МИНИСТЕРСТВО НАУКИ И ВЫСШЕГООБРАЗОВАНИЯ
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
_____ /Миронов А. И./
Подпись

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

Нижний Новгород
2024

Содержание

Введение

Лабораторная работа направлена на изучение обработки полиномов от трёх переменных (x, y, z). Полиномы могут быть использованы для решения многих задач математического анализа, теории вероятностей, линейной алгебры и других областей математики

В данной лабораторной работе студенты будут изучать основные принципы работы алгоритма обработки полиномов и реализовывать его на практике. Это позволит им лучше понять принципы работы связного списка и освоить навыки работы с алгоритмами обработки полиномов.

1 Постановка задачи

Цель:

Цель лабораторной работы – научиться представлять полиномы в виде связных списков, где каждый узел списка содержит моном. Такое представление позволяет эффективно решать задачи сложения, вычитания, умножения и вычисления значений полиномов.

Задачи:

1. Изучение основных принципов работы со связным списком.
2. Создание полинома на основе списка с головой, элементами которого являются мономы. Каждый моном определяется коэффициентом и набором степеней.
3. Написание программы на C++, использующей связный список для преобразования арифметического выражения.
4. Анализ времени выполнения программы и оценка эффективности использования связного для данной задачи.
5. Тестирование программы на различных входных данных, включая выражения с разными операциями и скобками.

Результатом выполнения лабораторной работы станет полнофункциональная реализация алгоритмов работы с полиномами на связных списках, которая может быть использована для решения задач математического анализа, теории вероятностей и других областей математики.

2 Руководство пользователя

2.1 Приложение для демонстрации работы связного списка

1. Запустите приложение с названием sample_tlist.exe. В результате появится окно, показанное ниже и вам будет предложено ввести два связного списка. Для каждого необходимо ввести целое число n и далее n (рис. 1).

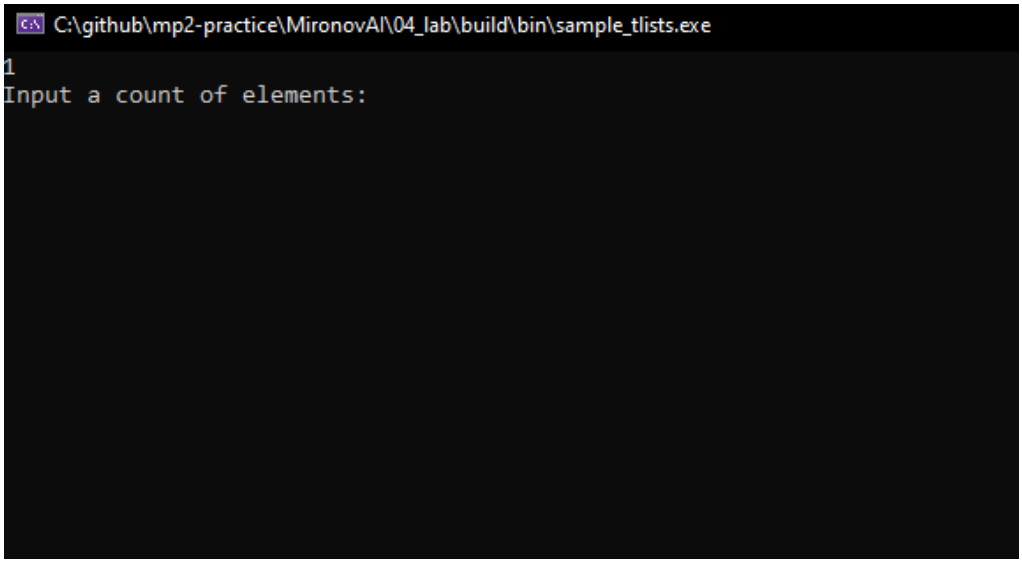


Рис. 1. Основное окно программы

2. После ввода будет выведены результаты соответствующих операций и функций стека (рис. 2).

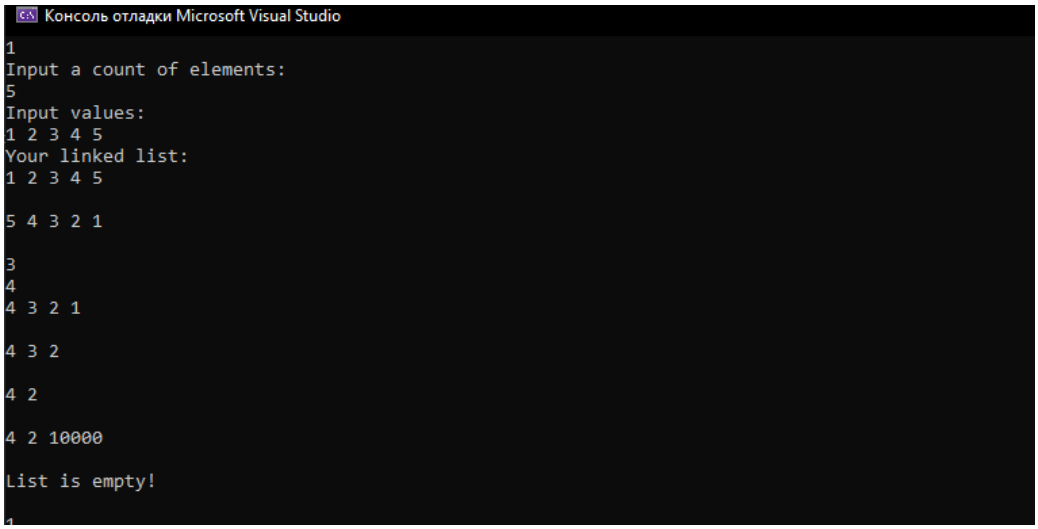


Рис. 2. Результат тестирования функций класса `TList`

2.2 Приложение для демонстрации работы полиномов.

1. Запустите приложение с названием sample_tpolynom.exe. В результате появится окно, показанное ниже, вам будет предложено ввести два полинома,в одну строку (рис. 3).

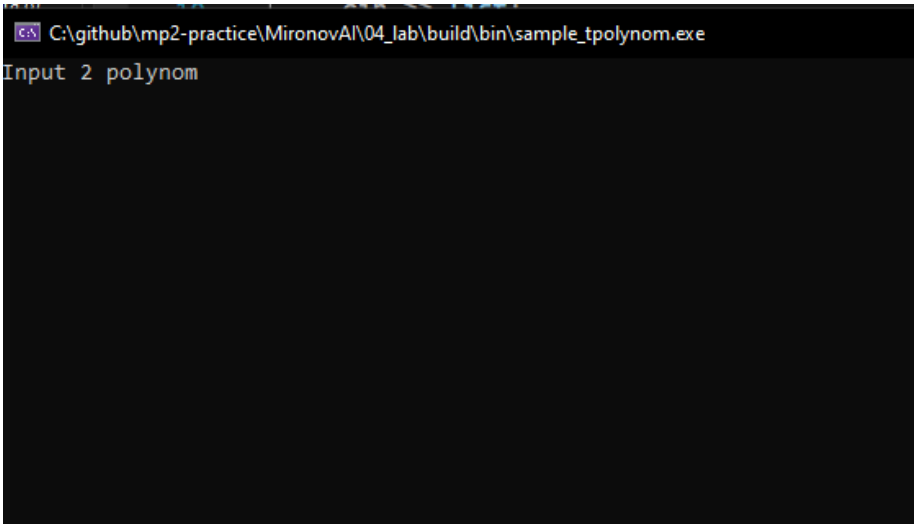


Рис. 3. Основное окно программы

2. После ввода арифметического выражения будет выведены результаты соответствующих операций и функций (рис. 4).

