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:

«АНАЛИТИЧЕСКИЕ ПРЕОБРАЗОВАНИЯ ПОЛИНОМОВ ОТ НЕСКОЛЬКИХ ПЕРЕМЕННЫХ (СПИСКИ)»

Выполнил: студент группы
3822Б1ФИ1
_____ / Созонов И.С. /
Подпись

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

Нижний Новгород
2024

Содержание

Введение

Наряду с привычным вычислительным применением компьютеры широко используются и для аналитической обработки данных. Среди примеров таких приложений – компьютерное доказательство теорем, логический вывод, анализ текстовой информации и многое другое. Среди таких примеров и задача обработки полиномов, задаваемых в общей аналитической форме. Полиномы являются хорошо изученной областью математики (алгебра полиномов), которая широко используется в приложениях (аппроксимация экспериментальных данных, построение функциональных зависимостей и т.п.).

Лабораторная работа направлена на изучение методов компьютерной обработки полиномов. С этой целью в лабораторной работе изучаются различные варианты структуры хранения и разрабатываются программы для обработки полиномов. Основной учебной целью работы является практическое освоение методов организации структур хранения данных с помощью списков. В ходе выполнения лабораторной работы разрабатывается общая форма представления линейных списков, разрабатываются программы работы со списками, которые могут быть использованы и в других областях приложений.

1 Постановка задачи

Цель – реализовать классы для представления линейных односвязных списков и обработки полиномов.

Задачи:

1. Разработать шаблонную структуру TNode для работы с узлами списка.
2. На основании структуры узла реализовать шаблонный класс TList для работы с линейными односвязными списками, который должен поддерживать следующие основные операции: поиск узла с заданным значением, вставка узла с заданным значением в начало списка, вставка узла с заданным значением в конец списка, вставка узла с заданным значением до определенного узла списка, вставка узла с заданным значением после определенного узла списка, удаление узла списка с заданным значением, очистка списка. Также следует добавить специфические операции: проверка на полноту списка, проверка на пустоту списка, проверка, достигли ли конца списка. Навигационные методы: получение текущего узла списка, переход к следующему узлу списка, переход в начало списка.
3. Дополнительно создать шаблонный класс THeadRingList (наследник класса TList) для работы с односвязными кольцевыми списками.
4. Разработать класс TMonom (параметр шаблонного класса THeadRingLis) для работы с мономами полинома. Перегрузить операторы сравнения мономов.
5. Разработать класс TPolynom на основе линейного односвязного кольцевого списка мономов. Написать следующие операции для работы с полиномами: сложение полиномов, вычитание полиномов, умножение полиномов, вычисление значения полинома в заданной точке, дифференцирование полинома по переменной «x», дифференцирование полинома по переменной «y», дифференцирование полинома по переменной «z».

2 Руководство пользователя

2.1 Приложение для демонстрации работы списков

1. Запустить sample_tlist.exe. В результате появится окно для ввода количества элементов, которые необходимо поместить в список (рис. 1).

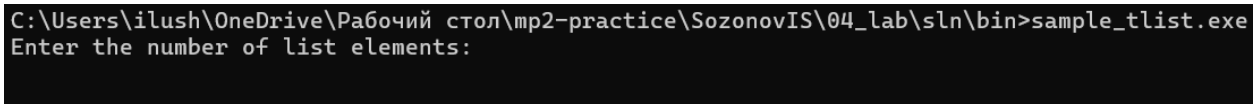


Рис. 1. Основное окно приложения

2. Ввести количество элементов. В результате появится окно для ввода элементов, которые необходимо поместить в список (Рис. 2).

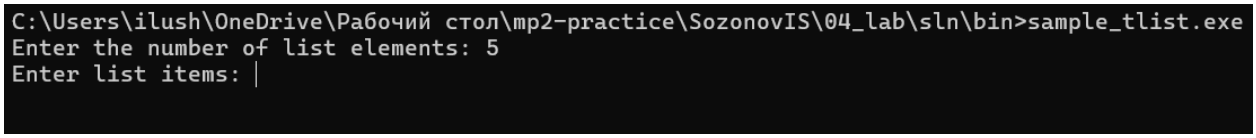


Рис. 2. Ввод количества элементов

3. Ввести элементы. В результате будет выполнена проверка списка на пустоту и появится окно для ввода элемента, который нужно вставить в начало списка (Рис. 3).

