Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение  
высшего образования «Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского»

Институт Информационных технологий, математики и механики

Отчёт по учебной практике

Операции над полиномами

Выполнил:

студент гр. 381806-3

Зайцев А. Р.

Проверила:

к.т.н., доцент каф. МОСТ, ИИТММ

Кустикова В.Д.

Нижний Новгород

2019 г.

**Содержание**

[Введение 3](#_Toc36041984)

[1 Постановка задачи 4](#_Toc36041985)

[2 Руководство пользователя 5](#_Toc36041986)

[3 Руководство программиста 6](#_Toc36041987)

[3.1 Описание структуры программы 6](#_Toc36041988)

[3.2 Описание структур данных 6](#_Toc36041989)

[3.2.1 Структура TNode 6](#_Toc36041990)

[3.2.2 Класс TList 7](#_Toc36041991)

[3.3 Описание алгоритмов 10](#_Toc36041992)

[3.3.1 Алгоритм работы парсинга строки для монома 10](#_Toc36041993)

[3.3.2 Алгоритм работы парсинга строки для полинома 11](#_Toc36041994)

[Заключение 12](#_Toc36041995)

[Литература 13](#_Toc36041996)

[Приложения 14](#_Toc36041997)

[Приложение 1. Структура TNode 14](#_Toc36041998)

[Приложение 2. Структура Monom 15](#_Toc36041999)

[Приложение 3. Класс TList 19](#_Toc36042000)

[Приложение 4. Класс Polynom 25](#_Toc36042001)

[Приложение 5. Класс Exception 34](#_Toc36042002)

# Введение

В математике нам довольно часто приходится иметь дело с полиномами. Данный термин знаком нам ещё с младших классов. Но прежде чем говорить о полиноме, дадим определение моному.

**Моном** - это многочлен со своим коэффициентом от одной или нескольких переменных с целыми степенями. В нашем случае будут мономы от 3 переменных (x, y, z). Именно сумма мономов образует **полином**. Также дам определение полинома с точки зрения нашего курса.

**Полином** – это список, составленный из мономов, и мономы упорядочены по возрастанию степеней.

Мы будем рассматривать сложение, вычитание и умножение полиномов.

Вычислять всё это на бумаге порой долго и утомительно, но благодаря разработанной программе вычисления происходят моментально.

# Постановка задачи

**Задача:** Разработать программу, умеющую обрабатывать операции над полиномами, а именно: сложение, вычитание и умножение. Также программа должна иметь возможность дать пользователю ввести собственное выражение в строке, которое после обработается, а на выходе получится полином, отсортированный по возрастанию степеней после приведения подобных.

**Входные данные:**

1. Строка с полиномами и операциями над ними.

**Выходные данные:**

1. Один полином, отсортированный по возрастанию степеней после приведения подобных.

# Руководство пользователя

Далее я представлю пошаговую инструкцию по работе с программой.

1. Запустите приложение Polynom.exe, при запуске программа попросит вас ввести выражение с полиномами в строке, то есть полиномы с какими-либо операциями над ними (см. Рис. 1).



1. Начало программы.
2. Далее вводим выражение (можно также использовать скобки, и ставить знак «\*» как захочется, так как учтён приоритет операций). (см. Рис. 2).



1. Ввод выражения.
2. И получаем результат. (см. Рис. 3).



1. Результирующий полином

# Руководство программиста

## Описание структуры программы

Код программы содержится в файлах: main.cpp, TNode.h, Monom.h, TList.h, Polynom.h, Exception.h.

В файле TNode.h содержится шаблонная структура TNode и реализация её методов (это узел для списка).

В файле Monom.h содержится реализация структуры TNode<unsigned int, double>, то есть структуры TNode с фиксированными типами данных.

В файле TList.h содержится шаблонная реализация класса TList (это сам список, составленный из узлов).

В файле Polynom.h содержится реализация класса Polynom, который использует в качестве поля объект шаблонного класса с фиксированными типами данных TList<unsigned int, double>.

В файле Exception.h содержится класс исключений, позволяющий работать с сообщениями об ошибке.

## Описание структур данных

### Структура TNode

**Объявление структуры**

template<typename TKey, typename TData>

struct TNode

{

TKey key;

TData data;

TNode<TKey, TData>\* pNext;

TNode();

TNode(TKey key, TData data);

TNode(const TNode<TKey, TData>& node);

~TNode();

bool operator==(const TNode<TKey, TData>& node) const;

const TNode<TKey, TData>& operator=(const TNode<TKey, TData>& node);

friend ostream& operator<<(ostream& out, const TNode<TKey, TData>& node)

{

out << " {" << node.key << ", " << node.data << "} ";

return out;

}

};

**Описание полей**

key – значение ключа узла.

data – какие-либо данные, хранимые в узле.

pNext – указатель на следующий узел.

### Класс TList

**Объявление класса**

template<typename TKey, typename TData>

class TList

{

private:

TNode<TKey, TData>\* pFirst;

TNode<TKey, TData>\* pPrevious;

TNode<TKey, TData>\* pCurrent;

TNode<TKey, TData>\* pNext;

public:

TList();

TList(const TList<TKey, TData>& list);

TList(TNode<TKey, TData>\* node);

~TList();

TNode<TKey, TData>\* Search(TKey key);

TNode<TKey, TData>\* Search(TNode<TKey, TData>\* node);

void InsertBegin(TKey new\_key, TData data);

void InsertEnd(TKey new\_key, TData data);

void InsertAfter(TKey key, TKey new\_key, TData data);

void InsertBefore(TKey key, TKey new\_key, TData data);

void InsertBegin(TNode<TKey, TData>\* node);

void InsertEnd(TNode<TKey, TData>\* node);

void InsertAfter(TKey key, TNode<TKey, TData>\* node);

void InsertBefore(TKey key, TNode<TKey, TData>\* node);

void Remove(TKey key);

void Remove(TNode<TKey, TData>\* node);

// Служебные операции

bool IsEmpty() const;

bool IsEnded() const;

// Навигация

void Reset();

void Next();

TNode<TKey, TData>\* GetFirst() const;

TNode<TKey, TData>\* GetCurrent() const;

friend ostream& operator<<(ostream& out, const TList<TKey, TData>& list);

};

**Описание полей**

pFirst – указатель на первый узел.

pPrevious – указатель на предыдущий узел.

pCurrent – указатель на текущий узел.

pNext – указатель на следующий узел.

