CS224 Object Oriented Programming and Design Methodologies Lab Manual

Lab 7- Operator Overloading

Department of Computer Science

Dhanani School of Science and Engineering

Habib University

Fall 2025

Contents

Lab 7

Operator Overloading

7.1 Guidelines

1. Use of AI is strictly prohibited. This is not limited to the use of AI tools for code generation, debugging, or any form of assistance. If detected, it will result in immediate failure of the lab, student will be awarded 0 marks, reported to the academic integrity board and appropriate disciplinary action will be taken.
2. Absence in lab regardless of the submission status will result in 0 marks.
3. All assignments and lab work must be submitted by the specified deadline on Canvas. Late submissions will not be accepted.

7.2 Objectives

Following are the lab objectives of this lab:

1. Practice classes in C++.
2. Understand operator overloading.

7.3 Directory Structure

Labs will have following directory structure:

```
lab
├── manual
│   └── lab.pdf
├── src
│   ├── *.hpp
│   └── *.cpp
├── tests
│   ├── CMakeList.txt
│   └── *.cpp
├── makefile
└── CMakeList.txt
```

`manual` will contain the lab manual pdf. `src` will contain the source code files, and `tests` contains the test files. `makefile` contains the makefile for testing and running.

7.4 Operator Overloading

Just as a function can be overloaded, an operator can also be overloaded and perform different tasks. In fact, you've already encountered examples of operator overloading. For example, the * operator when applied to two numbers yields their product, while when applied to a pointer it yields the value stored in the address that the pointer points to. The compiler checks the number and type of arguments to determine how to use the operator. C++ allows us to extend the concept of operator overloading to user defined types. For example, we can overload the + operator to add two objects of our own class type. To overload an operator we declare an overloaded function using the `operator` keyword followed by the operator to be overloaded. The name of the function is `operator`op. The "op" can be any of the C++ operators (e.g., +, -, ...) with some exceptions. Like any ordinary function, it has a body, return type, and can accept parameters. The number of parameters is the same as the number of the operator's operands. That is, the overloaded function of a binary operator accepts two parameters, while that of a unary operator accepts one. As we said, operator overloading is usually applied to classes so that they operate in a more natural way. For example, the following program defines the `Rect` class, which contains a function that overloads the + operator:

```
#include <iostream>

class Rect {
private:

    float length, height;

public:

    Rect() = default;                       // Default constructor
    Rect(float l, float h);                 // Parameterized constructor
    Rect  operator+(const Rect& r) const;   // Overload + operator
    float area() const;                     // Calculate area
    void  show() const;                     // Display length and height
};

Rect::Rect(float l, float h) : length(l), height(h) {}

Rect Rect::operator+(const Rect& r) const {
    Rect tmp;
    tmp.length = length + r.length;
    tmp.height = height + r.height;
    return tmp;
}

float Rect::area() const { return length * height; }

void Rect::show() const {
    std::cout << "L:" << length << " H:" << height << std::endl;
}

int main() {
    Rect r1(10.0f, 20.0f), r2(30.0f, 40.0f), r3;
    r3 = r1 + r2;  // Equivalent to r3 = r1.operator+(r2);
    r3.show();

    return 0;
}
```

Listing 7.1: `Rect.cpp`, slightly modified example 18.1 from [1]

Before discussing the operator overloading, let's answer, since the `length` and `height` members are private, why the compilation does not fail when those members are accessed in the `operator+` function (e.g., `r.length`)? The answer is that C++ allows an object to access all the members of other objects of the same class, including the private members.

When the compiler encounters the `r1+r2` statement it examines the types of operands and realizes that it must call the corresponding `operator+` function. Specifically, the `r1+r2` statement is interpreted as `r1.operator+(r2)`, so the `operator+` function of the `r1` object is called with argument the `r2` object. In general, when the overloaded function of a binary operator is declared in the class, the left operand (e.g., `r1`) is the object that calls it, while the right operand (e.g., `r2`) is passed as an argument to the function. Note that if the `operator+` function was not declared, the compiler would display an error message for an illegal action, since objects cannot be added. Alternatively, we could explicitly call the function and write `r1.operator+(r2)`. Naturally, the usual use of the operator is an easier to read and write abbreviation of the explicit call. The function returns a new object with dimensions equal to the sum of the corresponding dimensions of the two objects. So, the program displays `40` and `60`.

The benefit of operator overloading is that the operation of the class looks more natural and the program becomes easier to read and write. It is programmatically more elegant for the class developer to provide such convenient services to class users than making them to implement those services. Yes, it does simplify the work of the class user (e.g., with statements like `r1+r2`), but it gives a hard time to the class programmer as (s)he has to write the operator functions to support these "natural" operations. After all, as we said, one of the goals of C++ was to make class objects behave like basic types.