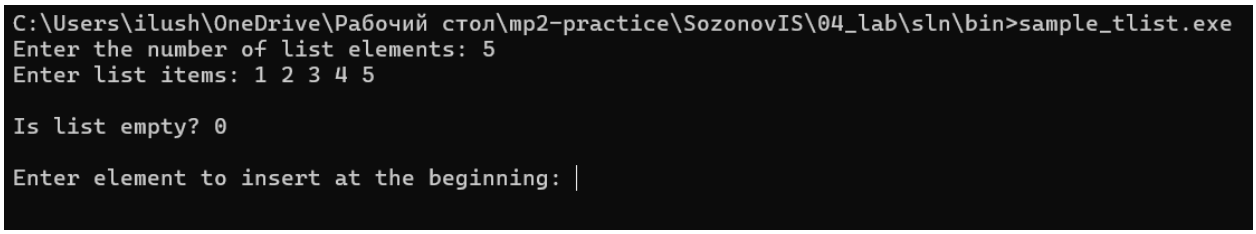


Рис. 3. Ввод элементов списка

4. Ввести элемент. В результате будет выведен список со вставленным элементом и появится окно для ввода элемента, который нужно вставить в конец списка (Рис. 4).

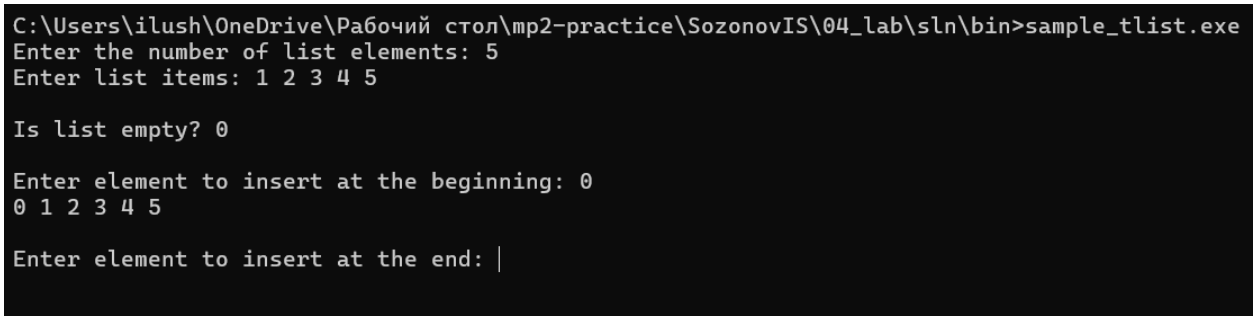


Рис. 4. Ввод элемента, который нужно вставить в начало списка

5. Ввести элемент. В результате будет выведен список со вставленным элементом и появится окно для ввода элемента, перед которым нужно вставить элемент в список (Рис. 5).

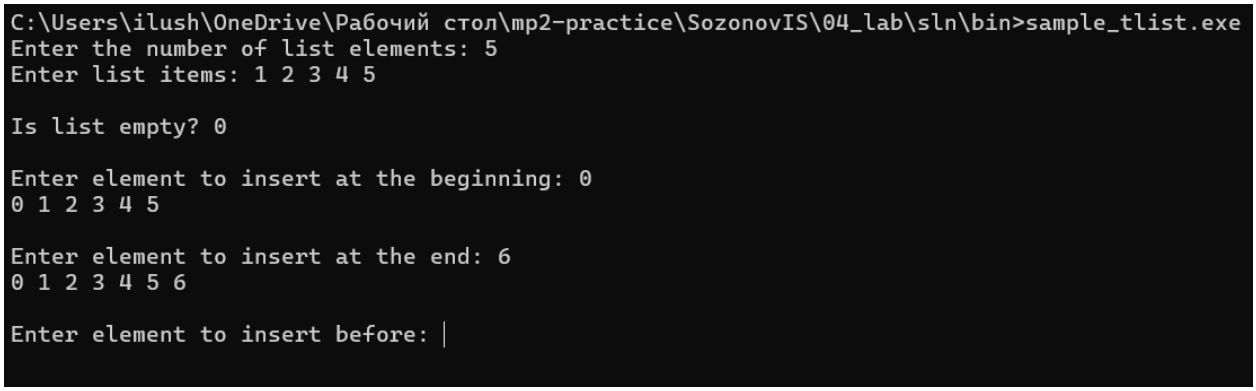


Рис. 5. Ввод элемента, который нужно вставить в конец списка

6. Ввести элемент. В результате появится окно для ввода элемента, который нужно вставить в список (Рис. 6).

