Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное

учреждение

высшего профессионального образования «Национальный исследовательский Нижегородский государственный

университет им. Н. И. Лобачевского»

**Институт информационный технологий, математики и механики**

Лабораторная работа №3 по курсу «Информатика и

программирование»

Выполнил: студент группы 381908-3

Борисов Максим

Преподаватель:

Карчков Д. А.

г. Нижний Новгород

2020

**Оглавление**

[**Теория** 3](#_Toc34820028)

[**Описание работы конструктора:** 3](#_Toc34820029)

[**Описание работы деструктора:** 3](#_Toc34820030)

[**Список реализованных методов:** 3](#_Toc34820031)

[**Код программы:** 4](#_Toc34820032)

[Файл Complex.h 4](#_Toc34820033)

[Файл Complex.cpp 4](#_Toc34820034)

[Файл Fraction.h 13](#_Toc34820035)

[Файл Fraction.cpp 14](#_Toc34820036)

[Файл Lab1.cpp 16](#_Toc34820037)

# **Теория**

1. **Комплексные числа** - числа вида  {\displaystyle a+bi}, где {\displaystyle a,b}— [вещественные числа](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D1%89%D0%B5%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%BE%D0%B5_%D1%87%D0%B8%D1%81%D0%BB%D0%BE), {\displaystyle i} — [мнимая единица](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%B8%D0%BC%D0%B0%D1%8F_%D0%B5%D0%B4%D0%B8%D0%BD%D0%B8%D1%86%D0%B0), то есть число, для которого выполняется равенство: .
2. **Конструктор класса** - это специальный метод класса, вызываемый при объявлении объекта. Конструктор есть как по умолчанию, так и созданный самостоятельно, который будет выполнить те инструкции, которые пишет разработчик.
3. **Деструктор класса** - это специальный метод класса, служащий для деинициализации объекта. Деструктор может служить, например, для освобождения памяти. Деструктор класса вызывается самостоятельно в конце блока, где создан объект.

 {\displaystyle i^{2}=-1.}

## **Описание работы конструктора:**

Конструктор вызывается в момент объявления объекта класса. Во время вызова конструктора выполняются определенные инструкции, описанные в теле конструктора. Например, выделение память для динамического массива, инициализация переменный, инициализация массивов.

## **Описание работы деструктора:**

Деструктор вызывается в конце блока, где создан объект. Во время вызова деструктора происходит удаление объектов и удаление памяти. Память освобождается автоматические, если объект лежит в стеке. Если же объект лежит в куче, то удаление памяти нужно прописать самостоятельно в теле деструктора. Например, высвобождение памяти динамического массива и затем приравнивание указателя на первый элемент к nullptr.

## **Список реализованных методов:**

1. Конструктор
2. Конструктор по умолчанию
3. Конструктор копирования
4. Деструктор
5. Перегрузка операторов (=, +, -, \*, /, ^, %, +=(object and number), -= (object and number), \*= (object and number), /= (object and number), %=, a++, ++a, a--, --a, >>, <<, ==, !=, >, <)
6. Метод show\_complex()

# **Код программы:**

### Файл Complex.h

#pragma once

#include "fraction.h"

using namespace std;

class Complex {

private:

Fraction Re;

Fraction Im;

public:

Complex(Fraction \_Re, Fraction \_Im); //Re+Im\*i

Complex();

Complex(const Complex& tmp);

~Complex();

void ComplexMulti(Complex tmp, int\* \_num, int\* \_den, int test) const;

Complex& operator=(const Complex& tmp);

Complex operator+(const Complex& tmp) const;

Complex operator-(const Complex& tmp) const;

Complex operator\*(const Complex& tmp) const; //

Complex operator/(const Complex& tmp) const; //

Complex operator^(const int& tmp) const; //

Complex operator%(const int& tmp) const;

Complex& operator+=(const Complex& tmp);

Complex& operator+=(int tmp);

Complex& operator-=(const Complex& tmp);

Complex& operator-=(int tmp);

