МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
**«Национальный исследовательский
Нижегородский государственный университет им. Н.И. Лобачевского»
(ННГУ)**

**Институт информационных технологий, математики и механики**

# ЛАБОРАТОРНАЯ РАБОТА

на тему:

## «Полиномы»

**Выполнил:** студент группы 3822Б1ФИ2
_____ /Рысев М. Д./
Подпись

**Проверил:** к.т.н, доцент каф. ВВиСП
_____ / Кустикова В.Д./
Подпись

Нижний Новгород
2024

# Содержание

# Введение

Лабораторная работа направлена на самостоятельную разработку полиномов на базе кольцевого линейного односвязного списка. Полиномы встречаются во многих областях математики, таких как линейная алгебра, математический анализ, дифференциальные уравения и т.д.

# 1 Постановка задачи

**Цель:**

Цель лабораторной работы – изучить способ представления мономах в виде линейного спска мономов.

**Задачи:**

1. Изучения и написание структуры данных линейный односвязный список и кольцевой односвязный список.
2. Выделение отдельной сущности - монома, для представления полинома в виде линейного списка мономов.
3. Тестирование программы.

# 2 Руководство пользователя

## 2.1 Приложение для демонстрации работы связного списка

1. Запустите приложение с названием TListSamplr.exe. В результате появится окно, в котором нужно будет ввести 5 элементов кольцевого списка. (рис. 1).

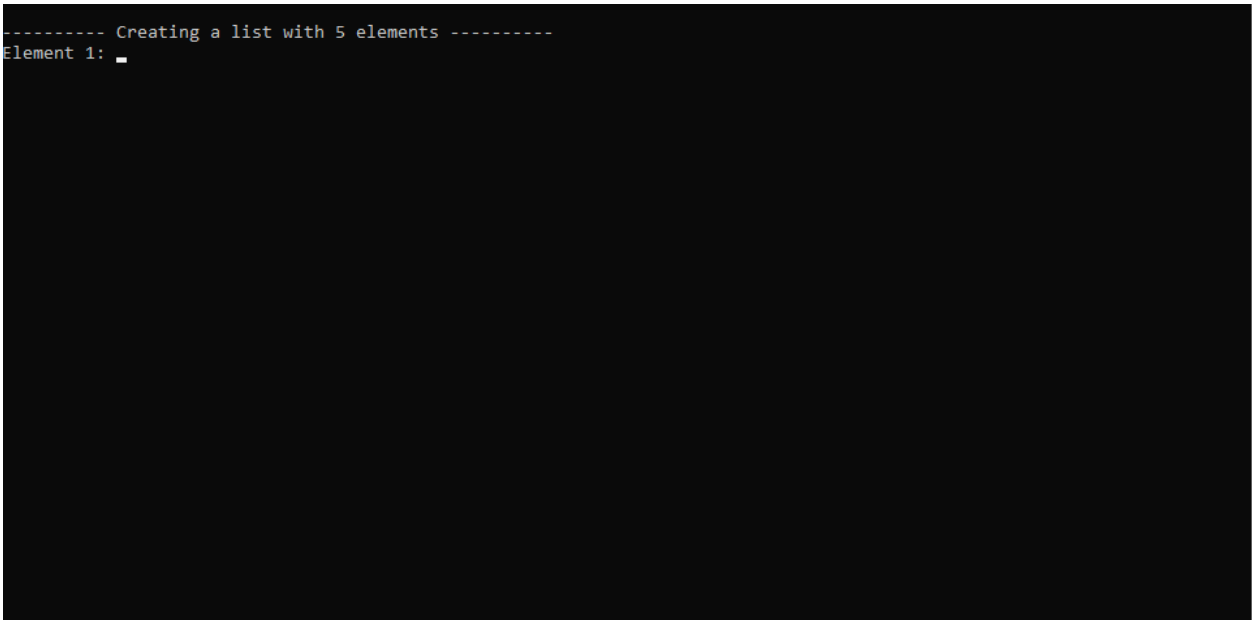



Рис. 1. Основное окно программы

2. Дальше предлагается ввести два числа, для проверки корректности работы операции поиска. Затем идёт демонстрация операций удаления и вставки. В них тоже предлагается ввести собственные данные. Затем следует демонстрация операций навигации по линейному списку. Завершается демонстрация, показом работы методом очистки списка. (рис. 2).

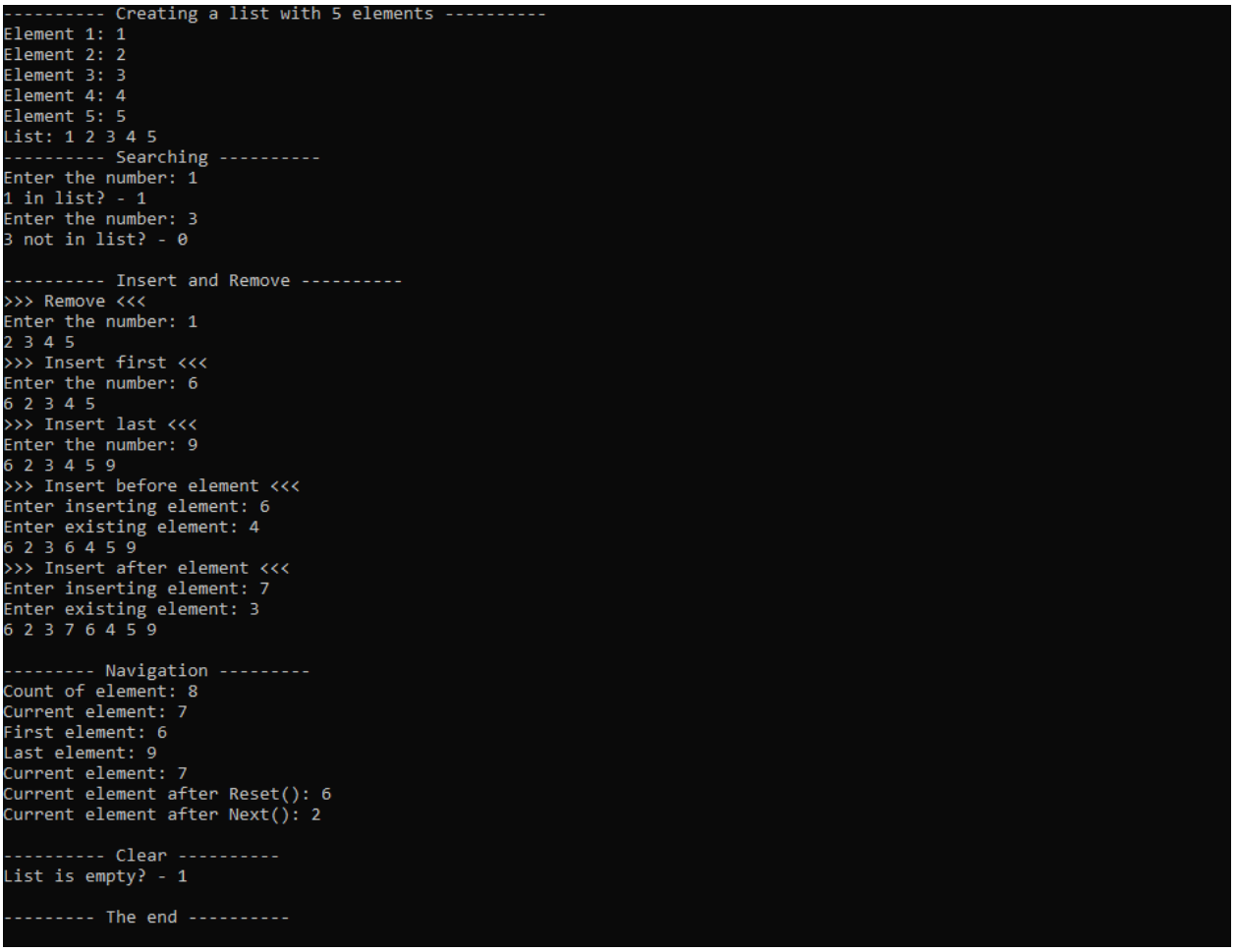



Рис. 2. Результат тестирования функций класса `TList`

## 2.2 Приложение для демонстрации работы полиномов.

1. Запустите приложение с названием TPolynom_sample.exe. В появившемся окне будет предложено ввести два полинома. (рис. 3).

