Міністерство освіти і науки України

Національний університет “Львівська політехніка”

Інститут комп’ютерних наук та інформаційних технологій

Кафедра програмного забезпечення



**Звіт**

Про виконання лабораторної роботи №3

**На тему:**

«Основи модульного тестування програм з використанням мови С++»

з дисципліни “Об’єктно-орієнтоване програмування”

**Лектор:**

доц. каф. ПЗ

Коротєєва Т.О.

**Виконав:**

ст. гр. ПЗ-18

Юшкевич А.І.

**Прийняв:**

асис. каф. ПЗ

Дивак І.В.

« … » … 2023 р.

∑ = \_\_\_\_\_\_\_\_\_\_\_\_

Львів – 2022

**Тема**: Створення та використання класів.

**Мета**: Навчитися створювати класи, використовувати конструктори для ініціалізації об’єктів, опанувати принципи створення функцій-членів. Навчитися використовувати різні типи доступу до полів та методів класів.

# Теоретичні відомості

Опис класу в першому наближенні виглядає так:

class <ім'я> {

[private:]

<Опис прихованих елементів> public:

<Опис доступних елементів>

}; //Опис закінчується крапкою з комою.

Замість ключового слова class може бути також struct. З точки зору C++ між class та struct практично немає різниці. Невеликі відмінності наступні:

1. Для class дані-члени, функції-члени та батьківські класи за замовчуванням приватні (private).
2. Для struct дані-члени, функції-члени та батьківські класи за замовчуванням публічні (public).

Специфікатор доступу private і public керують видимістю елементів класу. Елементи, описані після службового слова private, доступні тільки всередині класу. Цей вид доступу прийнятий у класі за замовчуванням. Інтерфейс класу описується після специфікатора public. Дія будь-якого специфікатора поширюється до наступного специфікатора або до кінця класу. Можна задавати кілька секцій private і public, їх порядок значення не має.

Поля класу:

* можуть мати будь-який тип, крім типу цього ж класу (але можуть бути вказівниками або посиланнями на цей клас);
* можуть бути описані з модифікатором const, при цьому вони ініціалізуються тільки один раз (за допомогою конструктора) і не можуть змінюватися;
* можуть бути описані з модифікатором static (розглядається в наступних лабораторних).

Починаючи із С++11 допускається ініціалізація полів при описі класу.

# Завдання

1. Створити клас відповідно до варіанту.
2. При створенні класу повинен бути дотриманий принцип інкапсуляції.
3. Створити конструктор за замовчуванням та хоча б два інших конструктори для початкової ініціалізації об’єкта.
4. Створити функції-члени для задавання та зчитування значень полів (getters/setters)
5. Створити інші функції члени згідно з варіантом.
6. Продемонструвати можливості класу написавши для нього модульні тести для Google Test.
7. У звіті до лабораторної намалювати UML-діаграму класу, яка відповідає варіанту.
8. Звіт має містити тексти програм, тестів та результат виконання тестів.

Варіант №12:

|  |  |
| --- | --- |
| 12 | Клас СPolynom4 – кубічний чотиричлен (ax3+bx2+cx+d).  Клас повинен містити функції-члени, які реалізовують:   * Знаходження значення виразу для заданого x • Знаходження значення похідної в заданій точці х. * Знаходження визначеного інтегралу на заданому проміжку * Знаходження коренів рівняння * Додавання двох поліномів * Віднімання двох поліномів |

# Код програми

## Lab\_03\_OOP:

UI:

#include <iostream>

#include "PolynomHeader.h"

int main() {

cout << "Crearting object 'my' of CPolynom4 class using base constructor" << endl;

CPolynom4 my;

SPolynom p = my.GetPolynom();

cout << "It's private field, accessed via getter: (" << p.a << ", " << p.b << ", " << p.c << ", " << p.d << ")" << endl;

cout << "It's result if x = 3: " << my.GetResult(3) << endl << endl;

cout << "Crearting object 'custom' of CPolynom4 class and passing values of polynom via constructor parameters" << endl;

CPolynom4 custom(1, -4, -7, 10);

p = custom.GetPolynom();

cout << "It's private field, accessed via getter: (" << p.a << ", " << p.b << ", " << p.c << ", " << p.d << ")" << endl;

cout << "It's result if x = 1: " << custom.GetResult(1) << endl << endl;

cout << "Crearting object 'sum' of CPolynom4 class and initializing it by sum of 'my' and 'custom' fields" << endl;

CPolynom4 sum = my + custom;

p = sum.GetPolynom();

cout << "It's private field, accessed via getter: (" << p.a << ", " << p.b << ", " << p.c << ", " << p.d << ")" << endl;

cout << "It's derivative if x = 2: " << sum.GetDerivative(2) << endl << endl;

cout << "Assigning to custom value of 'my' - 'custom' fields" << endl;

custom = my - custom;

p = custom.GetPolynom();

cout << "It's private field, accessed via getter: (" << p.a << ", " << p.b << ", " << p.c << ", " << p.d << ")" << endl;

cout << "It's integral if x1 = 0, x2 = 1: " << custom.GetIntegral(0, 1) << endl << endl;

cout << "Trying to find roots for 'custom' via standard function: " << endl;

SResult res = custom.GetRoots();

if (res.num) {

cout << "Result: ";

for (int i = res.num; i > 0; i--)

cout << res.roots[i - 1] << "\t";

cout << endl << endl;

