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 №4 по курсу «Информатика и

программирование»

Выполнил: студент группы 381908-3

Борисов Максим

Преподаватель:

Карчков Д. А.

г. Нижний Новгород

2020

**Оглавление**

[**Теория** 3](#_Toc35536677)

[**Описание работы конструктора:** 3](#_Toc35536678)

[**Описание работы деструктора:** 3](#_Toc35536679)

[**Основные отличия данной лабораторной работы от предыдущей:** 3](#_Toc35536680)

[**Код программы:** 4](#_Toc35536681)

[Файл Complex.h 4](#_Toc35536682)

[Файл Complex.cpp 4](#_Toc35536683)

[Файл Fraction.h 7](#_Toc35536684)

[Файл Fraction.cpp 8](#_Toc35536685)

[Файл Lab2.cpp 10](#_Toc35536686)

# **Теория**

1. **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.
2. **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.

## **Основные отличия данной лабораторной работы от предыдущей:**

1. Поля класса Complex являются указателями класса Fraction, тем самым поля находятся в куче.
2. При тестировании класса Complex для каждого объекта этого класса выделялось место в динамической памяти.
3. При описании методов обращение к полям класса Complex происходило через оператор -> обращение к методам класса Fraction так же происходило через оператор ->

# **Код программы:**

### Файл Complex.h

#pragma once

#include "Fraction.h"

class Complex

{

private:

Fraction\* Re;

Fraction\* Im;

public:

Complex(Fraction \*\_Re, Fraction \*\_Im);

Complex();

Complex(const Complex\* tmp);

Complex(const Complex& tmp);

~Complex();

void show\_complex();

void ComplexMulti(const 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++();

Complex& operator--();

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

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

};

### Файл Complex.cpp

#include "Complex.h"

//CONSTRUCTOR

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

this->Re = new Fraction(\_Re);

this->Im = new Fraction (\_Im);

}

//DEFAULT CONSTRUCTOR

Complex::Complex() {

this->Re = new Fraction(0, 1);

this->Im = new Fraction(0, 1);

}

//COPY CONSTRUCTOR

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

this->Re = tmp->Re;

this->Im = tmp->Im;

}

Complex::Complex(const Complex& tmp) {

this->Re = tmp.Re;

this->Im = tmp.Im;

}

Complex::~Complex() {

}

//PRINT

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;

}

//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 {

Fraction RESULTRE = new Fraction();

RESULTRE = \*this->Re + tmp->Re;

Fraction RESULTIM = new Fraction();

RESULTIM = \*this->Im + tmp->Im;

Complex TEMP = new Complex(&RESULTRE, &RESULTIM);

return TEMP;

}

//OVERLOAD OF -

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

Fraction RESULTRE = new Fraction();

RESULTRE = \*this->Re - tmp->Re;

Fraction RESULTIM = new Fraction();

RESULTIM = \*this->Im - tmp->Im;

Complex TEMP = new Complex(&RESULTRE, &RESULTIM);

return TEMP;

}

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 RESULTRE = new Fraction(fNum, fDen);

Fraction RESULTIM = new Fraction(sNum, sDen);

Complex TEMP = new Complex(&RESULTRE, &RESULTIM);

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 RESULTRE = new Fraction(num, den); //temp object of Re

Fraction RESULTIM = new Fraction(num1, den1); //Temp object of Im

Complex TEMP = new Complex(&RESULTRE, &RESULTIM);

return TEMP;

}

//OVERLOAD OF ++

Complex& Complex::operator++() {

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

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

return \*this;

}

//OVERLOAD OF --

Complex& Complex::operator--() {

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

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

return \*this;

}

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

int ReNum = tmp->Re->getNum();

int ReDen = tmp->Re->getDen();

int ImNum = tmp->Im->getNum();

int ImDen = tmp->Im->getDen();

out << ReNum << "/" << ReDen << "+(" << ImNum << "/" << ImDen << ")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;