Complex& operator\*=(const Complex& tmp);

Complex& operator\*=(int tmp);

Complex& operator/=(const Complex& tmp);

Complex& operator/=(int tmp);

Complex& operator%=(int tmp);

Complex& operator++(); //

Complex& operator++(int);

Complex& operator--(); //

Complex& operator--(int);

friend std::ostream& operator<<(std::ostream& out, Complex& tmp);

friend std::istream& operator>>(std::istream& in, Complex& tmp);

bool operator==(const Complex& tmp) const;

bool operator!=(const Complex& tmp) const;

bool operator>(const Complex& tmp) const;

bool operator<(const Complex& tmp) const;

void show\_complex();

};

### Файл Complex.cpp

#include "Complex.h"

//CONSTRUCTOR

Complex::Complex(Fraction \_Re, Fraction \_Im) {

this->Re = \_Re;

this->Im = \_Im;

}

//DEFAULT CONSTRUCTOR

Complex::Complex() {

this->Re = Fraction(0);

this->Im = Fraction(0);

}

//COPY CONSTRUCTOR

Complex::Complex(const Complex& tmp) {

this->Re = tmp.Re;

this->Im = tmp.Im;

}

//DESTRUCTOR

Complex::~Complex() {

this->Re = Fraction(0);

this->Im = Fraction(0);

}

//COMPLEX MULTIPLICATION

void Complex::ComplexMulti(Complex tmp, int\* \_num, int\* \_den, int test) const{

if (test == 1) {

\*\_num = this->Re.getNum() \* tmp.Re.getNum();

\*\_den = this->Re.getDen() \* tmp.Re.getDen();

}

if (test == 2) {

\*\_num = this->Re.getNum() \* tmp.Im.getNum();

\*\_den = this->Re.getDen() \* tmp.Im.getDen();

}

if (test == 3) {

\*\_num = this->Im.getNum() \* tmp.Re.getNum();

\*\_den = this->Im.getDen() \* tmp.Re.getDen();

}

if (test == 4) {

\*\_num = this->Im.getNum() \* tmp.Im.getNum();

\*\_den = this->Im.getDen() \* tmp.Im.getDen();

}

}

//OVERLOAD OF =

Complex& Complex::operator=(const Complex& tmp)

{

this->Re = tmp.Re;

this->Im = tmp.Im;

return \*this;

}

//OVERLOAD OF +

Complex Complex::operator+(const Complex& tmp) const{

int ImDen = 0;

int ImNumSum = 0;

int ReDen = 0;

int ReNumSum = 0;

if (this->Re.getDen() == tmp.Re.getDen()) {

ReNumSum = this->Re.getNum() + tmp.Re.getNum();

ReDen = tmp.Re.getDen();

}

else {

int ReNum1 = this->Re.getNum() \* tmp.Re.getDen();

int ReNum2 = tmp.Re.getNum() \* this->Re.getDen();

ReNumSum = ReNum1 + ReNum2;

ReDen = this->Re.getDen() \* tmp.Re.getDen();

}

if (this->Im.getDen() == tmp.Im.getDen()) {

ImNumSum = this->Im.getNum() + tmp.Im.getNum();

ImDen = tmp.Im.getDen();

}

else {

int ImNum1 = this->Im.getNum() \* tmp.Im.getDen();

int ImNum2 = tmp.Im.getNum() \* this->Im.getDen();

ImNumSum = ImNum1 + ImNum2;

ImDen = this->Im.getDen() \* tmp.Im.getDen();

}

Fraction TEMP1(ReNumSum, ReDen);

Fraction TEMP2(ImNumSum, ImDen);

Complex TEMP(TEMP1, TEMP2);

return TEMP;

}

//OVERLOAD OF -

Complex Complex::operator-(const Complex& tmp) const {

int ImDen = 0;

int ImNumSub = 0;

int ReDen = 0;

int ReNumSub = 0;

if (this->Re.getDen() == tmp.Re.getDen()) {

ReNumSub = this->Re.getNum() + tmp.Re.getNum();