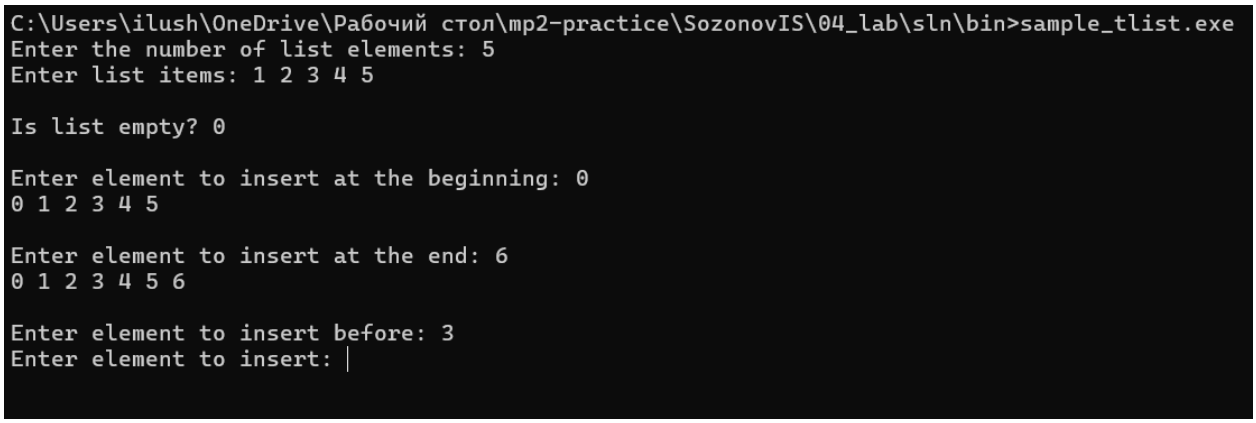


Рис. 6. Ввод элемента, перед которым нужно вставить элемент в список

7. Ввести элемент. В результате будет выведен список со вставленным элементом и появится окно для ввода элемента, после которого нужно вставить элемент в список (Рис. 7).

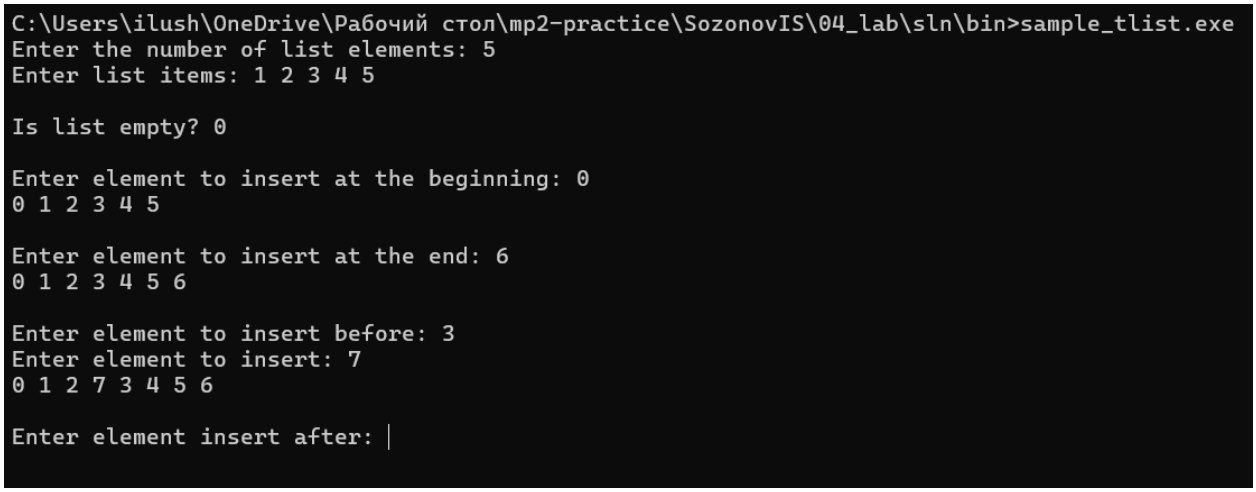


Рис. 7. Ввод элемента, который нужно вставить в список

8. Ввести элемент. В результате появится окно для ввода элемента, который нужно вставить в список (Рис. 8).

