Федеральное агентство по образованию Российской Федерации

Государственное образовательное учреждение

Высшего профессионального образования

Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий математики механики

Выполнил:

студент ИИТММ гр. 381706-2

Антипин А.С.\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Проверил:

ассистент каф. МОСТ, ИИТММ

Лебедев И.Г\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

Полиномы

Отчет по лабораторной работе

Нижний Новгород

2018 г.

**Содержание**

[Введение 3](#_Toc536786900)

[Постановка целей и задач 4](#_Toc536786901)

[Руководство пользователя 5](#_Toc536786902)

[Руководство программиста 6](#_Toc536786903)

[Описание структуры программы 6](#_Toc536786904)

[Описание структур данных 8](#_Toc536786905)

[Описание алгоритмов 10](#_Toc536786906)

[Эксперименты 11](#_Toc536786907)

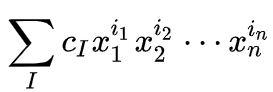
[Время выполнения 11](#_Toc536786908)

[Заключение 13](#_Toc536786909)

[Литература 14](#_Toc536786910)

[Приложение 15](#_Toc536786911)

# Введение

В математике огромное значение имеет удобное для визуального представления и правильное с технической точки зрения представление выражений и функций для их дальнейшего решения. Полиномы является одним из таких способов. Полиномом от n переменных называется мономов или, строго конечная формальная сумма вида, где - набор из целых неотрицательных чисел, являющимися степенями x-ов, а - число, именуемое коэффициент монома. Изучение полиномиальных уравнений и их решений составляло едва ли не главный объект «классической алгебры». Многочлены также играют ключевую роль в алгебраической геометрии, объектом которой являются множества, определённые как решения систем многочленов. Особые свойства преобразования коэффициентов при умножении многочленов используются в алгебраической геометрии, алгебре, теории узлов и других разделах математики для кодирования или выражения многочленами свойств различных объектов.

# Постановка целей и задач

Целью лабораторной работы является создание структуры хранения типа «Полином» и методов работы с ним, таких как:

* Математические операции с полиномами;
* Доступ к отдельному моному в полиноме;
* Добавление к полиному мономов.

Для реализации алгоритмов будет использоваться 2 класса:

* Monom;
* Polinom.

Для проверки правильности работы этих классов будут написаны тесты с использованием фреймворка Google Test, а также тестовый образец программы, которая будет использует класс список.

# Руководство пользователя

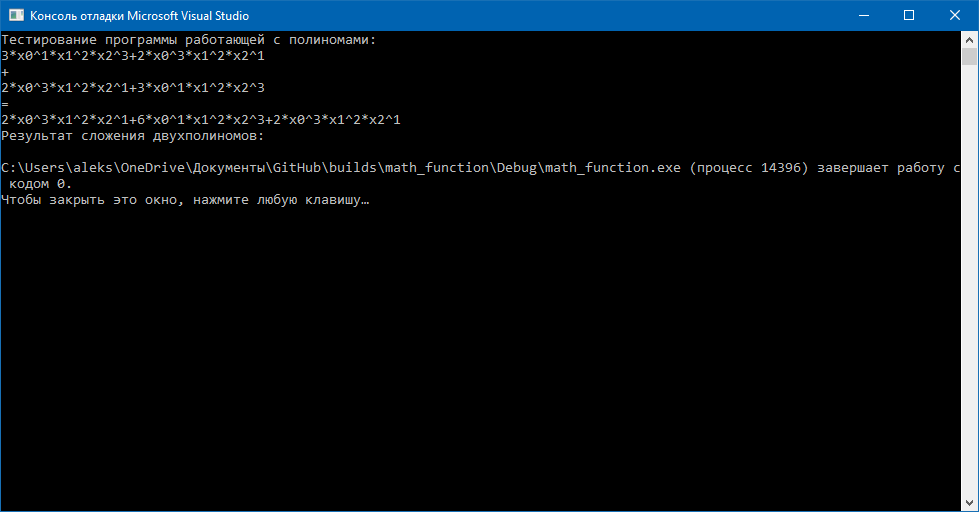
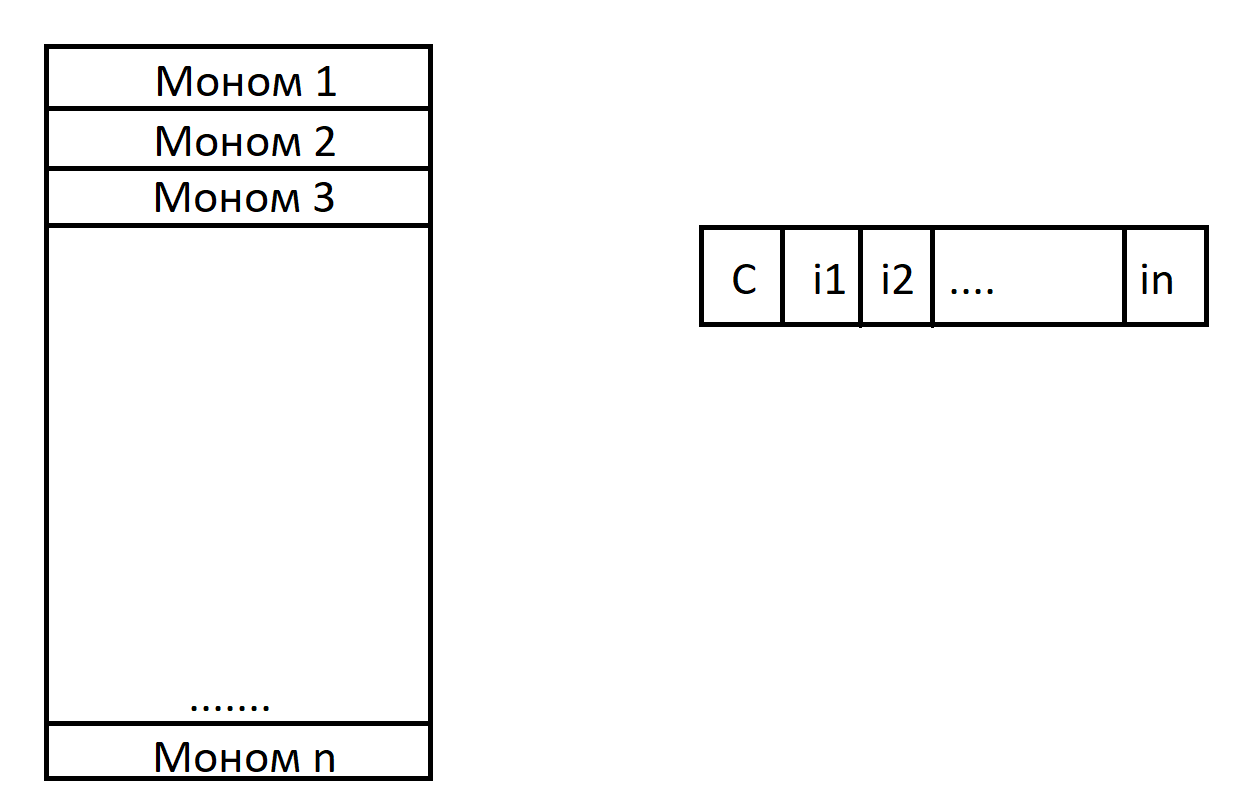
После запуска программы пользователя встречает консольное окно (рис. 1):