ReDen = tmp.Re.getDen();

}

else {

int ReNum1 = this->Re.getNum() \* tmp.Re.getDen();

int ReNum2 = tmp.Re.getNum() \* this->Re.getDen();

ReNumSub = ReNum1 - ReNum2;

ReDen = this->Re.getDen() \* tmp.Re.getDen();

}

if (this->Im.getDen() == tmp.Im.getDen()) {

ImNumSub = this->Im.getNum() - tmp.Im.getNum();

ImDen = tmp.Im.getDen();

}

else {

int ImNum1 = this->Im.getNum() \* tmp.Im.getDen();

int ImNum2 = tmp.Im.getNum() \* this->Im.getDen();

ImNumSub = ImNum1 - ImNum2;

ImDen = this->Im.getDen() \* tmp.Im.getDen();

}

Fraction TEMP1(ReNumSub, ReDen);

Fraction TEMP2(ImNumSub, ImDen);

Complex TEMP(TEMP1, TEMP2);

return TEMP;

}

//OVERLOAD OF \*

Complex Complex::operator\*(const Complex& tmp) const{

int ReNum, ReDen;

int ReImNum, ReImDen;

int ImReNum, ImReDen;

int ImNum, ImDen;

int fNum, fDen;

int sNum, sDen;

ComplexMulti(tmp, &ReNum, &ReDen, 1); //First Re \* Second Re

ComplexMulti(tmp, &ReImNum, &ReImDen, 2); //First Re \* Second Im

ComplexMulti(tmp, &ImReNum, &ImReDen, 3); //First Im \* Second Re

ComplexMulti(tmp, &ImNum, &ImDen, 4); //First Im \* Second Im

if (ReDen == ImNum) { //ReNum

fNum = ReNum - ImNum;

fDen = ReDen;

}

else {

fNum = (ReNum \* ImDen) - (ReDen \* ImNum);

fDen = ReDen \* ImDen;

}

if (ReImDen == ImReDen) {

sNum = ReImNum + ImReNum;

sDen = ReImDen;

}

else {

sNum = (ReImNum \* ImReDen) + (ImReNum \* ReImDen);

sDen = ReImDen \* ImReDen;

}

Fraction TEMP1(fNum, fDen);

Fraction TEMP2(sNum, sDen);

Complex TEMP(TEMP1, TEMP2);

return TEMP;

}

//OVERLOAD OF /

Complex Complex::operator/(const Complex& tmp) const{

int num, den, num1, den1;

//FIRST HALF

this->Re.multiplication(tmp.Re, &num, &den); //First Re \* Second Re

Fraction first(num, den); //Record result to the fraction first

this->Im.multiplication(tmp.Im, &num, &den); //First Im \* Second Im

Fraction second(num, den); //Record result to the fraction second

tmp.Re.degree(&num, &den); //Prepare for denominator. Get Second Re to higher degree

Fraction third(num, den); //Record result to the fraction third

tmp.Im.degree(&num, &den); //Get Second Im to higher degree

Fraction fouth(num, den); //Record result to the fraction fouth

first.addition(second, &num, &den); //Addition of first and second objects

Fraction fNum(num, den); //Result of numerator of first half

third.addition(fouth, &num, &den); //Addition if third and fourth objects

Fraction fDen(num, den); //Result of denominator of first half

fNum.division(fDen, &num, &den); // Result of fraction of first half

Fraction fResult(num, den); //Record to fraction fResult

//SECOND HALF

tmp.Re.multiplication(this->Im, &num1, &den1); //Second Re \* First Im

Fraction sfirst(num1, den1); //Record result

this->Re.multiplication(tmp.Im, &num1, &den1); //First Re \* Second Im

Fraction ssecond(num1, den1); //Record result

sfirst.substraction(ssecond, &num1, &den1); //sfirst - ssecond

Fraction sNum(num1, den1); //Record result of numerator of second half

sNum.division(fDen, &num1, &den1); //Result of fraction of second half

Fraction TEMP1(num, den); //temp object of Re

Fraction TEMP2(num1, den1); //Temp object of Im

Complex TEMP(TEMP1, TEMP2);

return TEMP;