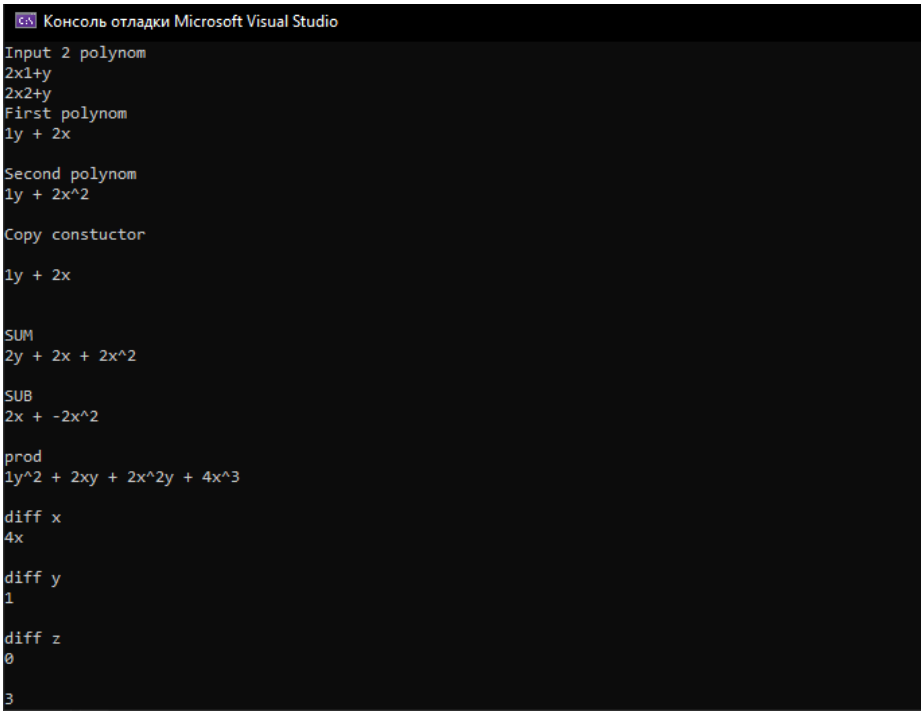


Рис. 4. Результат тестирования функций класса `TPolynom`

3 Руководство программиста

3.1 Описание алгоритмов

3.1.1 Линейный односвязный список

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

Операция добавления элемента реализуется при помощи указателя на первый элемент. Если структура хранения пуста, то мы просто создаем новый элемент, иначе создаём новый элемент и сдвигаем указатель на начало.

Пример:

4	2		

Операция добавления элемента (1) в начало:

1	4	2	

Операция добавления в конец

Операция добавления элемента реализуется при помощи указателя на последний элемент. Если структура хранения пуста, то мы просто создаем новый элемент, иначе создаём новый элемент и сдвигаем указатель на конец.

Пример:

4	2		

Операция добавления элемента (1) в конец:

4	2	1	

Операция добавления после текущего

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

Пример:

Текущий элемент: 4

4	2		

Операция добавления элемента (1) после текущего:

4	1	2	

Операция удаления первого элемента

Операция удаления элемента реализуется при помощи указателя на первый элемент.

Пример:

4	2		

Операция удаления первого элемента:

2			

Операция удаления последнего элемента

Операция удаления элемента реализуется при помощи указателя на последний элемент.

Пример:

4	2		

Операция удаления последнего элемента:

4			

Операция удаления текущего элемента

Операция удаления элемента реализуется при помощи указателя на текущий элемент.

Пример :

Текущий элемент 2

4	2	1	

Операция удаления текущего элемента:

4	1		

Операция удаления элемента

Операция удаления элемента при помощи перебора всех элементов списка.

Пример :

Удалить элемент 2

4	2	1	

Операция удаления элемента:

4	1		

Операция получения текущего элемента.

Операция взятия элемента с вершины также реализуется указателя на текущий элемент.

Пример:

4	2		

Операция взятия элемента если текущий по-умолчанию:

Результат: 4

Операция поиска.

Операция поиска ищет элемент в списке.

Пример:

4	2		

Операция поиска 2 элемента:

Результат: Указатель на 2 элемент

Операция проверки на пустоту.

Операция проверки на полноту проверяет, есть ли хотя бы один элемент в списке. Также реализуется при помощи указателя на первый элемент.

Пример 1:

4	2		

Операция проверки на полноту:

Результат: false

Пример 2:

Операция проверки на полноту:

Результат: true

Операция сортировки.

Операция сортировки позволяет сортировать список.

Пример :

4	2		

Сортировка по возрастанию:

2	4		

3.1.2 Кольцевой список с головой

Кольцевой односвязный список отличается от односвязного списка наличием указателя на фиктивную голову в конце. Это позволяет облегчить некоторые операции, и бесконечно сдвигать текущий элемент.

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

Операция добавления элемента реализуется при помощи указателя на первый элемент. Если структура хранения пуста, то мы просто создаем новый элемент, иначе создаём новый элемент и сдвигаем указатель на начало.

Пример:

4	2		

Операция добавления элемента (1) в начало:

1	4	2	

Операция добавления в конец

Операция добавления элемента реализуется при помощи указателя на последний элемент. Если структура хранения пуста, то мы просто создаем новый элемент, иначе создаём новый элемент и сдвигаем указатель на конец.

Пример:

4	2		

Операция добавления элемента (1) в конец:

4	2	1	

Операция добавления после текущего

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

Пример:

Текущий элемент: 4

4	2		

Операция добавления элемента (1) после текущего:

4	1	2	

Операция удаления первого элемента

Операция удаления элемента реализуется при помощи указателя на первый элемент.

Пример:

4	2		

Операция удаления первого элемента:

2			

Операция удаления последнего элемента

Операция удаления элемента реализуется при помощи указателя на последний элемент.

Пример:

4	2		

Операция удаления последнего элемента:

4			

Операция удаления текущего элемента

Операция удаления элемента реализуется при помощи указателя на текущий элемент.

Пример :

Текущий элемент 2

4	2	1	

Операция удаления текущего элемента:

4	1		

Операция удаления элемента

Операция удаления элемента при помощи перебора всех элементов списка.

Пример :

Удалить элемент 2

4	2	1	

Операция удаления элемента:

4	1		

Операция получения текущего элемента.

Операция взятия элемента с вершины также реализуется указателя на текущий элемент.

Пример:

4	2		

Операция взятия элемента если текущий по-умолчанию:

Результат: 4

Операция поиска.

Операция поиска ищет элемент в списке.

Пример:

4	2		

Операция поиска 2 элемента:

Результат: Указатель на 2 элемент

Операция проверки на пустоту.

Операция проверки на полноту проверяет, есть ли хотя бы один элемент в списке. Также реализуется при помощи указателя на первый элемент.

Пример 1:

4	2		

Операция проверки на полноту:

Результат: false

Пример 2:

Операция проверки на полноту:

Результат: true

Операция сортировки.

Операция сортировки позволяет сортировать список.

Пример :

4	2		

Сортировка по возрастанию:

2	4		

3.1.3 Полином

Программа предоставляет возможности для работы с полиномами: суммирование, произведение, дифференцирование полиномов.

Алгоритм на входе требует строку, которая представляет некоторый полином. Алгоритм допускает наличия трёх независимых переменных и положительные целые степени независимых переменных.

Операция суммирования полиномов

Операция суммирования полиномов согласно математическим правилам

Пример:

(-2x^2 + 3x*y*z + 1) + (3x^2+1)

Результат:

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

Операция вычитания полиномов

Операция вычитания полиномов согласно математическим правилам

Пример:

(-2x^2 + 3x*y*z + 1) - (3x^2+1)

Результат:

-5x^2 - 3x*y*z

Операция произведения полиномов

Операция произведения полиномов согласно математическим правилам

Пример:

(-2x^2 + 3x*y*z + 1) * (3x^2+1)

Результат:

-6x^4 + x^2 + 9x^2yz + 3xyz+ 3x*y*z + 2

Операция дифференцирования полиномов

Операция дифференцирования полинома согласно математическим правилам. Возможно дифференцирование по независимым переменным x, y или z.

