Lab 2

Instructions for Lab Submission:

1. File Naming and Submission:
    a. Submit your solution as a zip file named **Lab2_<ERP>.zip**, where *<ERP>* is your ERP number. For example, if your ERP number is *12345*, the file should be named *Lab2_12345.zip*.
    b. Ensure that the zip file:
        i. Contains the following directories:
            1. ***headers/***: All header files, named as ***<class/struct name>.hpp***
            2. ***src/***: All implementation files named as ***<class/struct name>.cpp***
            3. ***main/***: All files with main function for each task named as ***Task<number>.cpp***
        ii. Does not include unnecessary files like executables.
        iii. Includes the ***.vscode/*** directory with the ***tasks.json*** file if used
        iv. Includes a **.txt** file containing explanations where asked.
    c. Do not submit code in ***TEXT*** files.
2. Code Organization:
    a. Each class or struct used in a task must have its declaration in a *.hpp* file (in the ***headers/*** directory) and its implementation in a *.cpp* file (in the ***src/*** directory). The logic and function calls should be written in a separate main file (in the ***main/*** directory).
    b. Ensure that the code structure is modular and adheres to the principles of good software development (e.g., separation of concerns).
3. Code Neatness:
    a. Ensure your code is well-formatted and easy to read. Use proper indentation and meaningful variable names.
    b. Include appropriate comments to explain the logic where necessary (especially for functions and complex sections of the code).
    c. **Add clear explanations where required(or specified)** to make the logic of your program easy to follow.
4. Code Compilation and Functionality:
    a. Ensure your code compiles without errors or warnings. You should be able to compile and run your code on any system with a standard C++ compiler (e.g., GCC, Clang, MSVC).
    b. Double-check that the functionality of the program meets the requirements outlined in the task.
5. Additional Instructions:
    a. Do not use any libraries or features that have not been covered in the course material, unless specifically instructed.
    b. Make sure your code adheres to the principles of good software development (e.g., modular functions, minimal repetition, and no hardcoding).
    c. If you encounter any issues or require clarifications, reach out before the submission deadline.
6. Deadline:
    a. Ensure that you submit your lab on time, before the deadline. Late submissions will not be accepted
    b. Double-check that the zip file is correctly named and contains all necessary files before submitting.
7. Plagiarism Policy:
    a. The work submitted must be entirely your own. Any code or content copied from others (including online sources, classmates or AI) will result in an automatic zero for the task.

Lecture Review

Copy Constructor:

A copy constructor is a special constructor in C++ used to create a new object by copying the contents of an existing object. It is automatically invoked when an object is passed by value, returned from a function, or explicitly copied.

The syntax for a copy constructor is as follows:

```
ClassName(const ClassName& other) { /* Implementation */ }
```

- ClassName: The name of the class.
- other: The object being copied (passed by reference).

The copy constructor's purpose is to initialize a new object as a copy of an existing object. Here's an example:

```
class MyClass {
    int data;
public:

    // Constructor
    MyClass(int val) : data(val) {}

    // Copy constructor
    MyClass(const MyClass& other) {
        data = other.data;
    }
};
```

In the above example, the copy constructor ensures that the data of the new object is initialized to the value of data from the other object.

Shallow Copy:

A shallow copy simply copies the values of the member variables from one object to another. If the class contains pointers or dynamically allocated memory, both the original and copied objects will point to the same memory location, which can lead to issues like double deletion or data corruption. Example of shallow copy:

```
class MyClass {
    int* ptr;
public:

    // Constructor
    MyClass(int value) {
        ptr = new int(value);
    }

    // Copy constructor (shallow copy)
    MyClass(const MyClass& other) {
        ptr = other.ptr;  // Just copying the pointer
    }
};
```

Deep Copy:

A deep copy, on the other hand, creates a new copy of the data in the heap memory. This ensures that the original object and the copied object do not share the same memory locations, preventing issues like double free or unintended data modification. Example of deep copy:

```
class MyClass {
    int* ptr;
public:

    // Constructor
    MyClass(int value) {
        ptr = new int(value);
    }

    // Deep copy constructor
    MyClass(const MyClass& other) {
        ptr = new int(*(other.ptr));  // Creating a new memory location
    }

    // Destructor
    ~MyClass() {
        delete ptr;
    }
};
```

In this case, the copy constructor allocates new memory for the ptr in the copied object, preventing sharing of the pointer between the original and copied objects.

Friend Functions:

A **friend function** is a function (or method) that is allowed to access the private and protected members of a class. Even though the function is not a member of the class, it is still granted special access to the class's internals.