}

Complex Complex::operator^(const int& tmp) const{

int ReNum = this->Re.getNum();

int ReDen = this->Re.getDen();

int ImNum = this->Im.getNum();

int ImDen = this->Im.getDen();

int ReNumTmp = ReNum;

int ReDenTmp = ReDen;

int ImNumTmp = ImNum;

int ImDenTmp = ImDen;

if (tmp == 0) {

ReNum = 1;

ReDen = 1;

ImNum = 1;

ImDen = 1;

}

else {

for (int i = 1; i < tmp; i++) {

ReNum \*= ReNumTmp;

ReDen \*= ReDenTmp;

ImNum \*= ImNumTmp;

ImDen \*= ImDenTmp;

}

}

Fraction TEMP1(ReNum, ReDen);

Fraction TEMP2(ImNum, ImDen);

Complex TEMP(TEMP1, TEMP2);

return TEMP;

}

Complex Complex::operator%(const int& tmp) const{

int ReNum = this->Re.getNum() % tmp;

int ReDen = this->Re.getDen() % tmp;

int ImNum = this->Im.getNum() % tmp;

int ImDen = this->Im.getDen() % tmp;

Fraction TEMP1(ReNum, ReDen);

Fraction TEMP2(ImNum, ImDen);

Complex TEMP(TEMP1, TEMP2);

return TEMP;

}

Complex& Complex::operator+=(const Complex& tmp) {

int ImDen = 0;

int ImNumSum = 0;

int ReDen = 0;

int ReNumSum = 0;

if (this->Re.getDen() == tmp.Re.getDen()) {

ReNumSum = this->Re.getNum() + tmp.Re.getNum();

ReDen = tmp.Re.getDen();

}

else {

int ReNum1 = this->Re.getNum() \* tmp.Re.getDen();

int ReNum2 = tmp.Re.getNum() \* this->Re.getDen();

ReNumSum = ReNum1 + ReNum2;

ReDen = this->Re.getDen() \* tmp.Re.getDen();

}

if (this->Im.getDen() == tmp.Im.getDen()) {

ImNumSum = this->Im.getNum() + tmp.Im.getNum();

ImDen = tmp.Im.getDen();

}

else {

int ImNum1 = this->Im.getNum() \* tmp.Im.getDen();

int ImNum2 = tmp.Im.getNum() \* this->Im.getDen();

ImNumSum = ImNum1 + ImNum2;

ImDen = this->Im.getDen() \* tmp.Im.getDen();

}

Fraction TEMP1(ReNumSum, ReDen);

Fraction TEMP2(ImNumSum, ImDen);

this->Re = Fraction(TEMP1);

this->Im = Fraction(TEMP2);

return \*this;

}

Complex& Complex::operator+=(int tmp) {

int TmpNum = tmp;

int TmpDen = 1;

int ReDen = 0;

int ReNumSum = 0;

if (this->Re.getDen() == TmpDen) {

ReNumSum = this->Re.getNum() + TmpNum;

ReDen = TmpDen;

}

else {

int ReNum1 = this->Re.getNum() \* TmpDen;

int ReNum2 = TmpNum \* this->Re.getDen();

ReNumSum = ReNum1 + ReNum2;

ReDen = this->Re.getDen() \* TmpDen;

}

Fraction TEMP1(ReNumSum, ReDen);

this->Re = Fraction(TEMP1);

return \*this;

}

Complex& Complex::operator-=(const Complex& tmp) {

int ImDen = 0;

int ImNumSub = 0;

int ReDen = 0;

int ReNumSub = 0;

if (this->Re.getDen() == tmp.Re.getDen()) {

ReNumSub = this->Re.getNum() + tmp.Re.getNum();

ReDen = tmp.Re.getDen();

}

else {

int ReNum1 = this->Re.getNum() \* tmp.Re.getDen();

int ReNum2 = tmp.Re.getNum() \* this->Re.getDen();

