Lab 11

Items	Description
Course Title	Object Oriented Programming
Lab Title	Classes (Operator Overloading + Association)
Duration	3 Hours
Tools	Eclipse/ C++
Objective	To get familiar with the use of different concepts in classes in c++

Input and Output Operator Overloading

In C++, operators like `<<` (for output) and `>>` (for input) are commonly used with built-in types (e.g., `int`, `double`). By overloading these operators, we enable the use of `cin` and `cout` to work with user-defined classes. This is especially useful in making custom types behave in a way that feels native in C++.

Key Points:

- **Operator Overloading**: Allows defining how operators should behave with custom classes.
- **Friend Functions**: `<<` and `>>` operators are usually overloaded as friend functions to allow access to private members of the class.

Example Walkthrough:

Consider a `Complex` class to represent complex numbers, which consist of a real and an imaginary part.

1. **Declaring the Class**: We declare `Complex` with two private members, `real` and `imag`, for storing real and imaginary parts of the complex number.
2. **Friend Functions for Operator Overloading**:

- We define `operator<<` to output the `Complex` number in a readable format (e.g., `3 + 4i`).
- We define `operator>>` to take user input for both `real` and `imag`.

3. **Usage in `main`**:
   - When `cin >> c1;` is used, the overloaded `operator>>` prompts the user to input values.
   - When `cout << c1;` is used, the overloaded `operator<<` outputs the values in a formatted way.

```
#include <iostream>

using namespace std;

class Complex {

    int real, imag;

public:

    Complex(int r = 0, int i = 0) : real(r), imag(i) {}

    // Overload the >> operator for input

    friend istream& operator>>(istream &input, Complex &c) {

        cout << "Enter real part: ";

        input >> c.real;

        cout << "Enter imaginary part: ";

        input >> c.imag;
```

```
        return input;
    }
    // Overload the << operator for output
    friend ostream& operator<<(ostream &output, const Complex &c) {
        output << c.real << " + " << c.imag << "i";
        return output;
    }
};
int main() {
    Complex c1;
    cin >> c1; // Uses the overloaded >> operator
    cout << "Complex number: " << c1 << endl; // Uses the overloaded << operator
    return 0;
}
```

Aggregation

Aggregation is a type of association that represents a "has-a" relationship, where one class contains references to objects of another class. The key aspect of aggregation is that the contained objects have an independent lifecycle from the container class.

Key Points:

- **Independence**: The lifetime of the contained object (e.g., `Student`) is independent of the containing class (e.g., `Department`).
- **Non-Ownership**: The container class does not "own" the contained objects; they can exist independently.

Example Walkthrough:

Suppose we have a `Department` class that contains a list of `Student` objects. However, the `Department` does not "own" the `Student` objects—students can exist outside the context of a department.

1. **Declaring `Student`**: The `Student` class has a name.
2. **Declaring `Department`**: The `Department` class has a list of students, implemented as a `vector<Student>`.
3. **Adding Students to the Department**: Students are added to the `Department` by passing a `Student` object.
4. **Displaying Students**: The department can display its students using a member function.

National University
of computer and emerging sciences

```cpp
#include <iostream>

#include <vector>

using namespace std;

class Student {

    string name;

public:

    Student(string n) : name(n) {}

    string getName() const { return name; }

};

class Department {

    vector<Student> students;

public:

    void addStudent(const Student& s) {

        students.push_back(s);

    }
```

National University
of computer and emerging sciences

```
    void displayStudents() const {
        for (const auto& student : students) {
            cout << student.getName() << endl;
        }
    }
};
int main() {
    Department dept;
    Student s1("Alice"), s2("Bob");
    dept.addStudent(s1);
    dept.addStudent(s2);
    cout << "Department Students:" << endl;
    dept.displayStudents();
    return 0;
}
```

Composition

Composition also represents a "has-a" relationship but with stronger ownership semantics. Here, the container class owns the lifecycle of the contained objects. If the container object is destroyed, the contained objects are also destroyed.

Key Points:

- **Dependence**: The contained object cannot exist independently of the container.
- **Strong Ownership**: The lifecycle of the contained object (e.g., `Engine`) is directly tied to the lifecycle of the container (e.g., `Car`).

Example Walkthrough:

Consider a `Car` class that contains an `Engine`. If the `Car` object is destroyed, the `Engine` will be destroyed as well. This shows that the `Engine` is an essential part of `Car` and cannot exist without it.

1. **Declaring `Engine`**: The `Engine` class represents the engine of a car.
2. **Declaring `Car`**: The `Car` class contains an `Engine` object directly.
3. **Object Creation and Destruction**: When `Car` is created, its `Engine` is also created. When `Car` goes out of scope, its `Engine` is automatically destroyed.

```
#include <iostream>

using namespace std;

class Engine {

public:

    Engine() {

        cout << "Engine created." << endl;

    }
```

```
    ~Engine() {
        cout << "Engine destroyed." << endl;
    }
};
class Car {
    Engine engine; // Engine is part of the Car, created and destroyed with it
public:
    Car() {
        cout << "Car created." << endl;
    }
    ~Car() {
        cout << "Car destroyed." << endl;
    }
};
```

```
int main() {

    Car myCar;

    return 0;

}
```

In this code:

- When `myCar` is created, the `Car` constructor is called, which in turn initializes `Engine` because `engine` is a member of `Car`.
- When `myCar` goes out of scope, the `Car` destructor is called, and the `Engine` destructor is automatically called afterward.

National University
of computer and emerging sciences

Lab Task

1. Task for Input/Output Operator Overloading

- **Task**: Create a class `Date` that represents a date with day, month, and year attributes. Overload the `>>` operator to take date input in the format `DD/MM/YYYY` and the `<<` operator to display the date in a long format, e.g., "1st of January, 2023".
- **Requirements**:
    - Overload the `>>` operator to parse a string date input and set `day`, `month`, and `year`.
    - Overload the `<<` operator to output the date in a human-readable form, converting the numeric month into its name.
    - **Extra Challenge**: Add logic to handle valid date ranges for day and month (e.g., day should be between 1-31, month between 1-12) and validate leap years for February.

2. Task for Aggregation

- **Task**: Design a `Library` class that contains a collection of `Book` objects. Each `Book` has attributes like `title`, `author`, and `ISBN`. Implement methods in `Library` to add books, find a book by ISBN, and display the list of books. Demonstrate that the `Library` aggregates `Book` objects but does not "own" them (i.e., books can exist independently).
- **Requirements**:
    - Implement a `Library` class with methods for adding books, searching for a book by its ISBN, and displaying all books.
    - Use a collection (e.g., `vector<Book>`) to hold books in the library.
    - **Extra Challenge**: Add a feature to count books by a particular author and to remove a book by ISBN from the library.

3. Task for Composition

- **Task**: Create a `House` class that includes two other classes: `Room` and `Address`. Each `House` has a specific number of `Room` objects and one `Address` object. The `Address` should contain street name, city, and postal code, while `Room` should have a type (e.g., "Bedroom", "Kitchen") and area in square feet.
- **Requirements**:
    - Implement the `House` class to contain a fixed number of `Room` objects created within it and an `Address` object as part of its composition.
    - Include methods in `House` to display its address and all room details.
    - **Extra Challenge**: Implement a method to calculate the total area of all rooms in the house, demonstrating the "tight coupling" between the house and its rooms.