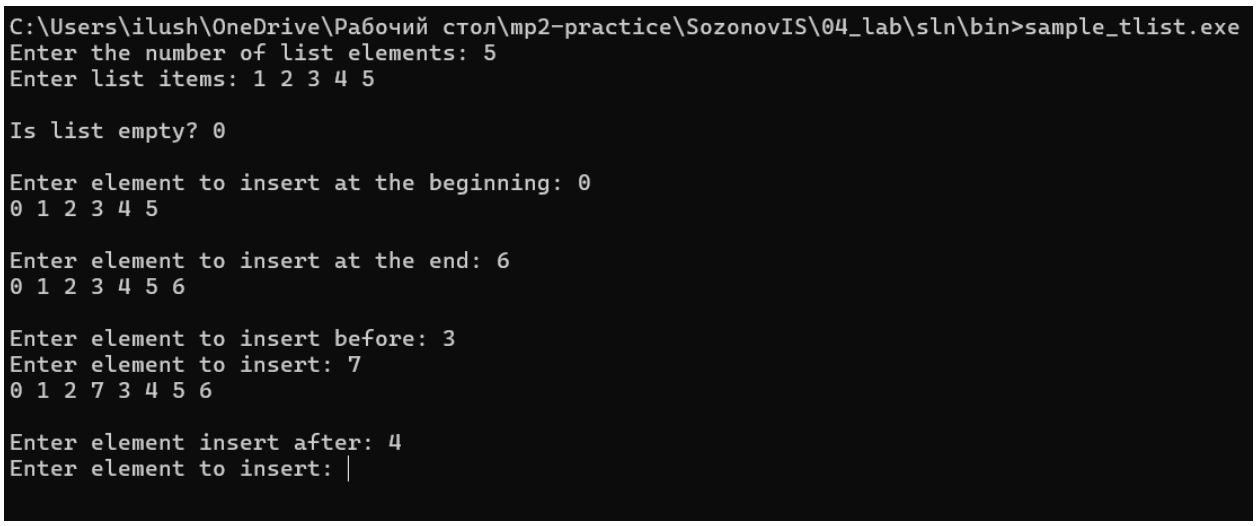


Рис. 8. Ввод элемента, после которого нужно вставить элемент в список

9. Ввести элемент. В результате будет выведен список со вставленным элементом и появится окно для ввода элемента, который нужно удалить из списка (Рис. 9).

```
C:\Users\ilush\OneDrive\Рабочий стол\mp2-practice\SozonovIS\04_lab\sln\bin>sample_tlist.exe
Enter the number of list elements: 5
Enter list items: 1 2 3 4 5

Is list empty? 0

Enter element to insert at the beginning: 0
0 1 2 3 4 5

Enter element to insert at the end: 6
0 1 2 3 4 5 6

Enter element to insert before: 3
Enter element to insert: 7
0 1 2 7 3 4 5 6

Enter element insert after: 4
Enter element to insert: 11
0 1 2 7 3 4 11 5 6

Enter element to remove:
```

Рис. 9. Ввод элемента, который нужно вставить в список

10. Ввести элемент. В результате будет выведен список с удаленным элементом (Рис. 10).

```
C:\Users\ilush\OneDrive\Рабочий стол\mp2-practice\SozonovIS\04_lab\sln\bin>sample_tlist.exe
Enter the number of list elements: 5
Enter list items: 1 2 3 4 5

Is list empty? 0

Enter element to insert at the beginning: 0
0 1 2 3 4 5

Enter element to insert at the end: 6
0 1 2 3 4 5 6

Enter element to insert before: 3
Enter element to insert: 7
0 1 2 7 3 4 5 6

Enter element insert after: 4
Enter element to insert: 11
0 1 2 7 3 4 11 5 6

Enter element to remove: 2
0 1 7 3 4 11 5 6

C:\Users\ilush\OneDrive\Рабочий стол\mp2-practice\SozonovIS\04_lab\sln\bin>
```

Рис. 10. Ввод элемента, который нужно удалить из списка

2.2 Приложение для демонстрации работы полиномов

1. Запустить sample_tpolynom.exe. В результате появится окно для ввода первого полинома (Рис. 11).

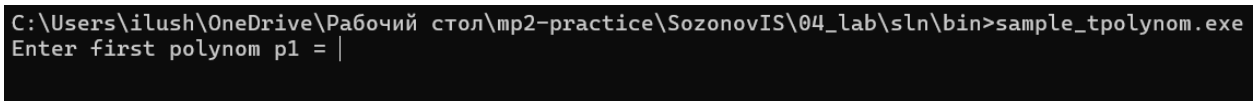


Рис. 11. Основное окно приложения

2. Ввести первый полином. В результате появится окно для ввода второго полинома (Рис. 12).

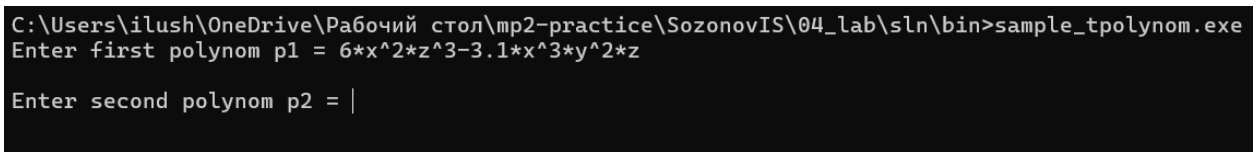


Рис. 12. Ввод первого полинома

3. Ввести второй полином. В результате появится окно для ввода значения переменной x (Рис. 13).