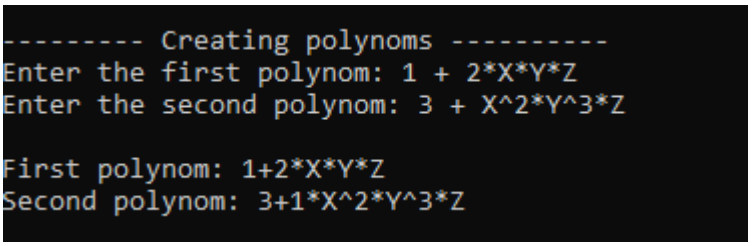



Рис. 3. Основное окно программы

2. После ввода полиномов будут выведены результаты математических операций над ними. При демонстрации функции вычисления значений полинома, будет предложено ввести значения переменных.(рис. 4).

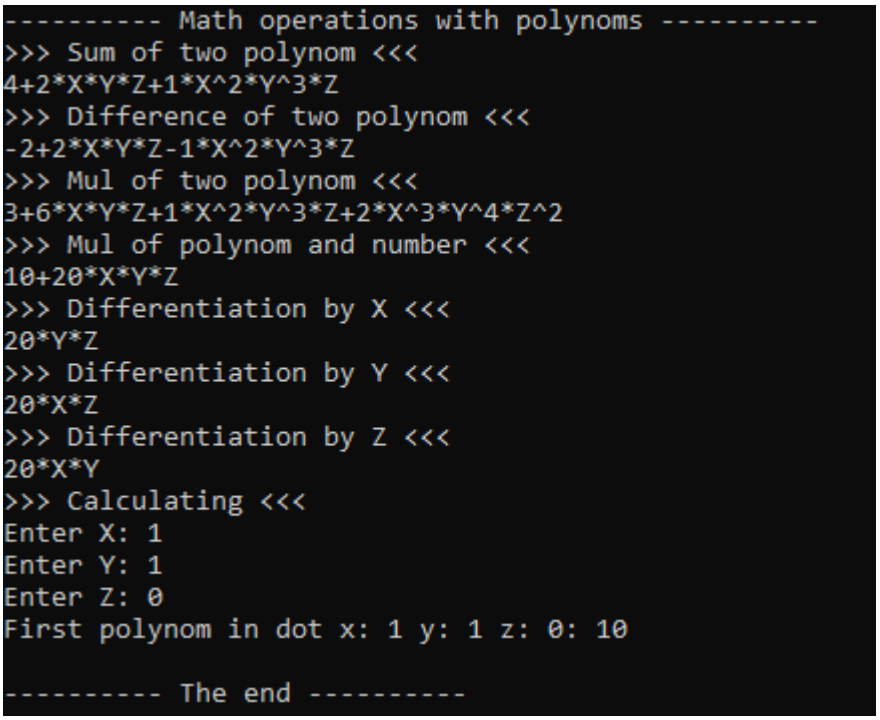



Рис. 4. Результат тестирования функций класса `TPolynom`

# 3 Руководство программиста

## 3.1 Описание алгоритмов

### 3.1.1 Линейный односвязный список

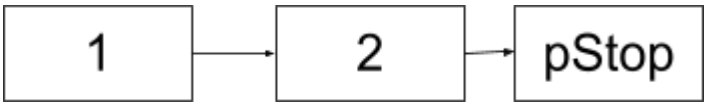
Линейный односвязный представлен указателем на первый узел, текущий узел, последний узел. Узел односвязного списка – это структура, которая хранит в себе сам элемент, указатель на следующий узел. Если узел является последним, то указатель на следующий элемент равен pStop, который равен nullptr.

Операции, доступные с данной структурой хранения, следующие: добавление элемента , удаление элемента, взять текущий элемент (первый элемент по-умолчанию), проверка на пустоту, сортировка, отчистка списка.

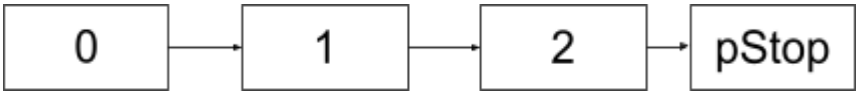
**Операция добавления в начало**

Добавление элемента в начало происходит следующим образом: создается новый узел, указателем на следующий элемент которого является первый первый элемент исходного списка.

Пример:



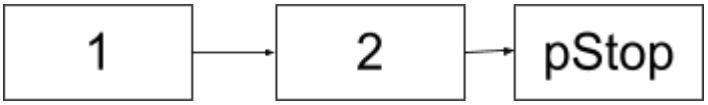

Операция добавления элемента (0) в начало:



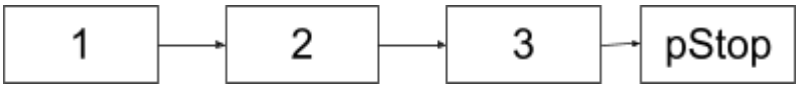

**Операция добавления в конец**

Добавление элемента в конец происходит следующим образом: создается новый узел, который кладется по указателю на следующий элемент у последнего элемента.

Пример:




Операция добавления элемента (3) в конец:



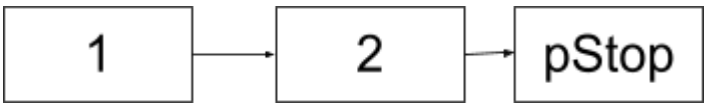

**Операция добавления после текущего**

Операция добавления элемента реализуется при помощи указателя на текущий элемент (по-умолчанию первый элемент, далее можно двигать). Создаём новый узел и

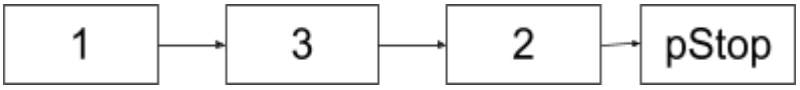
вставляем его после текущего, изменив указатель на следующий элемент у текущего и нового элементов.

Пример:

Пусть текущий элемент - 1.



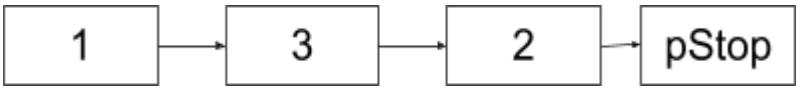

Операция добавления элемента (3) после текущего:



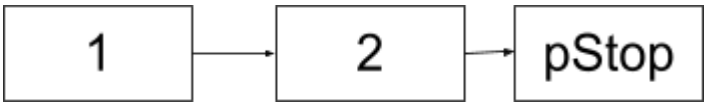

**Операция удаления элемента**

При удалении элемента ищется удаляемый элемент, указатель на следующий элемент предыдущего, относительно удаляемого элемента, перезаписывается на указатель на следующий элемент для удаляемого..

Пример:




Операция удаления элемента со значением (3):

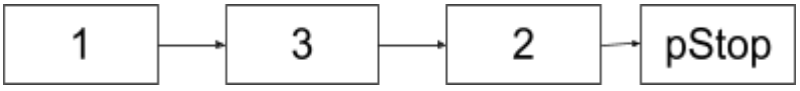



**Операция получения текущего элемента**

Реализуется путём взятия значения, лежащего по указателю на текущий элемент.

Пример:

Допустим, что текущим является второй по счёту элемент списка?