ReNumSub = ReNum1 - ReNum2;

ReDen = this->Re.getDen() \* tmp.Re.getDen();

}

if (this->Im.getDen() == tmp.Im.getDen()) {

ImNumSub = this->Im.getNum() - tmp.Im.getNum();

ImDen = tmp.Im.getDen();

}

else {

int ImNum1 = this->Im.getNum() \* tmp.Im.getDen();

int ImNum2 = tmp.Im.getNum() \* this->Im.getDen();

ImNumSub = ImNum1 - ImNum2;

ImDen = this->Im.getDen() \* tmp.Im.getDen();

}

Fraction TEMP1(ReNumSub, ReDen);

Fraction TEMP2(ImNumSub, ImDen);

this->Re = Fraction(TEMP1);

this->Im = Fraction(TEMP2);

return \*this;

}

Complex& Complex::operator-=(int tmp) {

int TmpNum = tmp;

int TmpDen = 1;

int ImNumSub = 0;

int ReDen = 0;

int ReNumSub = 0;

if (this->Re.getDen() == TmpDen) {

ReNumSub = this->Re.getNum() + TmpNum;

ReDen = TmpDen;

}

else {

int ReNum1 = this->Re.getNum() \* TmpDen;

int ReNum2 = TmpNum \* this->Re.getDen();

ReNumSub = ReNum1 - ReNum2;

ReDen = this->Re.getDen() \* TmpDen;

}

Fraction TEMP1(ReNumSub, ReDen);

this->Re = Fraction(TEMP1);

return \*this;

}

Complex& Complex::operator\*=(const Complex& tmp) {

int ReNum, ReDen;

int ReImNum, ReImDen;

int ImReNum, ImReDen;

int ImNum, ImDen;

int fNum, fDen;

int sNum, sDen;

ComplexMulti(tmp, &ReNum, &ReDen, 1);

ComplexMulti(tmp, &ReImNum, &ReImDen, 2);

ComplexMulti(tmp, &ImReNum, &ImReDen, 3);

ComplexMulti(tmp, &ImNum, &ImDen, 4);

if (ReDen == ImDen) {

fNum = ReNum - ImNum;

fDen = ReDen;

}

else {

fNum = (ReNum \* ImDen) - (ReDen \* ImNum);

fDen = ReDen \* ImDen;

}

if (ReImDen == ImReDen) {

sNum = ReImNum + ImReNum;

sDen = ReImDen;

}

else {

sNum = (ReImNum \* ImReDen) + (ImReNum \* ReImDen);

sDen = ReImDen \* ImReDen;

}

Fraction TEMP1(fNum, fDen);

Fraction TEMP2(sNum, sDen);

this->Re = Fraction(TEMP1);

this->Im = Fraction(TEMP2);

return \*this;

}

Complex& Complex::operator\*=(int tmp) {

Fraction TEMP1(this->Re.getNum() \* tmp, this->Re.getDen());

Fraction TEMP2(this->Im.getNum() \* tmp, this->Re.getDen());

this->Re = Fraction(TEMP1);

this->Im = Fraction(TEMP2);

return \*this;

}

Complex& Complex::operator/=(const Complex& tmp) {

int num, den, num1, den1;

//FIRST HALF

this->Re.multiplication(tmp.Re, &num, &den); //First Re \* Second Re

Fraction first(num, den); //Record result to the fraction first

this->Im.multiplication(tmp.Im, &num, &den); //First Im \* Second Im

Fraction second(num, den); //Record result to the fraction second

tmp.Re.degree(&num, &den); //Prepare for denominator. Get Second Re to higher degree