Friend functions are typically used when you need a function outside of the class to interact with its private or protected members.

Key points about friend functions:

- They are not members of the class but are declared inside the class using the friend keyword.
- They can access all private and protected members of the class, just like member functions can.
- They can be regular functions, other classes' methods, or even entire classes.

Example:

```
#include <iostream>

class MyClass {
private:
    int privateData;

public:
    MyClass(int value) : privateData(value) {}

    // Declare a friend function
    friend void accessPrivateData(MyClass& obj);
};

// Friend function can access private members of MyClass
void accessPrivateData(MyClass& obj) {
    std::cout << "The private data is: " << obj.privateData << std::endl;
}

int main() {
    MyClass obj(123);
    accessPrivateData(obj);  // Friend function accessing private member
    return 0;
}
```

Explanation:

- The class MyClass has a private member privateData.
- The function accessPrivateData is declared as a friend of the class using the friend keyword.
- Even though accessPrivateData is not a member of MyClass, it can access the private data directly because it's a friend of the class.

When to use friend functions:

- When you want to allow a function (outside of the class) to access private or protected members of the class.
- They're useful for operator overloading, especially when you need to access the internals of two different classes for operations like addition or comparison.

Overloading the '=' Operator

The assignment operator (=) is used to assign one object to another. By default, C++ performs a shallow copy when the = operator is used, which can lead to problems when dealing with dynamic memory. Therefore, it's common to overload the assignment operator to ensure a deep copy.

The syntax for overloading the = operator is:

```
ClassName& operator=(const ClassName& other) { /* Implementation */ }
```

Operator Overloading(Recap):

Operator overloading allows defining how operators like +, -, *, etc., behave for user-defined types. This helps make custom types more intuitive to use.

Basics of Operator Overloading

- Operators are overloaded using the **operator** keyword.
- Overloading can be done as a member function or as a non-member friend function.
- Some operators (e.g., =, (), [], ->) can only be overloaded as member functions.

General Syntax

```
ReturnType operatorSYMBOL(/* Arguments */) {
    // Define operation
}
```

Example: Overloading the + Operator

```
class MyINT {
private:
    int x;

public:
    MyINT(int x = 0) : x(x) {}

    // Overloading + operator
    MyINT operator+(const MyINT& other) {
        return MyINT(x + other.x);
    }

};

int main() {
    MyINT obj1(10), obj2(5);
    MyINT obj3 = obj1 + obj2;  // Using overloaded + operator

    return 0;
}
```

Example: Overloading the >> and << operators

```
class MyFLOAT {
private:
    float x;

public:
    MyFLOAT(float x = 0) : x(x) {}

    // Overloading >> operator
    friend std::istream& operator>>(std::istream& in, MyFLOAT& obj) {
        in >> obj.x;
        return in;
    }

    // Overloading << operator
    friend std::ostream& operator<<(std::ostream& out, const MyFLOAT& obj) {
        out << "x: " << obj.x;
        return out;
    }
};
```

A **friend function** allows an operator to access the private and protected members of a class while being defined outside the class. This is particularly useful for **binary operators** (e.g., +, *, <<, >>) when:

- The operator needs to handle two operands, where the **left-hand operand** is not an object of the class (e.g., *2.3* MyFLOATObject*).
- The operator requires direct access to private or protected data members of the class.

By declaring the operator as a friend, it can "bypass" normal encapsulation to perform its task effectively.

```
class SomeClass {
    // Data members
public:
    // Overloading + operator: RetunType Object = SomeClass + RightHandOperand
    ReturnType operator+(const RightHandOperand& other) { /* implementation */ }

    // When the left-hand operand is a different data type: ReturnType Object = LeftHandOperand + SomeClass
    friend ReturnType operator+(const LeftHandOperand& lho, const SomeClass& rho) { /* implementation */ }

};
```

Lab Tasks

Task 1

Define a class `DynamicArray` that contains:

- A **dynamically allocated integer array**.
- A **constructor** to initialize the array with a given size.
- A **method** **`set(int index, int value)`** to set a value at a specific index.
- A **method** **`get(int index)`** to retrieve a value at an index.
- A **default copy constructor** (compiler-generated) to demonstrate **shallow copying**.
- A **manual deep copy constructor** that allocates new memory and copies elements individually.

In the main function:

1. Create an object and fill it with values.
2. Copy it using the default copy constructor (shallow copy).
3. Modify the original object and observe the effect on the copied object.
4. Implement a **deep copy constructor** and repeat step 3 to confirm that the copied object is independent.