рис. 1 (вывод программы тестирования полиномов для пользователя)

в которой сначала приводятся два полинома, а затем выводится сумма этих полиномов.

# Руководство программиста

## Описание структуры программы

Класс «полином» будет реализован с помощью набора элементов «моном», соответственно каждый моном будет состоять из вещественного коэффициента и массива степеней x-ов:

Где с – коэффициент монома;

i1, i2, …., in – степени монома.

То есть для реализации алгоритмов будет использовано 2 класса:

* Класс «Моном» (Monom).
* Класс «Полином» (Polinom), который будет наследоваться от класса List с шаблоном в виде Monom.

А также проект использующий фреймворк Google Test, для проверки правильности работы этих классов и тесовый проект, который будет показываться пользователю.

Древо классов

**Класс gtest**

**test\_main.cpp**

**Monom\_test.cpp**

**Polinom\_test.cpp**

**Проект для пользователя**

**main.cpp**

**Polinom.h**

**Класс List**

**Monom.h**

**Класс TDatLink**

**Polinom.cpp**

**Monom.cpp**

**Класс Monom:**

Класс «Моном» содержит представление отдельных мономов полинома. Также в этом классе перегружены математические операции присущие мономам, такие как сложение, вычитание и др., а также операции сравнения.

**Класс Polinom:**

В классе «Полином» организована структура полинома – списка мономов. Был взят класс List и в качестве шаблона использовался моном. Здесь перегружены математические операции с многочленами и вывод этих многочленов на консоль.

**Класс gtest:**

Класс gtest реализует тестирование классов Monom и Polinom, по средствам фреймворка Google Test. Тесты пишутся для каждого метода классов, каждого ветвления этих методов и для всех возможных исключений этих методов.

**Проект List:**

В данном проекте реализован примет использования многочленов, в котором складываются два монома и результат выводится на консоль.

## Описание структур данных

**Класс Monom:**

int n – количество переменных х;

double c – коэффициент монома;

unsigned int\* power – массив степеней монома;

Exceptions\_from\_polinom\_and\_calculator exception – переменная для вызова исключений;

**Класс Polinom:**

Exceptions\_from\_polinom\_and\_calculator exception – переменная для вызова исключений;

**Описание методов:**

|  |  |
| --- | --- |
| Метод: | Описание: |
| Monom::Monom() | Конструктор по умолчанию для класса Monom. |
| Monom::Monom(int \_n, unsigned int\* \_power, double \_c) | Конструктор с параметром для класса Monom. Первый параметр – количество переменных х, второй – степени этих переменных, третий – коэффициент монома. |
| Monom::Monom(const Monom& monom) | Конструктор копирования класса Monom. |
| Monom::~Monom() | Деструктор монома. |
| int Monom::GetN() | Метод, который возвращает количество переменных х. |
| double Monom::GetC() | Метод, который возвращает коэффициент монома. |
| unsigned int\* Monom::GetPower() | Метод, который возвращает указатель на массив степеней монома. |
| void Monom::SetN(int \_n) | Метод, который позволяет изменить количество переменных в мономе. Если их становится больше, то добавляются новые с единичными степенями, если меньше, то удаляются последние. |
| void Monom::SetC(double \_c) | Метод, который позволяет изменить коэффициент монома. |
| void Monom::SetPower(unsigned int\* \_power) | Метод, позволяющий изменить степени монома. |
| Monom& Monom::operator=(const Monom& monom) | Перегрузка оператора присваивания для монома. |
| Monom Monom::operator+(const Monom& monom) | Перегрузка операции сложения мономов. |
| Monom Monom::operator-(const Monom& monom) | Перегрузка оператора вычитание для мономов. |
| Monom Monom::operator\*(const Monom& monom) | Перегрузка оператора умножения для мономов. |
| bool Monom::operator==(const Monom& monom) | Перегрузка операции сравнения для мономов. |
| bool Monom::operator>(const Monom& monom) | Перегрузка операции больше для мономов. |
| bool Monom::operator<(const Monom& monom) | Перегрузка операции меньше для мономов. |
| std::ostream& operator<<(std::ostream& os, Monom& monom) | Перегрузка вывода монома на консоль. |
| Polinom::Polinom() : List<Monom>() | Конструктор по умолчанию для полинома, который наследуется от конструктора по умолчанию списка с шаблоном монома. |
| Polinom::Polinom(const Polinom& polinom) | Конструктор копирования для полинома. |
| Polinom::Polinom(const List<Monom>& ls) | Конструктор преобразования типа из списка мономов в полином. |
| Polinom& Polinom::operator=(Polinom& polinom) | Перегрузка оператора присваивания полиномов. |
| Polinom Polinom::operator+(Polinom& polinom) | Перегрузка оператора сложения для полиномов. |
| Polinom Polinom::operator-(Polinom& polinom) | Перегрузка оператора вычитания для полиномов. |
| Polinom Polinom::operator\*(Polinom& polinom) | Перегрузка оператора умножение для полиномов. |
| Polinom& Polinom::operator+=(Monom& monom) | Прибавление к текущему полиному монома. |
| Monom Polinom::operator[](const int nomber) | Доступ к отдельному моному в полиноме. |
| std::ostream& operator<<(std::ostream& os, Polinom& polinom) | Перегрузка вывода полинома на консоль. |