Пример:

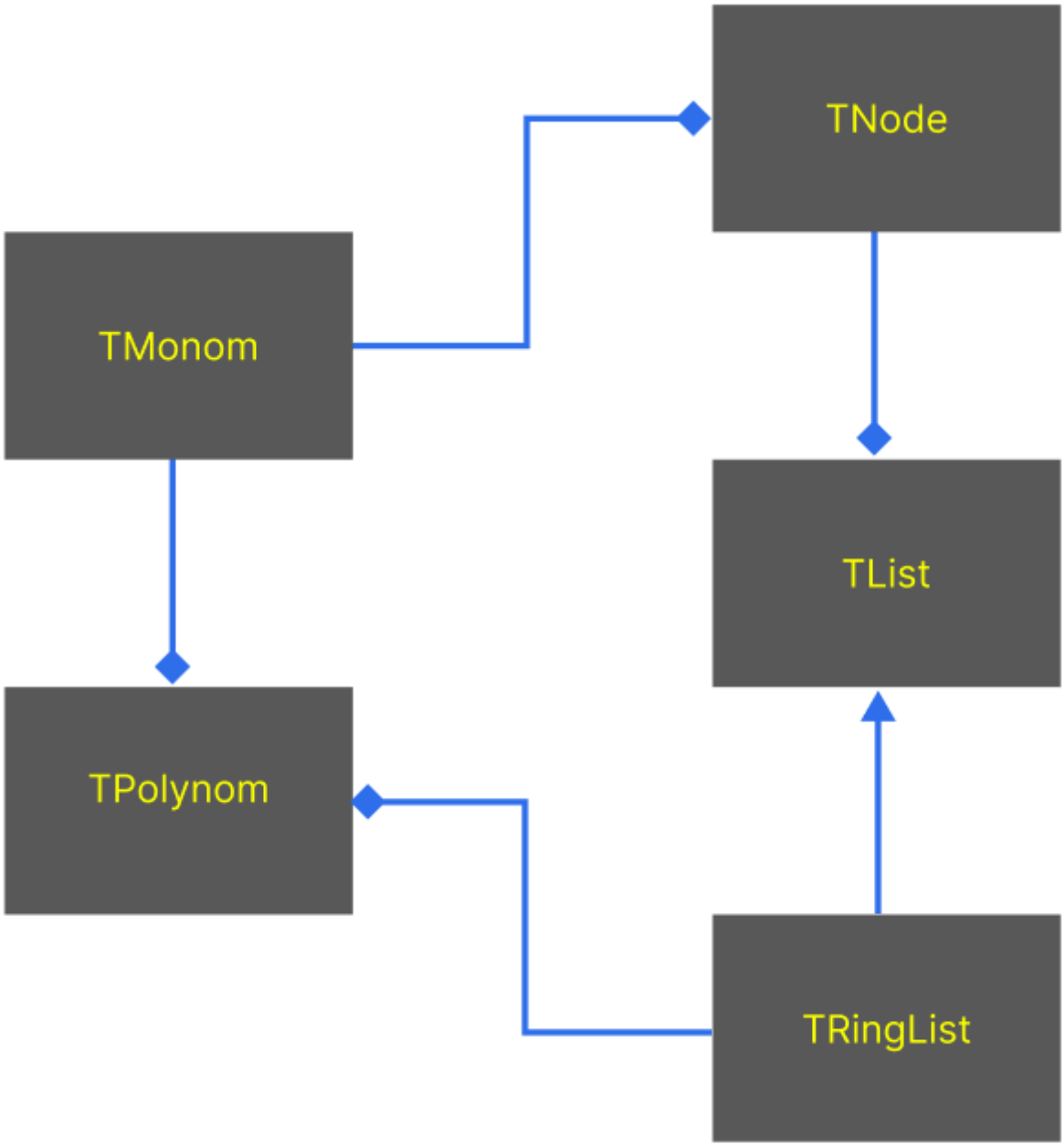
3x*y*z

Результат дифференцирования (по x, y и z):

3yz, 3xz, 3yz.

3.2 Описание программной реализации

3.2.1 Схема наследования классов



3.2.2 Описание структуры TNode

```
template <class Type>
struct TNode
{
	Type value;
	TNode<Type>* next;

	TNode(const Type& value = Type());
```

```
        bool operator<(const TNode<Type>& node) const;
        bool operator>(const TNode<Type>& node) const;
        bool operator==(const TNode<Type>& node) const;
        bool operator!=(const TNode<Type>& node) const;

};
```

Назначение: представление узла списка

Поля:

value – данные, которые хранит узел

next – указатель на следующий узел

Методы:

TNode(const Type& value = Type());

Назначение: конструктор по умолчанию, конструктор с параметрами.

Входные параметры отсутствуют:

Выходные параметры: отсутствуют.

bool operator<(const TNode<Type>& node) const;

Назначение: перегрузка оператора меньше
Входные параметры:
node **– узел, который сравниваем**
Выходные параметры:
true или false

bool operator==(const TNode<Type>& node) const;

Назначение: перегрузка оператора меньше
Входные параметры:
node **– узел, который сравниваем**
Выходные параметры:
true или false

3.2.3 Описание класса TList

```
template <class Type>
class TList
{
protected:
        TNode<Type>* head;              // first element
        TNode<Type>* curr;              // current node
        TNode<Type>* last;              // last element
        TNode<Type>* stop;

public:

        TList();
        TList(const TList<Type>& list);
        TList(const TNode<Type>* node);
        virtual ~TList();
        virtual void pop_first();
        virtual void pop_last();
        virtual void pop_curr();
```

```
        void remove(const TNode<Type>* node);
        void remove(const Type& value);

        virtual void push_back(const Type& value);
        virtual void push_front(const Type& value);
        virtual void push_after_curr(const Type& value);

        virtual TNode<Type>* find_prev(const Type& value) const;
        virtual TNode<Type>* find(const Type& value) const;


        TNode<Type>* get_curr() const;
        int get_size() const;

        void start();
        bool empty() const;
        virtual void next();
        void sort(bool reverse=true);
        virtual void clear();
        virtual void copy(const TNode<Type>* node);

        friend istream& operator>>(istream& buf, TList<Type>& list);
        friend ostream& operator<<(ostream& buf, TList<Type>& list);


};
```

Назначение: представление списка.

Поля:

head – указатель на первый элемент списка.

curr – указатель на текущий элемент списка (по-умолчанию равен указателю на первый элемент).

last – указатель на последний элемент списка стой).

Методы:

TList();

Назначение: конструктор по умолчанию.

Входные параметры отсутствуют:

Выходные параметры: отсутствуют.

TList(const TList<Type>& list);

Назначение: конструктор копирования.

Входные параметры:

list – список, на основе которого создаем новый список.

Выходные параметры: отсутствуют.

TList(const TNode<Type>& node);

Назначение: конструктор с параметрами.

Входные параметры:

node – узел, на основе которого создаем новый список.

Выходные параметры: отсутствуют.

virtual ~TList();

Назначение: деструктор.

Входные параметры отсутствуют:

Выходные параметры: отсутствуют.

virtual void copy();

Назначение: копирование списка.

Входные параметры отсутствуют:

Выходные параметры: отсутствуют.

virtual void pop_first();

Назначение: удаление первого элемента.

Входные параметры отсутствуют.

Выходные параметры отсутствуют.

virtual void pop_last();

Назначение: удаление последнего элемента.

Входные параметры отсутствуют.

Выходные параметры отсутствуют.

virtual void pop_curr();

Назначение: удаление текущего элемента.

Входные параметры отсутствуют.

Выходные параметры отсутствуют.

void remove(const TNode<Type>* node);

Назначение: удаление элемента.

Входные параметры:

node – узел, который хотим удалить.

Выходные параметры отсутствуют.

void remove(const Type& value);

Назначение: удаление элемента.

Входные параметры:

value – элемент, который хотим удалить.

Выходные параметры отсутствуют.

virtual void push_front(const Type& value);

Назначение: добавление элемента в начало.

Входные параметры:

value – добавляемый элемент.

Выходные параметры отсутствуют.

virtual void push_back(const Type& value);

Назначение: добавление элемента в конец.

Входные параметры:

value – добавляемый элемент.

Выходные параметры отсутствуют.

virtual void push_after_curr(const Type& value);

Назначение: добавление элемента после текущего элемента.

Входные параметры:

value – добавляемый элемент.

Выходные параметры отсутствуют.

TNode<Type>* find(const Type& value) const;

Назначение: поиск элемента.

Входные параметры:

value – элемент, который ищем.

Выходные параметры:

Указатель на элемент

bool empty() const;

Назначение: проверка на пустоту.

Входные параметры отсутствуют.

Выходные параметры отсутствуют.

void sort(int reverse) const;

Назначение: сортировка списка.

Входные параметры:

reverse – флаг, показывающий в каком порядке сортировать (по возрастанию по-умолчанию).

Выходные параметры отсутствуют.

virtual void clear();

Назначение: отчистка списка.

Входные параметры отсутствуют.

Выходные параметры отсутствуют.