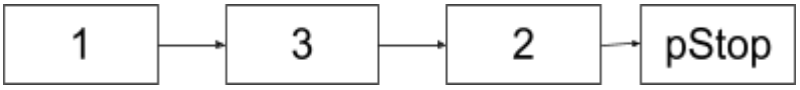



Операция взятия элемента текущего  вернёт значение 3.

**Операция поиска**

При операции поиска мы последовательно пробегаемся по списку до тех пор, пока не встретим искомый элемент Если элемент найден возвращается указатель на него, иначе возвращается nullptr.

Пример:

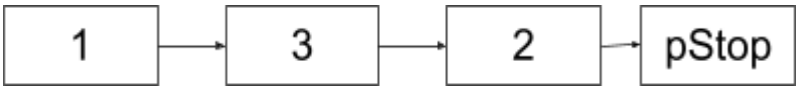



Операция поиска  элемента с ключём 3 вернёт указатель на второй по счёту элемент.

**Операция проверки на пустоту.**

Операция проверки на полноту вернёт 0 если в списке нет элементов, иначе вернёт 1.

Пример:




Результат: 0

### 3.1.2 Кольцевой список

Кольцевой список отличается от обычного тем, что в кольцевом списке указатель на следующий элемент для последнего указывает на фиктивный узел, который в свою очередь указывает на  узел начала списка.

В связи с этим для кольцевого списка сохраняются те же самые операции, что и для обычного списка, только последний элемент указывает на первый.

### 3.1.3 Полином

Полином реализуется на базе кольцевого списка. Каждый узел списка является мономом.

**Операция суммирования полиномов**

Суммирование происходит следующим образом: создаётся результирующий полином, в который записываются результаты сложения подобных мономов. При вставке в результирующий моном происходит поиск подобного слагаемого, и в случае успеха, коэффициенты найденного и вставляемого складываются, а в случае неуспеха, вставляемый моном ставиться после последнего меньшго для него.

Пример:

(1 + X) + (1 - X)

Результат:

2

**Операция вычитания полиномов**

При операции вычитания, вычитаемый моном умножается на -1, а затем происходит вызов операции суммирования.

Пример:

(1 + X) - (1 - X)

Результат:

2*X

**Операция произведения полиномов**

При умножении полиномов, все мономы первого полинома перемножаются со всеми мономаха второго полинома. При вставке нового элемента в результирующий список, происходит поиск подобного слагаемого с последующим сложением коэффициентов найденного и вставляемого. Если же подобного слагаемого не нашлось, то вставка происходит после последнего меньшего.

Пример:

(1 - X) * (1 + X)

Результат:

1 - X^2

**Операция дифференцирования полиномов**

При дифференцировании монома, каждый моном дифференцируется по заданной переменной в соответствии с математическим правилами. Если в текущем мономе нет нужной переменной или коэффициент равен 0, то вместо этого монома записывается пустой моном.

Возможно дифференцирование по переменным x, y или z.

Пример:

2*x^2*y^3*z^2

Результат дифференцирования:

По X: 4*x*y^3*z^2,

По Y: 6*x^2*y^2*z^2,

По Z: 4*x^2*y^3*z.

## 3.2 Описание программной реализации

### 3.2.1 Схема наследования классов

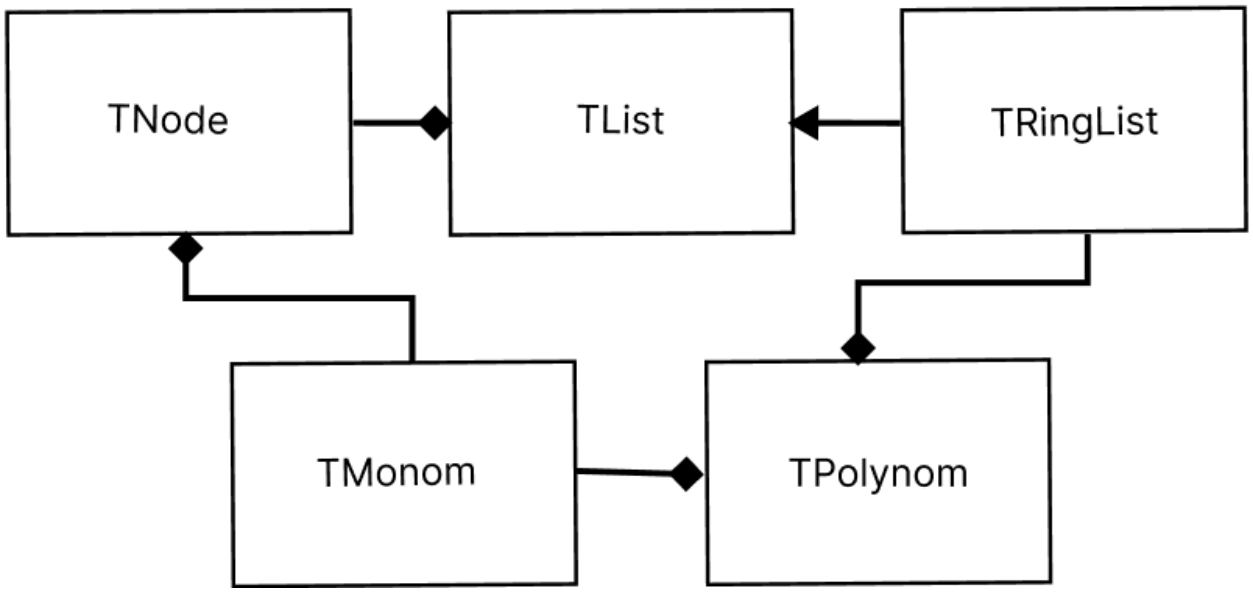



Рис. 5. Схема наследования классов

### 3.2.2 Описание структуры TNode

```
#pragma once

template <typename T>
struct TNode
{
	T key;
	TNode<T>* pNext;

	TNode() : key(), pNext(nullptr) {}
	TNode(const T& _key) : key(_key), pNext(nullptr) {}
	TNode(const T& _key, TNode<T>* _pNext) : key(_key), pNext(_pNext) {}
	TNode(TNode<T>* node) : key(node->key), pNext(node->pNext) {}
};
```

Назначение: представление узла списка

Поля:

key – данные, которые хранит узел

pNext – указатель на следующий узел

Методы:

**TNode();**

Назначение: конструктор по умолчанию.

Входные параметры отсутствуют:

Выходные параметры: отсутствуют.

```
TNode(const T& _key);
```

Назначение: конструктор конструктор с параметром.

Входные параметры отсутствуют:

Выходные параметры: отсутствуют.

```
TNode(const T& _key, TNode<T>* _pNext);
```

Назначение: конструктор конструктор с параметром.

Входные параметры отсутствуют:

Выходные параметры: отсутствуют.

```
TNode(TNode<T>* node);
```

Назначение: конструктор конструктор с параметром.

Входные параметры отсутствуют:

Выходные параметры: отсутствуют.

### 3.2.3 Описание класса TList

```
#pragma once
#include "TNode.h"
#include <iostream>

template <typename T>
class TList {
protected:
	TNode<T>* pFirst;
	TNode<T>* pCurrent;
	TNode<T>* pLast;
	TNode<T>* pStop;
public:
	TList();
	TList(TNode<T>* _pFirst);
	TList(const TList<T>& lst);
	virtual ~TList();

	TNode<T>* Search(const T& key) const;
	virtual void insertFirst(const T& key);
	void insertLast(const T& key);
	void insertBefore(const T& key);
	void insertBefore(const T& old_key, const T& new_key);
	void insertAfter(const T& key);
	void insertAfter(const T& old_key, const T& new_key);
	virtual void Remove(const T& key);
	virtual void Clear();

	size_t getSize() const;
	TNode<T>* getCurrent() const;
	TNode<T>* getFirst() const;
	TNode<T>* getLast() const;
	TNode<T>* getStop() const;
	bool isFull() const;
	bool isEmpty() const;

	virtual void Next();
	void Reset();

	friend std::ostream& operator << (std::ostream& out, TList<T>& lst) {
		lst.Reset();
```

```
            while (lst.getCurrent() != lst.getStop()) {
                  out << lst.getCurrent()->key << " ";
                  lst.Next();
            }
            return out;
      }
};
```

Назначение: представление списка.