}

cout << "Trying to find roots for 'custom' setting own borders ([-15; 20]): " << endl;

res = custom.GetRoots(-15, 20);

if (res.num) {

cout << "Result: ";

for (int i = res.num; i > 0; i--)

cout << res.roots[i - 1] << "\t";

cout << endl << endl;

}

cout << "Trying to find roots for 'custom' setting own borders ([-15; 20]) and epsilon (0.01): " << endl;

res = custom.GetRoots(-15, 20, 0.01);

if (res.num) {

cout << "Result: ";

for (int i = res.num; i > 0; i--)

cout << res.roots[i - 1] << "\t";

cout << endl << endl;

}

return 0;

}

Lab\_03\_OOP.cpp:

#include <iostream>

#include "PolynomHeader.h"

CPolynom4::CPolynom4(double a, double b, double c, double d) {

polynom.a = a;

polynom.b = b;

polynom.c = c;

polynom.d = d;

}

CPolynom4::CPolynom4(SPolynom polynom) {

this->polynom = polynom;

}

CPolynom4::CPolynom4() {

this->polynom.a = 1;

this->polynom.b = 1;

this->polynom.c = 1;

this->polynom.d = 1;

}

double CPolynom4::GetResult(double x) {

return polynom.a \* x \* x \* x + polynom.b \* x \* x + polynom.c \* x + polynom.d;

}

double CPolynom4::GetDerivative(double x) {

return 3 \* polynom.a \* x \* x + 2 \* polynom.b \* x + polynom.c;

}

double CPolynom4::GetIntegral(double x1, double x2) {

return polynom.a \* x2 \* x2 \* x2 \* x2 / 4 + polynom.b \* x2 \* x2 \* x2 / 3 + polynom.c \* x2 \* x2 / 2 + polynom.d \* x2

- (polynom.a \* x1 \* x1 \* x1 \* x1 / 4 + polynom.b \* x1 \* x1 \* x1 / 3 + polynom.c \* x1 \* x1 / 2 + polynom.d \* x1);

}

SResult CPolynom4::GetRoots() {

RootsFinder finder(polynom);

return finder.FindByDichotomy();

}

SResult CPolynom4::GetRoots(ldouble leftLim, ldouble rightLim) {

RootsFinder finder(leftLim, rightLim, polynom);

return finder.FindByDichotomy();

}

SResult CPolynom4::GetRoots(ldouble leftLim, ldouble rightLim, ldouble epsilon) {

RootsFinder finder(leftLim, rightLim, epsilon, polynom);

return finder.FindByDichotomy();

}

SPolynom CPolynom4::GetPolynom() {

return polynom;

}

void CPolynom4::SetPolynom(ldouble a, ldouble b, ldouble c, ldouble d) {

polynom.a = a;

polynom.b = b;

polynom.c = c;

polynom.d = d;

}

void CPolynom4::SetPolynom(SPolynom polynom) {

this->polynom = polynom;

}

CPolynom4 CPolynom4::operator + (const CPolynom4& other) {

double a, b, c, d;

a = this->polynom.a + other.polynom.a;

b = this->polynom.b + other.polynom.b;

c = this->polynom.c + other.polynom.c;

d = this->polynom.d + other.polynom.d;

CPolynom4 NewPol(a, b, c, d);

return NewPol;

}

CPolynom4 CPolynom4::operator - (const CPolynom4& other) {

double a, b, c, d;

a = this->polynom.a - other.polynom.a;

b = this->polynom.b - other.polynom.b;

c = this->polynom.c - other.polynom.c;

d = this->polynom.d - other.polynom.d;

CPolynom4 NewPol(a, b, c, d);

return NewPol;

}

bool CPolynom4::operator == (const CPolynom4& other) {

if ( this->polynom.a == other.polynom.a

&& this->polynom.b == other.polynom.b

&& this->polynom.c == other.polynom.c

&& this->polynom.d == other.polynom.d

)

{

return true;

}

else

return false;

}

bool CPolynom4::operator != (const CPolynom4& other) {

if ( this->polynom.a != other.polynom.a

|| this->polynom.b != other.polynom.b

|| this->polynom.c != other.polynom.c

|| this->polynom.d != other.polynom.d

)

{

return true;

}

else

return false;

}

### PolynomHeader.h:

#pragma once

#include "..\Lib\RootsHeader.h"

class CPolynom4 {

private:

SPolynom polynom;

public:

CPolynom4(double a, double b, double c, double d);

CPolynom4(SPolynom polynom);

CPolynom4();

double GetResult(double x);

double GetDerivative(double x);

double GetIntegral(double x1, double x2);

SResult GetRoots();

SResult GetRoots(ldouble leftLim, ldouble rightLim);

SResult GetRoots(ldouble leftLim, ldouble rightLim, ldouble epsilon);

SPolynom GetPolynom();

void SetPolynom(ldouble a, ldouble b, ldouble c, ldouble d);

void SetPolynom(SPolynom polynom);

CPolynom4 operator + (const CPolynom4& other);

CPolynom4 operator - (const CPolynom4& other);

bool operator == (const CPolynom4& other);

bool operator != (const CPolynom4& other);

};

## Lab\_03\_Tests\_OOP:

### pch.h:

#pragma once

#include "gtest/gtest.h"

#include "..\Lab\_03\_OOP\PolynomHeader.h"