}

//COMPLEX MULTIPLICATION

void Complex::ComplexMulti(const 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();

}

}

### Файл Fraction.h

#pragma once

#include <iostream>

using namespace std;

class Fraction

{

//Fields

private:

int numerator;

int denominator;

public:

//Constructor

Fraction(int \_num, int \_den);

Fraction(const Fraction& tmp);

Fraction(const Fraction\* tmp);

Fraction() {

this->numerator = 1;

this->denominator = 1;

cout << "CONSTRUCTOR DEFAULT" << this << endl;

}

~Fraction() {

cout << "DESTRUCTOR" << this << endl;

}

//Get

int getNum() const;

int getDen() const;

//Set

void setNum(int \_Num);

void setDen(int \_Den);

//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);

Fraction& operator=(const Fraction& tmp);

Fraction operator+(const Fraction& tmp) const;

Fraction operator-(const Fraction & tmp) const;

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

};

### Файл Fraction.cpp

#include "Fraction.h"

using namespace std;

//Constructor

Fraction::Fraction(const Fraction& tmp) {

this->numerator = tmp.numerator;

this->denominator = tmp.denominator;

cout << "CONSTRUCTOR COPY &" << this << endl;

}

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

this->numerator = tmp->numerator;

this->denominator = tmp->denominator;

cout << "CONSTRUCTOR COPY \*" << this << endl;

}

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

this->numerator = \_a;

this->denominator = \_b;

cout << "CONSTRUCTOR PARAM" << this << endl;

}

//Get

int Fraction::getNum() const {

return this->numerator;

}

int Fraction::getDen() const {

return this->denominator;

}

//Set

void Fraction::setNum(int \_Num) {

this->numerator = \_Num;

}

void Fraction::setDen(int \_Den) {

this->denominator = \_Den;

}

//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;

}

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

this->numerator = tmp.numerator;

this->denominator = tmp.denominator;

return \*this;

}

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

int NumF;

int NumS;

int NumSum;

int Den;

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

NumSum = this->numerator + tmp.numerator;

Den = tmp.denominator;

}

else {

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

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

NumSum = NumF + NumS;

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

}

Fraction TEMP = new Fraction(NumSum, Den);

return TEMP;

}

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

int NumF;

int NumS;

int NumSub;

int Den;

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

NumSub = this->numerator - tmp.numerator;

Den = tmp.denominator;

}

else {

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

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

NumSub = NumF - NumS;

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

}

Fraction TEMP = new Fraction(NumSub, Den);

return TEMP;

}

### Файл Lab2.cpp

#include "Complex.h"

int main()

{

Fraction First = new Fraction();

Fraction Second = new Fraction();

Fraction Third = new Fraction();

Fraction Fouth = new Fraction();

cout << "FIRST COMPLEX: " << endl;

cin >> &First;

cin >> &Second;

cout << "SECOND COMPLEX: " << endl;

cin >> &Third;

cin >> &Fouth;

Complex FIRSTCOMPLEX = new Complex(&First, &Second);

Complex\* SECONDCOMPLEX = new Complex(&Third, &Fouth);

Complex RESULT = new Complex();

cout << "FIRST COMPLEX: " << endl;

cout << &FIRSTCOMPLEX;

cout << "SECOND COMPLEX: " << endl;

cout << SECONDCOMPLEX;

cout << "===OVERLOAD OF +===" << endl;

RESULT = FIRSTCOMPLEX + SECONDCOMPLEX;

cout << &RESULT;

cout << "===OVERLOAD OF -===" << endl;

RESULT = FIRSTCOMPLEX - SECONDCOMPLEX;

cout << &RESULT;

cout << "===OVERLOAD OF \*===" << endl;

RESULT = FIRSTCOMPLEX \* SECONDCOMPLEX;

cout << &RESULT;

cout << "===OVERLOAD OF /===" << endl;

RESULT = FIRSTCOMPLEX / SECONDCOMPLEX;

cout << &RESULT;

}