Поля:

**pFirst** – указатель на первый элемент списка.

**pCurrent**– указатель на текущий элемент списка

**pLast** – указатель на последний элемент списка.

Методы:

**TList();**

Назначение: конструктор по умолчанию.

Входные параметры отсутствуют:

Выходные параметры: отсутствуют.

**TList(TNode<T>* _pFirst);**

Назначение: конструктор с параметром.

Входные параметры:

**_pFirst –** указатель на первый элемент списка**.**

Выходные параметры: отсутствуют.

**TList(const TList<T>& lst);**

Назначение: конструктор копирования.

Входные параметры:

**lst –** копируемый список**.**

Выходные параметры: отсутствуют.

**virtual ~TList();**

Назначение: деструктор.

Входные параметры отсутствуют:

Выходные параметры: отсутствуют.

**virtual void Remove(const T& key);**

Назначение: удаление элемента.

Входные параметры:

**key –** удаляемый элемент**.**

Выходные параметры отсутствуют.

```
virtual void insertFirst(const T& key);
```

Назначение: добавление элемента в начало.

Входные параметры:

**key –** добавляемый элемент.

Выходные параметры отсутствуют.

```
void insertLast(const T& key);
```

Назначение: добавление элемента в конец.

Входные параметры:

**key –** добавляемый элемент.

Выходные параметры отсутствуют.

```
void insertBefore(const T& key);
```

Назначение: добавление элемента перед текущим элементам.

Входные параметры:

**key –** добавляемый элемент.

Выходные параметры отсутствуют.

```
void insertBefore(const T& old_key, const T& new_key);
```

Назначение: добавление элемента перед текущим элементам.

Входные параметры:

**old_key –** старый элемент.

**new_key –** новый элемент.

Выходные параметры отсутствуют.

```
void insertAfter(const T& key);
```

Назначение: добавление элемента после текущего.

Входные параметры:

**key –** добавляемый элемент.

Выходные параметры отсутствуют.

```
void insertAfter(const T& old_key, const T& new_key);
```
Назначение: добавление элемента после имеющегося.

Входные параметры:

**old_key –** старый элемент.

**new_key –** новый элемент.

Выходные параметры отсутствуют.

```
TNode<T>* Search(const T& key) const;
```
Назначение: поиск элемента.

Входные параметры:

**key –** искомый элемент.

Выходные параметры:

Указатель на элемент

```
bool isEmpty();
```
Назначение: проверка на пустоту.

Входные параметры отсутствуют.

Выходные параметры отсутствуют.

```
bool isFull();
```
Назначение: проверка на полнту.

Входные параметры отсутствуют.

Выходные параметры отсутствуют.

```
virtual void Clear();
```
Назначение: отчистка списка.

Входные параметры отсутствуют.

Выходные параметры отсутствуют.

```
virtual void Next();
```
Назначение: отчистка списка.

Входные параметры отсутствуют.

Выходные параметры отсутствуют.

### 3.2.4 Описание класса TRingList

```
#pragma once
#include "TList.h"

template <typename T>
class TRingList : public TList<T> {
private:
	TNode<T>* pHead;
public:
	TRingList();
	TRingList(const TList<T>& lst);
	TRingList(const TRingList<T>& lst);
```

```
        TRingList(TNode<T>* _pFirst);
        virtual ~TRingList();

        void insertFirst(const T& key);
        void Remove(const T& key);
        void Clear();
};
```

Назначение: представление кольцевого списка.

Поля:

**pHead** – указатель на фиктивную голову (не является элементом списка, предназначен для итерации).

Методы:

**TRingList();**

Назначение: конструктор по умолчанию.

Входные параметры отсутствуют:

Выходные параметры: отсутствуют.

**TRingList(const TList<T>& lst);**

Назначение: конструктор копирования.

Входные параметры:

**lst –** список, на основе которого создаем новый список**.**

Выходные параметры: отсутствуют.

**TRingList(const TRingList<T>& lst);**

Назначение: конструктор с параметрами.

Входные параметры:

**node –** узел, на основе которого создаем новый список**.**

Выходные параметры: отсутствуют.

**virtual ~TRingList();**

Назначение: деструктор.

Входные параметры отсутствуют:

Выходные параметры: отсутствуют.

**void insertFirst(const T& key);**

Назначение: вставка в начало.

Входные параметры:

**key –** новый элемент.

Выходные параметры отсутствуют.

```
void Remove(const T& key);
```

Назначение: удаление элемента.

Входные параметры:

**key –** удаляемый элемент.

Выходные параметры отсутствуют.

```
void Clear();
```

Назначение: отчистка списка.

Входные параметры отсутствуют.

Выходные параметры отсутствуют.

### 3.2.5 Описание класса TMonom

```
#pragma once
#include <iostream>
#include <string>
#include <math.h>
using namespace std;

class Monom
{
public:

	double coef;
	int degX;
	int degY;
	int degZ;

	Monom(const Monom& monom);
	Monom(const double coef = 1.0, const int _degX = 0, const int _degY =
0, const int _degZ = 0);

	const Monom& operator=(const Monom& monom);
	bool operator==(const Monom& monom) const;
	bool operator!=(const Monom& monom) const;
	bool operator>(const Monom& monom) const;
	bool operator<(const Monom& monom) const;
	Monom operator*(const Monom& monom) const;
	friend std::ostream& operator << (std::ostream& out, Monom& monom) {
		out << monom.coef;
		if (monom.degX != 0 && monom.degX == 1) out << "*X";
		else if (monom.degX > 1) out << "*X^" << monom.degX;
		if (monom.degY != 0 && monom.degY == 1) out << "*Y";
		else if (monom.degY > 1) out << "*Y^" << monom.degY;
		if (monom.degZ != 0 && monom.degZ == 1) out << "*Z";
		else if (monom.degZ > 1) out << "*Z^" << monom.degZ;
		return out;
	}

	double getCoef() const;
	int getDegX() const;
	int getDegY() const;
	int getDegZ() const;
```

```
        c
        Monom difX() const;
        Monom difY() const;
        Monom difZ() const;

        string toString() const;
};
```

Назначение: представление монома

Поля:

coef – коэффициент монома

degX – степень при независимой переменной x

degY – степень при независимой переменной y

degZ – степень при независимой переменной z

Методы:

```
Monom(const Monom& monom);
```

Назначение: конструктор по копирования.

Входные параметры:

**monom** – копируемый моном.

Выходные параметры: отсутствуют.

```
Monom(const double coef = 1.0, const int _degX = 0, const int _degY = 0,
const int _degZ = 0);
```

Назначение: конструктор с параметрами, конструктор по-умолчанию.

Входные параметры:

**coef – коэффициент монома,**

**_degX – степень при x,**

**_degY – степень при y,**

**_degZ – степень при z.**

Выходные параметры: отсутствуют.

```
bool operator<(const Monom& monom) const;
```

Назначение: перегрузка оператора меньше.

Входные параметры:

**monom** – моном, который сравниваем.

Выходные параметры:

true или false.

```
bool operator>(const Monom& monom) const;
```

Назначение: перегрузка оператора меньше.

Входные параметры:

**monom** – моном, который сравниваем.

Выходные параметры:

true или false.

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

Назначение: перегрузка оператора меньше.

Входные параметры:

**monom** – моном, который сравниваем.

Выходные параметры:

true или false.

```
Monom operator*(const Monom& monom) const;
```

Назначение: перегрузка оператора умножения мономов.

Входные параметры:

**monom** – умножаемый моном.

Выходные параметры:

Произведение мономов.

```
double getCoef() const;
```

Назначение: получение коэффициента монома.