### test.cpp:

#include "pch.h"

TEST(RootsFinder, Dichotomy0000) {

SPolynom pol = { 0, 0, 0, 0 };

RootsFinder test(pol);

EXPECT\_TRUE(isnan(test.FindByDichotomy().roots[0]));

EXPECT\_TRUE(isnan(test.FindByDichotomy().roots[1]));

EXPECT\_TRUE(isnan(test.FindByDichotomy().roots[2]));

}

TEST(RootsFinder, Dichotomy0001) {

SPolynom pol = { 0, 0, 0, 1 };

RootsFinder test(pol);

EXPECT\_TRUE(isnan(test.FindByDichotomy().roots[0]));

EXPECT\_TRUE(isnan(test.FindByDichotomy().roots[1]));

EXPECT\_TRUE(isnan(test.FindByDichotomy().roots[2]));

}

TEST(RootsFinder, Dichotomy0010) {

SPolynom pol = { 0, 0, 1, 0 };

RootsFinder test(pol);

SResult test\_result = { NULL, NULL, NULL, 0 };

EXPECT\_DOUBLE\_EQ(test.FindByDichotomy().roots[0], 0);

EXPECT\_TRUE(isnan(test.FindByDichotomy().roots[1]));

EXPECT\_TRUE(isnan(test.FindByDichotomy().roots[2]));

}

TEST(RootsFinder, Dichotomy0100) {

SPolynom pol = { 0, 1, 0, 0 };

RootsFinder test(pol);

EXPECT\_DOUBLE\_EQ(test.FindByDichotomy().roots[0], 0);

EXPECT\_TRUE(isnan(test.FindByDichotomy().roots[1]));

EXPECT\_TRUE(isnan(test.FindByDichotomy().roots[2]));

}

TEST(RootsFinder, Dichotomy1000) {

SPolynom pol = { 1, 0, 0, 0 };

RootsFinder test(pol);

EXPECT\_DOUBLE\_EQ(test.FindByDichotomy().roots[0], 0);

EXPECT\_TRUE(isnan(test.FindByDichotomy().roots[1]));

EXPECT\_TRUE(isnan(test.FindByDichotomy().roots[2]));

}

TEST(RootsFinder, Dichotomy0011) {

SPolynom pol = { 0, 0, 1, 1 };

RootsFinder test(pol);

EXPECT\_DOUBLE\_EQ(test.FindByDichotomy().roots[0], -1);

EXPECT\_TRUE(isnan(test.FindByDichotomy().roots[1]));

EXPECT\_TRUE(isnan(test.FindByDichotomy().roots[2]));

}

TEST(RootsFinder, Dichotomy0110) {

SPolynom pol = { 0, 1, 1, 0 };

RootsFinder test(pol);

EXPECT\_TRUE(test.FindByDichotomy().roots[0] <= 0.001);

EXPECT\_DOUBLE\_EQ(test.FindByDichotomy().roots[1], -1);

EXPECT\_TRUE(isnan(test.FindByDichotomy().roots[2]));

}

TEST(RootsFinder, Dichotomy1100) {

SPolynom pol = { 1, 1, 0, 0 };

RootsFinder test(pol);

EXPECT\_TRUE(test.FindByDichotomy().roots[0] <= 0.001);

EXPECT\_DOUBLE\_EQ(test.FindByDichotomy().roots[1], -1);

EXPECT\_TRUE(isnan(test.FindByDichotomy().roots[2]));

}

TEST(RootsFinder, Secant0000) {

SPolynom pol = { 0, 0, 0, 0 };

RootsFinder test(pol);

EXPECT\_TRUE(isnan(test.FindBySecant().roots[0]));

EXPECT\_TRUE(isnan(test.FindBySecant().roots[1]));

EXPECT\_TRUE(isnan(test.FindBySecant().roots[2]));

}

TEST(RootsFinder, Secant0001) {

SPolynom pol = { 0, 0, 0, 1 };

RootsFinder test(pol);

EXPECT\_TRUE(isnan(test.FindBySecant().roots[0]));

EXPECT\_TRUE(isnan(test.FindBySecant().roots[1]));

EXPECT\_TRUE(isnan(test.FindBySecant().roots[2]));

}

TEST(RootsFinder, Secant0010) {

SPolynom pol = { 0, 0, 1, 0 };

RootsFinder test(pol);

SResult test\_result = { NULL, NULL, NULL, 0 };

EXPECT\_DOUBLE\_EQ(test.FindBySecant().roots[0], 0);

EXPECT\_TRUE(isnan(test.FindBySecant().roots[1]));

EXPECT\_TRUE(isnan(test.FindBySecant().roots[2]));

}

TEST(RootsFinder, Secant0100) {

SPolynom pol = { 0, 1, 0, 0 };

RootsFinder test(pol);

EXPECT\_DOUBLE\_EQ(test.FindBySecant().roots[0], 0);

EXPECT\_TRUE(isnan(test.FindBySecant().roots[1]));

EXPECT\_TRUE(isnan(test.FindBySecant().roots[2]));

}

TEST(RootsFinder, Secant1000) {

SPolynom pol = { 1, 0, 0, 0 };

RootsFinder test(pol);

EXPECT\_DOUBLE\_EQ(test.FindBySecant().roots[0], 0);

EXPECT\_TRUE(isnan(test.FindBySecant().roots[1]));