Question: Why does modifying the original object affect the shallow copy but not the deep copy?

```
PS D:\OOP Github\Object-Oreinted-Programming-CPP\lab-solutions\Lab2> .\Task1.exe
Creating Array 1
10 20 30 40 50
Using shallow copy...
10 20 30 40 50
Modifying arr1...
11 22 33 44 55
Array 2 after the modifying Array 1:
10 20 30 40 50
```

Task 2

Extend the `DynamicArray` class by implementing:

- An overloaded assignment operator (`operator=`) that performs a deep copy.

In the main function:

1. Create two DynamicArray objects and use `operator=` to assign one object to another.
2. Modify one of them and verify that the other remains unchanged.

Note: Put a self-assignment check to prevent the data deleting itself during copying.

```
PS D:\OOP Github\Object-Oreinted-Programming-CPP\lab-solutions\Lab2> .\Task2.exe
Creating Array 1
10 20 30 40 50
Copying Array...
10 20 30 40 50
Modifying Array 1...
11 22 33 44 55
Array 2 after the modifying Array 1:
10 20 30 40 50
```

Task 3:

Create a class `Fraction` that represents a mathematical fraction with:

- Two private attributes: `numerator` and `denominator`.
- A friend function `addFractions(const Fraction&, const Fraction&)` that adds two `Fraction` objects and returns a new `Fraction` object.
- A friend function `multiplyFractions(const Fraction&, const Fraction&)` that multiplies two `Fraction` objects.

In the main function:

- Define two `Fraction` objects.
- Use the friend functions to add and multiply them.
- Print the results using a `display()` method.

Question: Why do we use a friend function instead of a regular member function here?

```
PS D:\OOP Github\Object-Oreinted-Programming-CPP\lab-solutions\Lab2> .\Task3.exe
Fraction 1: 1/2
Fraction 2: 3/4
Sum of the fractions: 10/8
Product of the fractions: 3/8
```

Task 4:

Implement a Matrix2x2 class in C++ that supports the following operations using operator overloading:

1. Matrix Addition (+): Add two 2x2 matrices.
2. Matrix Subtraction (-): Subtract one 2x2 matrix from another.
3. Matrix Multiplication (*): Multiply two 2x2 matrices.
4. Output (<<): Overload the << operator to display the matrix in a readable format.

Note:

For Matrix Addition and subtraction, add and subtract the numbers in the same positions.

For matrix multiplication: Multiply rows of the first matrix by columns of the second matrix and add the results.

Step-by-Step:

1. **Take the first row of the first matrix and the first column of the second matrix.**
2. **Multiply the corresponding numbers and add them together.**
3. **Repeat for all rows and columns.**

```
PS D:\OOP Github\Object-Oreinted-Programming-CPP\lab-solutions\Lab2> .\Task4.exe
Matrix 1:
1 2
3 4
Matrix 2:
5 6
7 8
Sum of Matrix 1 and Matrix 2:
6 8
10 12
Difference of Matrix 1 and Matrix 2:
-4 -4
-4 -4
Product of Matrix 1 and Matrix 2:
19 22
43 50
```

Task 5:

Implement a program for a mechanic shop that operates on different cars. The program should include the following features:

1. `Car` Class:
   - Each car has a make, model, year, and a dynamically allocated array to store repair costs (to demonstrate deep copy).
   - Provide a constructor, destructor, copy constructor, and assignment operator to ensure proper memory management.
2. `MechanicShop` Class:
   - The mechanic shop maintains a list of cars and performs operations on them.
   - Use a dynamically allocated array to store the cars (to demonstrate deep copy).
3. Friend Function:
   - Implement a friend function calculateTotalRepairCost that calculates the total repair cost for a given car.
4. Operator Overloading:
   - Overload the + operator to combine the repair costs of two cars into a new car (simulating merging repair histories).
   - Overload the << operator to display car details and repair costs.
5. Deep Copy:
   - Ensure that all dynamically allocated memory (e.g., repair costs, list of cars) is properly copied using deep copy in the copy constructor and assignment operator.

```
PS D:\OOP Github\Object-Oreinted-Programming-CPP\lab-solutions\Lab2> .\Task5.exe
Cars in the shop:
Car: Suzuki Mehran (2000)
Repair Costs: 500 405.99
Total Repair Cost: 905.99

Car: Suzuki Alto (1979)
Repair Costs: 458.78 259.99 300.2
Total Repair Cost: 1018.97


Merged Car Repair History:
Car: Suzuki Mehran (2000)
Repair Costs: 500 405.99 458.78 259.99 300.2
Total Repair Cost: 1924.96
```