Входные параметры отсутствуют.

Выходные параметры:

Коэффициент.

```
int getDegX() const;
```

Назначение: получение степени при X..

Входные параметры отсутствуют:

Выходные параметры:

Степень при X.

```
int getDegY() const;
```

Назначение: получение степени при Y.

Входные параметры отсутствуют:

Выходные параметры:

Степень при Y.

```
int getDegZ() const;
```

Назначение: получение степени при Z.

Входные параметры отсутствуют:

Выходные параметры:

Степень при Z.

```
Monom difX() const;
```
Назначение: производная по x.

Входные параметры отсутствуют:

Выходные параметры:

Моном, являющийся производной по x исходного монома.

```
Monom difY() const;
```
Назначение: производная по y.

Входные параметры отсутствуют:

Выходные параметры:

Моном, являющийся производной по x исходного монома.

```
Monom difZ() const;
```
Назначение: производная по z.

Входные параметры отсутствуют:

Выходные параметры:

Моном, являющийся производной по x исходного монома.

```
double Calculate(double x, double y, double z);
```
Назначение: вычисление значения монома.

Входные параметры:

**x** – значение переменной X.

**y** – значение переменной Y.

**z** – значение переменной Z.

Выходные параметры:

Результат вычисления.

```
string toString() const;
```
Назначение: перевод монома в строку.

Входные параметры:

Выходные параметры:

Строка.

### 3.2.6 Описание класса TPolynom

```
#pragma once
#include <iostream>
#include "TRingList.h"
#include "TMonom.h"
#include "arexp.h"
#include <string>
```

```
using namespace std;

class TPolynom : public ArithmeticExpression {
protected:
	TRingList<Monom>* monoms;
	void toPolynome();
	bool findLess(Monom& m, TNode<Monom>*& g);
	void setDegree(Monom&, string& param, int deg);
	void giveSimTer();
	bool isSimilar(Monom& a, Monom& b) const;
	void SortInsert(Monom& m);
public:
	TPolynom();
	TPolynom(const string& _polynome);
	TPolynom(const TRingList<Monom>& ringlist);
	TPolynom(const TPolynom& polynom);
	~TPolynom();

	const TPolynom& operator=(const TPolynom& polynom);

	TPolynom operator+(TPolynom& polynom);
	TPolynom operator-(TPolynom& polynom);
	TPolynom operator*(TPolynom& polynom);
	TPolynom operator*=(int num);
	TPolynom operator*=(float num);
	double operator()(double x, double y, double z);
	friend std::ostream& operator << (std::ostream& out, TPolynom& polynom)
{
		polynom.monoms->Reset();
		while (polynom.monoms->getCurrent() != polynom.monoms->getStop())
{
			Monom monom = polynom.monoms->getCurrent()->key;
			if (monom.coef > 0 && polynom.monoms->getCurrent() !=
polynom.monoms->getFirst()) out << "+";
			if (monom.coef != 0) out << monom;
			polynom.monoms->Next();
		}
		return out;
	}
	friend std::istream& operator >> (std::istream& in, TPolynom& polynom)
{
		string pol;
		getline(in, pol);
		polynom = TPolynom(pol);
		return in;
	}

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

	TPolynom dif_x() const;
	TPolynom dif_y() const;
	TPolynom dif_z() const;

	void ToString();
};
```

Назначение: работа с полиномами

Поля:

**monoms** – список мономов.

Методы:

**`TPolynom();`**

Назначение: конструктор по умолчанию.

Входные параметры отсутствуют:

Выходные параметры: отсутствуют.

**`TPolynom(const string& _polynome);`**

Назначение: конструктор с параметрами.

Входные параметры:

**`_polynome`** – копируемый полином.

Выходные параметры: отсутствуют.

**`TPolynom(const TRingList<Monom>& ringlist);`**

Назначение: конструктор с параметрами.

Входные параметры:

**`ringlist`** – копируемый список.

Выходные параметры: отсутствуют.

**`TPolynom(const TPolynom& polynom);`**

Назначение: конструктор копирования.

Входные параметры:

**`polynom`** – строка, на основе которого создаем новый полином.

Выходные параметры: отсутствуют.

**`const TPolynom& operator=(const TPolynom& polynom);`**

Назначение: операция присваивания.

Входные параметры:

**`polynom`** – полином, на основе которого создаем новый полином.

Выходные параметры: ссылка на присвоенный полином.

**`bool operator == (const TPolynom& polynom) const;`**

Назначение: операция равенства.

Входные параметры:

**`polynom`** – полином, с которым сравниваем.

Выходные параметры: true или false – равны полиномы или нет.

**`TPolynom operator+(TPolynom& polynom);`**

Назначение: сложение полиномов.

Входные параметры:

**`polynom`** – второе слагаемое.

Выходные параметры:

сумма полиномов.

**TPolynom operator-(TPolynom& polynom);**

Назначение: разность полиномов.

Входные параметры:

**polynom –** вычитаемое**.**

Выходные параметры:

разность полиномов.

**TPolynom operator*(TPolynom& polynom);**

Назначение: произведение полиномов.

Входные параметры:

**polynom –** второй множитель**.**

Выходные параметры:

произведение полиномов.

**TPolynom operator*=(int num);**

Назначение: произведение полинома на число.

Входные параметры:

**num –** число**.**

Выходные параметры:

произведение полинома и числа.

**TPolynom operator*=(float num);**

Назначение: произведение полинома на число.

Входные параметры:

**num –** число**.**

Выходные параметры:

произведение полинома и числа.

**double operator()(double x, double y, double z);**

Назначение: вычисление значения.

Входные параметры:

**x** – значение переменной X.

**y** – значение переменной Y.

**z** – значение переменной Z.

Выходные параметры:

результат вычисления.

**TPolynom dif_x() const;**

Назначение: производная по переменной x.

Входные параметры:

Выходные параметры:

результат дифференцирования.

**`TPolynom dif_y() const;`**

Назначение: производная по переменной y.

Входные параметры:

Выходные параметры:

результат дифференцирования.

**`TPolynom dif_z() const;`**

Назначение: производная по переменной z.

Входные параметры:

Выходные параметры:

результат дифференцирования.

**`TPolynom dif_z() const;`**

Назначение: производная по переменной z.

Входные параметры:

Выходные параметры:

результат дифференцирования.

**`void ToString();`**

Назначение: перевод полинома в строку.

Входные параметры:

Выходные параметры:

строка.

**`void toPolynome();`**

Назначение: конвертирование массива токенов в полином.

Входные параметры:

Выходные параметры:

**`bool findLess(Monom& m, TNode<Monom>*& g);`**

Назначение: поиск последнего меньшего или подобного слагаемого.

Входные параметры:

**`m`** – вставляемый моном.

**`g`** – указатель на найденный моном.

Выходные параметры:

true - если моном найден, false - иначе

**`void setDegree(Monom& m, string& param, int deg);`**

Назначение: установка значения степени.

Входные параметры:

**`m`** – моном.

**`param`** – параметр, у которого устанавливается степень.

**`deg`** – указатель на найденный моном.

Выходные параметры:

**`void giveSimTer();`**

Назначение: приведение подобных слагаемых.

Входные параметры:

Выходные параметры:

**`bool isSimilar(Monom& a, Monom& b) const;`**

Назначение: проверка мономов на подобие.

Входные параметры:

**`a`** – моном.

**`b`** – моном.

Выходные параметры:

true - если мономы подобные, false - иначе.

**`void SortInsert(Monom& m);`**

Назначение: вставка монома полином.

Входные параметры:

**`m`** – моном.

Выходные параметры:

# Заключение

В ходе выполнения работы были написаны классы списка, кольцевого списка и полиномов. Освоены первичные навыки работы с линейными односвязными списками..

# Литература

1. Связный список [https://ru.wikipedia.org/wiki/Связный_список].
2. Полином [https://ru.wikipedia.org/wiki/Многочлен].