## Описание алгоритмов

**Подробное описание некоторых методов**

Изменение количества переменных в мономе:

* Проверка переданного значения на не отрицательность;
* Копирование предыдущих значений степеней во временную переменную и очистка основного поля. Выделение нового количества памяти;
* Выяснение какая из переменных (старая или новая) больше;
* Копирование в заново выделенную память значений из временного хранилища.
* При необходимости добавление единичных степеней для новых переменных х.

Сложение полиномов:

* Проверка на совпадение длин полиномов, если длины не совпадают, то вызывается исключение;
* Создаются указатели на первые элементы полиномов, и результирующий полином;
* В цикле происходит проверка на равенство мономов в полиноме и в случае успеха, они складываются. Если моном в полиноме больше соответствующего монома в другом полиноме, то сначала записывается больший, а затем дописываются меньшие;
* Возвращается результирующий полином.

# Эксперименты

## Время выполнения

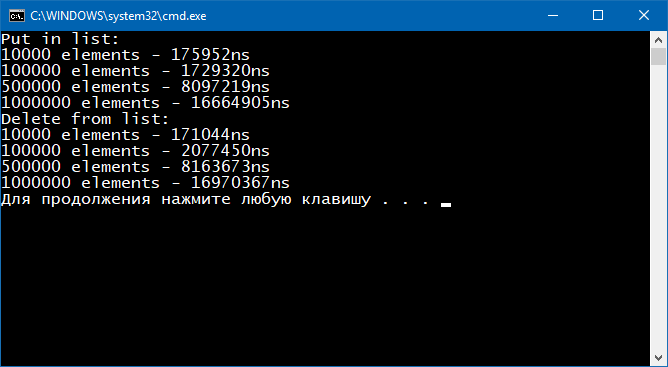
Рассмотрим время выполнения некоторых методов. Сложение мономов – сложность O(n) и умножение двух полиномов – сложность O(n\*m\*z), где n и m – количество мономов в полиномах, а z – количество переменных в мономах (рис. 2) (время приведено в наносекундах).

рис. 2 (время работы сложения мономов и умножения полиномов)

Составим диаграмму с временем работы:

С учетом погрешности измерения, можно сказать, что теоретическая сложность алгоритмов совпадает с практическими измерениями.

Тесты проводились на системе:

Процессор Intel Core i5 7200U;

Оперативная память 12 GB.

# Заключение

В заключении можно сказать, что все поставленные цели и задачи были выполнены, а именно: созданы классы «Monom» и «Polinom» с реализованными математическими операциями над ними, а также написаны к ним тесты, и они успешно пройдены.

# Литература

* Учебные материалы к учебному курсу «Методы программирования» - Гергель В.П.
* https://ru.wikipedia.org/wiki/Многочлен

# Приложение

**Monom.h:**

# #pragma once

#include <iostream>

#include "Exception\_class.h"

class Monom

{

protected:

int n;//размер

double c;//коэффициент

unsigned int\* power;//степени

Exceptions\_from\_polinom\_and\_calculator exception;

public:

Monom();

Monom(const int \_n, unsigned int\* \_power, double \_c);

Monom(const Monom& monom);

~Monom();

int GetN();//вернуть размер

double GetC();//вернуть коэффициент

unsigned int\* GetPower();//вернуть указатель на степени

void SetN(int \_n);//изменить размер

void SetC(double \_c);//изменить коэффициент

void SetPower(unsigned int\* \_power);//изменить степени

Monom& operator=(const Monom& monom);

Monom operator+(const Monom& monom);

Monom operator-(const Monom& monom);

Monom operator\*(const Monom& monom);

bool operator==(const Monom& monom);

bool operator>(const Monom& monom);

bool operator<(const Monom& monom);

friend std::ostream& operator<<(std::ostream& os, Monom& monom);

};

**Monom.cpp:**

#include "Monom.h"

#include <iostream>

Monom::Monom()

{

}

Monom::Monom(const int \_n, unsigned int\* \_power, double \_c)

{

if (\_n <= 0)

exception.except\_throw(101);

n = \_n;

c = \_c;

power = new unsigned int[n];

if (power != NULL)

for (int i = 0; i < n; i++)

power[i] = \_power[i];

else

for (int i = 0; i < n; i++)

power[i] = 0;

}

Monom::Monom(const Monom& monom)

{

n = monom.n;

c = monom.c;

power = new unsigned int[n];

for (int i = 0; i < n; i++)

power[i] = monom.power[i];

}

Monom::~Monom()

{

if (power != NULL)

delete[] power;

}

int Monom::GetN()

{

return n;

}

double Monom::GetC()

{

return c;

}

unsigned int\* Monom::GetPower()

{

return power;

}

void Monom::SetN(int \_n)

{

if (\_n <= 0)

exception.except\_throw(101);

unsigned int\* tmp = new unsigned int[n];

for (int i = 0; i < n; i++)

tmp[i] = power[i];

delete[] power;

power = new unsigned int[\_n];

bool flag = true;

int more = \_n;

int less = n;

if (n > \_n)

