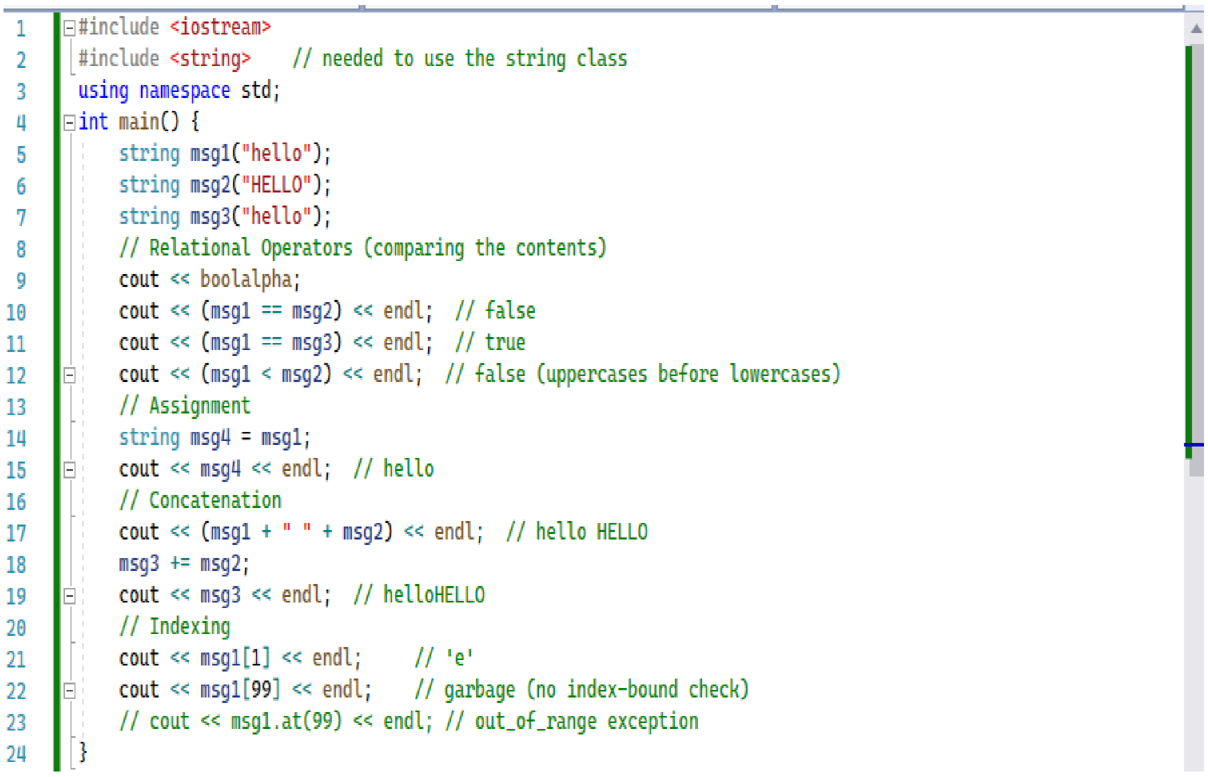
In-Lab Activities:

Overloaded Operators of Standard Library Class string:

The C++ string class overloads these operators to work on string objects:

- String comparison (==, !=, >, <, >=, <=): For example, you can use str1 == str2 to compare the contents of two string objects
- Stream insertion and extraction (<<, >>): For example, you can use cout << str1 and cin >> str2 to output/input string objects
- Strings concatenation (+, +=): For example, str1 + str2 concatenates two string objects to produce a new string object; str1 += str2 appends str2 into str1
- Character indexing or subscripting []: For example, you can use str[*n*] to get the char at index *n*; or str[*n*] = c to modify the char at index *n*. Take note that [] operator does not perform index-bound check, i.e., you have to ensure that the index is within the bounds. To perform index-bound check, you can use string's at() member function
- Assignment (=): For example, str1 = str2 assigns str2 into str1