# Приложения

## Приложение А. Реализация класса TNode

```
#pragma once

template <typename T>
struct TNode
{
	T key;
	TNode<T>* pNext;

	TNode() : key(), pNext(nullptr) {}
	TNode(const T& _key) : key(_key), pNext(nullptr) {}
	TNode(const T& _key, TNode<T>* _pNext) : key(_key), pNext(_pNext) {}
	TNode(TNode<T>* node) : key(node->key), pNext(node->pNext) {}
};
```

## Приложение Б. Реализация класса TList

```
template <typename T>
TList<T>::TList() {
	pFirst = nullptr;
	pCurrent = nullptr;
	pLast = nullptr;
	pStop = nullptr;
}

template <typename T>
TList<T>::TList(TNode<T>* _pFirst) {
	if (_pFirst == nullptr) return;
	pFirst = new TNode<T>(_pFirst);
	TNode<T>* tmp1 = _pFirst;
	TNode<T>* tmp2 = pFirst;
	while (tmp1->pNext != nullptr) {
		tmp2->pNext = new TNode<T>(tmp1->pNext->key);
		tmp1 = tmp1->pNext;
		tmp2 = tmp2->pNext;
	}
	pLast = tmp2;
	pCurrent = pFirst;
	pStop = pLast->pNext;
}

template <typename T>
TList<T>::TList(const TList<T>& lst) {
	if (lst.pFirst == nullptr) return;
	pFirst = new TNode<T>(lst.getFirst());
	TNode<T>* tmp1 = lst.getFirst();
	TNode<T>* tmp2 = pFirst;
	while (tmp1->pNext != lst.getStop()) {
		tmp2->pNext = new TNode<T>(tmp1->pNext->key);
		tmp1 = tmp1->pNext;
		tmp2 = tmp2->pNext;
	}
	pLast = tmp2;
	pCurrent = pFirst;
	pStop = pLast->pNext;
```

```
}

template <typename T>
TList<T>::~TList() {
	if (pStop != pFirst) Clear();
}

template <typename T>
TNode<T>* TList<T>::Search(const T& key) const{
	TNode<T>* tmp = pFirst;
	while (tmp != pStop && tmp->key != key) tmp = tmp->pNext;
	if (tmp == pStop) tmp = nullptr;
	return tmp;
}

template <typename T>
void TList<T>::insertFirst(const T& key) {
	TNode<T>* tmp = new TNode<T>(key, pFirst);
	if (pFirst == pStop) pLast = tmp;
	pFirst = tmp;
	pCurrent = pFirst;
}

template <typename T>
void TList<T>::insertLast(const T& key) {
	if (pFirst == pStop) {
		insertFirst(key);
		return;
	}
	TNode<T>* tmp = new TNode<T>(key, pStop);
	pLast->pNext = tmp;
	pLast = pLast->pNext;
	pCurrent = pLast;
}

template <typename T>
void TList<T>::insertBefore(const T& key) {
	if (pCurrent == pFirst) {
		insertFirst(key);
		return;
	}
	TNode<T>* tmp = pFirst;
	while (tmp->pNext != pCurrent) tmp = tmp->pNext;
	tmp->pNext = new TNode<T>(key, pCurrent);
	pCurrent = tmp->pNext;
}

template <typename T>
void TList<T>::insertBefore(const T& old_key, const T& new_key) {
	TNode<T>* tmp = pCurrent;
	pCurrent = Search(old_key);
	if (pCurrent == nullptr) throw "ERROR: element not exist";
	insertBefore(new_key);
}

template <typename T>
void TList<T>::insertAfter(const T& old_key, const T& new_key) {
	TNode<T>* tmp = pCurrent;
	pCurrent = Search(old_key);
	if (pCurrent == nullptr) throw "ERROR: element not exist";
	insertAfter(new_key);
}
```

```
template <typename T>
void TList<T>::insertAfter(const T& key) {
	if (pCurrent->pNext == pStop) {
		insertLast(key);
		return;
	}
	pCurrent->pNext = new TNode<T>(key, pCurrent->pNext);
	pCurrent = pCurrent->pNext;
}

template <typename T>
void TList<T>::Remove(const T& key) {
	if (pFirst == pStop) throw "ERROR: list is empty";
	TNode<T>* tmp = pFirst, *prev = nullptr;
	while (tmp->pNext != pStop && tmp->key != key) {
		prev = tmp;
		tmp = tmp->pNext;
	}
	if (tmp->pNext == pStop && tmp->key != key) throw "ERROR: element not
exist";
	if (prev == nullptr) {
		pFirst = tmp->pNext;
		pCurrent = pFirst;
		delete tmp;
		return;
	}
	if (tmp->pNext == pStop && tmp->key == key) {
		pLast = prev;
	}
	prev->pNext = tmp->pNext;
	pCurrent = prev;
	delete tmp;

}

template <typename T>
void TList<T>::Clear() {
	if (pFirst == pStop) throw "ERROR: list already empty";
	TNode<T>* tmp = pFirst;
	while (pFirst != pStop) {
		pFirst = pFirst->pNext;
		delete tmp;
		tmp = pFirst;
	}
	pFirst = nullptr;
	pCurrent = nullptr;
	pStop = nullptr;
}

template <typename T>
size_t TList<T>::getSize() const{
	size_t count = 0;
	TNode<T>* tmp = pFirst;
	while (tmp != pStop) {
		tmp = tmp->pNext;
		count += 1;
	}
	return count;
}

template <typename T>
TNode<T>* TList<T>::getCurrent() const {
	return pCurrent;
```

```
}

template <typename T>
TNode<T>* TList<T>::getFirst() const {
	return pFirst;
}

template <typename T>
TNode<T>* TList<T>::getLast() const {
	return pLast;
}

template <typename T>
TNode<T>* TList<T>::getStop() const {
	return pStop;
}

template <typename T>
bool TList<T>::isFull() const {
	TNode<T>* tmp = new TNode<T>();
	return tmp == nullptr;
}

template <typename T>
bool TList<T>::isEmpty() const {
	return pFirst == pStop;
}

template <typename T>
void TList<T>::Next() {
	//if (pCurrent == pLast) return;
	pCurrent = pCurrent->pNext;
}

template <typename T>
void TList<T>::Reset() {
	pCurrent = pFirst;
}
```

# Приложение B. Реализация класса TRingList

```
template <typename T>
TRingList<T>::TRingList() : TList<T>() {
	pHead = nullptr;
	pStop = pHead;
}

template <typename T>
TRingList<T>::TRingList(const TList<T>& lst) : TList<T>(lst)
{
	pHead = new TNode<T>(0, lst.getFirst());
	if (pLast != nullptr) pLast->pNext = pHead;
	pStop = pHead;
}

template <typename T>
TRingList<T>::TRingList(const TRingList<T>& lst) : TList<T>(lst)
{
	pHead = new TNode<T>(0, lst.getFirst());
	if (pLast != nullptr) pLast->pNext = pHead;
	pStop = pHead;
```

```
}

template <typename T>
TRingList<T>::TRingList(TNode<T>* _pFirst) : TList<T>(_pFirst) {
	pHead = new TNode<T>(pFirst->key, pFirst);
	pLast->pNext = pHead;
	pStop = pHead;
}

template <typename T>
TRingList<T>::~TRingList() {
	if (pFirst == nullptr) return;
	delete pHead;
}

template <typename T>
void TRingList<T>::insertFirst(const T& key) {
	TList<T>::insertFirst(key);
	pHead = new TNode<T>(-1, pFirst);
	pLast->pNext = pHead;
	pStop = pHead;
}

//template <typename T>
//void TRingList<T>::Next() {
//    pCurrent = pCurrent->pNext;
//}

template <typename T>
void TRingList<T>::Remove(const T& key) {
	TList<T>::Remove(key);
	pHead->pNext = pFirst;
	TNode<T>* tmp = pFirst;
	while (tmp->pNext != pStop) tmp = tmp->pNext;
	pLast = tmp;
}

template <typename T>
void TRingList<T>::Clear() {
	TList<T>::Clear();
	if (pFirst != nullptr) delete pFirst;
	pFirst = nullptr;
	pHead = nullptr;
}
```