{

flag = false;

more = \_n;

less = n;

}

for (int i = 0; i < less; i++)

power[i] = tmp[i];

if (flag = true)

for (int i = less; i < more; i++)

power[i] = 1;

delete[] tmp;

n = \_n;

}

void Monom::SetC(double \_c)

{

c = \_c;

}

void Monom::SetPower(unsigned int\* \_power)

{

try

{

for (int i = 0; i < n; i++)

if (\_power[i] >= 0)

power[i] = \_power[i];

else

exception.except\_throw(102);

}

catch (int ex)

{

exception.except\_throw(100);

}

}

Monom& Monom::operator=(const Monom& monom)

{

if (this != &monom)

{

n = monom.n;

c = monom.c;

if (power)

{

power = NULL;

delete[] power;

}

power = new unsigned int[n];

for (int i = 0; i < n; i++)

power[i] = monom.power[i];

}

return \*this;

}

Monom Monom::operator+(const Monom& monom)

{

if (n != monom.n)

exception.except\_throw(103);

Monom tmp(monom);

bool flag = true;

for (int i = 0; i < n; i++)

if (this->power[i] != monom.power[i])

flag = false;

if (flag == true)

tmp.c = c + monom.c;

else

exception.except\_throw(104);

return tmp;

}

Monom Monom::operator-(const Monom& monom)

{

if (n != monom.n)

exception.except\_throw(103);

Monom tmp(monom);

bool flag = true;

for (int i = 0; i < n; i++)

if (this->power[i] != monom.power[i])

flag = false;

if (flag == true)

tmp.c = c - monom.c;

else

exception.except\_throw(104);

return tmp;

}

Monom Monom::operator\*(const Monom& monom)

{

if (n != monom.n)

exception.except\_throw(103);

Monom tmp(monom);

tmp.c = c \* monom.c;

for (int i = 0; i < n; i++)

tmp.power[i] = power[i] + monom.power[i];

return tmp;

}

bool Monom::operator==(const Monom& monom)

{

bool res = true;

if (n != monom.n)

exception.except\_throw(103);

for (int i = 0; i < n; i++)

if (power[i] != monom.power[i])

res = false;

if (c != monom.c)

res = false;

return res;

}

bool Monom::operator>(const Monom& monom)

{

bool res = true;

if (n != monom.n)

exception.except\_throw(103);

for (int i = 0; i < n; i++)

if (power[i] > monom.power[i])

return true;

else if (power[i] < monom.power[i])

return false;

else

res = false;

if (res == false)

if (c < monom.c)

res = false;

else

res = true;

return res;

}

bool Monom::operator<(const Monom& monom)

{

bool res = true;

if (n != monom.n)

exception.except\_throw(103);

for (int i = 0; i < n; i++)

if (power[i] < monom.power[i])

return true;

else if (power[i] > monom.power[i])

return false;

else

res = false;

if (res == false)

if (c > monom.c)

res = false;

else

res = true;

return res;

}

std::ostream& operator<<(std::ostream& os, Monom& monom)

{

os << monom.c << '\*';

for (int i = 0; i < monom.n - 1; i++)

os << "x" << i << "^" << monom.power[i] << '\*';

os << "x" << monom.n - 1 << "^" << monom.power[monom.n - 1];

return os;

}

**Polinom.h:**

#pragma once

#include "Monom.h"

#include "List.h"

class Polinom : public List<Monom>

{

protected:

Exceptions\_from\_polinom\_and\_calculator exception;

public:

Polinom();

Polinom(const Polinom& polinom);

Polinom(const List<Monom>& MonomList);

Polinom& operator=(Polinom& polinom);

Polinom operator+(Polinom& polinom);

Polinom operator-(Polinom& polinom);

Polinom operator\*(Polinom& polinom);

Polinom& operator+=(Monom& monom);

Monom operator[](const int nomber);

friend std::ostream& operator<<(std::ostream& os, Polinom& polinom);

};

**Polinom.cpp:**

#include "Polinom.h"

Polinom::Polinom() : List<Monom>()

{

}

Polinom::Polinom(const Polinom& polinom)

{

ListLen = polinom.ListLen;

TDatLink<Monom>\* tmp = polinom.FirstItem;

FirstItem = new TDatLink<Monom>(tmp->GetMem(), NULL);

TDatLink<Monom>\* TMP = FirstItem;

TDatLink<Monom>\* tmpCurr;

for (int i = 1; i < polinom.ListLen; i++)

{

tmp = tmp->GetNextLink();

tmpCurr = new TDatLink<Monom>(tmp->GetMem(), NULL);

TMP->Set\_m(tmpCurr);

TMP = tmpCurr;

}

LastItem = tmpCurr;

}

Polinom::Polinom(const List<Monom>& ls)

{

ListLen = ls.GetListLen();

FirstItem = new TDatLink<Monom>(ls.GetValue(0), NULL);

TDatLink<Monom>\* tmp = FirstItem;

TDatLink<Monom>\* tmpCurr;

for (int i = 1; i < ListLen; i++)

{

tmpCurr = new TDatLink<Monom>(ls.GetValue(i), NULL);

tmp->Set\_m(tmpCurr);

tmp = tmp->GetNextLink();

}

LastItem = tmpCurr;

}

Polinom& Polinom::operator=(Polinom& polinom)

{

if (this != &polinom)

{

DelList();

ListLen = polinom.ListLen;

TDatLink<Monom>\* tmp = polinom.FirstItem;

FirstItem = new TDatLink<Monom>(tmp->GetMem(), NULL);

TDatLink<Monom>\* TMP = FirstItem;

TDatLink<Monom>\* tmpCurr;

for (int i = 1; i < polinom.ListLen; i++)