**Описание методов**

bool IsEmpty() const

**Назначение**: Проверка списка на пустоту.

**Входные данные**: Отсутствуют.

**Выходные данные**: true в случае пустоты и false в противном случае.

bool IsEnded() const

**Назначение**: Проверка того, что текущий узел существует.

**Входные данные**: Отсутствуют.

**Выходные данные**: true в случае, если узел не существует, и false в противном случае.

void Reset();

**Назначение**: Установка текущего элемента в начало.

**Входные данные**: Отсутствуют.

**Выходные данные**: Отсутствуют.

void Next();

**Назначение**: Сдвиг указателей на предыдущий, на текущий и на следующий на шаг вперёд.

**Входные данные**: Отсутствуют.

**Выходные данные**: Отсутствуют.

TNode<TKey, TData>\* GetFirst() const

**Назначение**: Возврат начального узла, если тот существует.

**Входные данные**: Отсутствуют.

**Выходные данные**: Начальный узел.

TNode<TKey, TData>\* GetCurrent() const

**Назначение**: Возврат текущего узла, если тот существует.

**Входные данные**: Отсутствуют.

**Выходные данные**: Текущий узел.

TNode<TKey, TData>\* Search(TKey key)

**Назначение**: Возврат узла по заданному ключу, если тот существует.

**Входные данные**: Ключ.

**Выходные данные**: Искомый узел.

TNode<TKey, TData>\* Search(TNode<TKey, TData>\* node)

**Назначение**: Возврат узла по адресу узла благодаря указателю, если тот существует.

**Входные данные**: Указатель на узел.

**Выходные данные**: Искомый узел.

void InsertBegin(TKey new\_key, TData data)

**Назначение**: Вставка в начало нового узла.

**Входные данные**: Ключ и данные нового узла.

**Выходные данные**: Отсутствуют.

void InsertEnd(TKey new\_key, TData data)

**Назначение**: Вставка в конец нового узла.

**Входные данные**: Ключ и данные нового узла.

**Выходные данные**: Отсутствуют.

void InsertAfter(TKey key, TKey new\_key, TData data)

**Назначение**: Вставка нового узла после узла с заданным ключом.

**Входные данные**: Ключ узла, после которого происходит вставка, затем ключ и данные нового узла.

**Выходные данные**: Отсутствуют.

void InsertBefore(TKey key, TKey new\_key, TData data)

**Назначение**: Вставка нового узла до узла с заданным ключом.

**Входные данные**: Ключ узла, до которого происходит вставка, затем ключ и данные нового узла.

**Выходные данные**: Отсутствуют.

void InsertBegin(TNode<TKey, TData>\* node)

**Назначение**: Вставка в начало нового узла.

**Входные данные**: Указатель на узел, ключ и данные которого хотим использовать для вставки узла.

**Выходные данные**: Отсутствуют.

void InsertEnd(TNode<TKey, TData>\* node)

**Назначение**: Вставка в конец нового узла.

**Входные данные**: Указатель на узел, ключ и данные которого хотим использовать для вставки узла.

**Выходные данные**: Отсутствуют.

void InsertAfter(TKey key, TNode<TKey, TData>\* node)

**Назначение**: Вставка нового узла после узла с заданным ключом.

**Входные данные**: Ключ узла, после которого происходит вставка, затем указатель на узел, ключ и данные которого хотим использовать для вставки узла.

**Выходные данные**: Отсутствуют.

void InsertBefore(TKey key, TNode<TKey, TData>\* node)

**Назначение**: Вставка нового узла до узла с заданным ключом.

**Входные данные**: Ключ узла, до которого происходит вставка, затем указатель на узел, ключ и данные которого хотим использовать для вставки узла.

**Выходные данные**: Отсутствуют.

void Remove(TKey key)

**Назначение**: Удаление узла по заданному ключу.

**Входные данные**: Ключ узла, если такой существует, по которому удалится узел.

**Выходные данные**: Отсутствуют.

void Remove(TNode<TKey, TData>\* node)

**Назначение**: Удаление узла по адресу заданного узла.

**Входные данные**: Указатель на узел, если такой существует, по которому удалится узел.

**Выходные данные**: Отсутствуют.

## Описание алгоритмов

### Алгоритм работы парсинга строки для монома

**На входе:** Строка с мономом.

**На выходе:** Моном.

Идея работы со строкой состоит в следующем:

1. Создаётся новая строка, куда будет записана исходная, но без пробелов и знака «\*», который пользователь может написать между переменными, как мы это делаем иногда на бумаге.
2. Затем уже в новой строке просматривается первый символ на наличие знака «+» или «-», что будет свидетельствовать о том, положительный моном или отрицательный. Если какой-нибудь из знаков есть, то мы сохраняем его в отдельную символьную переменную под знак.
3. После этого мы проверяем, равен ли следующий символ одному из чисел «0123456789». Если равен, то мы сохраняем этот символ в специально созданную под коэффициент строку. На этом мы не останавливаемся, а проверяем, есть ли числа дальше. Так мы будем добавлять все символы с числами в эту строку, пока не дойдём до переменных «x», «y» или «z».
4. Как только мы дойдём до этих переменных, останавливаемся. Затем извлекаем из строки полученное число, применяем к нему, если имеются, знак «+» или «-» из сохранённой ранее переменной под знак и полученное вновь число записываем в переменную data в узел (моном). Стоит добавить, что в случае отсутствия знака, число считается автоматически положительным. А если мы не встретили числа для коэффициента, а сразу попали на переменные, то тогда коэффициентом будет 1, которая также запишется в data.
5. Далее приступаем к обработке самих переменных. Создаём заранее 3 целочисленные переменные, в которые будем прибавлять степени от «x», «y», «z». Проходим их одну за другой, и если после одной переменной идёт сразу следующая, то у первой степень будет 1, а если после первой идёт знак степени «^» и следом за ним число от 1 до 9, то это число и будет являться степенью данной переменной, которое мы сразу прибавляем в нашу созданную переменную. Каждый раз будет проверяться, не превысило ли максимального значения (это 9) число в созданной переменной (если превысило, то выкидывается ошибка).
6. Итак, дойдя до конца строки, мы запомним итоговое значение степени для каждой из 3 переменных. По данным трёх созданных переменных получаем значение для поля key в узле (мономе) – это и будет степень монома.
7. В конце концов мы получим моном как объект со своими полями – коэффициентом и степенью.

### Алгоритм работы парсинга строки для полинома

**На входе:** Строка с выражением из полиномов и операций над ними.

**На выходе:** Один результирующий полином, составленный из отсортированных по возрастанию мономов после приведения подобных.