3.2.4 Описание класса TRingList

```
template <class Type>
class TRingList: public TList<Type>
{
private:
	TNode<Type>* fict_head;
public:

	TRingList();
	TRingList(const TRingList<Type>& list);
	TRingList(const TNode<Type>* node);
	virtual ~TRingList();
	void pop_first();
	void pop_last();
	void pop_curr();

	void next();
	void clear();
	void push_after_curr(const Type& value);
	void push_back(const Type& value);
	void push_front(const Type& value);
	bool operator==(const TRingList<Type>& list) const;
	friend ostream& operator<<(ostream& buf, TRingList<Type>& list)
	{
		TNode<Type>* tmp = list.fict_head->next;

		if (list.empty())
		{
			buf << "List is empty!\n";
			return buf;
		}

		while (tmp != list.fict_head)
		{
			buf << tmp->value << endl;
			tmp = tmp->next;
		}


		return buf;
	}
};
```

Назначение: представление кольцевого списка.

Поля:

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

Методы:

Класс TRingList наследуется от TList, некоторые методы из TList также будут работать и с TRingList. Среди таких методов, которые не перекрываются в TRingList: remove, find, sort, empty, start. (смотреть абзац 3.2.2).

`TRingList();`

Назначение: конструктор по умолчанию.

Входные параметры отсутствуют:

Выходные параметры: отсутствуют.

`TRingList(const TRingList<Type>& list);`

Назначение: конструктор копирования.

Входные параметры:

`list –` список, на основе которого создаем новый список.

Выходные параметры: отсутствуют.

`TRingList(const TNode<Type>& node);`

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 pop_first();`

Назначение: удаление первого элемента.

Входные параметры отсутствуют.

Выходные параметры отсутствуют.

`void pop_last();`

Назначение: удаление последнего элемента.

Входные параметры отсутствуют.

Выходные параметры отсутствуют.

`void pop_curr();`

Назначение: удаление текущего элемента.

Входные параметры отсутствуют.

Выходные параметры отсутствуют.

void push_front(const Type& value);

Назначение: добавление элемента в начало.

Входные параметры:

value – добавляемый элемент.

Выходные параметры отсутствуют.

void push_back(const Type& value);

Назначение: добавление элемента в конец.

Входные параметры:

value – добавляемый элемент.

Выходные параметры отсутствуют.

void push_after_curr(const Type& value);

Назначение: добавление элемента после текущего элемента.

Входные параметры:

value – добавляемый элемент.

Выходные параметры отсутствуют.

TNode<Type>* find(const Type& value) const;

Назначение: поиск элемента.

Входные параметры:

value – элемент, который ищем.

Выходные параметры:

Указатель на элемент

void clear();

Назначение: отчистка списка.

Входные параметры отсутствуют.

Выходные параметры отсутствуют.

3.2.5 Описание класса TMonom

```
class TMonom
{
private:

      double coef;
      int degree_x;
      int degree_y;
```

```
      int degree_z;

public:
      TMonom(const TMonom& monom);
      TMonom(const double coef = 0.0, const int degreex = -1, const int degreey
= -1, const int degreez = -1);

      bool operator<(const TMonom& monom) const;
      bool operator>(const TMonom& monom) const;
      bool operator==(const TMonom& monom) const;
      bool operator!=(const TMonom& monom) const;
      TMonom operator*(const TMonom& monom) const;

      double get_coef() const;
      void set_coef(double num);
      void inc_coef(double digit);

      double eval(double x, double y, double z);
      TMonom dif_x() const;
      TMonom dif_y() const;
      TMonom dif_z() const;

      friend ostream& operator<<(ostream& buf, TMonom& monom)
```

Назначение: представление монома

Поля:

coef – коэффициент монома

degree_x – степень при независимой переменной x

degree_y – степень при независимой переменной y

degree_z – степень при независимой переменной z

Методы:

TMonom(const TMonom& monom);

Назначение: конструктор по копирования.

Входные параметры:

monom – моном, который копируем.

Выходные параметры: отсутствуют.

TMonom(const double coef = 0.0, const int degreex = -1, const int degreey = -1, const int degreez = -1);

Назначение: конструктор с параметрами, конструктор по-умолчанию.

Входные параметры:

coef – коэффициент монома,

degree_x – степень при x,

degree_y – степень при y,

degree_z – степень при z.

Выходные параметры: отсутствуют.

bool operator<(const TMonom& monom) const;

Назначение: перегрузка оператора меньше.

Входные параметры:

monom – моном, который сравниваем.

Выходные параметры:

true или false.

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

Назначение: перегрузка оператора меньше.

Входные параметры:

monom – моном, который сравниваем.

Выходные параметры:

true или false.

TMonom operator*(const TMonom& monom) const;

Назначение: перегрузка оператора умножения мономов.

Входные параметры:

monom – моном, который умножаем.

Выходные параметры:

Моном, который равен произведению мономов.

double get_coef() const;

Назначение: получение коэффициента монома.

Входные параметры отсутствуют.

Выходные параметры:

Коэффициент.

double eval(double x, double y, double z);

Назначение: нахождение значения монома в точке.

Входные параметры:

x, y, z – координаты точки.

Выходные параметры:

Значение монома в точке.

TMonom dif_x() const;

Назначение: производной по x.

Входные параметры отсутствуют:

Выходные параметры:

Моном, являющийся производной по x исходного монома.

```
TMonom dif_y() const;
```

Назначение: производной по y.

Входные параметры отсутствуют:

Выходные параметры:

Моном, являющийся производной по y исходного монома.

```
TMonom dif_z() const;
```

Назначение: производной по z.

Входные параметры отсутствуют:

Выходные параметры:

Моном, являющийся производной по z исходного монома.

3.2.6 Описание класса TPolynom

```
class TPolynom {
private:
	TRingList<TMonom> monoms;

	void del_zeros();
	void parse(string polynom);
	void x(string token, char& stage, int& i, double& coef,
		int& degreex, int& degreey, int& degreez);
	void y(string token, char& stage, int& i, double& coef,
		int& degreex, int& degreey, int& degreez);
	void z(string token, char& stage, int& i, double& coef,
		int& degreex, int& degreey, int& degreez);
	void c(string token, char& stage, int& i, double& coef,
		int& degreex, int& degreey, int& degreez);
	string preparation(string polynom);

public:
	TPolynom();
	TPolynom(const string& polynom_string);
	TPolynom(const TRingList<TMonom>& list);
	TPolynom(const TPolynom& polynom);

	const TPolynom& operator=(const TPolynom& polynom);
	bool operator==(const TPolynom& polynom) const;
	TPolynom operator+(const TPolynom& polynom);
	TPolynom operator-(const TPolynom& polynom);
	TPolynom operator*(const TPolynom& polynom);
	double operator()(double x, double y, double z);

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

	TRingList<TMonom> get_monoms();
	friend ostream& operator<<(ostream& buf, TPolynom& polynom);
};
```

Назначение: работа с полиномами

Поля:

RingList<TMonom> monoms – список мономов.

Методы:

TPolynom();

Назначение: конструктор по умолчанию.

Входные параметры отсутствуют:

Выходные параметры: отсутствуют.

TPolynom(const TPolynom& polynom);

Назначение: конструктор копирования.

Входные параметры:

polynom – полином, на основе которого создаем новый полином.

Выходные параметры: отсутствуют.

TPolynom(const string& polynom);

Назначение: конструктор с параметрами.

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+(const TPolynom& polynom);

Назначение: суммирование полиномов.

Входные параметры:

polynom – строка, на основе которого создаем новый полином.

Выходные параметры: сумма полиномов.

TPolynom operator-(const TPolynom& polynom);

Назначение: разность полиномов.

Входные параметры:

polynom – строка, на основе которого создаем новый полином.

Выходные параметры: разность полиномов.

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

Назначение: умножение полиномов.

Входные параметры:

polynom – строка, на основе которого создаем новый полином.

Выходные параметры: произведение полиномов.

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

Назначение: вычисления полинома в точке.

Входные параметры:

x – значение переменной x,

y – значение переменной y,

z – значение переменной z

Выходные параметры: результат вычисления полинома в точке.

```
TPolynom dif_x() const;
```

Назначение: дифференцирование полинома по x.

Входные параметры отсутствуют.

Выходные параметры: дифференциал полинома по x.

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

Назначение: дифференцирование полинома по y.

Входные параметры отсутствуют.

Выходные параметры: дифференциал полинома по y.

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

Назначение: дифференцирование полинома по z.