Fraction third(num, den); //Record result to the fraction third

tmp.Im.degree(&num, &den); //Get Second Im to higher degree

Fraction fouth(num, den); //Record result to the fraction fouth

first.addition(second, &num, &den); //Addition of first and second objects

Fraction fNum(num, den); //Result of numerator of first half

third.addition(fouth, &num, &den); //Addition if third and fourth objects

Fraction fDen(num, den); //Result of denominator of first half

fNum.division(fDen, &num, &den); // Result of fraction of first half

Fraction fResult(num, den); //Record to fraction fResult

//SECOND HALF

tmp.Re.multiplication(this->Im, &num1, &den1); //Second Re \* First Im

Fraction sfirst(num1, den1); //Record result

this->Re.multiplication(tmp.Im, &num1, &den1); //First Re \* Second Im

Fraction ssecond(num1, den1); //Record result

sfirst.substraction(ssecond, &num1, &den1); //sfirst - ssecond

Fraction sNum(num1, den1); //Record result of numerator of second half

sNum.division(fDen, &num1, &den1); //Result of fraction of second half

Fraction TEMP1(num, den); //temp object of Re

Fraction TEMP2(num1, den1); //Temp object of Im

this->Re = Fraction(TEMP1);

this->Im = Fraction(TEMP2);

return \*this;

}

Complex& Complex::operator/=(int tmp) {

Fraction TEMP1(this->Re.getNum(), this->Re.getDen() \* tmp);

Fraction TEMP2(this->Im.getNum(), this->Im.getDen() \* tmp);

this->Re = Fraction(TEMP1);

this->Im = Fraction(TEMP2);

return \*this;

}

Complex& Complex::operator%=(int tmp) {

return \*this;

}

Complex& Complex::operator++() {

int num = this->Re.getNum();

this->Re.setNum(num += this->Re.getDen());

return \*this;

}

Complex& Complex::operator++(int) {

int num = this->Re.getNum();

Complex temp(Re, Im);

this->Re.setNum(num += this->Re.getDen());

return temp;

}

Complex& Complex::operator--() {

int num = this->Re.getNum();

this->Re.setNum(num -= this->Re.getDen());

return \*this;

}

Complex& Complex::operator--(int) {

int num =this->Re.getNum();

Complex temp(Re, Im);

this -> Re.setNum(num -= this->Re.getDen());

return temp;

}

std::ostream& operator<<(std::ostream& out, Complex& tmp) {

out << tmp.Re.getNum() << "/" << tmp.Re.getDen() << " + (" << tmp.Im.getNum() << "+" << tmp.Im.getDen() << ")i" << endl;

return out;

}

std::istream& operator>>(std::istream& in, Complex& tmp) {

cout << "Enter real part" << endl;

in >> tmp.Re;

cout << "Enter imaginary part" << endl;

in >> tmp.Im;

return in;

}

bool Complex::operator==(const Complex& tmp) const {

Fraction thisX = this->Re \* this->Re;

Fraction thisY = this->Im \* this->Im;

Fraction thisXY = thisX + thisY;

Fraction tmpX = tmp.Re \* tmp.Re;

Fraction tmpY = tmp.Im \* tmp.Im;

Fraction tmpXY = tmpX \* tmpY;

if (thisXY == tmpXY)

return true;

else

return false;

}

bool Complex::operator!=(const Complex& tmp) const {

Fraction thisX = this->Re \* this->Re;

Fraction thisY = this->Im \* this->Im;

Fraction thisXY = thisX + thisY;

Fraction tmpX = tmp.Re \* tmp.Re;

Fraction tmpY = tmp.Im \* tmp.Im;

Fraction tmpXY = tmpX \* tmpY;

if (thisXY == tmpXY)

return false;

else

return true;

}

bool Complex::operator>(const Complex& tmp) const {

Fraction thisX = this->Re \* this->Re;

Fraction thisY = this->Im \* this->Im;

Fraction thisXY = thisX + thisY;

Fraction tmpX = tmp.Re \* tmp.Re;

Fraction tmpY = tmp.Im \* tmp.Im;

Fraction tmpXY = tmpX \* tmpY;

if (thisXY > tmpXY)

return true;

else

return false;

}

bool Complex::operator<(const Complex& tmp) const {

Fraction thisX = this->Re \* this->Re;