7.5 Testing

We have provided two ways to test the code: manual drivers and automated tests. It is preferred to use the automated tests, if you can, because they are more comprehensive and will help you find more bugs. However, if you are not able to use the automated tests, you can use the manual drivers.

7.5.1 Manual Drivers

Manual drivers are provided in the `src` directory, their names start with `main_*.cpp` (e.g., `main_MixedFraction.cpp`). Use following command to compile and run a manual driver,

```
>>> g++ main_MixedFraction.cpp MixedFraction.cpp -o main_MixedFraction
>>> ./main_MixedFraction
Running MixedFraction Tests...
[PASS] AdditionOperator
[PASS] SubtractionOperator
[PASS] MultiplicationOperator
[PASS] DivisionOperator
[PASS] PlusEqualOperator
[PASS] MinusEqualOperator
[PASS] TimesEqualOperator
[PASS] DivideEqualOperator
[PASS] EqualityOperator
[PASS] InequalityOperator
[PASS] Normalization

----------------------------------------
Total: 11 | Passed: 11 | Failed: 0
----------------------------------------
```

7.5.2 Automated Tests

Automated tests are provided in the `tests` directory. You can only do automated testing if you are using Linux, WSL, or MacOS. If you are using Windows, you can use WSL. To do automated testing, first install `googletest` library, `make`, and `cmake`, with the commands below,

```
# For Ubuntu/Debian based systems and WSL
sudo apt-get install -y libgtest-dev make cmake

# For MacOS
# run following command as super user
brew install make cmake
git clone https://github.com/google/googletest
cd googletest
mkdir build
cd build
cmake .. -DCMAKE_CXX_STANDARD=17
make
make install
```

Then, go to the root directory of the lab and run the following commands to compile and run the tests,

```
make TEST=TIME           # To test Time
make TEST=VECTOR2D       # To test Vector2D
make TEST=MIXEDFRACTION # To test MixedFraction
make                     # To test all
```

If you get any error, make sure you run `make clean` first, then try again.

7.6 Exercises

1. [25 points] You are required to implement the `Time` class defined in the provided header file (`src/Time.hpp`). This class represents a specific time of the day (in hours, minutes, and seconds) and supports operator overloading for arithmetic and comparison operations.

   Your task is to:

   - Implement the constructors.
   - Correctly define all the overloaded operators. See header file for the list of operators to overload.
   - Handle addition and subtraction between `Time` objects and between `Time` and `int` (seconds).
   - Ensure that all time values are normalized (e.g., 70 seconds → 1 minute 10 seconds).
   - Implement a `show()` method that displays the time in a human-readable format, e.g. `HH:MM:SS`.

   Constraints:

   - You are not allowed to use any standard time libraries like `<ctime>` or `<chrono>`. If you do so, you will get zero points for this exercise.
   - For subtraction assume that the first time is always greater than or equal to the second time.
   - $0 \leq$ `hours`, `minutes`, `seconds`
   - Prefer integer arithmetic only.
   - Ensure all overloaded operators return new `Time` objects (do not modify the operands).

2. [25 points] Implement the `Vector2D` class defined in the provided header file (`src/Vector2D.hpp`). This class represents a two-dimensional vector and supports various vector operations through operator overloading. Your task is to:

   - Implement the constructors.
   - Correctly implement all the overloaded operators. See header file for the list of operators to overload.
   - Implement vector addition, subtraction, scalar multiplication, dot product, and equality comparison.

   Constraints:

   - Ensure all overloaded operators return new `Vector2D` objects (do not modify the operands), except for the compound assignment operators (e.g., `+=`, `-=`).
   - Assume all scales are non-zero.

3. [50 points] You are required to implement the `MixedFraction` class defined in the provided header file (`src/MixedFraction.hpp`). This class represents a mixed fraction (a whole number and a proper fraction) and supports various arithmetic operations through operator overloading.

   Your task is to:

   - Implement the constructors.
   - Correctly define all the overloaded operators. See header file for the list of operators to overload.
   - Handle addition, subtraction, multiplication, and division between `MixedFraction` objects and between `MixedFraction` and `int`.
   - Ensure that all fractions are stored in their simplest form (e.g., 2/4 should be stored as 1/2).

   Constraints:

   - Ensure all overloaded operators return new `MixedFraction` objects (do not modify the operands), except for the compound assignment operators (e.g., `+=`, `-=`).
   - Prefer integer arithmetic only.
   - Assume all denominators are non-zero.

References

[1] G.S. Tselikis. *Introduction to C++*. CRC Press, 2022. ISBN: 9781000635744. URL: https://books.google.com.pk/books?id=LNx1EAAAQBAJ.