Входные параметры отсутствуют.

Выходные параметры: дифференциал полинома по z.

```
TPolynom dif_x() const;
```

Назначение: дифференцирование полинома по x.

Входные параметры отсутствуют.

Выходные параметры: дифференциал полинома по x.

```
TRingList<TMonom> get_monoms();
```

Назначение: получение списка мономов.

Входные параметры отсутствуют.

Выходные параметры: список мономов.

Заключение

В ходе выполнения лабораторной работы студенты изучили основные принципы работы алгоритма обработки полиномов от трех переменных (x, y, z) и реализовали его на практике. Были изучены основные принципы работы со связным списком, а также реализована возможность создания полинома на основе списка коэффициентов и степеней.

Также была проведена проверка программы на различных входных данных, включая выражения с разными операциями . Результатом выполнения лабораторной работы стала полнофункциональная реализация алгоритмов работы с полиномами на связных списках, которая может быть использована для решения задач математического анализа, теории вероятностей и других областей математики.

В процессе выполнения лабораторной работы студенты освоили навыки работы с алгоритмами обработки полиномов и научились представлять полиномы в виде связных списков, что позволит им решать более сложные задачи в будущем.

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

```
template <class Type>
TNode<Type>::TNode(const Type& new_value) : value(new_value), next(nullptr) {}

template <class Type>
bool TNode<Type>::operator<(const TNode<Type>& node) const
{
	return this->value < node.value;
}

template <class Type>
bool TNode<Type>::operator>(const TNode<Type>& node) const
{
	return this->value > node.value;;
}

template <class Type>
bool TNode<Type>::operator==(const TNode<Type>& node) const
{
	return this->value == node.value;;
}

template <class Type>
bool TNode<Type>::operator!=(const TNode<Type>& node) const
{
	return this->value != node.value;;
}
```

Приложение Б. Реализация класса TList

```
#ifndef _TLIST_H_
#define _TLIST_H_
#ifndef _TLIST_H_
#define _TLIST_H_

#include "tnode.h"
#include <iostream>

using namespace std;

template <class Type>
class TList
{
protected:
	TNode<Type>* head;		// first element
	TNode<Type>* curr;		// current node
	TNode<Type>* last;		// last element
	TNode<Type>* stop;

public:

	TList();
	TList(const TList<Type>& list);
	TList(const TNode<Type>* node);
	virtual ~TList();
	virtual void pop_first();
```

```
	virtual void pop_last();
	virtual void pop_curr();
	void remove(const TNode<Type>* node);
	void remove(const Type& value);

	virtual void push_back(const Type& value);
	virtual void push_front(const Type& value);
	virtual void push_after_curr(const Type& value);

	virtual TNode<Type>* find_prev(const Type& value) const;
	virtual TNode<Type>* find(const Type& value) const;


	TNode<Type>* get_curr() const;
	int get_size() const;

	void start();
	bool empty() const;
	virtual void next();
	void sort(bool reverse=true);
	virtual void clear();
	virtual void copy(const TNode<Type>* node);

	friend istream& operator>>(istream& buf, TList<Type>& list)
	{
		int count;
		cout << "Input a count of elements:\n";
		cin >> count;
		cout << "Input values:" << endl;
		while (count)
		{
			count--;
			Type value; cin >> value;
			list.push_back(value);
		}

		return buf;
	}
	friend ostream& operator<<(ostream& buf, TList<Type>& list)
	{
		TNode<Type>* tmp = list.head;
		if (list.empty())
		{
			buf << "List is empty!\n";
			return buf;
		}
		while (tmp != nullptr)
		{
			buf << tmp->value << " ";
			tmp = tmp->next;
		}
		buf << endl;

		return buf;
	}

};

template <class Type>
TList<Type>::TList()
{
	head = nullptr;
	last = nullptr;
```

```
	curr = nullptr;
	stop = nullptr;
}


template <class Type>
void TList<Type>::copy(const TNode<Type>* node)
{
	TNode<Type>* tmp = node->next;
	head = new TNode<Type>(node->value);
	curr = head;
	last = head;
	stop = nullptr;
	while (tmp != nullptr)
	{
		last->next = new TNode<Type>(tmp->value);
		last = last->next;
		tmp = tmp->next;
	}
}


template <class Type>
TList<Type>::TList(const TList<Type>& list): TList<Type>()
{
	if (list.empty())
	{
		return;
	}
	copy(list.head);
}


template <class Type>
TList<Type>::TList(const TNode<Type>* node)
{
	if (node == nullptr)
	{
		return;
	}
	copy(node);
}


template <class Type>
TList<Type>::~TList<Type>()
{
	clear();
}

template <class Type>
bool TList<Type>::empty() const
{
	return head == nullptr;
}


template <class Type>
void TList<Type>::next()
{
	if (curr == nullptr)
	{
		string ex = "next isn`t exist";
```

```
            throw ex;
      }
      curr = curr->next;
}


template <class Type>
int TList<Type>::get_size() const
{
      TNode<Type>* tmp = head;
      int size = 0;
      while (tmp != stop)
      {
            size++;
            tmp = tmp->next;
      }
      return size;
}


template <class Type>
TNode<Type>* TList<Type>::get_curr() const
{
      return curr;
}


template <class Type>
TNode<Type>* TList<Type>::find_prev(const Type& value) const
{
      TNode<Type>* tmp = head;

      if (head->value == value)
      {
            return nullptr;
      }

      while (tmp != stop)
      {
            if (tmp->next->value == value) break;
            tmp = tmp->next;
      }
      if (tmp == stop)
      {
            return nullptr;
      }
      return tmp;
}


template <class Type>
TNode<Type>* TList<Type>::find(const Type& value) const
{
      if (head == nullptr) return nullptr;
      TNode<Type>* tmp = head;

      while (tmp != stop)
      {
            if (tmp->value == value)
            {
                  break;
            }
            tmp = tmp->next;
```

```
	}
	if (tmp == stop)
	{
		return nullptr;
	}
	return tmp;
}


template <class Type>
void TList<Type>::pop_first()
{
	if (head == nullptr)
	{
		string ex = "SizeError: can`t remove empty list";
		throw ex;
	}

	if (last == head)
	{
		*this = TList<Type>();
		return;
	}
	if (curr == head)
	{
		curr = head->next;
	}
	TNode<Type>* tmp = head->next;
	delete head;
	head = tmp;
}


template <class Type>
void TList<Type>::pop_last()
{
	if (head == nullptr)
	{
		string ex = "SizeError: can`t remove empty list";
		throw ex;
	}

	if (last == head)
	{
		*this = TList<Type>();
		return;
	}



	TNode<Type>* tmp = head;
	while (tmp->next != last)
	{
		tmp = tmp->next;
	}

	if (curr == last)
	{
		curr = tmp;
	}

	delete tmp->next;
	tmp->next = nullptr;
```

```
	last = tmp;

}


template <class Type>
void TList<Type>::remove(const TNode<Type>* node)
{
	if (head == nullptr)
	{
		string ex = "SizeError: can`t remove empty list";
		throw ex;
	}

	if (node == nullptr)
	{
		return;
	}

	if (head == node)
	{
		this->pop_first();
		return;
	}

	if (last == node)
	{
		this->pop_last();
		return;
	}

	TNode<Type>* tmp = head;

	while (tmp->next != node && tmp != stop)
	{
		tmp = tmp->next;
	}

	if (tmp == stop)
	{
		return;
	}

	TNode<Type>* tmp1 = tmp->next->next;

	// rightward shift
	if (curr == node)
	{
		curr = tmp1;
	}

	delete tmp->next;
	tmp->next = tmp1;
}


template <class Type>
void TList<Type>::remove(const Type& value)
{
	TNode<Type>* tmp = head;
```

```
	if (head == nullptr)
	{
		string ex = "SizeError: can`t remove empty list";
		throw ex;
	}

	if (head->value == value)
	{
		this->pop_first();
		return;
	}

	while (tmp->next != stop)
	{
		if (tmp->next->value == value)
		{
			break;
		}
		tmp = tmp->next;
	}

	if (tmp->next == stop)
	{
		return;
	}
	if (tmp->next->next == stop && tmp->next->value == value)
	{
		this->pop_last();
		return;
	}
	TNode<Type>* tmp1 = tmp->next->next;

	// rightward shift
	if (curr == tmp->next)
	{
		curr = tmp1;
	}
	delete tmp->next;
	tmp->next = tmp1;
}


template <class Type>
void TList<Type>::pop_curr()
{
	if (head == curr)
	{
		this->pop_first();
		return;
	}
	if (last == curr)
	{
		this->pop_last();
		return;
	}
	TNode<Type>* tmp1 = head, *tmp = curr;
	while (tmp1->next != curr) tmp1 = tmp1->next;
	curr = curr->next;
	tmp1->next = curr;
	delete tmp;
}
```

```
template <class Type>
void TList<Type>::push_back(const Type& value)
{
	if (last == nullptr)
	{
		last = new TNode<Type>(value);
		head = last;
		curr = head;
		return;
	}

	last->next = new TNode<Type>(value);
	last = last->next;
}


template <class Type>
void TList<Type>::push_front(const Type& value)
{
	if (head == nullptr)
	{
		head = new TNode<Type>(value);
		last = head;
		curr = head;
		return;
	}
	TNode<Type>* new_head = new TNode<Type>(value);
	new_head->next = head;
	head = new_head;
}


template <class Type>
void TList<Type>::push_after_curr(const Type& value)
{
	if (curr == last)
	{
		this->push_back(value);
		return;
	}

	TNode<Type>* tmp = curr->next;
	curr->next = new TNode<Type>(value);
	curr->next->next = tmp;
}


template<class Type>
void TList<Type>::sort(bool reverse)
{
	if (head == nullptr)
		return;

	TNode<Type>* tmp1 = head;
	while (tmp1->next != stop)
	{
		TNode<Type>* tmp2 = tmp1->next;
		while (tmp2 != stop)
		{
			if (reverse)
			{
				if (tmp1->value < tmp2->value)
				{
```

```
                        Type tmp = tmp1->value;
                        tmp1->value = tmp2->value;
                        tmp2->value = tmp;
                  }
                  tmp2 = tmp2->next;
            }
            else
            {
                  if (tmp1->value > tmp2->value)
                  {
                        Type tmp = tmp1->value;
                        tmp1->value = tmp2->value;
                        tmp2->value = tmp;
                  }
                  tmp2 = tmp2->next;
            }
      }
      tmp1 = tmp1->next;
   }
}


template<class Type>
void TList<Type>::clear()
{

   while (head != stop)
   {
      TNode<Type>* tmp;
      tmp = head;
      head = head->next;
      delete tmp;
   }
   curr = nullptr;
   last = nullptr;
   stop = nullptr;
}


template<class Type>
void TList<Type>::start()
{
   curr = head;
}


#endif //
```