{

tmp = tmp->GetNextLink();

tmpCurr = new TDatLink<Monom>(tmp->GetMem(), NULL);

TMP->Set\_m(tmpCurr);

TMP = tmpCurr;

}

LastItem = tmpCurr;

}

return \*this;

}

Polinom Polinom::operator+(Polinom& polinom)

{

if (ListLen != polinom.ListLen)

exception.except\_throw(105);

TDatLink<Monom>\* tmp\_1 = FirstItem;

TDatLink<Monom>\* tmp\_2 = polinom.FirstItem;

Polinom res;

while (tmp\_1 && tmp\_2)

if (tmp\_1->GetMem() == tmp\_2->GetMem())

{

res.InstLast(tmp\_1->GetMem() + tmp\_2->GetMem());

tmp\_1 = tmp\_1->GetNextLink();

tmp\_2 = tmp\_2->GetNextLink();

}

else if (tmp\_1->GetMem() > tmp\_2->GetMem())

{

res.InstLast(tmp\_1->GetMem());

tmp\_1 = tmp\_1->GetNextLink();

}

else

{

res.InstLast(tmp\_2->GetMem());

tmp\_2 = tmp\_2->GetNextLink();

}

if (!tmp\_1)

tmp\_1 = tmp\_2;

while (tmp\_1)

{

res.InstLast(tmp\_1->GetMem());

tmp\_1 = tmp\_1->GetNextLink();

}

return res;

}

Polinom Polinom::operator-(Polinom& polinom)

{

if (ListLen != polinom.ListLen)

exception.except\_throw(105);

TDatLink<Monom>\* tmp\_1 = FirstItem;

TDatLink<Monom>\* tmp\_2 = polinom.FirstItem;

Polinom res;

while (tmp\_1 && tmp\_2)

if (tmp\_1->GetMem() == tmp\_2->GetMem())

{

res.InstLast(tmp\_1->GetMem() - tmp\_2->GetMem());

tmp\_1 = tmp\_1->GetNextLink();

tmp\_2 = tmp\_2->GetNextLink();

}

else if (tmp\_1->GetMem() > tmp\_2->GetMem())

{

Monom A(tmp\_1->GetMem().GetN(), tmp\_1->GetMem().GetPower(), 0);

res.InstLast(A - tmp\_1->GetMem());

tmp\_1 = tmp\_1->GetNextLink();

}

else

{

Monom A(tmp\_2->GetMem().GetN(), tmp\_2->GetMem().GetPower(), 0);

res.InstLast(A - tmp\_2->GetMem());

tmp\_2 = tmp\_2->GetNextLink();

}

if (!tmp\_1)

tmp\_1 = tmp\_2;

while (tmp\_1)

{

res.InstLast(tmp\_1->GetMem());

tmp\_1 = tmp\_1->GetNextLink();

}

return res;

}

Polinom Polinom::operator\*(Polinom& polinom)

{

Polinom res;

TDatLink<Monom>\* tmp\_1 = FirstItem;

TDatLink<Monom>\* tmp\_2 = polinom.FirstItem;

for (int i = 0; i < ListLen; i++)

{

for (int i = 0; i < polinom.ListLen; i++)

{

res.InstLast(tmp\_1->GetMem() \* tmp\_2->GetMem());

tmp\_2 = tmp\_2->GetNextLink();

}

tmp\_1 = tmp\_1->GetNextLink();

tmp\_2 = polinom.FirstItem;

}

return res;

}

Polinom& Polinom::operator+=(Monom& monom)

{

for (int i = 0; i < ListLen; i++)

{

Monom tmp = GetValue(i);

if (tmp == monom)

{

DelCustom(i);

InstCustom(i, monom + tmp);

return \*this;

}

else if (tmp < monom)

{

InstCustom(i, monom);

return \*this;

}

}

InstLast(monom);

return \*this;

}

Monom Polinom::operator[](const int nomber)

{

return GetValue(nomber);

}

std::ostream& operator<<(std::ostream& os, Polinom& polinom)

{

TDatLink<Monom>\* tmp = polinom.FirstItem;

for (int i = 0; i < polinom.ListLen - 1; i++)

{

os << polinom.GetValue(i) << '+';

tmp = tmp->GetNextLink();

}

os << polinom.GetValue(polinom.ListLen - 1) << std::endl;

return os;

}

**Monom\_test.cpp:**

#include <gtest.h>

#include "Monom.h"

TEST(monom, can\_create\_monom)

{

unsigned int B[]{ 3, 2, 1 };

ASSERT\_NO\_THROW(Monom A(3, B, 3.14));

}

TEST(monom, can\_get\_monom\_size)

{

unsigned int B[]{ 3, 2, 1 };

Monom A(3, B, 3.14);

EXPECT\_EQ(A.GetN(), 3);

}

TEST(monom, can\_get\_monom\_coefficient)

{

unsigned int B[]{ 3, 2, 1 };

Monom A(3, B, 3.14);

EXPECT\_EQ(A.GetC(), 3.14);

}

TEST(monom, can\_get\_monom\_power)

{

unsigned int B[]{ 3, 2, 1 };

Monom A(3, B, 3.14);

EXPECT\_EQ(A.GetPower()[0], B[0]);

EXPECT\_EQ(A.GetPower()[1], B[1]);

EXPECT\_EQ(A.GetPower()[2], B[2]);

}

TEST(monom, can\_throw\_if\_set\_new\_negetive\_size)

{

unsigned int B[]{ 3, 2, 1 };

Monom A(3, B, 3.14);

ASSERT\_ANY\_THROW(A.SetN(-1));

}

TEST(monom, can\_set\_new\_size\_without\_throws)

{

unsigned int B[]{ 3, 2, 1 };

Monom A(3, B, 3.14);

ASSERT\_NO\_THROW(A.SetN(4));

}

TEST(monom, can\_set\_new\_size)