Идея работы с таким выражением состоит в следующем:

1. Создаётся результирующий полином, куда будут прибавляться все мономы. Строка просматривается слева направо. Мы должны обойти каждый её символ. Запоминаем индекс первого из символов, который может быть равен операциям «+» или «-», любому из чисел «0123456789» или буквам «x», «y», «z», которые характеризуют переменные для мономов. Запоминаем индекс потому, что с него начинается моном («+» или «-» в начале покажут положителен или отрицателен моном).
2. Далее мы просматриваем символ за символом, пока не встретим операцию «+» или «-». Здесь мы останавливаемся и запоминаем индекс символа до одной из операций. Это и будет конец нашего монома, так как дальше начнётся уже следующий.
3. Сохраняем подстроку, ограниченную нашим первым и последним индексом, в отдельную строку. Затем, использовав эту строку, создаём моном. После чего прибавляем полученный моном с помощью операции «+» к результирующему полиному.
4. Таким образом повторяем пункты 1-3, пока не дойдём до конца строки и не обработаем все мономы.
5. В конце возвращаем результирующий полином.

# Заключение

В ходе лабораторной работы я разработал и реализовал программу, умеющую работать с подсчётом и вычислением полиномов. При написании программы использовался список – структура данных, составленная из узлов. С его помощью было удобно хранить все данные о каждом мономе для полиномов.

# Литература

1. Презентации по курсу «Алгоритмы и структуры данных» из учебно-методических материалов ННГУ им. Лобачевского.

# Приложения

## Приложение 1. Структура TNode

#pragma once

#include <iostream>

using namespace std;

template<typename TKey, typename TData>

struct TNode

{

TKey key;

TData data;

TNode<TKey, TData>\* pNext;

TNode();

TNode(TKey key, TData data);

TNode(const TNode<TKey, TData>& node);

~TNode();

bool operator==(const TNode<TKey, TData>& node) const;

const TNode<TKey, TData>& operator=(const TNode<TKey, TData>& node);

friend ostream& operator<<(ostream& out, const TNode<TKey, TData>& node)

{

out << " {" << node.key << ", " << node.data << "} ";

return out;

}

};

template<typename TKey, typename TData>

TNode<TKey, TData>::TNode()

{

key = 0;

pNext = nullptr;

}

template<typename TKey, typename TData>

TNode<TKey, TData>::TNode(TKey key, TData data)

{

this->key = key;

this->data = data;

pNext = nullptr;

}

template<typename TKey, typename TData>

TNode<TKey, TData>::TNode(const TNode<TKey, TData>& node)

{

pNext = nullptr;

key = node.key;

data = node.data;

}

template<typename TKey, typename TData>

TNode<TKey, TData>::~TNode()

{

pNext = nullptr;

}

template<typename TKey, typename TData>

bool TNode<TKey, TData>::operator==(const TNode<TKey, TData>& node) const

{

return ((key = node.key) && (data = node.data));

}

template<typename TKey, typename TData>

const TNode<TKey, TData>& TNode<TKey, TData>::operator=

(const TNode<TKey, TData>& node)

{

if (\*this == node)

return \*this;

key = node.key;

data = node.data;

return \*this;

}

## Приложение 2. Структура Monom

#pragma once

#include "Exception.h"

#include "TNode.h"

#include <iostream>

#include <string>

using namespace std;

#define MAX\_KEY 999

#define MAX\_DEGREE 9

#define Monom TNode<unsigned int, double>

template<>

struct Monom

{

unsigned int key;

double data;

Monom\* pNext;

TNode();

TNode(int key, double data);

TNode(double data); // Нужен для умножения монома на константу

TNode(const Monom& monom);

TNode(const string& monom);

~TNode();

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

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

Monom operator-() const; // унарный минус

Monom operator+(const Monom& monom);

Monom operator-(const Monom& monom);

Monom operator\*(const Monom& monom);

friend ostream& operator<<(ostream& out, const Monom& monom);

};

Monom::TNode()

{

key = 0;

data = 0.0;

pNext = nullptr;

}

Monom::TNode(int key, double data)

{

if (key > MAX\_KEY)

throw Exception(" Error, key more than 999\n");

this->key = key;

this->data = data;

pNext = nullptr;

}

Monom::TNode(double data)

{

key = 0;

this->data = data;

pNext = nullptr;

}

Monom::TNode(const Monom& monom)

{

key = monom.key;

data = monom.data;

pNext = nullptr;

}

Monom::~TNode()

{

pNext = nullptr;

}

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

{

return ((this->key == monom.key) && (this->data == monom.data));

}

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

{

if (\*this == monom)

return \*this;

key = monom.key;

data = monom.data;

return \*this;

}

Monom Monom::operator-() const

{

return Monom(key, -data);

}

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

{

if (key != monom.key)

throw Exception(" Error of addition, degrees are different\n");

return Monom(key, data + monom.data);

}

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

{

if (key != monom.key)

throw Exception(" Error of subtraction, degrees are different\n");

return Monom(key, data - monom.data);

}

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

{

if ((key / 100 + monom.key / 100 > MAX\_DEGREE)

|| ((key % 100) / 10 + (monom.key % 100) / 10 > MAX\_DEGREE)

|| (key % 10 + monom.key % 10 > MAX\_DEGREE))

throw Exception

(" Error of multiplication, monom doesn't exist, degree > MAX\_DEGREE\n");

return Monom(key + monom.key, data \* monom.data);

}

Monom::TNode(const string& monom)

{

unsigned int \_key;

double \_data;

string new\_monom;

for (int i = 0; i < monom.length(); i++) // отбрасываем лишние символы

{

if ((monom[i] != ' ') && (monom[i] != '\*'))

new\_monom.push\_back(monom[i]);

}

int x\_degree = 0;

int y\_degree = 0;

int z\_degree = 0;

string coef;

char sign = '+'; // '+' ~ 43, '-' ~ 45

// Определяем знак

int j = 0;

if (new\_monom[j] == '+' || new\_monom[j] == '-')

if (('0' <= new\_monom[j + 1] && new\_monom[j + 1] <= '9')

|| (new\_monom[j + 1] == 'x') || (new\_monom[j + 1] == 'y')

|| (new\_monom[j + 1] == 'z'))

{

sign = new\_monom[j];

j++;

}

else

throw Exception(" Error, incorrect enter after '+' or '-'!\n");

// выделяем коэфициент и сохраняем его в отдельной строке

if ((new\_monom[j] == 'x') || (new\_monom[j] == 'y') || (new\_monom[j] == 'z'))

coef.push\_back('1'); // в таком случае коэфициент у монома равен 1

else

while ((new\_monom[j] != 'x') && (new\_monom[j] != 'y')