Приложение В. Реализация класса TRingList

```
#ifndef _TRingList_H_
#define _TRingList_H_

#include "tlist.h"
#include <iostream>
using namespace std;


template <class Type>
class TRingList: public TList<Type>
{
private:
```

```
	TNode<Type>* fict_head;
public:

	TRingList();
	TRingList(const TRingList<Type>& list);
	TRingList(const TNode<Type>* node);
	virtual ~TRingList();
	void pop_first();
	void pop_last();
	void pop_curr();

	void next();
	void clear();
	void push_after_curr(const Type& value);
	void push_back(const Type& value);
	void push_front(const Type& value);
	bool operator==(const TRingList<Type>& list) const;
	friend ostream& operator<<(ostream& buf, TRingList<Type>& list)
	{
		TNode<Type>* tmp = list.fict_head->next;

		if (list.empty())
		{
			buf << "List is empty!\n";
			return buf;
		}

		while (tmp != list.fict_head)
		{
			buf << tmp->value << endl;
			tmp = tmp->next;
		}


		return buf;
	}
};

template <class Type>
TRingList<Type>::TRingList()
{
	fict_head = new TNode<Type>;
	last = fict_head;
	last->next = fict_head;
	stop = fict_head;
}


template <class Type>
TRingList<Type>::TRingList(const TRingList<Type>& list): TRingList<Type>()
{
	if (list.empty())
	{
		return;
	}
	TNode<Type>* tmp = list.head->next;
	head = new TNode<Type>(list.head->value);
	curr = head;
	last = head;

	while (tmp != list.stop)
	{
		last->next = new TNode<Type>(tmp->value);
```

```
            last = last->next;
            tmp = tmp->next;
      }
      last->next = fict_head;
}


template <class Type>
TRingList<Type>::TRingList(const TNode<Type>* node) : TRingList<Type>()
{
      if (node == nullptr)
      {
            return;
      }
      TNode<Type>* tmp = node->next;
      head = new TNode<Type>(node->value);
      curr = head;
      last = head;

      while (tmp != nullptr)
      {
            if (tmp->value == Type() && tmp->next == head)
            {
                  break;
            }
            last->next = new TNode<Type>(tmp->value);
            last = last->next;
            tmp = tmp->next;
      }
      last->next = fict_head;
}


template <class Type>
TRingList<Type>::~TRingList()
{
      if (fict_head != nullptr)
      {
            delete fict_head;
      }
      if (last != nullptr) last->next = nullptr;
      stop = nullptr;
}


template<class Type>
bool TRingList<Type>::operator==(const TRingList<Type>& list) const
{
      TNode<Type>* t = head, *t2 = list.head;
      if (t == nullptr || t2 == nullptr)
      {
            throw "Your polynoms must not be empty!\n";
      }
      while ((t != fict_head) && (t2 != list.fict_head))
      {
            if (t->value != t2->value)
            {
                  return false;
            }
            t = t->next;
            t2 = t2->next;
      }
      return (t == fict_head) && (t2 == list.fict_head);
```

```
}


template <class Type>
void TRingList<Type>::next()
{
	curr = curr->next;
}


template <class Type>
void TRingList<Type>::pop_first()
{
	if (head == nullptr)
	{
		string ex = "SizeError: can`t remove empty list";
		throw ex;
	}

	if (last == head)
	{
		*this = TRingList<Type>();
		return;
	}
	if (curr == head)
	{
		curr = head->next;
	}
	TNode<Type>* tmp = head->next;
	fict_head->next = tmp;
	delete head;
	head = tmp;
}


template <class Type>
void TRingList<Type>::pop_last()
{
	if (head == nullptr)
	{
		string ex = "SizeError: can`t remove empty list";
		throw ex;
	}

	if (last == head)
	{
		*this = TRingList<Type>();
		return;
	}

	TNode<Type>* tmp = head;
	while (tmp->next != last)
	{
		tmp = tmp->next;
	}

	if (curr == last)
	{
		curr = tmp;
	}

	delete tmp->next;
	tmp->next = fict_head;
```

```
	last = tmp;

}


template <class Type>
void TRingList<Type>::push_back(const Type& value)
{
	if (head == nullptr)
	{
		head = new TNode<Type>(value);
		last = head;
		last->next = fict_head;
		fict_head->next = head;
		curr = head;
		return;
	}

	TNode<Type>* tmp = new TNode<Type>(value);
	tmp->next = fict_head;
	last->next = tmp;
	last = last->next;
}


template <class Type>
void TRingList<Type>::push_front(const Type& value)
{
	if (head == nullptr)
	{
		head = new TNode<Type>(value);
		last = head;
		curr = head;
		last->next = fict_head;
		fict_head->next = head;
		return;
	}
	TNode<Type>* new_head = new TNode<Type>(value);
	new_head->next = head;
	head = new_head;
	fict_head->next = head;
}


template <class Type>
void TRingList<Type>::pop_curr()
{
	if (head == curr)
	{
		this->pop_first();
		return;
	}
	if (last == curr)
	{
		this->pop_last();
		return;
	}
	TNode<Type>* tmp1 = head, * tmp = curr;
	while (tmp1->next != curr) tmp1 = tmp1->next;
	curr = curr->next;
	tmp1->next = curr;
```

```
	delete tmp;

}


template <class Type>
void TRingList<Type>::push_after_curr(const Type& value)
{
	if (curr == last)
	{
		this->push_back(value);
		return;
	}

	TNode<Type>* tmp = curr->next;
	curr->next = new TNode<Type>(value);
	curr->next->next = tmp;


}


template<class Type>
void TRingList<Type>::clear()
{
	if (head == nullptr)
	{
		return;
	}
	while (head != fict_head)
	{
		TNode<Type>* tmp;
		tmp = head;
		head = head->next;
		delete tmp;
	}
	head = nullptr;
	curr = nullptr;
	last = fict_head;
	last->next = fict_head;
}


#endif //
```