{

unsigned int B[]{ 3, 2, 1 };

Monom A(3, B, 3.14);

A.SetN(4);

EXPECT\_EQ(A.GetN(), 4);

}

TEST(monom, can\_set\_new\_coefficient\_without\_throws)

{

unsigned int B[]{ 3, 2, 1 };

Monom A(3, B, 3.14);

ASSERT\_NO\_THROW(A.SetC(6.28));

}

TEST(monom, can\_set\_new\_coefficient)

{

unsigned int B[]{ 3, 2, 1 };

Monom A(3, B, 3.14);

A.SetC(6.28);

EXPECT\_EQ(A.GetC(), 6.28);

}

TEST(monom, can\_set\_new\_power\_without\_throws)

{

unsigned int B[]{ 3, 2, 1 };

unsigned int C[]{ 1, 2, 3 };

Monom A(3, B, 3.14);

ASSERT\_NO\_THROW(A.SetPower(C));

}

TEST(monom, can\_throw\_if\_set\_new\_power\_with\_negative\_element)

{

unsigned int B[]{ 3, 2, 1 };

unsigned int C[]{ 1, -2, 3 };

Monom A(3, B, 3.14);

ASSERT\_NO\_THROW(A.SetPower(C));

}

TEST(monom, can\_set\_new\_power)

{

unsigned int B[]{ 3, 2, 1 };

unsigned int C[]{ 1, 2, 3 };

Monom A(3, B, 3.14);

A.SetPower(C);

EXPECT\_EQ(A.GetPower()[0], C[0]);

EXPECT\_EQ(A.GetPower()[1], C[1]);

EXPECT\_EQ(A.GetPower()[2], C[2]);

}

TEST(monom, can\_assign\_monom\_to\_yourself\_without\_throws)

{

unsigned int B[]{ 3, 2, 1 };

Monom A(3, B, 3.14);

ASSERT\_NO\_THROW(A = A);

}

TEST(monom, can\_assign\_monom\_to\_yourself)

{

unsigned int B[]{ 3, 2, 1 };

Monom A(3, B, 3.14);

A = A;

EXPECT\_EQ(A.GetC(), 3.14);

EXPECT\_EQ(A.GetN(), 3);

EXPECT\_EQ(A.GetPower()[0], B[0]);

EXPECT\_EQ(A.GetPower()[1], B[1]);

EXPECT\_EQ(A.GetPower()[2], B[2]);

}

TEST(monom, can\_assign\_without\_throws)

{

unsigned int B[]{ 3, 2, 1 };

unsigned int C[]{ 1, 2, 3 };

Monom A(3, B, 3.14);

Monom D(3, C, 5);

ASSERT\_NO\_THROW(A = D);

}

TEST(monom, can\_assign\_monoms)

{

unsigned int B[]{ 3, 2, 1 };

unsigned int C[]{ 1, 2, 3 };

Monom A(3, B, 3.14);

Monom D(3, C, 5);

A = D;

EXPECT\_EQ(A.GetC(), 5);

EXPECT\_EQ(A.GetN(), 3);

EXPECT\_EQ(A.GetPower()[0], C[0]);

EXPECT\_EQ(A.GetPower()[1], C[1]);

EXPECT\_EQ(A.GetPower()[2], C[2]);

}

TEST(monom, can\_add\_monoms\_with\_different\_size)

{

unsigned int B[]{ 3, 2, 1 };

unsigned int C[]{ 1, 2, 3, 4 };

Monom A(3, B, 3.14);

Monom D(4, C, 5);

ASSERT\_ANY\_THROW(A + D);

}

TEST(monom, can\_add\_monoms\_with\_different\_power)

{

unsigned int B[]{ 3, 2, 1 };

unsigned int C[]{ 1, 2, 3,};

Monom A(3, B, 3.14);

Monom D(3, C, 5);

ASSERT\_ANY\_THROW(A + D);

}

TEST(monom, can\_add\_monoms)

{

unsigned int B[]{ 3, 2, 1 };

unsigned int C[]{ 3, 2, 1 };

Monom A(3, B, 3.14);

Monom D(3, C, 5);

A = A + D;

EXPECT\_EQ(A.GetC(), 8.14);

EXPECT\_EQ(A.GetN(), 3);

EXPECT\_EQ(A.GetPower()[0], C[0]);

EXPECT\_EQ(A.GetPower()[1], C[1]);

EXPECT\_EQ(A.GetPower()[2], C[2]);

}

TEST(monom, can\_subtract\_monoms\_with\_different\_size)

{

unsigned int B[]{ 3, 2, 1 };

unsigned int C[]{ 1, 2, 3, 4 };

Monom A(3, B, 3.14);

Monom D(4, C, 5);

ASSERT\_ANY\_THROW(A - D);

}

TEST(monom, can\_subtract\_monoms\_with\_different\_power)

{

unsigned int B[]{ 3, 2, 1 };

unsigned int C[]{ 1, 2, 3, };

Monom A(3, B, 3.14);

Monom D(3, C, 5);

ASSERT\_ANY\_THROW(A - D);

}

TEST(monom, can\_subtract\_monoms)

{

unsigned int B[]{ 3, 2, 1 };

unsigned int C[]{ 3, 2, 1 };

Monom A(3, B, 3);

Monom D(3, C, 5);

A = D - A;

EXPECT\_EQ(A.GetC(), 2);

EXPECT\_EQ(A.GetN(), 3);

EXPECT\_EQ(A.GetPower()[0], C[0]);

EXPECT\_EQ(A.GetPower()[1], C[1]);

EXPECT\_EQ(A.GetPower()[2], C[2]);

}

TEST(monom, can\_multiply\_monoms\_with\_different\_size)

{

unsigned int B[]{ 3, 2, 1 };

unsigned int C[]{ 1, 2, 3, 4 };

Monom A(3, B, 3.14);

Monom D(4, C, 5);