EXPECT\_TRUE(isnan(test.FindBySecant().roots[2]));

}

TEST(RootsFinder, Secant0011) {

SPolynom pol = { 0, 0, 1, 1 };

RootsFinder test(pol);

EXPECT\_DOUBLE\_EQ(test.FindBySecant().roots[0], -1);

EXPECT\_TRUE(isnan(test.FindBySecant().roots[1]));

EXPECT\_TRUE(isnan(test.FindBySecant().roots[2]));

}

TEST(RootsFinder, Secant0110) {

SPolynom pol = { 0, 1, 1, 0 };

RootsFinder test(pol);

EXPECT\_TRUE(test.FindBySecant().roots[0] <= 0.001);

EXPECT\_DOUBLE\_EQ(test.FindBySecant().roots[1], -1);

EXPECT\_TRUE(isnan(test.FindBySecant().roots[2]));

}

TEST(RootsFinder, Secant1100) {

SPolynom pol = { 1, 1, 0, 0 };

RootsFinder test(pol);

EXPECT\_TRUE(test.FindBySecant().roots[0] <= 0.001);

EXPECT\_DOUBLE\_EQ(test.FindBySecant().roots[1], -1);

EXPECT\_TRUE(isnan(test.FindBySecant().roots[2]));

}

//////////////////////////////////////////////////

TEST(CPolynom4, AccessorHandlesZero) {

SPolynom input = { 0, 0, 0, 0 };

CPolynom4 polynom(input);

ASSERT\_TRUE(input.a == polynom.GetPolynom().a);

ASSERT\_TRUE(input.b == polynom.GetPolynom().b);

ASSERT\_TRUE(input.c == polynom.GetPolynom().c);

ASSERT\_TRUE(input.d == polynom.GetPolynom().d);

}

TEST(CPolynom4, AccessorHandlesBigInput) {

SPolynom input = { 1000000, 1000000, 1000000, 1000000 };

CPolynom4 polynom(input);

ASSERT\_TRUE(input.a == polynom.GetPolynom().a);

ASSERT\_TRUE(input.b == polynom.GetPolynom().b);

ASSERT\_TRUE(input.c == polynom.GetPolynom().c);

ASSERT\_TRUE(input.d == polynom.GetPolynom().d);

}

TEST(CPolynom4, AccessorHandlesNegativeInput) {

SPolynom input = { -4245, -252, -74225, -3742 };

CPolynom4 polynom(input);

ASSERT\_TRUE(input.a == polynom.GetPolynom().a);

ASSERT\_TRUE(input.b == polynom.GetPolynom().b);

ASSERT\_TRUE(input.c == polynom.GetPolynom().c);

ASSERT\_TRUE(input.d == polynom.GetPolynom().d);

}

TEST(CPolynom4, AccessorHandlesFloatInput) {

SPolynom input = { 4245.346, -252.2626, -74225.35643, -3742.2457 };

CPolynom4 polynom(input);

ASSERT\_TRUE(input.a == polynom.GetPolynom().a);

ASSERT\_TRUE(input.b == polynom.GetPolynom().b);

ASSERT\_TRUE(input.c == polynom.GetPolynom().c);

ASSERT\_TRUE(input.d == polynom.GetPolynom().d);

}

TEST(CPolynom4, MutatorHandlesZero) {

CPolynom4 polynom1(4345.36, 734.3568, 34783.445, 95563.4537);

SPolynom input = { 0, 0, 0, 0 };

polynom1.SetPolynom(input);

ASSERT\_TRUE(input.a == polynom1.GetPolynom().a);

ASSERT\_TRUE(input.b == polynom1.GetPolynom().b);

ASSERT\_TRUE(input.c == polynom1.GetPolynom().c);

ASSERT\_TRUE(input.d == polynom1.GetPolynom().d);

CPolynom4 polynom2(4345.36, 734.3568, 34783.445, 95563.4537);

polynom2.SetPolynom(0, 0, 0, 0);

ASSERT\_TRUE(input.a == polynom2.GetPolynom().a);

ASSERT\_TRUE(input.b == polynom2.GetPolynom().b);

ASSERT\_TRUE(input.c == polynom2.GetPolynom().c);

ASSERT\_TRUE(input.d == polynom2.GetPolynom().d);

}

TEST(CPolynom4, MutatorHandlesBigInput) {

CPolynom4 polynom1(4345.36, 734.3568, 34783.445, 95563.4537);

SPolynom input = { 1000000, 1000000, 1000000, 1000000 };

polynom1.SetPolynom(input);

ASSERT\_TRUE(input.a == polynom1.GetPolynom().a);

ASSERT\_TRUE(input.b == polynom1.GetPolynom().b);

ASSERT\_TRUE(input.c == polynom1.GetPolynom().c);

ASSERT\_TRUE(input.d == polynom1.GetPolynom().d);

CPolynom4 polynom2(4345.36, 734.3568, 34783.445, 95563.4537);

polynom2.SetPolynom(1000000, 1000000, 1000000, 1000000);

ASSERT\_TRUE(input.a == polynom2.GetPolynom().a);

ASSERT\_TRUE(input.b == polynom2.GetPolynom().b);

ASSERT\_TRUE(input.c == polynom2.GetPolynom().c);

ASSERT\_TRUE(input.d == polynom2.GetPolynom().d);