Приложение Г. Реализация класса TMonom

```
#include "tmonom.h"
#include <algorithm>


TMonom::TMonom(const TMonom& monom)
{
	coef = monom.coef;
	degree_x = monom.degree_x;
	degree_y = monom.degree_y;
	degree_z = monom.degree_z;
}


TMonom::TMonom(const double coef_, const int degreex_, const int degreey_, const
int degreez_)
{
```

```
	coef = coef_;
	degree_x = degreex_;
	degree_y = degreey_;
	degree_z = degreez_;
}


bool TMonom::operator<(const TMonom& monom) const {
	return ((degree_x < monom.degree_x) || (degree_x == monom.degree_x &&
degree_y < monom.degree_y) ||
		(degree_x == monom.degree_x && degree_y == monom.degree_y &&
degree_z < monom.degree_z));
}


bool TMonom::operator>(const TMonom& monom) const {
	return ((degree_x > monom.degree_x) || (degree_x == monom.degree_x &&
degree_y > monom.degree_y) ||
		(degree_x == monom.degree_x && degree_y == monom.degree_y &&
degree_z > monom.degree_z));
}


bool TMonom::operator==(const TMonom& monom) const {
	return degree_x == monom.degree_x && degree_y == monom.degree_y &&
degree_z == monom.degree_z;
}


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


TMonom TMonom::operator*(const TMonom& monom) const
{
	return TMonom(monom.coef * this->coef, monom.degree_x + this->degree_x,
		this->degree_y + monom.degree_y, this->degree_z + monom.degree_z);
}


double TMonom::eval(double x, double y, double z)
{
	return coef * std::pow(x, degree_x) * std::pow(y, degree_y) * std::pow(z,
degree_z);
}


double TMonom::get_coef() const
{
	return coef;
}


void TMonom::set_coef(double digit)
{
	coef = digit;
}


void TMonom::inc_coef(double digit)
{
	coef += digit;
```

```
}


TMonom TMonom::dif_x() const
{
	if (*this == TMonom())
	{
		return TMonom();
	}
	if (degree_x == 0)
	{
		return TMonom(0, 0, 0, 0);
	}
	return TMonom(coef * degree_x, degree_x - 1, degree_y, degree_z);
}


TMonom TMonom::dif_y() const
{
	if (*this == TMonom())
	{
		return TMonom();
	}
	if (degree_y == 0)
	{
		return TMonom(0, 0, 0, 0);
	}
	return TMonom(coef * degree_y, degree_x, degree_y - 1, degree_z);
}


TMonom TMonom::dif_z() const
{
	if (*this == TMonom())
	{
		return TMonom();
	}
	if (degree_z == 0)
	{
		return TMonom(0, 0, 0, 0);
	}
	return TMonom(coef * degree_z, degree_x, degree_y, degree_z - 1);
}
```