# Приложение Г. Реализация класса TMonom

```
#include "TMonom.h"

double dc(double num, int deg) {
	double res = 1.0;
	for (int i = 0; i < deg; i++) res *= num;
	return num;
}

Monom::Monom(const Monom& monom) {
	coef = monom.coef;
	degX = monom.degX;
	degY = monom.degY;
	degZ = monom.degZ;
}
```

```
Monom::Monom(const double _coef, const int _degX, const int _degY, const int
_degZ) {
	coef = _coef;
	degX = _degX;
	degY = _degY;
	degZ = _degZ;
}

const Monom& Monom::operator=(const Monom& monom)
{
	coef = monom.coef;
	degX = monom.degX;
	degY = monom.degY;
	degZ = monom.degZ;
	return (*this);
}

bool Monom::operator==(const Monom& monom) const
{
	return (coef == monom.coef) && (degX == monom.degX) && (degY ==
monom.degY) && (degZ == monom.degZ);
}

bool Monom::operator!=(const Monom& monom) const
{
	return !(*this == monom);
}

bool Monom::operator>(const Monom& monom) const
{
	return ((degX > monom.degX) || (degX == monom.degX && degY >
monom.degY) ||
		(degX == monom.degX && degY == monom.degY && degZ >
monom.degZ));
}

bool Monom::operator<(const Monom& monom) const
{
	return ((degX < monom.degX) || (degX == monom.degX && degY <
monom.degY) ||
		(degX == monom.degX && degY == monom.degY && degZ < monom.degZ)
|| (degX == monom.degX && degY == monom.degY && degZ == monom.degZ && coef <
monom.coef));
}

Monom Monom::operator*(const Monom& monom) const
{
	return Monom(monom.coef * coef, monom.degX + degX, degY + monom.degY,
degZ + monom.degZ);
}

double Monom::getCoef() const { return coef; }

int Monom::getDegX() const { return degX; }

int Monom::getDegY() const { return degY; }

int Monom::getDegZ() const { return degZ; }

double Monom::Calculate(double x, double y, double z)
{
	return coef * dc(x, degX) * dc(y, degY) * dc(z, degZ);
}
```

```
Monom Monom::difX() const
{
	if (*this == Monom()) return Monom();
	return Monom(coef * degX, degX - 1, degY, degZ);
}

Monom Monom::difY() const
{
	if (*this == Monom()) return Monom();
	return Monom(coef * degY, degX, degY - 1, degZ);
}

Monom Monom::difZ() const
{
	if (*this == Monom()) return Monom();
	return Monom(coef * degZ, degX, degY, degZ - 1);
}

string Monom::toString() const {
	string m = "";
	if (coef != 0) {
		m += to_string(coef);
		if (degX != 0 && degX == 1) m += "*X";
		else if (degX > 1) m += "*X^" + to_string(degX);
		if (degY != 0 && degY == 1) m += "*Y";
		else if (degY > 1) m += "*Y^" + to_string(degY);
		if (degZ != 0 && degZ == 1) m += "*Z";
		else if (degZ > 1) m += "*Z^" + to_string(degZ);
	}
	return m;
}
```