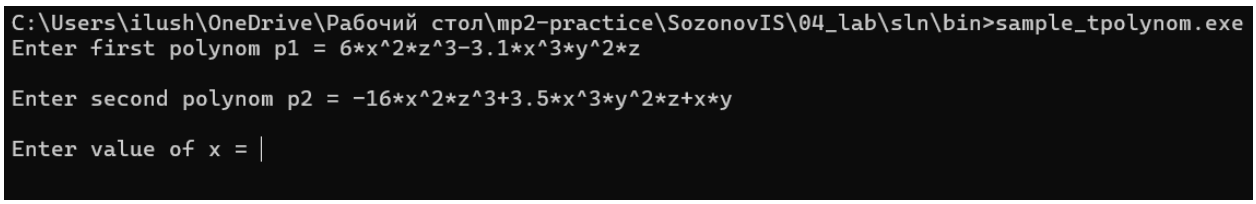


Рис. 13. Ввод второго полинома

4. Ввести значение переменной x. В результате появится окно для ввода значения переменной y (Рис. 14).

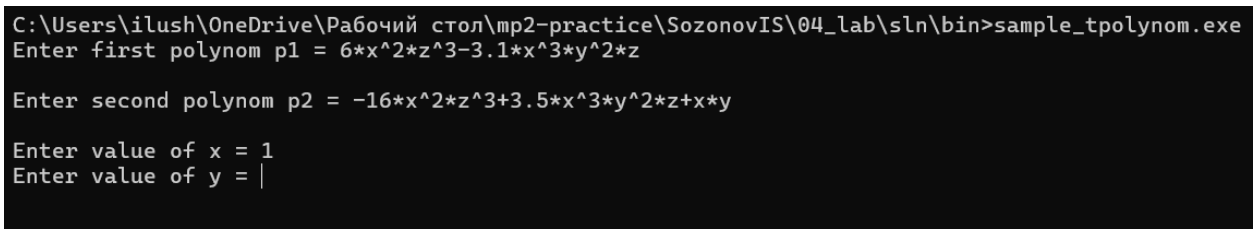


Рис. 14. Ввод значения переменной x

5. Ввести значение переменной y. В результате появится окно для ввода значения переменной z (Рис. 15).

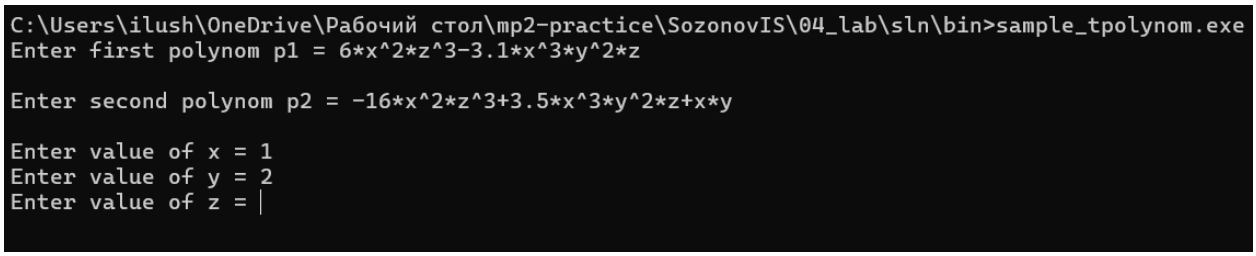


Рис. 15. Ввод значения переменной y

1. Ввести значение переменной z. В результате будут выведены сумма, разность и произведение полиномов, а также производные первого полинома по переменным x, y, z соответственно (Рис. 16).

