Department of Computer and Software Engineering- ITU
CE101L: Object Oriented Programming Lab

Course Instructor: Nadir Abbas	Dated: 18/02/2025
Teaching Assistant: Zainab, Nahal, Bilal	Semester: Spring 2025
Session: 2024-2028	Batch: BSCE24

Lab 5. Operator Overloading in Object-Oriented Programming

Name	Roll number	Obtained Marks/35

Checked on: ____________________

Signature: ____________________

Objective

The objective of this lab is to help students understand and apply concepts of operator overloading.

Equipment and Component

Component Description	Value	Quantity
Computer	Available in lab	1

Conduct of Lab

1. Students are required to perform this experiment individually.
2. In case the lab experiment is not understood, the students are advised to seek help from the course instructor, lab engineers, assigned teaching assistants (TA) and lab attendants.

Theory and Background

Operator Overloading

Operator overloading allows built-in operators to be redefined for user-defined types. This makes operations on objects more intuitive and similar to primitive data types.

Constructors and Destructor

- **Default Constructor:** Initializes object properties with default values.
- **Parameterized Constructor:** Allows initializing objects with specific values.
- **Copy Constructor:** Creates a new object by copying an existing one.
- **Destructor:** Automatically called when an object goes out of scope to free resources.

Overloading Arithmetic Operators (+, -,)

Operators such as +, -, and * can be overloaded to perform arithmetic operations on complex numbers:

```
Complex c1(2, 3), c2(1, 4);
Complex c3 = c1 + c2; // Adds two complex numbers
```

Overloading Comparison Operators (==, !=, <, >)

Comparison operators allow logical comparisons between complex numbers:

- == checks if two complex numbers are equal.
- != checks if two complex numbers are different.
- < and > compare the magnitudes of complex numbers.

Overloading Input and Output Operators (<< and >>)

The << operator enables displaying complex numbers, and the >> operator allows user input:

```
cout << c1;  // Outputs: "2 + 3i"
cin >> c2;   // User enters values for real and imaginary parts
```

These operators are implemented as **friend functions** to access private attributes directly.

Tasks

Task 1: Accept the assignment posted in Google Classroom and after accepting clone the repository to your computer for this ensure you have logged into github app with your account.

Task 2: Solve the given problems written after task instructions, write code through IDE like CLion

Task 3: Ensure your code/solution is in the cloned folder.

Task 4: Commit and Push the changes through the Github App

Write code in functions, in files **functions.cpp**, **functions.h** and **main.cpp** after completing each part, verify through running code using **"make run"** on Cygwin.

Question 1

Define a class **Complex** with the following specifications:

Members:

- **real (int)** - represents the real part of a complex number.
- **imaginary (int)** - represents the imaginary part of a complex number.

Constructors:

- **Default Constructor**: Initializes real and imaginary to 0.
- **Parameterized Constructor**: Initializes real and imaginary with given values.
- **Copy Constructor**: Copies values from another Complex object.
- **Destructor**: Releases resources if any.

Question 2

Implement setter and getter functions for each data member of the class:

- **void setReal(int r)**: Sets the real part of the complex number.
- **void setImaginary(int i):** Sets the imaginary part of the complex number.
- **int getReal() const**: Returns the real part.
- **int getImaginary() const**: Returns the imaginary part.

Question 3

Implement the following operator overloads:

- **Complex operator+(const Complex& c):** Overloads the + operator to add two complex numbers.
- **Complex operator-(const Complex& c):** Overloads the - operator to subtract two complex numbers.
- **Complex operator*(const Complex& c):** Overloads the * operator to multiply two complex numbers.

Question 4

Overload the comparison operators:

- **bool operator==(const Complex& c):** Overloads == operator to compare two complex numbers.
- **bool operator!=(const Complex& c):** Overloads != operator to check inequality.

Question 5

Overload additional relational operators:

- **bool operator<(const Complex& c):** Overloads < to compare magnitudes of complex numbers.
- **bool operator>(const Complex& c):** Overloads > to compare magnitudes of complex numbers.

Question 6

Implement input and output stream overloading:

- **friend ostream& operator<<(std::ostream& out, const Complex& c):** Overloads << operator to display a complex number.
- **friend istream& operator>>(std::istream& in, Complex& c):** Overloads >> operator to input a complex number.

Note:

The << and >> operators are overloaded as **friend functions** because they need to access private members of the Complex class. Since ostream and istream objects cannot be members of the Complex class, making them friends allows direct access to private attributes while maintaining encapsulation.

Expected Input and Output for Console Testing:

Example Input:

```
Enter real and imaginary part: 3 4
```

Example Output:

```
Complex number: 3 + 4i
```

Question 7:

Draw a UML class diagram representing the Complex class. Your diagram should include:

- **Class Name**
- **Attributes** (data members)
- **Methods** (constructors, setters, getters, operator overloads, etc.)
- **Access Specifiers** (public/private)

Save the UML diagram as an image and include it in your lab submission folder.

Assessment Rubric for Lab

Performance metric	CLO	Able to complete the task over 80% (4-5)	Able to complete the task 50-80% (2-3)	Able to complete the task below 50% (0-1)	Marks
1. Realization of experiment	1	Executes without errors excellent user prompts, good use of symbols, spacing in output. Through testing has been completed.	Executes without errors, user prompts are understandable, minimum use of symbols or spacing in output. Some testing has been completed.	Does not execute due to syntax errors, runtime errors, user prompts are misleading or non-existent. No testing has been completed.	
2. Conducting experiment	1	Able to make changes and answered all questions.	Partially able to make changes and few incorrect answers.	Unable to make changes and answer all questions.	
3. Computer use	2	Document submission timely.	Document submission late.	Document submission not done.	
4. Teamwork	3	Actively engages and cooperates with other group member(s) in effective manner.	Cooperates with other group member(s) in a reasonable manner but conduct can be improved.	Distracts or discourages other group members from conducting the experiment	
5. Laboratory safety and disciplinary rules	3	Code comments are added and does help the reader to understand the code.	Code comments are added and does not help the reader to understand the code.	Code comments are not added.	
6. Data collection	3	Excellent use of white space, creatively organized work, excellent use of variables and constants, correct identifiers for constants, No line-wrap.	Includes name, and assignment, white space makes the program fairly easy to read. Title, organized work, good use of variables.	Poor use of white space (indentation, blank lines) making code hard to read, disorganized and messy.	
7. Data analysis	4	Solution is efficient, easy to understand, and maintain.	A logical solution that is easy to follow but it is not the most efficient.	A difficult and inefficient solution.	
Total (out of 35):					