&& (new\_monom[j] != 'z') && (j < new\_monom.length()))

coef.push\_back(new\_monom[j++]);

// а теперь сохраняем в переменной коэфициент из строки

// в случае, если не найдём символ из приведённых,

// значение должно равняться npos (-1) (см. ниже)

if ((!coef.empty()) && (coef.find\_first\_not\_of("0123456789.") == coef.npos))

\_data = stod(coef) \* (44 - (int)sign); // stod извлекает из строки число

else

throw Exception

(" Error, monom doesn't exist, incorrect enter or incorrect coefficient\n");

// далее работаем со степенями у переменных x,y,z

for (int i = j; i < new\_monom.length(); i++)

{

switch (new\_monom[i])

{

case 'x':

{

if (new\_monom[i + 1] == '^')

if ('0' <= new\_monom[i + 2] && new\_monom[i + 2] <= '9')

if (x\_degree + new\_monom[i + 2] - '0' > MAX\_DEGREE)

throw Exception

(" Error, monom doesn't exist, degree > MAX\_DEGREE!\n");

else

{

x\_degree += new\_monom[i + 2] - '0';

i += 2;

}

else

throw Exception(" Error, after ^ must be digit (0 - 9)\n");

else if ((new\_monom[i + 1] == 'y') || (new\_monom[i + 1] == 'z')

|| (i + 1 >= new\_monom.length()))

if (x\_degree + 1 > MAX\_DEGREE)

throw Exception

(" Error, monom doesn't exist, degree > MAX\_DEGREE\n");

else

x\_degree++;

else

throw Exception(" Error, monom doesn't exist\n");

break;

}

case 'y':

{

if (new\_monom[i + 1] == '^')

if ('0' <= new\_monom[i + 2] && new\_monom[i + 2] <= '9')

if (y\_degree + new\_monom[i + 2] - '0' > MAX\_DEGREE)

throw Exception

(" Error, monom doesn't exist, degree > MAX\_DEGREE!!\n");

else

{

y\_degree += new\_monom[i + 2] - '0';

i += 2;

}

else

throw Exception(" Error, after ^ must be digit (0 - 9)\n");

else if ((new\_monom[i + 1] == 'x') || (new\_monom[i + 1] == 'z')

|| (i + 1 >= new\_monom.length()))

if (y\_degree + 1 > MAX\_DEGREE)

throw Exception

(" Error, monom doesn't exist, degree > MAX\_DEGREE\n");

else

y\_degree++;

else

throw Exception(" Error, monom doesn't exist\n");

break;

}

case 'z':

{

if (new\_monom[i + 1] == '^')

if ('0' <= new\_monom[i + 2] && new\_monom[i + 2] <= '9')

if (z\_degree + new\_monom[i + 2] - '0' > MAX\_DEGREE)

throw Exception

(" Error, monom doesn't exist, degree > MAX\_DEGREE!!!\n");

else

{

z\_degree += new\_monom[i + 2] - '0';

i += 2;

}

else

throw Exception(" Error, after ^ must be digit (0 - 9)\n");

else if ((new\_monom[i + 1] == 'x') || (new\_monom[i + 1] == 'y')

|| (i + 1 >= new\_monom.length()))

if (z\_degree + 1 > MAX\_DEGREE)

throw Exception

(" Error, monom doesn't exist, degree > MAX\_DEGREE\n");

else

z\_degree++;

else

throw Exception(" Error, monom doesn't exist\n");

break;

}

default:

throw Exception(" Error, monom doesn't exist, incorrect enter!\n");

break;

}

}

\_key = x\_degree \* 100 + y\_degree \* 10 + z\_degree;

key = \_key;

data = \_data;

}

ostream& operator<<(ostream& out, const Monom& monom)

{

if (monom.data == 0.0)

{

out << " 0 (your monom is empty)";

return out;

}

if (monom.data < 0.0)

out << " -";

if (monom.data > 0.0)

out << " +";

if ((abs(monom.data) != 1.0) || (monom.key == 0))

out << " " << abs(monom.data);

if (monom.key / 100 == 1)

{

if (abs(monom.data) == 1)

out << " x";

else

out << " \* x";

}

else if (monom.key / 100 != 0)

{

if (abs(monom.data) == 1)

out << " x^" << monom.key / 100;

else

out << " \* x^" << monom.key / 100;

}

if ((monom.key % 100) / 10 == 1)

{

if ((monom.key / 100 == 0) && (abs(monom.data) == 1))

out << " y";

else

out << " \* y";

}

else if ((monom.key % 100) / 10 != 0)

{

if ((monom.key / 100 == 0) && (abs(monom.data) == 1))

out << " y^" << (monom.key % 100) / 10;

else

out << " \* y^" << (monom.key % 100) / 10;

}

if (monom.key % 10 == 1)

{

if ((monom.key / 100 == 0) && ((monom.key % 100) / 10 == 0)

&& (abs(monom.data) == 1))

out << " z";

else

out << " \* z";

}

else if (monom.key % 10 != 0)

{

if ((monom.key / 100 == 0) && ((monom.key % 100) / 10 == 0)

&& (abs(monom.data) == 1))

out << " z^" << monom.key % 10;

else

out << " \* z^" << monom.key % 10;

}

return out;

}

## Приложение 3. Класс TList

#pragma once

#include "Exception.h"

#include "Monom.h"

using namespace std;

template<typename TKey, typename TData>

class TList

{

private:

TNode<TKey, TData>\* pFirst;

TNode<TKey, TData>\* pPrevious;

TNode<TKey, TData>\* pCurrent;

TNode<TKey, TData>\* pNext;

public:

TList();

TList(const TList<TKey, TData>& list);

TList(TNode<TKey, TData>\* node);

~TList();

TNode<TKey, TData>\* Search(TKey key);

TNode<TKey, TData>\* Search(TNode<TKey, TData>\* node);

void InsertBegin(TKey new\_key, TData data);

void InsertEnd(TKey new\_key, TData data);

void InsertAfter(TKey key, TKey new\_key, TData data);

void InsertBefore(TKey key, TKey new\_key, TData data);

void InsertBegin(TNode<TKey, TData>\* node);

void InsertEnd(TNode<TKey, TData>\* node);

void InsertAfter(TKey key, TNode<TKey, TData>\* node);

void InsertBefore(TKey key, TNode<TKey, TData>\* node);

void Remove(TKey key);

void Remove(TNode<TKey, TData>\* node);