Fraction thisY = this->Im \* this->Im;

Fraction thisXY = thisX + thisY;

Fraction tmpX = tmp.Re \* tmp.Re;

Fraction tmpY = tmp.Im \* tmp.Im;

Fraction tmpXY = tmpX \* tmpY;

if (thisXY > tmpXY)

return false;

else

return true;

}

void Complex::show\_complex() {

int ImNum = this->Im.getNum();

int ImDen = this->Im.getDen();

int ReNum = this->Re.getNum();

int ReDen = this->Re.getDen();

cout << ReNum << "/" << ReDen << " + (" << ImNum << "/" << ImDen << ")i" << endl;

}

### Файл Fraction.h

#pragma once

#include <iostream>

#include <Windows.h>

#include <string>

class Fraction {

//Fields

private:

int numerator;

int denominator;

public:

//Constructor

Fraction(int \_num, int \_den);

Fraction(const Fraction& tmp);

Fraction(int a);

Fraction() {

this->numerator = 0;

this->denominator = 0;

}

//Get

int getNum() const;

int getDen() const;

//Set

void setNum(int newNum);

void setDen(int newDen);

//Other Methods

void addition(Fraction tmp, int\* \_num, int\* \_den) const;

void substraction(Fraction tmp, int\* \_num, int\* \_den) const;

void multiplication(Fraction tmp, int\* \_num, int\* \_den) const;

void division(Fraction tmp, int\* \_num, int\* \_den) const;

void degree(int\* \_num, int\* \_den) const;

void show\_fraction();

void record(int \_num, int \_den);

friend std::istream& operator>> (std::istream& in, Fraction& tmp);

bool operator==(const Fraction& tmp) const;

bool operator>(const Fraction& tmp) const;

Fraction operator+(const Fraction& tmp) const;

Fraction operator\*(const Fraction& tmp) const;

~Fraction() {

}

};

### Файл Fraction.cpp

#include "fraction.h"

using namespace std;

//Constructor

Fraction::Fraction(int \_a) {

this->numerator = 0;

this->denominator = 1;

}

Fraction::Fraction(const Fraction& tmp) {

this->numerator = tmp.numerator;

this->denominator = tmp.denominator;

}

Fraction::Fraction(int \_a, int \_b) {

this->numerator = \_a;

this->denominator = \_b;

}

//Get

int Fraction::getNum() const {

return this->numerator;

}

int Fraction::getDen() const {

return this->denominator;

}

//Set

void Fraction::setNum(int newNum) {

this->numerator = newNum;

}

void Fraction::setDen(int newDen) {

this->denominator = newDen;

}

//Other Methods

void Fraction::show\_fraction() {

cout << this->numerator << "/" << this->denominator << endl;

}

void Fraction::addition(Fraction tmp, int \* \_num, int\* \_den) const { // +

int Num1;

int Num2;

if (this->denominator == tmp.denominator) {

\*\_num = this->numerator + tmp.numerator;

\*\_den = tmp.denominator;

}

else {

Num1 = this->numerator \* tmp.denominator;

Num2 = tmp.numerator \* this->denominator;

\*\_num = Num1 + Num2;

\*\_den = this->denominator \* tmp.denominator;

}

}

void Fraction::substraction(Fraction tmp, int\* \_num, int\* \_den) const { // -

int Num1;

int Num2;

if (this->denominator == tmp.denominator) {

\*\_num = this->numerator - tmp.numerator;

\*\_den = tmp.denominator;

}

else {

Num1 = this->numerator \* tmp.denominator;

Num2 = tmp.numerator \* this->denominator;

\*\_num = Num1 - Num2;

\*\_den = this->denominator \* tmp.denominator;

}

}

void Fraction::multiplication(Fraction tmp, int\* \_num, int\* \_den) const { // \*

\*\_num = this->numerator \* tmp.numerator;

\*\_den = this->denominator \* tmp.denominator;

}

void Fraction::division(Fraction tmp, int\* \_num, int\* \_den) const { // /

if (tmp.denominator == 0) {

cout << "ERROR: SECOND FRACTION IS ZERO";

return;