# Приложение Д. Реализация класса TPolynom

```
#include "TPolynom.h"

double pow(double num, int k) {
	double res = 1.0;
	for (int i = 0; i < k; i++) res *= num;
	return res;
}

void TPolynom::SortInsert(Monom& m) {
	TNode<Monom>* less = new TNode<Monom>;

	if (findLess(m, less)) {
		if (m.degX == less->key.degX && m.degY == less->key.degY &&
m.degZ == less->key.degZ) less->key.coef += m.coef;
		else monoms->insertAfter(less->key, m);
	}
	else if (m == less->key) monoms->insertFirst(m);
	else monoms->insertLast(m);

}

bool TPolynom::isSimilar(Monom& a, Monom& b) const {
	return (a.degX == b.degX && a.degY == b.degY && a.degZ == b.degZ);
}
```

```
bool TPolynom::findLess(Monom& m, TNode<Monom>*& g)
{
	monoms->Reset();
	TNode<Monom>* l = new TNode<Monom>(m);
	g = new TNode<Monom>(m);
	while (monoms->getCurrent() != nullptr && monoms->getCurrent() !=
monoms->getStop() &&
				(monoms->getCurrent()->key < m ||
isSimilar(monoms->getCurrent()->key, m)))
	{
		l = monoms->getCurrent();
		monoms->Next();
	}
	if (monoms->getCurrent() == nullptr || monoms->getCurrent() ==
monoms->getFirst()) return false;
	else {
		g = l;
		return true;
	}
}

void TPolynom::setDegree(Monom& m ,string& param, int deg) {
	if (param == "x" || param == "X") m.degX = deg;
	if (param == "y" || param == "Y") m.degY = deg;
	if (param == "z" || param == "Z") m.degZ = deg;
}

void TPolynom::toPolynome()
{
	Monom m;
	string last_elem = mas[0];
	string last_param = "";
	int cf = 1;
	if (IsConst(last_elem)) m.coef = stod(last_elem);
	else if (last_elem == "-") cf = -1;
	else if (this->IsParam(last_elem)) last_param = last_elem;
	for (int i = 1; i < mas.size(); i++) {
		string tok = mas[i];
		if (tok == "+" || tok == "-") {
			if (last_param != "" && !IsConst(last_elem)) setDegree(m,
last_param, 1);
			SortInsert(m);
			(tok == "+") ? cf = 1 : cf = -1;
			m = Monom(1.0, 0, 0, 0);
		}
		else if (tok == "*" && IsParam(last_elem)) setDegree(m,
last_elem, 1);
		if (this->IsParam(tok)) last_param = tok;
		else if (IsConst(tok)) {
			if (last_elem == "^") setDegree(m, last_param, stoi(tok));
			else if (last_elem == "+" || last_elem == "-") {
				m.coef = stod(tok) * cf;
				cf = 1;
			}
		}
		last_elem = tok;
	}
	m.coef *= cf;
	if (last_param != "" && !IsConst(last_elem)) setDegree(m, last_param,
1);
	SortInsert(m);
	//giveSimTer();
```

```
}

void TPolynom::giveSimTer() {
	monoms->Reset();
	while (monoms->getCurrent() != monoms->getStop()) {
		TNode<Monom>* curr = monoms->getCurrent();
		TNode<Monom>*		tmp		=		new
TNode<Monom>(monoms->getCurrent()->pNext);
		while	(tmp	!=	monoms->getStop()	&&	curr->pNext	!=
monoms->getStop()) {
			if	(tmp->key.degX	==	curr->key.degX	&&	tmp->key.degY	==
curr->key.degY && tmp->key.degZ == curr->key.degZ) {
				curr->key.coef += tmp->key.coef;
				TNode<Monom>* toDel = tmp;
				tmp = tmp->pNext;
				monoms->Remove(toDel->key);
			}
			else tmp = tmp->pNext;
		}
		monoms->Next();
	}
}

TPolynom::TPolynom()
{
	arexp = "";
	monoms = new TRingList<Monom>;
}

TPolynom::TPolynom(const string& _polynome)
{
	arexp = _polynome + "\0";
	monoms = new TRingList<Monom>;
	GetTokens();
	toPolynome();
}

TPolynom::TPolynom(const TRingList<Monom>& ringlist) {
	TRingList<Monom> ringListCopy(ringlist);
	ringListCopy.Reset();
	TNode<Monom>* tmp = new TNode<Monom>(ringlist.getFirst());
	monoms = new TRingList<Monom>(tmp);
	while (tmp != ringlist.getStop()) {
		ringListCopy.Next();
		tmp = new TNode<Monom>(ringlist.getCurrent());
		TNode<Monom> *less = new TNode<Monom>;
		if (findLess(tmp->key, less)) {
			if	(tmp->key.degX	==	less->key.degX	&&	tmp->key.degY	==
less->key.degY	&&	tmp->key.degZ	==	less->key.degZ)	less->key.coef	+=
tmp->key.coef;
			else monoms->insertAfter(less->key, tmp->key);
		}
		else if (tmp->key == less->key) monoms->insertFirst(tmp->key);
		else monoms->insertLast(tmp->key);
	}
}

TPolynom::TPolynom(const TPolynom& polynom) {
	monoms = new TRingList<Monom>(*(polynom.monoms));
	arexp = polynom.arexp;
	mas = polynom.mas;
	post_form = polynom.post_form;
	values = polynom.values;
```

```
}

TPolynom::~TPolynom()
{
	if (monoms->isEmpty() == true) return;
	monoms->Clear();
}

const TPolynom& TPolynom::operator=(const TPolynom& polynom) {
	if (this == &polynom)
	{
		return *this;
	}
	monoms = new TRingList<Monom>(*(polynom.monoms));
	arexp = polynom.arexp;
	mas = polynom.mas;
	post_form = polynom.post_form;
	values = polynom.values;
	return *this;
}

TPolynom TPolynom::operator+(TPolynom& polynom) {
	TPolynom res(*this);
	TNode<Monom>* tmp = polynom.monoms->getFirst();
	while (tmp != polynom.monoms->getStop()) {
		TNode<Monom> *less = new TNode<Monom>;
		if (res.findLess(tmp->key, less)) {
			if (tmp->key.degX == less->key.degX && tmp->key.degY ==
less->key.degY && tmp->key.degZ == less->key.degZ) less->key.coef +=
tmp->key.coef;
			else res.monoms->insertAfter(less->key, tmp->key);
		}
		else if (tmp->key == less->key)
res.monoms->insertFirst(tmp->key);
		else res.monoms->insertLast(tmp->key);
		tmp = tmp->pNext;
	}
	//res.giveSimTer();
	res.arexp = res.InfixForm();
	return res;
}

TPolynom TPolynom::operator-(TPolynom& polynom) {
	polynom *= -1;
	TPolynom res = *this + polynom;
	return res;
}

TPolynom TPolynom::operator*(TPolynom& polynom) {
	TPolynom res;
	monoms->Reset();
	while (monoms->getCurrent() != monoms->getStop()) {
		TNode<Monom>* tmp = polynom.monoms->getFirst();
		while (tmp != polynom.monoms->getStop()) {
			TNode<Monom>* k = monoms->getCurrent();
			if (k->key.coef == 0) break;
			Monom m(k->key.coef * tmp->key.coef,
				k->key.degX + tmp->key.degX,
				k->key.degY + tmp->key.degY,
				k->key.degZ + tmp->key.degZ);
			res.SortInsert(m);
			tmp = tmp->pNext;
		}
```

```
            monoms->Next();
      }
      res.arexp = res.InfixForm();
      return res;
}

TPolynom TPolynom::operator*=(int num) {
      this->monoms->Reset();
      while (this->monoms->getCurrent() != this->monoms->getStop()) {
            this->monoms->getCurrent()->key.coef *= num;
            this->monoms->Next();
      }
      return *this;
}

TPolynom TPolynom::operator*=(float num) {
      this->monoms->Reset();
      while (this->monoms->getCurrent() != this->monoms->getStop()) {
            this->monoms->getCurrent()->key.coef *= num;
            this->monoms->Next();
      }
      return *this;
}

double TPolynom::operator()(double x, double y, double z) {
      double res = 0.0;
      monoms->Reset();
      TNode<Monom>* cur = monoms->getCurrent();
      while (cur != monoms->getStop()) {
            double num = 1.0;
            num *= cur->key.coef;
            num *= pow(x, cur->key.degX);
            num *= pow(y, cur->key.degY);
            num *= pow(z, cur->key.degZ);
            cur = cur->pNext;
            res += num;
      }
      return res;
}

TPolynom TPolynom::dif_x() const {
      TPolynom res;
      monoms->Reset();
      TNode<Monom>* cur = monoms->getCurrent();
      while (cur != monoms->getStop()) {
            if (cur->key.coef == 0 || cur->key.degX == 0) {
                  cur = cur->pNext;
                  Monom m = Monom(0, 0, 0, 0);
                  res.SortInsert(m);
                  continue;
            }
            Monom m(cur->key.coef * cur->key.degX, cur->key.degX - 1,
cur->key.degY, cur->key.degZ);
            res.monoms->insertLast(m);
            cur = cur->pNext;
      }
      res.arexp = res.InfixForm();
      if (res.monoms->getFirst()->key.coef == 0 &&
res.monoms->getFirst()->pNext != res.monoms->getStop())
res.monoms->Remove(Monom(0, 0, 0, 0));
      return res;
}
```

```
TPolynom TPolynom::dif_y() const {
	TPolynom res;
	monoms->Reset();
	TNode<Monom>* cur = monoms->getCurrent();
	while (cur != monoms->getStop()) {
		if (cur->key.coef == 0 || cur->key.degY == 0) {
			cur = cur->pNext;
			Monom m = Monom(0, 0, 0, 0);
			res.SortInsert(m);
			continue;
		}
		Monom	m(cur->key.coef	*	cur->key.degY,	cur->key.degX,
cur->key.degY - 1, cur->key.degZ);
		res.monoms->insertLast(m);
		cur = cur->pNext;
	}
	res.arexp = res.InfixForm();
	if	(res.monoms->getFirst()->key.coef	==	0	&&
res.monoms->getFirst()->pNext	!=	res.monoms->getStop())
res.monoms->Remove(Monom(0, 0, 0, 0));
	return res;
}

TPolynom TPolynom::dif_z() const {
	TPolynom res;
	monoms->Reset();
	TNode<Monom>* cur = monoms->getCurrent();
	while (cur != monoms->getStop()) {
		if (cur->key.coef == 0 || cur->key.degZ == 0) {
			cur = cur->pNext;
			Monom m = Monom(0, 0, 0, 0);
			res.SortInsert(m);
			continue;
		}
		Monom	m(cur->key.coef	*	cur->key.degZ,	cur->key.degX,
cur->key.degY, cur->key.degZ - 1);
		res.monoms->insertLast(m);
		cur = cur->pNext;
	}
	res.arexp = res.InfixForm();
	if	(res.monoms->getFirst()->key.coef	==	0	&&
res.monoms->getFirst()->pNext	!=	res.monoms->getStop())
res.monoms->Remove(Monom(0, 0, 0, 0));
	return res;
}

bool TPolynom::operator==(const TPolynom& polynom) const {
	monoms->Reset();
	polynom.monoms->Reset();
	while	(monoms->getCurrent()	!=	monoms->getStop()	&&
polynom.monoms->getCurrent() != polynom.monoms->getStop()) {
		if	(monoms->getCurrent()->key	!=
polynom.monoms->getCurrent()->key) return false;
		monoms->Next();
		polynom.monoms->Next();
	}
	if	(monoms->getCurrent()	!=	monoms->getStop()	&&
polynom.monoms->getCurrent() == polynom.monoms->getStop() ||
		monoms->getCurrent()	==	monoms->getStop()	&&
polynom.monoms->getCurrent() != polynom.monoms->getStop()) return false;
	return true;
}
```

```
void TPolynom::ToString() {
      arexp = "";
      monoms->Reset();
      while (monoms->getCurrent() != monoms->getStop()) {
            if (monoms->getCurrent()->key.coef > 0 && arexp != "") arexp +=
"+";
            arexp += monoms->getCurrent()->key.toString();
            monoms->Next();
      }
      if (arexp == "") arexp = "0";
}
```