// Служебные операции

bool IsEmpty() const;

bool IsEnded() const;

// Навигация

void Reset();

void Next();

TNode<TKey, TData>\* GetFirst() const;

TNode<TKey, TData>\* GetCurrent() const;

friend ostream& operator<<(ostream& out, const TList<TKey, TData>& list)

{

TNode<TKey, TData>\* tmp = list.GetFirst();

int cnt = 1;

while (tmp != nullptr)

{

out << \*(tmp);

if (tmp->pNext != nullptr)

{

out << "-->";

if (cnt != 5)

cnt++;

else

{

cnt = 0;

out << endl;

}

}

tmp = tmp->pNext;

}

return out;

}

};

template<typename TKey, typename TData>

TList<TKey, TData>::TList()

{

pFirst = nullptr;

pPrevious = nullptr;

pCurrent = nullptr;

pNext = nullptr;

}

template<typename TKey, typename TData>

TList<TKey, TData>::TList(const TList<TKey, TData>& list)

{

if (list.IsEmpty())

{

pPrevious = nullptr;

pCurrent = nullptr;

pNext = nullptr;

pFirst = nullptr;

return;

}

pFirst = new TNode<TKey, TData>(\*(list.pFirst));

TNode<TKey, TData>\* node = pFirst;

TNode<TKey, TData>\* tmp = list.pFirst;

while (tmp->pNext != nullptr)

{

node->pNext = new TNode<TKey, TData>(\*(tmp->pNext));

node = node->pNext;

tmp = tmp->pNext;

}

pPrevious = nullptr;

pCurrent = pFirst;

if (pFirst->pNext != nullptr)

pNext = pFirst->pNext;

else

pNext = nullptr;

}

template<typename TKey, typename TData>

TList<TKey, TData>::TList(TNode<TKey, TData>\* node)

{

pFirst = new TNode<TKey, TData>(\*node);

pCurrent = pFirst;

pPrevious = nullptr;

pNext = nullptr;

}

template<typename TKey, typename TData>

TList<TKey, TData>::~TList()

{

TNode<TKey, TData>\* del\_node = pFirst;

TNode<TKey, TData>\* next;

while (del\_node != nullptr) // пока не дошли до конца,

{ // удаляем последовательно узлы

next = del\_node->pNext;

delete del\_node;

del\_node = next;

}

}

template<typename TKey, typename TData>

bool TList<TKey, TData>::IsEmpty() const

{

return (pFirst == nullptr);

}

template<typename TKey, typename TData>

bool TList<TKey, TData>::IsEnded() const

{

return (pCurrent == nullptr);

}

template<typename TKey, typename TData>

void TList<TKey, TData>::Reset()

{

pCurrent = pFirst;

pPrevious = nullptr;

if (pFirst != nullptr)

pNext = pFirst->pNext;

}

template<typename TKey, typename TData>

void TList<TKey, TData>::Next()

{

if (IsEnded())

throw Exception

(" Error, Next() is impossible to perform, list ended\n");

pPrevious = pCurrent;

pCurrent = pNext;

if (pNext != nullptr)

pNext = pNext->pNext;

}

template<typename TKey, typename TData>

TNode<TKey, TData>\* TList<TKey, TData>::Search(TKey key)

{

if (pFirst == nullptr)

throw Exception(" Error, List is empty\n");

if (pFirst->key == key)

return pFirst;

TNode<TKey, TData>\* tmp = pFirst;

while ((tmp->pNext != nullptr) && (tmp->pNext->key != key))

tmp = tmp->pNext;

if (tmp->pNext == nullptr)

throw Exception(" Error, key isn't found\n");

return tmp->pNext;

}

template<typename TKey, typename TData>

TNode<TKey, TData>\* TList<TKey, TData>::Search(TNode<TKey, TData>\* node)

{

if (pFirst == nullptr)

throw Exception(" Error, List is empty\n");

if (pFirst == node)

return pFirst;

TNode<TKey, TData>\* tmp = pFirst;

while ((tmp->pNext != nullptr) && (tmp->pNext != node))

tmp = tmp->pNext;

if (tmp->pNext == nullptr)

throw Exception(" Error, node isn't found\n");

return tmp->pNext;

}

template<typename TKey, typename TData>

void TList<TKey, TData>::InsertBegin(TKey new\_key, TData data)

{

if (pFirst == nullptr)

{

pFirst = new TNode<TKey, TData>(new\_key, data);

pCurrent = pFirst;

return;

}

TNode<TKey, TData>\* new\_node = new TNode<TKey, TData>(new\_key, data);

new\_node->pNext = pFirst;

if (pCurrent == pFirst)

pPrevious = new\_node;

pFirst = new\_node;

}

template<typename TKey, typename TData>

void TList<TKey, TData>::InsertEnd(TKey new\_key, TData data)

{

if (pFirst == nullptr)

{

InsertBegin(new\_key, data);

return;

}

TNode<TKey, TData>\* new\_node = new TNode<TKey, TData>(new\_key, data);

TNode<TKey, TData>\* tmp = pFirst;

while (tmp->pNext != nullptr)

tmp = tmp->pNext;

tmp->pNext = new\_node;

if (pCurrent == tmp)

pNext = new\_node;

}

template<typename TKey, typename TData>

void TList<TKey, TData>::InsertAfter(TKey key, TKey new\_key, TData data)

{

if (pFirst == nullptr)

throw Exception(" Error, List is empty\n");

TNode<TKey, TData>\* tmp = pFirst;

while ((tmp != nullptr) && (tmp->key != key))

tmp = tmp->pNext;

if (tmp == nullptr)

throw Exception(" Error, key isn't found\n");

TNode<TKey, TData>\* new\_node = new TNode<TKey, TData>(new\_key, data);

new\_node->pNext = tmp->pNext;

tmp->pNext = new\_node;

if (pCurrent == tmp)

pNext = new\_node;

if (pCurrent == new\_node->pNext)

pPrevious = new\_node;

}

template<typename TKey, typename TData>

void TList<TKey, TData>::InsertBefore(TKey key, TKey new\_key, TData data)

{

if (pFirst == nullptr)

throw Exception(" Error, List is empty\n");

if (pFirst->key == key)

{

InsertBegin(new\_key, data);

return;

}

TNode<TKey, TData>\* tmp = pFirst;

while ((tmp->pNext != nullptr) && (tmp->pNext->key != key))

tmp = tmp->pNext;

if (tmp->pNext == nullptr)

throw Exception(" Error, key isn't found\n");