ASSERT\_ANY\_THROW(A \* D);

}

TEST(monom, can\_multiply\_monoms)

{

unsigned int B[]{ 3, 2, 1 };

unsigned int C[]{ 1, 3, 5 };

unsigned int E[]{ 4, 5, 6 };

Monom A(3, B, 3);

Monom D(3, C, 5);

A = A \* D;

EXPECT\_EQ(A.GetC(), 15);

EXPECT\_EQ(A.GetN(), 3);

EXPECT\_EQ(A.GetPower()[0], E[0]);

EXPECT\_EQ(A.GetPower()[1], E[1]);

EXPECT\_EQ(A.GetPower()[2], E[2]);

}

TEST(monom, can\_compare\_monoms\_with\_different\_size)

{

unsigned int B[]{ 3, 2, 1 };

unsigned int C[]{ 1, 2, 3, 4 };

Monom A(3, B, 3.14);

Monom D(4, C, 5);

ASSERT\_ANY\_THROW(A == D);

}

TEST(monom, can\_compare\_monoms\_1)

{

unsigned int B[]{ 3, 2, 1 };

unsigned int C[]{ 1, 2, 3 };

Monom A(3, B, 3.14);

Monom D(3, C, 5);

EXPECT\_EQ(A == D, 0);

}

TEST(monom, can\_compare\_monoms\_2)

{

unsigned int B[]{ 3, 2, 1 };

unsigned int C[]{ 3, 2, 1 };

Monom A(3, B, 3.14);

Monom D(3, C, 5);

EXPECT\_EQ(A == D, 0);

}

TEST(monom, can\_compare\_monoms\_3)

{

unsigned int B[]{ 3, 2, 1 };

unsigned int C[]{ 3, 2, 1 };

Monom A(3, B, 5);

Monom D(3, C, 5);

EXPECT\_EQ(A == D, 1);

}

TEST(monom, can\_compare\_more\_monoms\_with\_defferent\_size)

{

unsigned int B[]{ 3, 2, 1 };

unsigned int C[]{ 1, 2, 3, 4};

Monom A(3, B, 3.14);

Monom D(4, C, 5);

ASSERT\_ANY\_THROW(A > D);

}

TEST(monom, can\_compare\_more\_monoms\_1)

{

unsigned int B[]{ 1, 2, 3 };

unsigned int C[]{ 3, 2, 1 };

Monom A(3, B, 3.14);

Monom D(3, C, 5);

EXPECT\_EQ(A > D, 0);

}

TEST(monom, can\_compare\_more\_monoms\_2)

{

unsigned int B[]{ 3, 2, 1 };

unsigned int C[]{ 3, 2, 1 };

Monom A(3, B, 3);

Monom D(3, C, 5);

EXPECT\_EQ(A > D, 0);

}

TEST(monom, can\_compare\_more\_monoms\_3)

{

unsigned int B[]{ 3, 2, 1 };

unsigned int C[]{ 3, 2, 1 };

Monom A(3, B, 6);

Monom D(3, C, 5);

EXPECT\_EQ(A > D, 1);

}

TEST(monom, can\_compare\_more\_monoms\_4)

{

unsigned int B[]{ 3, 2, 1 };

unsigned int C[]{ 1, 2, 3 };

Monom A(3, B, 3);

Monom D(3, C, 5);

EXPECT\_EQ(A > D, 1);

}

TEST(monom, can\_compare\_less\_monoms\_with\_defferent\_size)

{

unsigned int B[]{ 3, 2, 1 };

unsigned int C[]{ 1, 2, 3, 4 };

Monom A(3, B, 3.14);

Monom D(4, C, 5);

ASSERT\_ANY\_THROW(A < D);

}

TEST(monom, can\_compare\_less\_monoms\_1)

{

unsigned int B[]{ 1, 2, 3 };

unsigned int C[]{ 3, 2, 1 };

Monom A(3, B, 3.14);

Monom D(3, C, 5);

EXPECT\_EQ(D < A, 0);

}

TEST(monom, can\_compare\_less\_monoms\_2)

{

unsigned int B[]{ 3, 2, 1 };

unsigned int C[]{ 3, 2, 1 };

Monom A(3, B, 3);

Monom D(3, C, 5);

EXPECT\_EQ(D < A, 0);

}

TEST(monom, can\_compare\_less\_monoms\_3)

{

unsigned int B[]{ 3, 2, 1 };

unsigned int C[]{ 3, 2, 1 };

Monom A(3, B, 4);

Monom D(3, C, 5);

EXPECT\_EQ(A < D, 1);

}

TEST(monom, can\_compare\_less\_monoms\_4)

{

unsigned int B[]{ 1, 2, 3 };

unsigned int C[]{ 3, 2, 1 };

Monom A(3, B, 3);

Monom D(3, C, 5);

EXPECT\_EQ(A < D, 1);

}

**Polinom\_test.cpp:**

#include <gtest.h>

#include "Polinom.h"

TEST(polinom, can\_create\_polinom)

{

ASSERT\_NO\_THROW(Polinom A);

}

TEST(polinom, can\_transformate\_list\_monoms\_to\_polinom)

{

unsigned int B[]{ 3, 2, 1 };

Monom C(3, B, 3.14);

List<Monom> A;

A.InstFirst(C);

A.InstFirst(C);

A.InstFirst(C);

ASSERT\_NO\_THROW(Polinom P(A));

}

TEST(polinom, can\_assign\_polinom)

{

Polinom P;

unsigned int B[]{ 3, 2, 1 };

Monom C(3, B, 3.14);

P.InstFirst(C);

P.InstFirst(C);

Polinom Po;

ASSERT\_NO\_THROW(Po = P);

}

TEST(polinom, can\_throw\_if\_add\_polinoms\_with\_different\_lenght)

{

Polinom P;

unsigned int B[]{ 3, 2, 1 };

Monom C(3, B, 3.14);