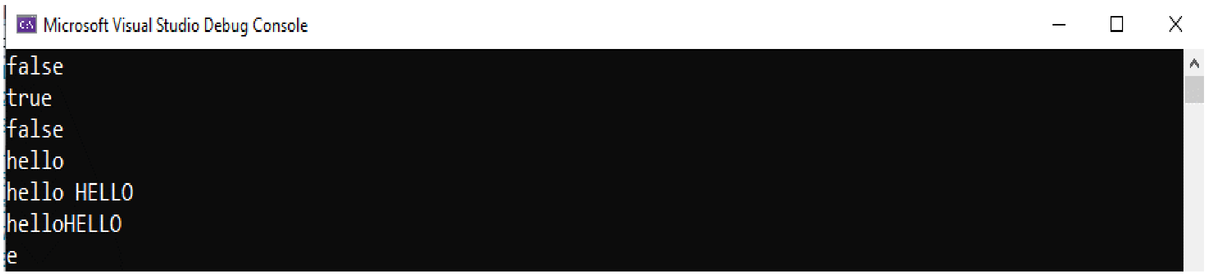
Example:



```cpp
#include <iostream>
#include <string>    // needed to use the string class
using namespace std;
int main() {
    string msg1("hello");
    string msg2("HELLO");
    string msg3("hello");
    // Relational Operators (comparing the contents)
    cout << boolalpha;
    cout << (msg1 == msg2) << endl;  // false
    cout << (msg1 == msg3) << endl;  // true
    cout << (msg1 < msg2) << endl;  // false (uppercases before lowercases)
    // Assignment
    string msg4 = msg1;
    cout << msg4 << endl;  // hello
    // Concatenation
    cout << (msg1 + " " + msg2) << endl;  // hello HELLO
    msg3 += msg2;
    cout << msg3 << endl;  // helloHELLO
    // Indexing
    cout << msg1[1] << endl;     // 'e'
    cout << msg1[99] << endl;    // garbage (no index-bound check)
    // cout << msg1.at(99) << endl; // out_of_range exception
}
```

Fig. 09 (Overloaded Operators of Standard Library Class string)

Output:



```
false
true
false
hello
hello HELLO
helloHELLO
e
```

Fig. 10 (Overloaded Operators of Standard Library Class string)

Operators that cannot be Overloaded:

In C++ following operators can't be overloaded:

Operator	Name
.	Member selection
.*	Member selection through pointer to function
::	Scope Resolution Operator
?:	Ternary Operator

Rules and Restrictions on Operator Overloading:

- An operator's precedence cannot be changed by overloading. Parentheses can be used to force the order of evaluation of overloaded operators in an expression.
- An operator's associativity cannot be changed by overloading—if an operator normally associates from left to right, then so do all of its overloaded versions.
- An operator's "arity" (that is, the number of operands an operator takes) cannot be changed—overloaded unary operators remain unary operators; overloaded binary operators remain binary operators. C++'s only ternary operator, ?:, cannot be overloaded. Operators &, *, + and - all have both unary and binary versions that can be separately overloaded.
- Only existing operators can be overloaded—you cannot create new ones.
- You cannot overload operators to change how an operator works on fundamentaltype values. For example, you cannot make the + operator subtract two int's. Operator overloading works only with objects of user-defined types or with a mixture of an object of a user-defined type and an object of a fundamental type.
- Related operators, like + and +=, must be overloaded separately.
- When overloading (), [], -> or any of the assignment operators, the operator overloading function must be declared as a class member. For all other overloadable operators, the operator overloading functions can be member functions or non-member functions.

Overloading Binary Operators:

When we overload an operator which works on two operands, it is known as binary operator overloading. It is used to manipulate the values of two objects of the same class. Following is the syntax of overloading binary operators:

```
return_type classname :: operator op(argument)
{
   // Function Body
}
```

For defining our function inside the class the syntax will be:

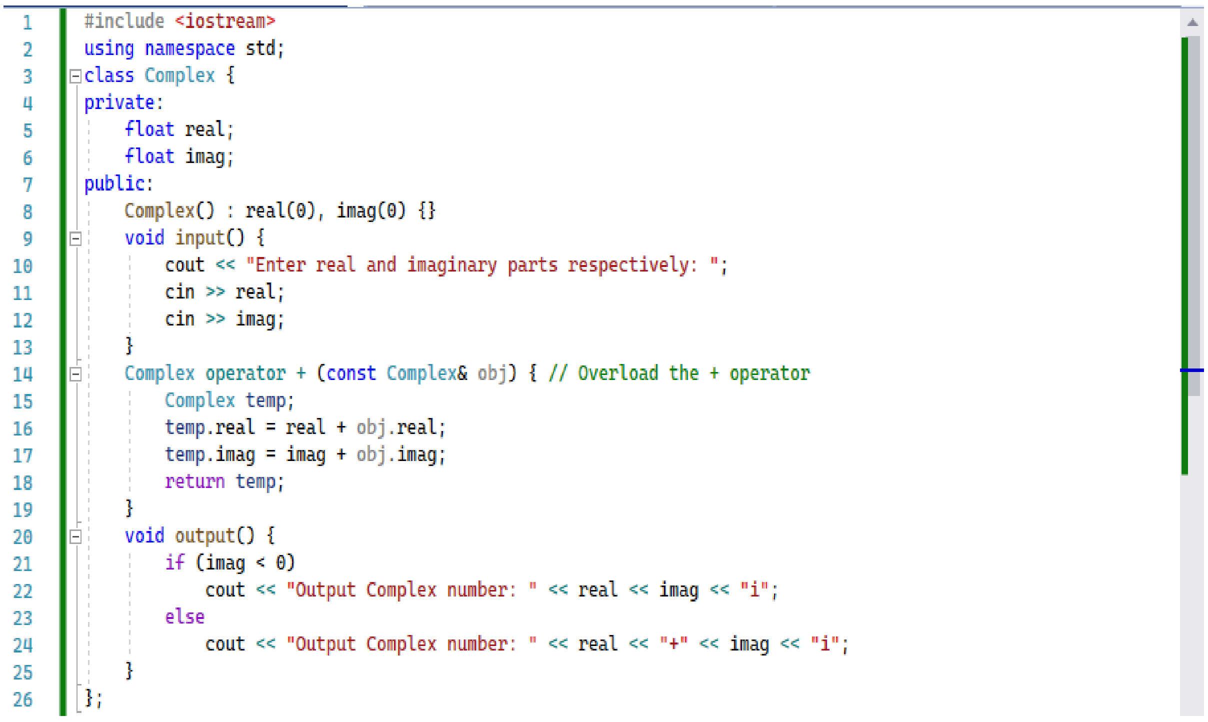
```
return_type operator op(argument)
{
   // Function Body
}
```

It is similar to our function definition in C++. We start with a return data type of our function and follow it with the name of our class. The catch comes after this, instead of writing the function name, we initialize it with the operator we want to “overload”. One more thing to keep in mind is that, unlike our default function, the binary operator overloaded function can only have one argument.

Algorithm:

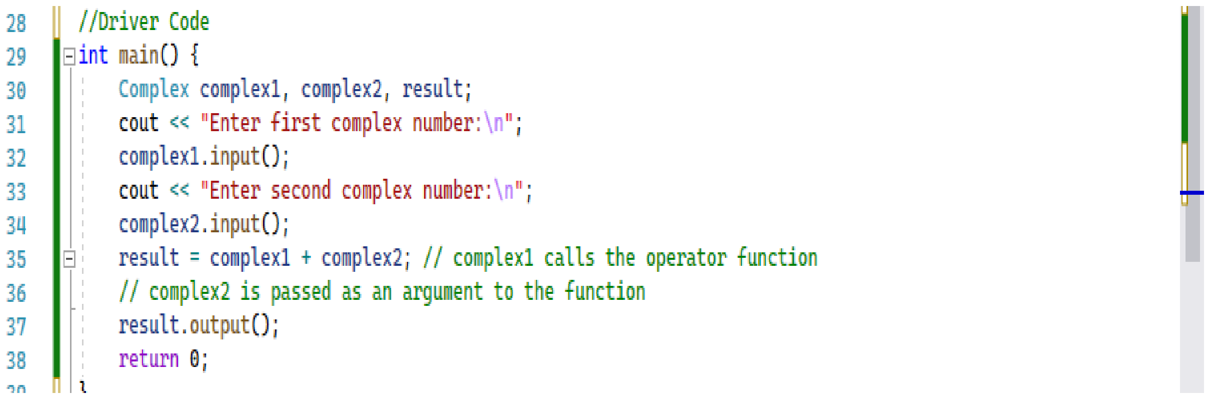
- Start with initializing the class name.
- Declare data members privately & member functions publicly.
- Create binary operator overloaded functions as required. With this, our class definition ends.
    - Decide the operator you want to overload.
    - Use the syntax discussed above to initialize the function.
    - Make sure to create an object of the class where the result will be stored.
    - Complete the function body by manipulating the objects and returning it.
- Start the driver code by initializing objects of the class.
- Make sure to create a resultant object as well. It will store the manipulation done by the binary operator.
- Finally print the resultant object using the output member function.