}

TEST(CPolynom4, MutatorHandlesNegativeInput) {

CPolynom4 polynom1(4345.36, 734.3568, 34783.445, 95563.4537);

SPolynom input = { -4245, -252, -74225, -3742 };

polynom1.SetPolynom(input);

ASSERT\_TRUE(input.a == polynom1.GetPolynom().a);

ASSERT\_TRUE(input.b == polynom1.GetPolynom().b);

ASSERT\_TRUE(input.c == polynom1.GetPolynom().c);

ASSERT\_TRUE(input.d == polynom1.GetPolynom().d);

CPolynom4 polynom2(4345.36, 734.3568, 34783.445, 95563.4537);

polynom2.SetPolynom(-4245, -252, -74225, -3742);

ASSERT\_TRUE(input.a == polynom2.GetPolynom().a);

ASSERT\_TRUE(input.b == polynom2.GetPolynom().b);

ASSERT\_TRUE(input.c == polynom2.GetPolynom().c);

ASSERT\_TRUE(input.d == polynom2.GetPolynom().d);

}

TEST(CPolynom4, MutatorHandlesFloatInput) {

CPolynom4 polynom1(4345.36, 734.3568, 34783.445, 95563.4537);

SPolynom input = { 4245.346, -252.2626, -74225.35643, -3742.2457 };

polynom1.SetPolynom(input);

ASSERT\_TRUE(input.a == polynom1.GetPolynom().a);

ASSERT\_TRUE(input.b == polynom1.GetPolynom().b);

ASSERT\_TRUE(input.c == polynom1.GetPolynom().c);

ASSERT\_TRUE(input.d == polynom1.GetPolynom().d);

CPolynom4 polynom2(4345.36, 734.3568, 34783.445, 95563.4537);

polynom2.SetPolynom(4245.346, -252.2626, -74225.35643, -3742.2457);

ASSERT\_TRUE(input.a == polynom2.GetPolynom().a);

ASSERT\_TRUE(input.b == polynom2.GetPolynom().b);

ASSERT\_TRUE(input.c == polynom2.GetPolynom().c);

ASSERT\_TRUE(input.d == polynom2.GetPolynom().d);

}

TEST(CPolynom4, GetResultZero) {

CPolynom4 polynom(0, 0, 0, 0);

EXPECT\_DOUBLE\_EQ(polynom.GetResult(5), 0);

EXPECT\_DOUBLE\_EQ(polynom.GetResult(36.2346), 0);

EXPECT\_DOUBLE\_EQ(polynom.GetResult(0), 0);

EXPECT\_DOUBLE\_EQ(polynom.GetResult(-34534.23), 0);

}

TEST(CPolynom4, GetResultNormal) {

CPolynom4 polynom(1, -4, -7, 10);

EXPECT\_DOUBLE\_EQ(polynom.GetResult(-2), 0);

EXPECT\_DOUBLE\_EQ(polynom.GetResult(1), 0);

EXPECT\_DOUBLE\_EQ(polynom.GetResult(5), 0);

EXPECT\_DOUBLE\_EQ(polynom.GetResult(2), -12);

EXPECT\_TRUE(fabs(polynom.GetResult(3.361) - (- 20.745)) <= 0.001);

}

TEST(CPolynom4, GetDerivativeCanFindZero) {

CPolynom4 polynom(1, -4, -7, 10);

EXPECT\_TRUE(fabs(polynom.GetDerivative(3.361) - 0) <= 0.001);

EXPECT\_TRUE(fabs(polynom.GetDerivative(-0.6943) - 0) <= 0.001);

}

TEST(CPolynom4, GetDerivativeHandlesNormalInput) {

CPolynom4 polynom(1, -4, -7, 10);

EXPECT\_DOUBLE\_EQ(polynom.GetDerivative(4), 9);

EXPECT\_DOUBLE\_EQ(polynom.GetDerivative(6), 53);

}

TEST(CPolynom4, GetIntegralHandlesZero) {

CPolynom4 polynom(1, -4, -7, 10);

EXPECT\_DOUBLE\_EQ(polynom.GetIntegral(0, 0), 0);

EXPECT\_DOUBLE\_EQ(polynom.GetIntegral(0, 1), 5.41666666666667);

EXPECT\_DOUBLE\_EQ(polynom.GetIntegral(-1, 0), 11.91666666666667);

}

TEST(CPolynom4, GetIntegralHandlesReverce) {

CPolynom4 polynom(1, -4, -7, 10);

EXPECT\_DOUBLE\_EQ(polynom.GetIntegral(1, -1), -17.33333333333333);

EXPECT\_DOUBLE\_EQ(polynom.GetIntegral(0, -2), -19.33333333333333);

}

TEST(CPolynom4, GetRootsHandlesZero) {

CPolynom4 polynom(1, -4, -7, 10);

EXPECT\_TRUE(fabs(polynom.GetRoots().roots[0] - 5) <= 0.001);

EXPECT\_TRUE(fabs(polynom.GetRoots().roots[1] - 1) <= 0.001);

EXPECT\_TRUE(fabs(polynom.GetRoots().roots[2] - (-2)) <= 0.001);

EXPECT\_TRUE(fabs(polynom.GetRoots(-20, 39).roots[0] - 5) <= 0.001);

EXPECT\_TRUE(fabs(polynom.GetRoots(-20, 39).roots[1] - 1) <= 0.001);