P.InstFirst(C);

P.InstFirst(C);

Polinom Po;

Po.InstFirst(C);

ASSERT\_ANY\_THROW(P + Po);

}

TEST(polinom, can\_add\_polinoms\_1)

{

Polinom P;

unsigned int B[]{ 3, 2, 1 };

Monom C(3, B, 2);

P.InstFirst(C);

P.InstFirst(C);

Polinom Po;

Po.InstFirst(C);

Po.InstFirst(C);

Polinom rez;

rez = P + Po;

EXPECT\_EQ(rez.GetValue(0).GetC(), 4);

EXPECT\_EQ(rez.GetValue(1).GetC(), 4);

}

TEST(polinom, can\_add\_polinoms\_2)

{

Polinom P;

unsigned int B[]{ 3, 2, 1 };

unsigned int A[]{ 1, 2, 3 };

Monom C(3, B, 2);

Monom D(3, A, 3);

P.InstFirst(C);

P.InstFirst(D);

Polinom Po;

Po.InstFirst(D);

Po.InstFirst(C);

Polinom rez;

rez = P + Po;

EXPECT\_EQ(rez.GetValue(0).GetC(), 2);

EXPECT\_EQ(rez.GetValue(1).GetC(), 6);

EXPECT\_EQ(rez.GetValue(2).GetC(), 2);

}

TEST(polinom, can\_throw\_if\_subtract\_polinoms\_with\_different\_lenght)

{

Polinom P;

unsigned int B[]{ 3, 2, 1 };

Monom C(3, B, 3.14);

P.InstFirst(C);

P.InstFirst(C);

Polinom Po;

Po.InstFirst(C);

ASSERT\_ANY\_THROW(P - Po);

}

TEST(polinom, can\_subtract\_polinoms\_1)

{

Polinom P;

unsigned int B[]{ 3, 2, 1 };

Monom C(3, B, 2);

P.InstFirst(C);

P.InstFirst(C);

Polinom Po;

Po.InstFirst(C);

Po.InstFirst(C);

Polinom rez;

rez = P - Po;

EXPECT\_EQ(rez.GetValue(0).GetC(), 0);

EXPECT\_EQ(rez.GetValue(1).GetC(), 0);

}

TEST(polinom, can\_subtract\_polinoms\_2)

{

Polinom P;

unsigned int B[]{ 3, 2, 1 };

unsigned int A[]{ 1, 2, 3 };

Monom C(3, B, 2);

Monom D(3, A, 3);

P.InstFirst(C);

P.InstFirst(D);

Polinom Po;

Po.InstFirst(D);

Po.InstFirst(C);

Polinom rez;

rez = P - Po;

EXPECT\_EQ(rez.GetValue(0).GetC(), -2);

EXPECT\_EQ(rez.GetValue(1).GetC(), 0);

EXPECT\_EQ(rez.GetValue(2).GetC(), 2);

}

TEST(polinom, can\_multiply\_polinoms)

{

Polinom P;

unsigned int B[]{ 3, 2, 1 };

unsigned int A[]{ 1, 2, 3 };

Monom C(3, B, 2);

Monom D(3, A, 3);

P.InstFirst(C);

P.InstFirst(D);

Polinom Po;

Po.InstFirst(D);

Po.InstFirst(C);

Polinom rez;

rez = P \* Po;

unsigned int rez\_rower[]{ 4, 4, 4 };

EXPECT\_EQ(rez.GetValue(0).GetC(), 6);

EXPECT\_EQ(rez.GetValue(0).GetN(), 3);

EXPECT\_EQ(rez.GetValue(0).GetPower()[0], rez\_rower[0]);

EXPECT\_EQ(rez.GetValue(0).GetPower()[1], rez\_rower[1]);

EXPECT\_EQ(rez.GetValue(0).GetPower()[2], rez\_rower[2]);

}

TEST(polinom, can\_add\_polinom\_with\_monom)

{

Polinom P;

unsigned int B[]{ 3, 2, 1 };

unsigned int A[]{ 1, 2, 3 };

Monom C(3, B, 2);

Monom D(3, A, 3);

P.InstFirst(C);

P.InstFirst(C);

ASSERT\_NO\_THROW(P += D);

}

TEST(polinom, can\_add\_polinom\_with\_monom\_2)

{

Polinom P;

unsigned int B[]{ 3, 2, 1 };

unsigned int A[]{ 1, 2, 3 };

Monom C(3, B, 2);

Monom D(3, A, 3);

P.InstFirst(D);

P.InstFirst(C);

P += D;

EXPECT\_EQ(P.GetValue(1).GetC(), 6);

}

TEST(polinom, can\_return\_monom)

{

Polinom P;

unsigned int B[]{ 3, 2, 1 };

Monom C(3, B, 2);

P.InstFirst(C);

P.InstFirst(C);

EXPECT\_EQ(P[1].GetC(), 2);

EXPECT\_EQ(P[1].GetN(), 3);

EXPECT\_EQ(P[1].GetPower()[0], B[0]);

EXPECT\_EQ(P[1].GetPower()[1], B[1]);

EXPECT\_EQ(P[1].GetPower()[2], B[2]);

}

**main.cpp:**

#include <iostream>

#include "Polinom.h"

int main()

{

setlocale(LC\_ALL, "RUS");

std::cout << "Тестирование программы работающей с полиномами:\n";

Polinom P;

unsigned int B[]{ 3, 2, 1 };

unsigned int A[]{ 1, 2, 3 };

Monom C(3, B, 2);

Monom D(3, A, 3);

P.InstFirst(C);

P.InstFirst(D);

Polinom Po;

Po.InstFirst(D);

Po.InstFirst(C);

Polinom rez;

rez = P + Po;

std::cout << P << "+\n" << Po << "=\n";

std::cout << rez;

std::cout << "Результат сложения двухполиномов:\n";

return 0;

}