TNode<TKey, TData>\* new\_node = new TNode<TKey, TData>(new\_key, data);

new\_node->pNext = tmp->pNext;

tmp->pNext = new\_node;

if (pCurrent == tmp)

pNext = new\_node;

if (pCurrent == new\_node->pNext)

pPrevious = new\_node;

}

template<typename TKey, typename TData>

void TList<TKey, TData>::InsertBegin(TNode<TKey, TData>\* node)

{

InsertBegin(node->key, node->data);

}

template<typename TKey, typename TData>

void TList<TKey, TData>::InsertEnd(TNode<TKey, TData>\* node)

{

InsertEnd(node->key, node->data);

}

template<typename TKey, typename TData>

void TList<TKey, TData>::InsertAfter(TKey key, TNode<TKey, TData>\* node)

{

InsertAfter(key, node->key, node->data);

}

template<typename TKey, typename TData>

void TList<TKey, TData>::InsertBefore(TKey key, TNode<TKey, TData>\* node)

{

InsertBefore(key, node->key, node->data);

}

template<typename TKey, typename TData>

void TList<TKey, TData>::Remove(TKey key)

{

if (pFirst == nullptr)

throw Exception(" Error remove, List is empty\n");

if (pFirst->key == key)

{

bool CurrentWasFirst = (pCurrent == pFirst);

bool PreviousWasFirst = (pPrevious == pFirst);

TNode<TKey, TData>\* tmp = pFirst;

pFirst = pFirst->pNext;

delete tmp;

if (CurrentWasFirst)

{

pCurrent = pFirst;

if (pCurrent != nullptr)

pNext = pFirst->pNext;

}

if (PreviousWasFirst)

pPrevious = nullptr;

return;

}

TNode<TKey, TData>\* tmp = pFirst;

while ((tmp->pNext != nullptr) && (tmp->pNext->key != key))

tmp = tmp->pNext;

if (tmp->pNext == nullptr)

throw Exception(" Error, key isn't found\n");

TNode<TKey, TData>\* node = tmp->pNext;

bool PreviousWasNode = (pPrevious == node);

bool CurrentWasNode = (pCurrent == node);

bool NextWasNode = (pNext == node);

tmp->pNext = tmp->pNext->pNext;

delete node;

if (PreviousWasNode)

pPrevious = tmp;

if (NextWasNode)

pNext = tmp->pNext;

if (CurrentWasNode)

{

pCurrent = tmp->pNext;

if (pCurrent != nullptr)

pNext = tmp->pNext->pNext;

}

}

template<typename TKey, typename TData>

void TList<TKey, TData>::Remove(TNode<TKey, TData>\* node)

{

if (pFirst == nullptr)

throw Exception(" Error remove, List is empty\n");

if (pFirst == node)

{

bool CurrentWasFirst = (pCurrent == pFirst);

bool PreviousWasFirst = (pPrevious == pFirst);

TNode<TKey, TData>\* tmp = pFirst;

pFirst = pFirst->pNext;

delete tmp;

if (CurrentWasFirst)

{

pCurrent = pFirst;

if (pCurrent != nullptr)

pNext = pFirst->pNext;

}

if (PreviousWasFirst)

pPrevious = nullptr;

return;

}

TNode<TKey, TData>\* tmp = pFirst;

while ((tmp->pNext != nullptr) && (tmp->pNext != node))

tmp = tmp->pNext;

if (tmp->pNext == nullptr)

throw Exception(" Error, address of node isn't found\n");

bool PreviousWasNode = (pPrevious == node);

bool CurrentWasNode = (pCurrent == node);

bool NextWasNode = (pNext == node);

tmp->pNext = tmp->pNext->pNext;

delete node;

if (PreviousWasNode)

pPrevious = tmp;

if (NextWasNode)

pNext = tmp->pNext;

if (CurrentWasNode)

{

pCurrent = tmp->pNext;

if (pCurrent != nullptr)

pNext = tmp->pNext->pNext;

}

}

template<typename TKey, typename TData>

TNode<TKey, TData>\* TList<TKey, TData>::GetFirst() const

{

if (pFirst == nullptr)

throw Exception(" Error get, pFirst = nullptr\n");

return pFirst;

}

template<typename TKey, typename TData>

TNode<TKey, TData>\* TList<TKey, TData>::GetCurrent() const

{

if (pCurrent == nullptr)

throw Exception(" Error get, pCurrent = nullptr\n");

return pCurrent;

}

## Приложение 4. Класс Polynom

#pragma once

#include "TList.h"

#include <iostream>

#include <string>

#include <stack>

using namespace std;

class Polynom

{

private:

TList<unsigned int, double>\* monoms;

int Priority(string sign\_); // для конструктора полином от строки

bool SignComparison(string sign, const stack<string> &tmp);

public:

Polynom();

Polynom(const Monom& monom);

Polynom(const TList<unsigned int, double>& list);

Polynom(const Polynom& polynom);

~Polynom();

Polynom(const string& exspression);

Polynom& operator=(const Polynom& polynom);

bool operator==(const Polynom& polynom) const;

Polynom operator+(const Polynom& polynom) const;

Polynom operator+(const Monom& monom) const;

Polynom operator-() const;

Polynom operator-(const Polynom& polynom) const;

Polynom operator-(const Monom& monom) const;

Polynom operator\*(const Polynom& polynom) const;

Polynom operator\*(const Monom& monom) const;

friend ostream& operator<<(ostream& out, const Polynom& polynom);

friend istream& operator>>(istream& in, Polynom& polynom);

};

Polynom::Polynom()

{

monoms = new TList<unsigned int, double>();

}

Polynom::Polynom(const Monom& monom)

{

monoms = new TList<unsigned int, double>();

monoms->InsertBegin(monom.key, monom.data);

}

Polynom::Polynom(const TList<unsigned int, double>& list)

{

TList<unsigned int, double>\* tmp = new TList<unsigned int, double>(list);

tmp->Reset();

unsigned int min\_key, current\_key;

TNode<unsigned int, double>\* min\_node;

TNode<unsigned int, double>\* current\_node;

int cnt = 0;

//Сортировка

tmp->Reset();

while (!tmp->IsEnded())

{

min\_key = tmp->GetCurrent()->key;

current\_key = tmp->GetCurrent()->key;

min\_node = tmp->GetCurrent();

current\_node = tmp->GetCurrent();

tmp->Next();

if (tmp->IsEnded())

break;

while (!tmp->IsEnded())

{

if (min\_key > tmp->GetCurrent()->key)

{

min\_key = tmp->GetCurrent()->key;

min\_node = tmp->GetCurrent();

}

tmp->Next();