}

else {

\*\_num = this->numerator \* tmp.denominator;

\*\_den = this->denominator \* tmp.numerator;

}

}

void Fraction::degree(int\* \_num, int\* \_den) const {

\*\_num = this->numerator \* this->numerator;

\*\_den = this->denominator \* this->denominator;

}

void Fraction::record(int \_num, int \_den) {

this->numerator = \_num;

this->denominator = \_den;

}

std::istream& operator>>(std::istream& in, Fraction& tmp) {

int temp;

cout << "Enter numerator: " << endl;

in >> temp;

tmp.setNum(temp);

cout << "Enter denumerator: " << endl;

in >> temp;

tmp.setDen(temp);

return in;

}

bool Fraction::operator==(const Fraction& tmp) const {

if (this->denominator == tmp.denominator)

return (this->numerator == tmp.denominator);

else

return (this->numerator \* tmp.denominator == tmp.numerator \* this->denominator);

}

Fraction Fraction::operator+(const Fraction& tmp) const {

int num;

int den;

if (this->denominator == tmp.denominator) {

num = this->numerator + tmp.numerator;

den = tmp.denominator;

}

else {

int Num1 = this->numerator \* tmp.denominator;

int Num2 = tmp.numerator \* this->denominator;

num = Num1 + Num2;

den = this->denominator \* tmp.denominator;

}

Fraction temp(num, den);

return temp;

}

Fraction Fraction::operator\*(const Fraction& tmp) const {

int num = this->numerator \* tmp.numerator;

int den = this->denominator \* tmp.denominator;

Fraction temp(num, den);

return temp;

}

bool Fraction::operator>(const Fraction& tmp) const {

if ((this->numerator / this->denominator) > (tmp.numerator / tmp.denominator))

return true;

else

return false;

}

### Файл Lab1.cpp

#include <iostream>

#include "fraction.h"

#include "Complex.h"

void entering(int\* \_num, int\* \_den) {

do {

cout << "Enter numerator: ";

cin >> \*\_num;

if (\*\_num == 0) {

cout << "ERROR: NUMERATOR IS ZERO" << endl;

}

} while (\*\_num == 0);

do {

cout << "Enter denominator: ";

cin >> \*\_den;

if (\*\_den == 0) {

cout << "ERROR: DENOMINATOR IS ZERO" << endl;

}

} while (\*\_den == 0);

}

int main()

{

using namespace std;

int num;

int den;

cout << "RE" << endl;

entering(&num, &den);

Fraction First(num, den);

cout << "IM" << endl;

entering(&num, &den);

Fraction Second(num, den);

Complex FirstComplex(First, Second);

cout << "==========" << endl;

cout << "RE" << endl;

entering(&num, &den);

Fraction Third(num, den);

cout << "IM" << endl;

entering(&num, &den);

Fraction Fouth(num, den);

Complex SecondComplex(Third, Fouth);

cout << "==========" << endl;

cout << "First: ";

FirstComplex.show\_complex();

cout << "Second: ";

SecondComplex.show\_complex();

cout << "==========" << endl;

Complex ResultComplex = FirstComplex \* SecondComplex;

cout << "Resut of \*";

ResultComplex.show\_complex();

cout << "==========" << endl;

ResultComplex = FirstComplex / SecondComplex;

cout << "Resut of /";

ResultComplex.show\_complex();

cout << "==========" << endl;

cout << "Enter degree: " << endl;

int degree;

cin >> degree;

ResultComplex = FirstComplex ^ degree;

cout << "Resut of ^";

ResultComplex.show\_complex();

cout << "==========" << endl;

cout << "Number: ";

FirstComplex.show\_complex();

cout << "Result of ++";

FirstComplex++;

FirstComplex.show\_complex();

cout << "==========" << endl;

cout << "Number: ";

FirstComplex.show\_complex();

cout << "Result of --";

FirstComplex--;

FirstComplex.show\_complex();

}