Приложение Д. Реализация класса TPolynom

```
#include "tpolynom.h"
#include <iostream>
#include <sstream>
using namespace std;

void TPolynom::del_zeros()
{
	int was = 0;
	monoms.start();
	TNode<TMonom>* t = monoms.get_curr();
	if (t == nullptr)
	{
		return;
	}
```

```
	while (t->value != TMonom())
	{
		if (t->value.get_coef() != 0)
		{
			was = 1;
			break;
		}
		t = t->next;
	}
	t = monoms.get_curr();
	if (was == 1)
	{
		while (t->value != TMonom())
		{
			if (t->value.get_coef() == 0)
			{
				TNode<TMonom>* t1 = t;
				t = t->next;
				monoms.remove(t1);
			}
			t = t->next;
		}
		return;
	}
	// if all of elements equal 0 we create new list with only 1 "0"
	monoms.clear();
	monoms.push_front(TMonom(0, 0, 0, 0));
	return;


}
string TPolynom::preparation(string polynom)
{
	string new_string, new_string1;

	int i = 0;
	while (i < polynom.size())
	{
		if (polynom[i] == ' ')
		{
			++i;
			continue;
		}
		new_string += polynom[i];
		++i;
	}
	i = 0;
	while (i < new_string.size())
	{
		if (new_string[i] == '+' || new_string[i] == '-')
		{
			new_string1 += " ";
			new_string1 += new_string[i];
			new_string1 += " ";
			++i;
			continue;
		}
		new_string1 += new_string[i];
		++i;
	}
	return new_string1;
}
```

```
void TPolynom::x(string token, char& stage, int& i, double& coef, int& degreex,
int& degreey, int& degreez)
{
	// we here --> token[i] == 'x'
	string digit;
	stage = 'x';
	i++;
	if (token[i] == '^')
	{
		i++;
		if (i >= token.size() || token[i] < '0' || token[i] > '9')
		{
			// string like 2x^yz
			throw "Wrong string\n";
		}
	}
	while (i < token.size() && ((token[i] >= '0' && token[i] <= '9') ||
token[i] == '.'))
	{
		digit += token[i];
		i++;
	}

	if (digit != "")
	{
		degreex += stod(digit);
	}
	else
	{
		degreex = 1;
	}
	if (i >= token.size())
	{
		return;
	}
	if (token[i] == 'x')
	{
		x(token, stage, i, coef,degreex, degreey, degreez);
	}
	else if (token[i] == 'y')
	{
		y(token, stage, i, coef, degreex, degreey, degreez);
	}
	else if (token[i] == 'z')
	{
		z(token, stage, i, coef, degreex, degreey, degreez);
	}
	else
	{
		throw "Wrong string";
	}

}


void TPolynom::y(string token, char& stage, int& i, double& coef, int& degreex,
int& degreey, int& degreez)
{
	string digit;
	stage = 'y';
	i++;
	if (token[i] == '^')
```

```
	{
		i++;
		if (i >= token.size() || token[i] < '0' || token[i] > '9')
		{
			// string like 2x^yz
			throw "Wrong string\n";
		}
	}
	while (i < token.size() && ((token[i] >= '0' && token[i] <= '9') ||
token[i] == '.'))
	{
		digit += token[i];
		i++;
	}

	if (digit != "")
	{
		degreey += stod(digit);
	}
	else
	{
		degreey = 1;
	}

	if (i >= token.size())
	{
		return;
	}
	if (token[i] == 'x')
	{
		x(token, stage, i, coef, degreex, degreey, degreez);
	}
	else if (token[i] == 'y')
	{
		y(token, stage, i, coef, degreex, degreey, degreez);
	}
	else if (token[i] == 'z')
	{
		z(token, stage, i, coef, degreex, degreey, degreez);
	}
	else
	{
		throw "Wrong string";
	}
}


void TPolynom::z(string token, char& stage, int& i, double& coef, int& degreex,
int& degreey, int& degreez)
{
	string digit;
	stage = 'z';
	i++;
	if (token[i] == '^')
	{
		i++;
		if (i >= token.size() || token[i] < '0' || token[i] > '9')
		{
			// string like 2x^yz
			throw "Wrong string\n";
		}
	}
```

```
	while (i < token.size() && ((token[i] >= '0' && token[i] <= '9') ||
token[i] == '.'))
	{
		digit += token[i];
		i++;
	}

	if (digit != "")
	{
		degreez += stod(digit);
	}
	else
	{
		degreez = 1;
	}

	if (i >= token.size())
	{
		return;
	}
	if (token[i] == 'x')
	{
		x(token, stage, i, coef, degreex, degreey, degreez);
	}
	else if (token[i] == 'y')
	{
		y(token, stage, i, coef, degreex, degreey, degreez);
	}
	else if (token[i] == 'z')
	{
		z(token, stage, i, coef, degreex, degreey, degreez);
	}
	else
	{
		throw "Wrong string";
	}
}


void TPolynom::c(string token, char& stage, int& i, double& coef, int& degreex,
int& degreey, int& degreez)
{
	string digit;
	stage = 'c';
	while (i < token.size() && ((token[i] >= '0' && token[i] <= '9') ||
token[i] == '.'))
	{
		digit += token[i];
		i++;
	}

	if (digit != "")
	{
		coef *= stod(digit);
	}
	if (i >= token.size())
	{
		return;
	}
	if (token[i] == 'x')
	{
		x(token, stage, i, coef, degreex, degreey, degreez);
```

```
	}
	else if (token[i] == 'y')
	{
		y(token, stage, i, coef, degreex, degreey, degreez);
	}
	else if (token[i] == 'z')
	{
		z(token, stage, i, coef, degreex, degreey, degreez);
	}
	else
	{
		throw "Wrong string";
	}

}


void TPolynom::parse(string polynom)
{
	polynom = preparation(polynom); // пробелы добавить при +-


	string token;
	double curr_coef = 1;
	int degreex = 0, degreey = 0, degreez = 0;
	char stage = 's';
	if (polynom.size() >= 3)
	{
		string t;
		t += (char)polynom[0];
		t += (char)polynom[1];
		t += (char)polynom[2];
		if (t == " - ")
		{
			curr_coef = -1;
			string tmp;
			for (int i = 3; i < polynom.size(); ++i) tmp += polynom[i];
			polynom = tmp;
		}

		if (t == " + ")
		{
			string tmp;
			for (int i = 3; i < polynom.size(); ++i) tmp += polynom[i];
			polynom = tmp;
		}
	}

	stringstream stream(polynom);
	while (stream >> token)
	{

		if (token == "-" || token == "+")
		{

			TMonom tmp(curr_coef, degreex, degreey, degreez);
			monoms.push_back(tmp);
			curr_coef = 1;
			degreex = 0; degreey = 0; degreez = 0;
			stage = 's';
			if (token == "-")
			{
```

```
                        curr_coef = -1;
                }
                continue;

        }
        int i = 0;
        while (i < token.size())
        {
                if (token[i] == 'x' || token[i] == 'y' || token[i] == 'z')
                {
                        if (token[i] == 'x')
                        {
                                x(token, stage, i, curr_coef, degreex, degreey,
degreez);
                        }
                        else if(token[i] == 'y')
                        {
                                y(token, stage, i, curr_coef, degreex, degreey,
degreez);
                        }
                        else
                        {
                                z(token, stage, i, curr_coef, degreex, degreey,
degreez);
                        }
                }
                else if (token[i] >= '0' && token[i] <= '9')
                {
                        c(token,  stage,  i,  curr_coef,  degreex,  degreey,
degreez);
                }
                else
                {
                        throw "Wrong string";
                }
        }

    }
    if (stage != 's')
    {
        TMonom tmp(curr_coef, degreex, degreey, degreez);
        monoms.push_back(tmp);
    }
    monoms.sort();
}


TPolynom::TPolynom()
{
    monoms = TRingList<TMonom>();
}

TPolynom::TPolynom(const string& polynom_string)
{
    monoms = TRingList<TMonom>();
    parse(polynom_string);
    this->del_zeros(); this->monoms.sort(); this->monoms.start();
}

TPolynom::TPolynom(const TRingList<TMonom>& list)
{
    monoms = TRingList<TMonom>(list);
    this->del_zeros(); this->monoms.sort(); this->monoms.start();
```

```
}

TPolynom::TPolynom(const TPolynom& polynom) : monoms(polynom.monoms)
{
}


const TPolynom& TPolynom::operator=(const TPolynom& polynom) {

	(*this).monoms.clear();
	(*this).monoms = TRingList<TMonom>(polynom.monoms);
	return *this;
}

TPolynom TPolynom::operator+(const TPolynom& polynom) {

	TPolynom sum(*this);
	sum.monoms.start();
	TRingList<TMonom> tmp(polynom.monoms);
	tmp.start();

	TNode<TMonom>* tmp1 = tmp.get_curr();

	if (tmp1 == nullptr)
	{
		return *this;
	}

	while (tmp1->value != TMonom())
	{
		TNode<TMonom>* t = sum.monoms.find(tmp1->value);

		if (t == nullptr)
		{
			sum.monoms.push_back(tmp1->value);
		}
		else
		{
			t->value.inc_coef(tmp1->value.get_coef());
		}
		tmp1 = tmp1->next;
	}
	sum.del_zeros(); sum.monoms.sort();
	return sum;
}
TPolynom TPolynom::operator-(const TPolynom& polynom) {

	TPolynom sum(*this);
	sum.monoms.start();
	TRingList<TMonom> tmp(polynom.monoms);
	tmp.start();

	TNode<TMonom>* tmp1 = tmp.get_curr();

	if (tmp1 == nullptr)
	{
		return *this;
	}

	while (tmp1->value != TMonom())
	{
		TNode<TMonom>* t = sum.monoms.find(tmp1->value);
		tmp1->value.set_coef(-(tmp1->value.get_coef()));
```

```
            if (t == nullptr)
            {
                  sum.monoms.push_back(tmp1->value);
            }
            else
            {
                  t->value.inc_coef(tmp1->value.get_coef());
            }
            tmp1 = tmp1->next;
      }
      sum.del_zeros(); sum.monoms.sort();
      return sum;

}
TPolynom TPolynom::operator*(const TPolynom& polynom) {


      TPolynom prod;

      TRingList<TMonom> l1(this->monoms), l2(polynom.monoms);
      l1.start();
      l2.start();
      TNode<TMonom>* t1 = l1.get_curr(), * t2 = l2.get_curr();
      if (t1 == nullptr || t2 == nullptr)
      {
            throw "Cant multiply empty polynom";
      }

      while (t1->value != TMonom())
      {
            t2 = l2.get_curr();
            while (t2->value != TMonom())
            {
                  // find list1[i] * list2[j] in prod
                  // if it is in list, we need sum coeff
                  // else we need add new monom
                  TMonom monom = t1->value * t2->value;
                  TNode<TMonom>* t = prod.monoms.find(monom);
                  if (t == nullptr)
                  {
                        prod.monoms.push_back(monom);
                  }
                  else
                  {
                        t->value.inc_coef(monom.get_coef());
                  }

                  t2 = t2->next;
            }

            t1 = t1->next;
      }
      prod.del_zeros(); prod.monoms.sort();
      return prod;

}
double TPolynom::operator()(double x, double y, double z) {

      double res = 0;
      monoms.start();
      TNode<TMonom>* t = monoms.get_curr();
      while (t->value != TMonom())
      {
```

```
            res += t->value.eval(x, y, z);
            t = t->next;
      }
      return res;
}
bool TPolynom::operator==(const TPolynom& polynom) const
{
      return monoms == polynom.monoms;
}

TPolynom TPolynom::dif_x() const {

      TPolynom polynom(*this);
      polynom.monoms.start();
      TPolynom res;
      TNode<TMonom>* t = polynom.monoms.get_curr();
      if (t == nullptr)
      {
            throw "Your polynom is empty!";
      }
      while(t->value != TMonom())
      {
            TMonom diff_x = t->value.dif_x();
            TNode<TMonom>* tmp = res.monoms.find(diff_x);
            if (tmp == nullptr)
            {
                  res.monoms.push_back(diff_x);
            }
            else
            {

                  tmp->value.inc_coef(diff_x.get_coef());

            }
            t = t->next;
      }
      res.del_zeros(); res.monoms.sort();
      return res;
}
TPolynom TPolynom::dif_y() const {

      TPolynom polynom(*this);
      polynom.monoms.start();
      TPolynom res;
      TNode<TMonom>* t = polynom.monoms.get_curr();
      if (t == nullptr)
      {
            throw "Your polynom is empty!";
      }
      while(t->value != TMonom())
      {
            TMonom diff_y = t->value.dif_y();
            TNode<TMonom>* tmp = res.monoms.find(diff_y);
            if (tmp == nullptr)
            {
                  res.monoms.push_back(diff_y);
            }
            else
            {
                  tmp->value.inc_coef(diff_y.get_coef());
            }
            t = t->next;
      }
```

```
	res.del_zeros(); res.monoms.sort();
	return res;
}
TPolynom TPolynom::dif_z() const {

	TPolynom polynom(*this);
	polynom.monoms.start();
	TPolynom res;
	TNode<TMonom>* t = polynom.monoms.get_curr();
	if (t == nullptr)
	{
		throw "Your polynom is empty!";
	}

	while(t->value != TMonom())
	{
		TMonom diff_z = t->value.dif_z();

		TNode<TMonom>* tmp = res.monoms.find(diff_z);
		if (tmp == nullptr)
		{
			res.monoms.push_back(diff_z);
		}
		else
		{
			tmp->value.inc_coef(diff_z.get_coef());

		}
		t = t->next;
	}
	res.del_zeros(); res.monoms.sort();
	return res;
}
```