}

if (min\_key != current\_key)

{

Monom monom(\*(tmp->Search(min\_node)));

\*(tmp->Search(min\_node)) = \*(tmp->Search(current\_node));

\*(tmp->Search(current\_node)) = monom;

}

cnt++;

tmp->Reset();

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

tmp->Next();

}

//Приводятся подобные

tmp->Reset();

while (!tmp->IsEnded())

{

TNode<unsigned int, double>\* node = tmp->GetCurrent();

tmp->Next();

if (tmp->IsEnded())

break;

while ((!tmp->IsEnded()) && (node->key == tmp->GetCurrent()->key))

{

node->data = node->data + tmp->GetCurrent()->data;

tmp->Remove(tmp->GetCurrent());

}

}

monoms = new TList<unsigned int, double>(\*tmp);

delete tmp;

}

Polynom::Polynom(const Polynom& polynom)

{

monoms = new TList<unsigned int, double>(\*polynom.monoms);

}

Polynom::~Polynom()

{

delete monoms;

}

bool Polynom::operator==(const Polynom& polynom) const

{

Polynom polynom1(\*this);

Polynom polynom2(polynom);

polynom1.monoms->Reset();

polynom2.monoms->Reset();

while ((!polynom1.monoms->IsEnded() && !polynom2.monoms->IsEnded()) &&

((\*polynom1.monoms->GetCurrent()) == (\*polynom2.monoms->GetCurrent())))

{

polynom1.monoms->Next();

polynom2.monoms->Next();

}

if (polynom1.monoms->IsEnded() && polynom2.monoms->IsEnded())

return true;

return false;

}

Polynom& Polynom::operator=(const Polynom& polynom)

{

if (\*this == polynom)

{

return \*this;

}

delete monoms;

monoms = new TList<unsigned int, double>(\*polynom.monoms);

return \*this;

}

// Данный оператор позволяет создавать полиномы отсортированными

// по возрастанию, а также не имеющих подобных

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

{

if (this->monoms->IsEmpty())

{

TList<unsigned int, double> tmp;

tmp.InsertBegin(monom.key, monom.data);

return Polynom(tmp);

}

Polynom result(\*this);

result.monoms->Reset();

while ((!result.monoms->IsEnded()) &&

(result.monoms->GetCurrent()->key < monom.key))

result.monoms->Next();

if (result.monoms->IsEnded()) // дошли до конца, и все мономы меньше нового

result.monoms->InsertEnd(monom.key, monom.data);

else if (result.monoms->GetCurrent()->key == monom.key)

{ // прибавляем как подобный

if ((result.monoms->GetCurrent()->data + monom.data) != 0.0)

result.monoms->GetCurrent()->data =

result.monoms->GetCurrent()->data + monom.data;

else

result.monoms->Remove(result.monoms->GetCurrent()->key);

}

else if (result.monoms->GetCurrent()->key > monom.key)

result.monoms->InsertBefore(result.monoms->GetCurrent()->key,

monom.key, monom.data);

result.monoms->Reset();

return result;

}

Polynom Polynom::operator+(const Polynom& polynom) const

{

Polynom polynom1(\*this);

Polynom polynom2(polynom);

polynom1.monoms->Reset();

polynom2.monoms->Reset();

while (!polynom2.monoms->IsEnded())

{

polynom1 = polynom1 + (\*polynom2.monoms->GetCurrent());

polynom2.monoms->Next();

}

return polynom1;

}

Polynom Polynom::operator-() const

{

Polynom tmp(\*this);

tmp.monoms->Reset();

while (!tmp.monoms->IsEnded())

{

\*tmp.monoms->GetCurrent() = -(\*tmp.monoms->GetCurrent());

tmp.monoms->Next();

}

return tmp;

}

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

{

Monom tmp(monom); // создаём копию монома, так как менять

return (\*this + (-tmp)); // передаваемый не можем из-за const

}

Polynom Polynom::operator-(const Polynom& polynom) const

{

Polynom tmp(polynom);

return (\*this + (-tmp));

}

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

{

Polynom result;

if (monom.data == 0.0)

return result;

Polynom polynom(\*this);

polynom.monoms->Reset();

while (!polynom.monoms->IsEnded())

{

Monom tmp = Monom(\*polynom.monoms->GetCurrent());

tmp = tmp \* monom;

if (tmp.data != 0.0)

result = result + tmp;

polynom.monoms->Next();

}

return result;

}

Polynom Polynom::operator\*(const Polynom& polynom) const

{

Polynom result;

Polynom polynom1(\*this);

Polynom polynom2(polynom);

polynom2.monoms->Reset();

while (!polynom2.monoms->IsEnded())

{

Monom tmp = Monom(\*polynom2.monoms->GetCurrent());

result = result + polynom1 \* tmp;

polynom2.monoms->Next();

} // подобные при умножении (которые например пол-cя при сумм-нии

// рез-тов от умн-ий одного пол-ма на разные мономы) прив-ся в сумме пол-ов

return result;

}

ostream& operator<<(ostream& out, const Polynom& polynom)

{

if (polynom.monoms->IsEmpty())

{

out << " 0 (your polynom is empty)";

return out;

}

Polynom tmp(polynom);

tmp.monoms->Reset();

while (!tmp.monoms->IsEnded())

{

out << \*(tmp.monoms->GetCurrent());

tmp.monoms->Next();

}

return out;

}

istream& operator>>(istream& in, Polynom& polynom)

{

string str;

cout << " Enter expression with polynomials in string:\n ";

getline(cin, str);

polynom = Polynom(str);

return in;

}

int Polynom::Priority(string sign\_)

{

if (sign\_.empty())

throw Exception(" Error, string is empty\n");

char sign = sign\_[0]; // с таким присваиванием работает, но нужно

switch (sign) // знать наверняка, что по 0 индексу лежит операция,

{ // возможны ошибки

case '+':

return 1;

case '-':

return 1;

case '\*':

return 2;

default:

return 0;

}

}

bool Polynom::SignComparison(string sign, const stack<string> &tmp)

{

if (Priority(sign) <= Priority(tmp.top()))

return true;

return false;

}

Polynom::Polynom(const string& exspression\_)

{

stack<string> stack1;

stack<string> stack2;

if (exspression\_[0] == '\0')

throw Exception(" Error, string is empty\n");

string exspression;

for (int i = 0; i < exspression\_.length(); i++) // избавляемся от пробелов

{

if (exspression\_[i] != ' ')

exspression = exspression + exspression\_[i];

}

for (int i = 0; i < exspression.length(); i++)