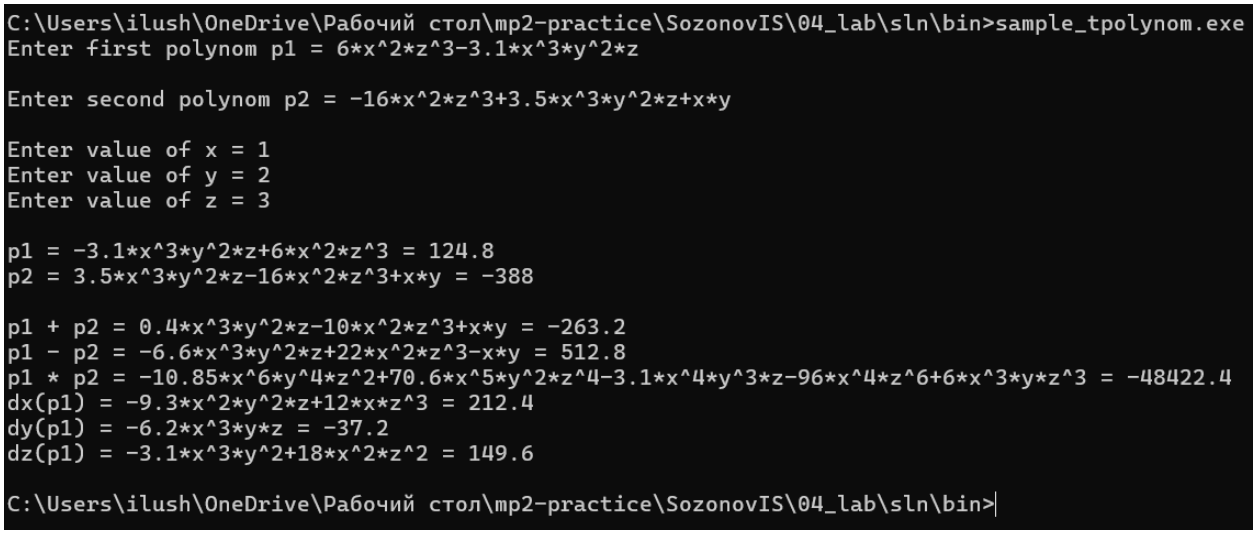


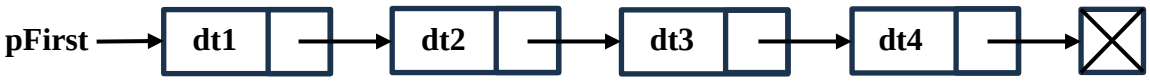
Рис. 16. Ввод значения переменной z

3 Руководство программиста

3.1 Использованные алгоритмы

3.1.1 Линейные односвязные списки

Односвязный (линейный) список – структура хранения, состоящая из узлов, содержащих данные и ссылку на следующий узел списка**.**



Узел списка называется **звеном**.

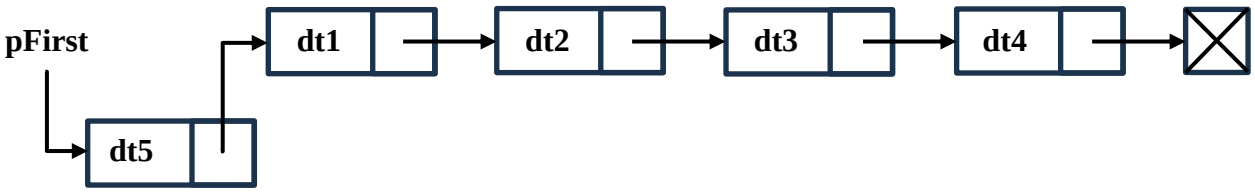
Для работы со списком предлагается реализовать следующие операции:

- **Поиск узла с заданным значением**

Для поиска узла по заданному значению в линейном односвязном списке можно использовать следующий алгоритм:

1) Начинаем с первого узла списка.
2) Пока не достигнем конца списка или не найдем узел с заданным значением, перемещаемся по списку, просматривая каждый узел.
3) Если нашли узел с заданным значением, возвращаем этот узел.
4) Если дошли до конца списка и не нашли узел с заданным значением, возвращаем nullptr.

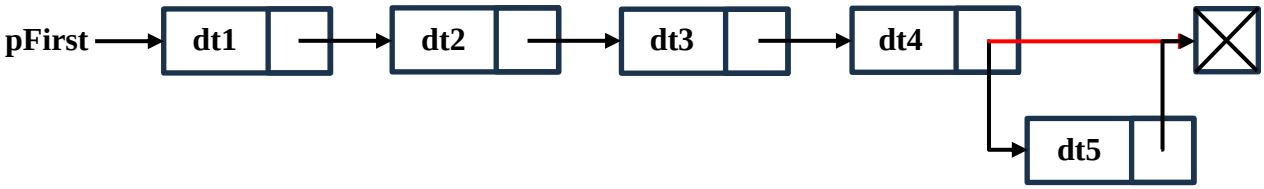
- **Вставка узла с заданным значением в начало списка**



Для вставки элемента в начало линейного односвязного списка можно использовать следующий алгоритм:

1) Создать новый узел с заданным значением.
2) Присвоить указателю нового узла указатель на текущий первый элемент списка.
3) Обновить указатель на первый элемент списка так, чтобы он указывал на новый узел.