Example:



```cpp
#include <iostream>
using namespace std;
class Complex {
private:
    float real;
    float imag;
public:
    Complex() : real(0), imag(0) {}
    void input() {
        cout << "Enter real and imaginary parts respectively: ";
        cin >> real;
        cin >> imag;
    }
    Complex operator + (const Complex& obj) { // Overload the + operator
        Complex temp;
        temp.real = real + obj.real;
        temp.imag = imag + obj.imag;
        return temp;
    }
    void output() {
        if (imag < 0)
            cout << "Output Complex number: " << real << imag << "i";
        else
            cout << "Output Complex number: " << real << "+" << imag << "i";
    }
};
```

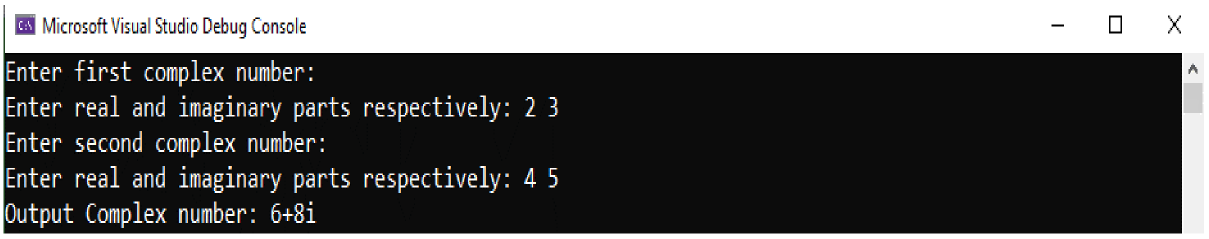
Fig. 10 (Binary Operator Overloading)



```cpp
//Driver Code
int main() {
    Complex complex1, complex2, result;
    cout << "Enter first complex number:\n";
    complex1.input();
    cout << "Enter second complex number:\n";
    complex2.input();
    result = complex1 + complex2; // complex1 calls the operator function
    // complex2 is passed as an argument to the function
    result.output();
    return 0;
}
```

Fig. 11 (Binary Operator Overloading)

Output:



```
Enter first complex number:
Enter real and imaginary parts respectively: 2 3
Enter second complex number:
Enter real and imaginary parts respectively: 4 5
Output Complex number: 6+8i
```

Fig. 12 (Binary Operator Overloading)

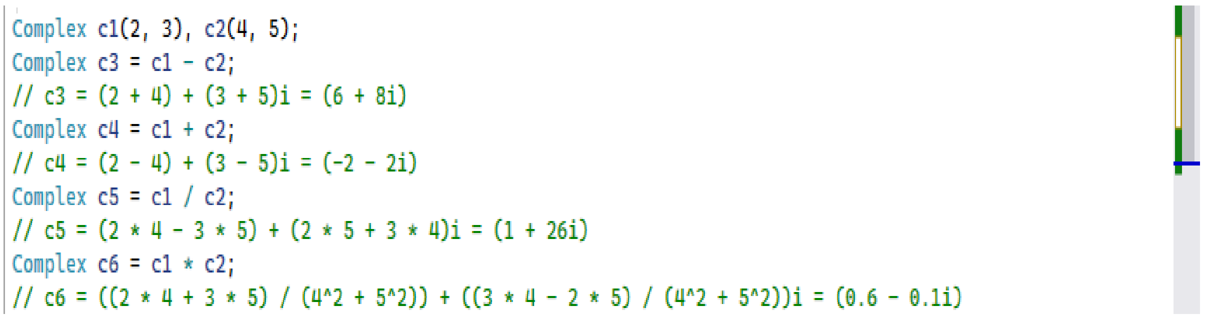
Task 01: Copy Students **[Estimated time 30 minutes / 20 marks]**

- Create a class "Student" with the following attributes:
  - Name (string)
  - Roll number (int)
  - Age (int)
- Implement the copy constructor for the "Student" class. The copy constructor should create a new object that is a copy of an existing "Student" object.
- In main, create a vector of "Student" objects and initialize it with a few student records.
- Create a function that takes a vector of "Student" objects as an argument and prints the details of each student in the vector.
- Create a new vector that is a copy of the original vector using the copy constructor.
- Verify that the copy constructor correctly created a new vector with a copy of each student record.
- Modify one of the student records in the new vector and verify that the changes only affect the new vector and not the original vector.