{

if (!(('0' <= exspression[i] && exspression[i] <= '9') ||

exspression[i] == '(' || exspression[i] == ')' ||

exspression[i] == '+' || exspression[i] == '-' ||

exspression[i] == '\*' || exspression[i] == '^' ||

exspression[i] == 'x' || exspression[i] == 'y' ||

exspression[i] == 'z'))

throw Exception(" Error, incorrecting symbol or low register!\n");

}

int cnt1 = 0;

int cnt2 = 0;

for (int i = 0; i < exspression.length(); i++)

{

if (exspression[i] == '(')

cnt1++;

if (exspression[i] == ')')

cnt2++;

if (cnt2 > cnt1)

throw Exception(" Error, ) meets before (");

}

if (cnt1 != cnt2)

throw Exception(" Error, number ( and ) isn't equal");

if (exspression[0] == '\*' || exspression[0] == '^')

throw Exception(" Error, You began with \* or ^\n");

for (int i = 0; i < exspression.length() - 1; i++)

{

if ((exspression[i] == '+' || exspression[i] == '-' ||

exspression[i] == '\*') && (exspression[i + 1] == '+' ||

exspression[i + 1] == '-' || exspression[i + 1] == '\*'))

throw Exception(" Error, 2 or more operations go together\n");

}

int check = 1;

for (int i = 0; i < exspression.length(); i++)

{

int start = i;

if (('0' <= exspression[i] && exspression[i] <= '9') ||

exspression[i] == 'x' || exspression[i] == 'y' ||

exspression[i] == 'z')

{

if (check == 1)

check = 0;

int end;

string for\_monom;

i++;

while ((i < exspression.length()) && (exspression[i] != '+') &&

(exspression[i] != '-') && (exspression[i] != '\*') &&

(exspression[i] != ')'))

i++;

end = i - 1; // должны сохранить позицию посл-го эл-та на 1 меньше

for\_monom = exspression.substr(start, end - start + 1);

if (check == 2) // знак у первого монома '-'

{

for\_monom.insert(0, "-");

check = 0;

}

stack1.push(for\_monom);

i--;

continue;

}

if (exspression[i] == '(')

{

string str;

str = str + exspression[i];

stack2.push(str);

continue;

}

if (exspression[i] == ')')

{

int flag = 0;

while (!stack2.empty())

{

if (stack2.top() != "(")

{

stack1.push(stack2.top());

stack2.pop();

continue;

}

stack2.pop();

flag = 1;

break;

}

if ((stack2.empty()) && (flag != 1))

throw Exception(" Wasn't found '('\n");

continue;

}

if (exspression[i] == '+' || exspression[i] == '-' ||

exspression[i] == '\*')

{

if (check == 1) // если знак перед первым мономом

{

if (exspression[i] == '\*')

throw Exception(" Error, operation '\*' before 1 monom\n");

if (exspression[i] == '-')

check = 2;

continue;

}

if (stack2.empty())

{

string str;

str = str + exspression[i];

stack2.push(str);

}

else

{

string str;

str = str + exspression[i];

if (SignComparison(str, stack2)) // true if <=

{

while (!stack2.empty())

{

if (Priority(str) == Priority(stack2.top()))

{

stack1.push(stack2.top());

stack2.pop();

break;

}

if (stack2.top() == "(")

{

break;

}

stack1.push(stack2.top());

stack2.pop();

}

stack2.push(str);

}

else

stack2.push(str);

continue;

}

}

else

throw Exception(" Error, incorrect enter, try again\n");

}

while (!stack2.empty())

{

stack1.push(stack2.top());

stack2.pop();

}

stack<string> stack3;

while (!stack1.empty()) // делаю swap

{

stack3.push(stack1.top());

stack1.pop();

}

stack<Polynom> result;

while (!stack3.empty())

{

if (stack3.top().find\_first\_of("xyz^0123456789.") != stack3.top().npos)

{ // выполнится, если это моном

Monom mon = Monom(stack3.top());

Polynom pol(mon);

result.push(pol);

if (!stack3.empty())

stack3.pop();

else

throw Exception

(" Error, stack is empty, you cannot do pop()\n");

}

else if (stack3.top().find\_first\_of("+-\*") != stack3.top().npos)

{

Polynom p1 = result.top();

if (!result.empty())

result.pop();

else

throw Exception

(" Error, stack is empty, you cannot do pop() !\n");

Polynom p2 = result.top();

if (!result.empty())

result.pop();

else

throw Exception

(" Error, stack is empty, you cannot do pop() !!\n");

char sign = stack3.top()[0]; // с таким присваиванием работает,

stack3.pop(); // но нужно знать наверняка, что по 0 индексу

switch (sign) // лежит операция, возможны ошибки!

{

case '+':

{

result.push(p2 + p1);

break;

}

case '-':

{

result.push(p2 - p1);

break;

}

case '\*':

{

result.push(p2 \* p1);

break;

}

default:

throw Exception(" Error, it can't be\n");

}

}

else

throw Exception(" Error, there are incorrect symbols in string\n");

}

monoms = new TList<unsigned int, double>(\*result.top().monoms);

}

// в данном конструкторе работает только сложение и вычитание мономов в строке

//Polynom::Polynom(const string& exspression)

//{

// string line = exspression;

// Polynom result;

// while (line.length())

// {

// int start, end;

// string for\_monom;

// for (int i = 0; i < line.length(); i++)

// {

// if (line[i] == ' ') // пропускаем пробелы

// continue;

// if ((line[i] == '+') || (line[i] == '-') ||

// ('0' <= line[i] && line[i] <= '9') || (line[i] == 'x') ||

// (line[i] == 'y') || (line[i] == 'z'))

// {

// start = i;

// i++;

// while ((i < line.length()) && (line[i] != '+') &&

// (line[i] != '-'))

// i++;

// end = i - 1; // должны сохр-ть поз-ю посл-го эл-та на 1 меньше

// break;

// }

// else

// throw Exception(" Error, incorrect enter or symbol\n");

// }

// for\_monom = line.substr(start, end - start + 1);

// result = result + Monom(for\_monom);

// if ((end + 1) == line.length())

// {

// break;

// }

// line = line.substr(end + 1, line.length() - end - 1);

// }

// monoms = new TList<unsigned int, double>(\*result.monoms);

//}

## Приложение 5. Класс Exception

#pragma once

#include <exception>

#include <iostream>

#include <string>

using namespace std;

class Exception : public exception

{

private:

const string message;

public:

Exception(string ex) : message(ex) {};

char const\* what() const

{

return message.c\_str();

}

};