МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСОЙ ФЕДЕРАЦИИ МОСКОВСКИЙ АВЦИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЬЕЛЬСКИЙ УНИВЕРСТИТЕТ)

ЛАБОРАТОРНАЯ РАБОТА №2 по курсу

объектно-ориентированное программирование I семестр, 2019/20 уч. год

Студент *Попов Данила Андреевич, группа 08-208Б-18*

Преподаватель *Журавлёв Андрей Андреевич*

Условие

Задание №1: написать класс, который реализует комплексное число в алгебраической форме со следующими функциями:

1. Сложение
2. Вычитание
3. Умножение
4. Деление
5. Сравнение
6. Сопряжённое число

Описание программы

Код программы состоит из 3-х файлов:

1. apps/main.cpp: файл, содержащий точку входа приложения
2. include/lab/complex.hpp: файл, содержащий объявление и реализацию inline-функций
3. src/complex.cpp: реализация не-inline методов класса Complex.h

Недочёты

Метод Str() работает не с внешним буффером, а создаёт каждый раз минимум два:

1. Для std::stringstream объекта, который предоставляет удобный интерфейс приведения стандартных типов к строке.
2. Для std::string объекта, который является возвращаемым значением.

Данный метод может использовать достаточно много процессорного времени при приведении объектов типа Complex к строке.

Выводы

В CMake с зависимостями очень сложно работать, в отличие от, например, системы модулей для Golang. Для того, чтобы сделать автоматическую загрузку google test, потребовалось немало часов изучения мануалов.

Исходный код

Complex.hpp

```
#pragma once

#include <cmath>
#include <cassert>
#include <string>
#include <sstream>
#include <ostream>

class Complex;

class Complex {
public:
    Complex() = default;

    explicit Complex(double real, double imaginary) noexcept
        : re{ real }
        , im{ imaginary }
    {}

    Complex(const Complex& other) noexcept
        : re(other.re)
        , im(other.im)
    {}


    double& Real() noexcept { return re; }
    const double& Real() const noexcept { return re; }

    double& Imag() noexcept { return im; }
    const double& Imag() const noexcept { return im; }

    Complex operator+(const Complex& other) const noexcept {
        Complex tmp = *this;
        tmp.Add(other);
        return tmp;
    }

    Complex operator-(const Complex& other) const noexcept {
        Complex tmp = *this;
```

```
    tmp.Sub(other);
    return tmp;
}

Complex operator*(const Complex& other) const noexcept {
    Complex tmp = *this;
    tmp.Mul(other);
    return tmp;
}

Complex operator/(const Complex& other) const noexcept {
    Complex tmp = *this;
    tmp.Div(other);
    return tmp;
}

Complex operator/(const double deno) const noexcept {
    return Complex{ re / deno, im / deno };
}

Complex& operator+=(const Complex& other) noexcept {
    Add(other);
    return *this;
}

Complex& operator-=(const Complex& other) noexcept {
    Sub(other);
    return *this;
}

Complex& operator*=(const Complex& other) noexcept {
    Mul(other);
    return *this;
}

Complex& operator/=(const Complex& other) noexcept {
    Div(other);
    return *this;
}

Complex& operator/=(const double deno) noexcept {
    return *this = *this / deno;
```

```
    }

    bool operator==(const Complex& other) noexcept {
        return (re == other.re) && (im == other.im);
    }

    double& operator[](size_t ix) {
        assert(ix < (sizeof(values) / sizeof(*values)));
        return values[ix];
    }

    const double& operator[](size_t ix) const {
        assert(ix < (sizeof(values) / sizeof(*values)));
        return values[ix];
    }

    // IO methods
    void Read(std::istream& stream);
    void Write(std::ostream& stream) const;

private:
    void Add(const Complex& other) noexcept {
        this->re += other.re;
        this->im += other.im;
    }

    void Sub(const Complex& other) noexcept {
        Add(Complex{ -other.re, -other.im });
    }

    void Mul(const Complex& other) noexcept {
        double re = this->re * other.re - this->im * other.im;
        double im = this->re * other.im + this->im * other.re;
        this->re = re;
        this->im = im;
    }

    void Div(const Complex& other) noexcept {
        double denominator = (other * other.Conj()).re;
        Complex numerator = *this * other.Conj();
        *this = numerator / denominator;
    }
```

```
    bool Equ(const Complex& other) noexcept {
        return (this->re == other.re) && (this->im == other.im);
    }

    Complex Conj() const noexcept {
        return Complex{ this->re, -this->im };
    }

    double Mod() const noexcept {
        return std::sqrt((*this * this->Conj()).re);
    }

    std::string Str() const {
        std::stringstream string;
        Write(string);
        return string.str();
    }

    union {
        double values[2];
        struct {
            double re, im;
        };
    };

    friend int Compare(const Complex& left, const Complex& right);
};

inline std::ostream& operator<<(std::ostream& out, const Complex& c) {
    c.Write(out);
    return out;
}

inline std::istream& operator>>(std::istream& input, Complex& c) {
    c.Read(input);
    return input;
}

int Compare(const Complex& left, const Complex& right);
```

Complex.cpp

```cpp
#include <lab/complex.hpp>

void Complex::Read(std::istream& stream) {
    stream >> re >> im;
}

void Complex::Write(std::ostream& stream) const {
    stream << re << " " << im;
}

int Compare(const Complex& left, const Complex& right) {
    double leftMod = left.Mod();
    double rightMod = right.Mod();

    return (leftMod < rightMod)
        ? -1
        : (leftMod == rightMod
            ? 0
            : 1);
}
```

main.cpp

```cpp
// stdlib headers:
#include <cstdio>
#include <iostream>
#include <string>

// LabLib headers:
#include <lab/complex.hpp>

using namespace std;

void ToUpper(string& str);

int main() {
    string input;
    while (cin) {
        cin >> input;
        ToUpper(input);
        Complex left, right;
        left.Read(cin);
        right.Read(cin);

        if (cin.fail()) {
            break;
        }

        if (input == "ADD") {
            cout << left + right;
        }
        else if (input == "SUB") {
            cout << left - right;
        }
        else if (input == "MUL") {
            cout << left * right;
        }
        else if (input == "DIV") {
            cout << left / right;
        }
        else if (input == "EQU") {
            cout << ((left == right) ? "True" : "False");
        }
        else if (input == "CMP") {
```

```
            cout << Compare(left, right);
        }
        cout << endl;
        cout.flush();
    };
}

void ToUpper(string& str) {
    for (auto& c : str) {
        c = static_cast<remove_reference_t<decltype(c)>>(toupper(c));
    }
}
```