Task 02: Complex Numbers **[Estimated time 30 minutes / 30 marks]**

- Write a program to make a class of Complex numbers with data members and member functions:
  - real number
  - imaginary number
  - getreal() to get real numbers
  - getimg() to get imaginary numbers
  - show() to show data
  - operator+() "subtract two objects"
  - operator-() "add two objects"
  - operator/()"multiply two objects"
  - operator*() "divide first object by second object".
- Also write a driver code to test the above functionalities

Example Usage:



```
Complex c1(2, 3), c2(4, 5);
Complex c3 = c1 - c2;
// c3 = (2 + 4) + (3 + 5)i = (6 + 8i)
Complex c4 = c1 + c2;
// c4 = (2 - 4) + (3 - 5)i = (-2 - 2i)
Complex c5 = c1 / c2;
// c5 = (2 * 4 - 3 * 5) + (2 * 5 + 3 * 4)i = (1 + 26i)
Complex c6 = c1 * c2;
// c6 = ((2 * 4 + 3 * 5) / (4^2 + 5^2)) + ((3 * 4 - 2 * 5) / (4^2 + 5^2))i = (0.6 - 0.1i)
```

Fig. 12 (In-Lab Task)

Task 03: Matrix Operations **[Estimated time 40 minutes / 30 marks]**

- Create a class Matrix to represent matrices. The class should have two data members:
  - row to represent the number of rows
  - col to represent the number of columns

  - A dynamic 2-D array mat to store the elements of the matrix.
- Overload the following binary operators for the Matrix class:
  - + operator to add two matrices
  - - operator to subtract two matrices
  - * operator to multiply two matrices

Post-Lab Activities:

Task 01: Polynomials **[Estimated time 60 minutes / 40 marks]**

A polynomial is an expression consisting of variables and coefficients e.g. x 2 − 4x + 7. Implement a class of Polynomial with two data members: 1) dynamic array of coefficients and 2) degree. Coefficients are the constant values multiplied with variable in a polynomial. E.g. in x 2 − 4x + 7, coefficient of 'x' is 4. If coef is an array of coefficients then:

coef[0] would hold all coefficients of x^0

coef[1] would hold all x^1

coef[n] = x^n .

Degree is largest exponent of that variable. E.g. . in x 2 − 4x + 7, degree is 2.

For polynomial P = 3X^4 + 4X^2 + 16, if someone tries to access P[4] then it should get 3. Similarly if someone tries to access P[3] it should get 0 as there is no term with degree 3.

Write a C++ program to create a class named **"Polynomial"** which must do the following tasks:

- In default constructor, initialize coefficient array with 0s.
- Implement setter method that takes three parameters,
  - coefficient value (int c, e.g.)
  - index of array at which this value (c) should be stored at
  - degree of whole polynomial
- Implement following functions:
  - Additions of polynomials (operator +)
  - Subtraction of polynomials (operator -)
- Implement the default, parameterized, and copy constructors, destructors and accessor methods as needed.

Main function should do the following:

- Declare two polynomials and set their values ONLY by taking input from the user.
- Declare third polynomial and use copy constructor to set it to polynomial 1's contents. Make sure you perform deep copy otherwise ZERO will be granted for this part!
- Declare fourth polynomial, one by one apply the following functions, and display the resultant fourth object side by side.
- Make sure to have a destructor that deallocates the allocated memory.

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 Part(a): Object Copy	[10 marks]
▪ Task 02 Part(b): Object Copy	[10 marks]
• **Division of In-Lab marks:**	**[80 marks]**
▪ Task 01: Copy Students	[20 marks]
▪ Task 02: Complex Numbers	[30 marks]
▪ Task 03: Matrix Operations	[30 marks]
• **Division of Post-Lab marks:**	**[40 marks]**
▪ Task 01: Polynomial	[40 marks]

References and Additional Material:

- **Copy Constructor**
  https://www.geeksforgeeks.org/copy-constructor-in-cpp/
- **Binary Operator Overloading**
  https://www.javatpoint.com/binary-operator-overloading-in-cpp

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