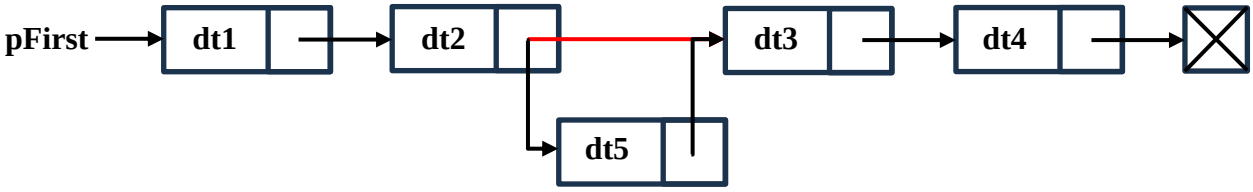
- **Вставка узла с заданным значением в конец списка**



Для вставки элемента в конец линейного односвязного списка нужно выполнить следующие шаги:

1) Создать новый узел с заданным значением.
2) Пройти по всем элементам списка, начиная с головного элемента, до тех пор, пока не найдем последний элемент.
3) Установить указатель последнего элемента на новый узел.
4) Установить указатель нового узла на `NULL`, чтобы он стал последним элементом списка.

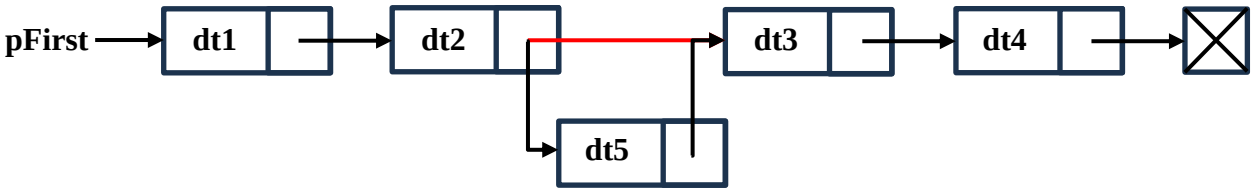
- **Вставка узла с заданным значением до определенного узла списка**



Для вставки элемента до определенного элемента в линейном односвязном списке, нужно выполнить следующие шаги:

1) Найти указатель на элемент, до которого нужно вставить новый элемент.
2) Создать новый узел с данными, которые нужно добавить в список.
3) Установить указатель предыдущего элемента на новый узел.
4) Установить указатель нового узла на найденный элемент.

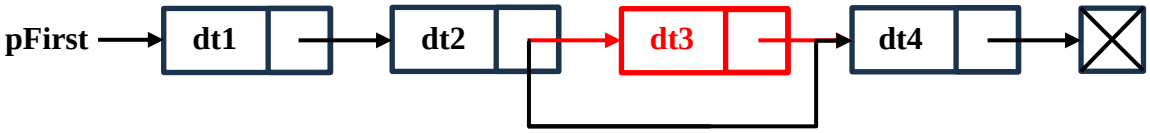
- **Вставка узла с заданным значением после определенного узла списка**



Для вставки нового элемента после определенного элемента в линейном односвязном списке необходимо выполнить следующие шаги:

5) Найти указатель на элемент, после которого нужно вставить новый элемент.
6) Создать новый узел с данными, которые нужно добавить в список.
7) Установить указатель нового узла на следующий элемент после найденного.
8) Установить указатель найденного элемента на новый узел.

- **Удаление узла списка с заданным значением из списка**



Для удаления узла из линейного односвязного списка необходимо выполнить следующие шаги:

1) Найти узел, который необходимо удалить.
2) Изменить указатель предыдущего узла так, чтобы он указывал на следующий узел после удаляемого узла.
3) Освободить память, занимаемую удаляемым узлом.

- **Очистка списка**

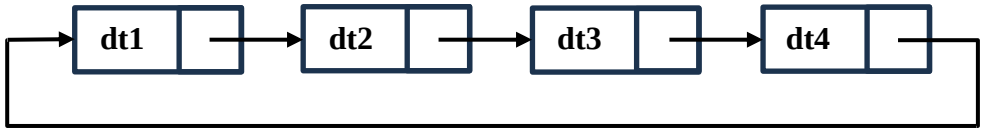
Для очистки линейного односвязного списка необходимо пройти по всем узлам списка и освободить память, занимаемую каждым узлом. Это можно сделать следующим образом:

1) Начать с первого узла списка.
2) Пока не достигнут конец списка:
- Сохранить ссылку на следующий узел.
- Освободить память, занимаемую текущим узлом.
- Перейти к следующему узлу (используя сохраненную ссылку).
3) После того как все узлы списка будут удалены, необходимо обнулить указатель на начало списка.