EXPECT\_TRUE(fabs(polynom.GetRoots(-20, 39).roots[2] - (-2)) <= 0.001);

EXPECT\_TRUE(fabs(polynom.GetRoots(-23, 13, 0.0000000001).roots[0] - 5) <= 0.0000000001);

EXPECT\_TRUE(fabs(polynom.GetRoots(-23, 13, 0.0000000001).roots[1] - 1) <= 0.0000000001);

EXPECT\_TRUE(fabs(polynom.GetRoots(-23, 13, 0.0000000001).roots[2] - (-2)) <= 0.0000000001);

}

## Lib:

### RootsHeader.h:

#pragma once

#include <cmath>

using namespace std;

typedef long double ldouble;

typedef struct{

ldouble a;

ldouble b;

ldouble c;

ldouble d;

} SPolynom;

typedef struct {

ldouble roots[3];

int num;

} SResult;

class RootsFinder {

private:

ldouble leftLim, rightLim, eps, uLeft, uRight;

SResult result;

SPolynom polynom;

ldouble f(ldouble x);

int D(SPolynom pol);

ldouble FindSecant(ldouble leftLim, ldouble rightLim);

bool SetLimits(bool isFirst);

int numOfRoots;

public:

RootsFinder(SPolynom polynom);

RootsFinder(ldouble leftLim, ldouble rightLim, SPolynom polynom);

RootsFinder(ldouble leftLim, ldouble rightLim, ldouble epsilon, SPolynom polynom);

SResult FindBySecant();

SResult FindByDichotomy();

};

### GetRoots.cpp:

#include <iostream>

#include "RootsHeader.h"

ldouble RootsFinder::f(ldouble x) {

return polynom.a \* x \* x \* x + polynom.b \* x \* x + polynom.c \* x + polynom.d;

}

int RootsFinder::D(SPolynom pol) {

int result = 0;

ldouble D = 0;

if (fabs(pol.a) <= 0.000000000000001) {

if (fabs(pol.b) <= 0.000000000000001 && fabs(pol.c) <= 0.000000000000001) {

result = 0;

cout << "Equation has no roots" << endl << endl;

}

else if (fabs(pol.b) <= 0.000000000000001) {

result = 1;

}

else {

D = pol.c \* pol.c - 4 \* pol.b \* pol.d;

if (D < 0)

result = 0;

else if (fabs(D) < 0.000000000000001)

result = 1;

else

result = 2;

}

}

else if (fabs(pol.a) > 0.000000000000001 && fabs(pol.b) <= 0.000000000000001 && fabs(pol.c) <= 0.000000000000001 && fabs(pol.d) <= 0.000000000000001) {

result = 1;

}

else {

ldouble r = pol.b / pol.a, s = pol.c / pol.a, t = pol.d / pol.a;

ldouble p = (3 \* s - r \* r) / 3, q = (2 \* r \* r \* r) / 27 - r \* s / 3 + t;

D = p \* p \* p / 27 + q \* q / 4;

if (D > 0)

result = 1;

else if (fabs(D) < 0.000000000000001)

result = 2;

else

result = 3;

}

return result;

}

ldouble RootsFinder::FindSecant(ldouble leftLim, ldouble rightLim) {

return leftLim - (f(leftLim) \* (rightLim - leftLim)) / (f(rightLim) - f(leftLim));

}

SResult RootsFinder::FindBySecant() {

ldouble savedRightLim = rightLim;

bool resultIsFound = false, isFirst = true;

while (numOfRoots) {

resultIsFound = SetLimits(isFirst);

isFirst = false;

if (!result.num)

break;

if (resultIsFound) {

result.roots[numOfRoots - 1] = leftLim;

}

else {

result.roots[numOfRoots - 1] = FindSecant(leftLim, rightLim);

ldouble staticPoint = 0;

ldouble xPrev = result.roots[numOfRoots - 1];

while (!resultIsFound) {

if (f(result.roots[numOfRoots - 1]) \* f(leftLim) <= 0)

staticPoint = leftLim;

else

staticPoint = rightLim;

xPrev = result.roots[numOfRoots - 1];

result.roots[numOfRoots - 1] = FindSecant(result.roots[numOfRoots - 1], staticPoint);

if (fabs(result.roots[numOfRoots - 1] - xPrev) <= eps)

resultIsFound = true;

}

}

numOfRoots--;

resultIsFound = false;

leftLim = rightLim;

rightLim = savedRightLim;

}

return result;

}

SResult RootsFinder::FindByDichotomy()

{

ldouble midLim = 0, savedRightLim = rightLim;

bool BordersAreSetting = true, resultIsFound = false, isFirst = true;

while (numOfRoots) {

resultIsFound = SetLimits(isFirst);

isFirst = false;

if (!result.num)

break;

if (resultIsFound)

midLim = leftLim;

while (!resultIsFound) {

midLim = (leftLim + rightLim) / 2;

if (f(leftLim) \* f(midLim) <= 0) {

rightLim = midLim;

}

else {

leftLim = midLim;

}

midLim = (leftLim + rightLim) / 2;

if (fabs(leftLim - rightLim) < eps)

resultIsFound = true;

}

result.roots[numOfRoots - 1] = midLim;

numOfRoots--;

resultIsFound = false;

leftLim = rightLim;

rightLim = savedRightLim;

}

return result;

}

bool RootsFinder::SetLimits(bool isFirst) {

bool onGoing = true, resultIsFound = false;

ldouble step = (fabs(leftLim) + fabs(rightLim)) / 10, fX = 0, fLeft = 0;

if (leftLim > rightLim) {

ldouble temp = leftLim;

leftLim = rightLim;

rightLim = temp;

}

if (step >= 1)

step = 1;

fLeft = f(leftLim);

if (fLeft == 0 && isFirst) {

resultIsFound = true;

onGoing = false;

}

while (onGoing) {

for (ldouble x = leftLim + step; x < rightLim; x += step) {

fX = f(x);

if (fabs(fX) <= eps) {

leftLim = x;

ldouble k = x + eps;

while (fabs(k) <= eps)

k += eps;

rightLim = k;

resultIsFound = true;

onGoing = false;

break;

}

if (fLeft \* fX <= 0) {

leftLim = x - step;

rightLim = x;

onGoing = false;

break;

}

fLeft = fX;

}

step /= 10;

if (step <= eps) {

result.num = 0;

result.roots[0] = NULL;

result.roots[1] = NULL;

result.roots[2] = NULL;

cout << "There are no roots in range [" << uLeft << "; " << uRight << "], or not all of them. Please, set your range" << endl << endl;

onGoing = false;

}

}

return resultIsFound;

}

RootsFinder::RootsFinder(SPolynom polynom) {

this->leftLim = -100;

this->rightLim = 100;

this->polynom = polynom;

eps = 0.000001;

uLeft = leftLim;

uRight = rightLim;

result.roots[0] = NAN;

result.roots[1] = NAN;

result.roots[2] = NAN;

result.num = D(polynom);

numOfRoots = result.num;

}

RootsFinder::RootsFinder(ldouble leftLim, ldouble rightLim, SPolynom polynom) {

this->leftLim = leftLim;

this->rightLim = rightLim;

this->polynom = polynom;

eps = 0.000001;

uLeft = leftLim;

uRight = rightLim;

result.roots[0] = NAN;

result.roots[1] = NAN;

result.roots[2] = NAN;

result.num = D(polynom);

numOfRoots = result.num;

}

RootsFinder::RootsFinder(ldouble leftLim, ldouble rightLim, ldouble epsilon, SPolynom polynom) {

this->leftLim = leftLim;

this->rightLim = rightLim;

this->polynom = polynom;

eps = epsilon;

uLeft = leftLim;

uRight = rightLim;

result.roots[0] = NAN;

result.roots[1] = NAN;

result.roots[2] = NAN;

result.num = D(polynom);

numOfRoots = result.num;

}

# Результат виконання

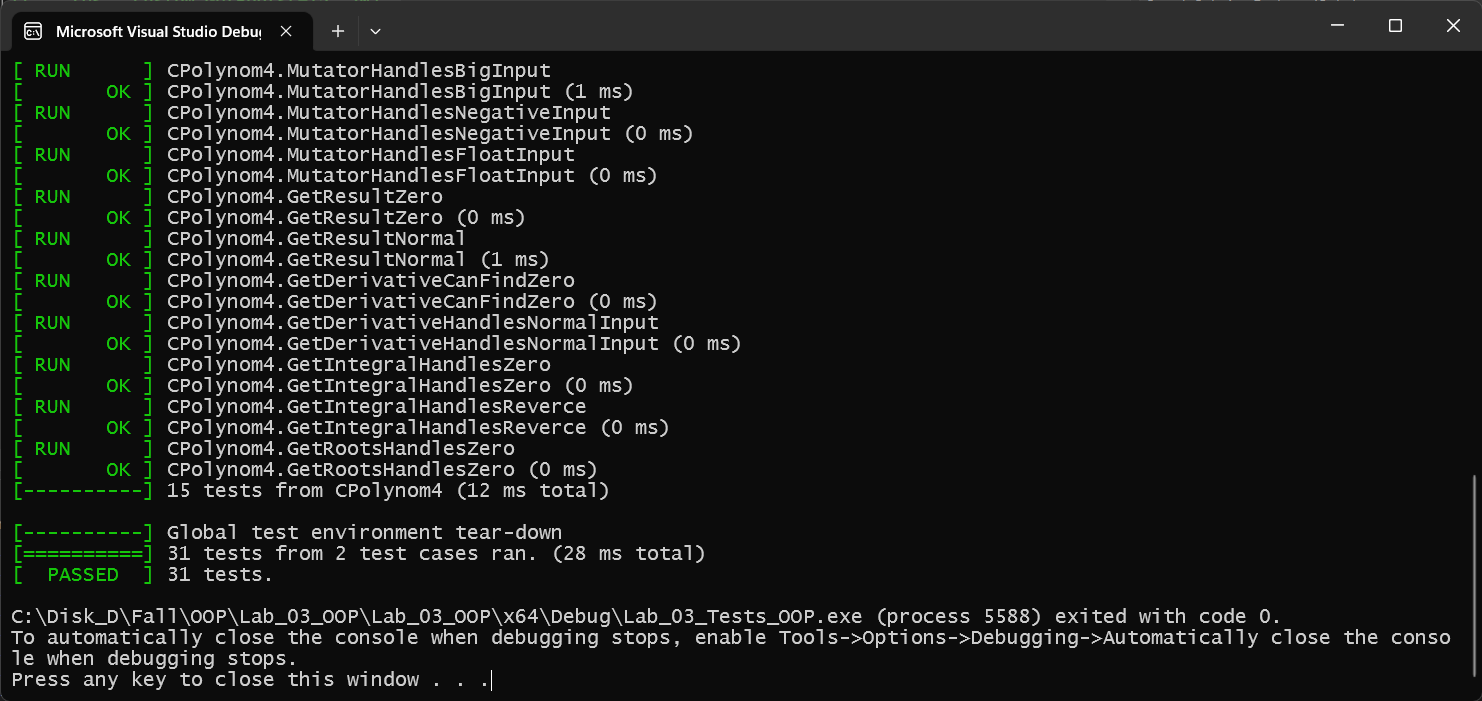


Рис 1. Виконання тестів

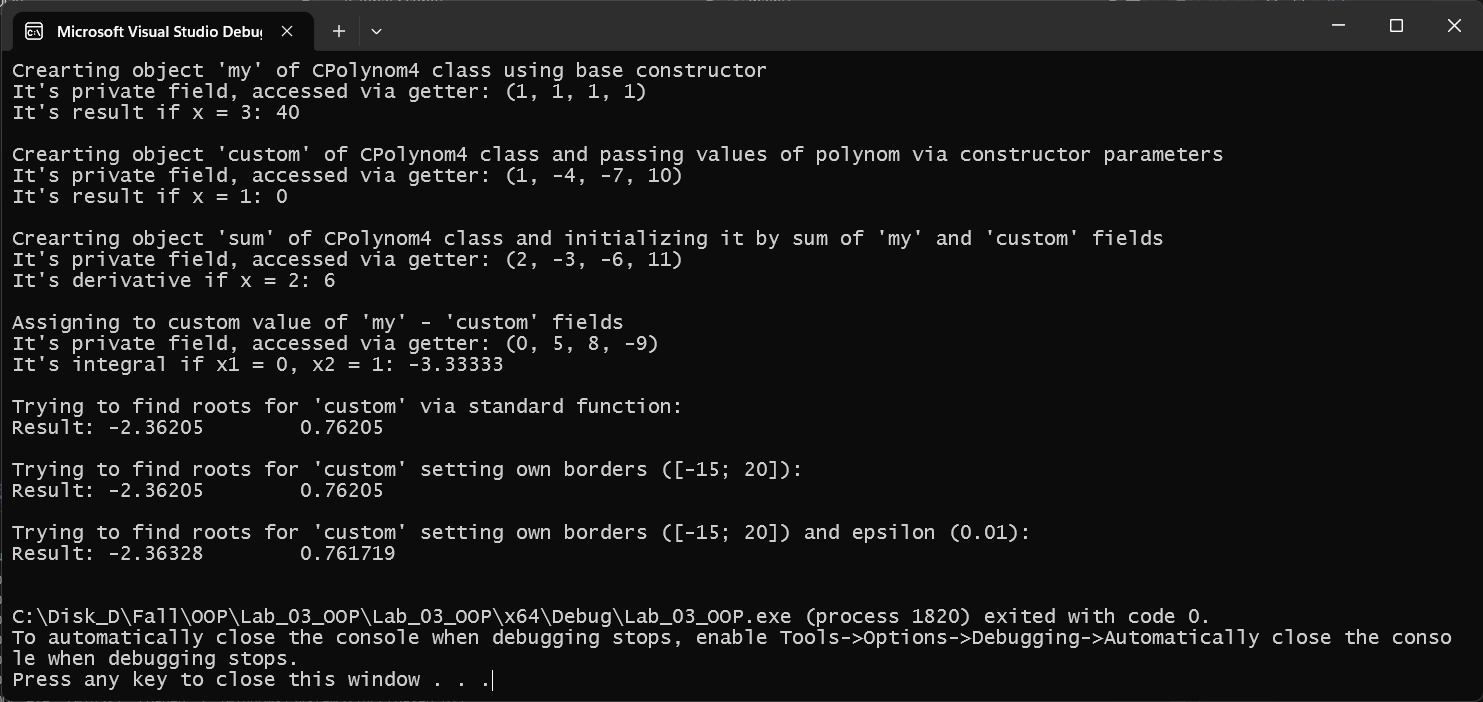


Рис 2. Виконання деяких методів класу



Рис 3. UML-діаграма класу

# Висновок

Під час виконання лабораторної роботи, я навчився створювати класи, використовувати конструктори для ініціалізації об’єктів, опанував принципи створення функцій-членів. Навчився використовувати різні типи доступу до полів та методів класів.

Зміст

[Теоретичні відомості 2](#_Toc130162182)

[Завдання 2](#_Toc130162183)

[Код програми 2](#_Toc130162184)

[Lab\_03\_OOP: 2](#_Toc130162185)

[UI: 2](#_Toc130162186)

[Lab\_03\_OOP.cpp: 2](#_Toc130162187)

[PolynomHeader.h: 2](#_Toc130162188)

[Lab\_03\_Tests\_OOP: 2](#_Toc130162189)

[pch.h: 2](#_Toc130162190)

[test.cpp: 2](#_Toc130162191)

[Lib: 2](#_Toc130162192)

[RootsHeader.h: 2](#_Toc130162193)

[GetRoots.cpp: 2](#_Toc130162194)

[Результат виконання 2](#_Toc130162195)

[Висновок 2](#_Toc130162196)

[Зміст 2](#_Toc130162197)