Этот алгоритм обеспечивает освобождение памяти, занимаемой всеми узлами списка, и предотвращает утечку памяти.

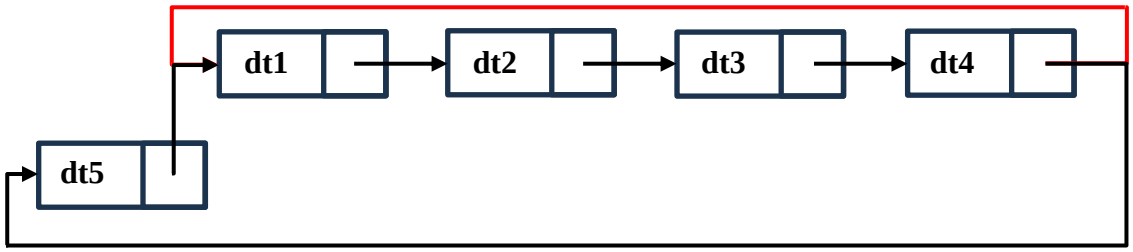
3.1.2 Односвязные кольцевые списки с головным узлом

Односвязный кольцевой список – это разновидность линейного односвязного списка, когда первый узел указывает на последний, а последний – на первый.



Для работы с кольцевым списком необходимо переопределить следующие операции линейного односвязного списка:

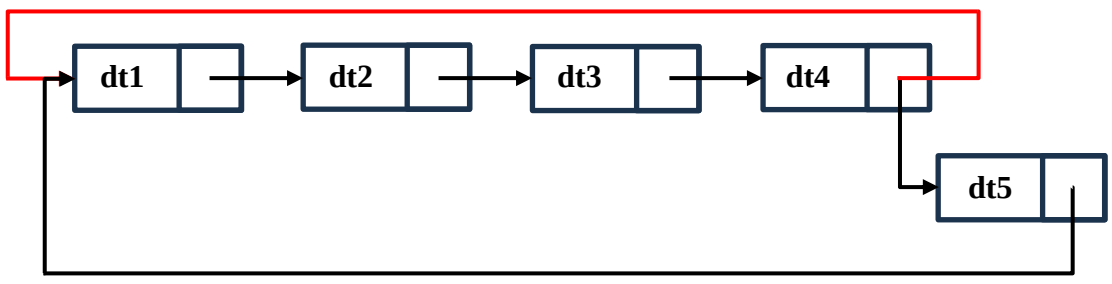
- **Вставка узла с заданным значением в начало кольцевого списка**



Для вставки элемента в начало кольцевого списка можно использовать следующий алгоритм:

1) Создать новый узел с данными, которые нужно добавить в список.
2) Если список пустой, то установить указатель на начало списка на только что созданный узел, а затем сделать его указатель на следующий узел равным самому себе (таким образом формируя кольцевую структуру).
3) Если список не пустой, то найти последний элемент списка (это можно сделать, например, с помощью цикла, проходя по каждому элементу и проверяя, равен ли указатель на следующий элемент самому первому элементу) и установить его указатель на следующий элемент равным только что созданному узлу.
4) Установить указатель на начало списка на только что созданный узел.

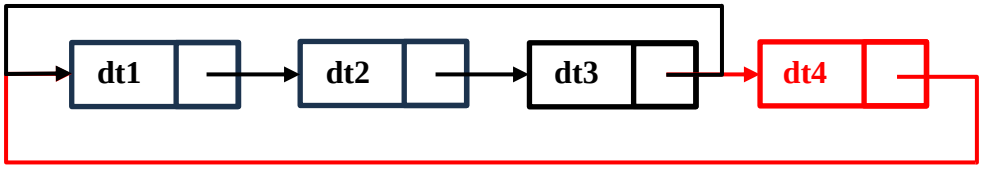
- **Вставка узла с заданным значением в конец кольцевого списка**



Для вставки элемента в конец кольцевого односвязного списка необходимо выполнить следующие шаги:

1) Создать новый узел, содержащий вставляемый элемент.
2) Если список пустой, то новый узел будет и сам список. Установить ссылку нового узла на самого себя.
3) В противном случае:
4) Найти последний узел списка (это можно сделать, перебирая узлы и проверяя, что ссылка на следующий узел не равна начальному узлу списка).
5) Установить ссылку последнего узла на новый узел.
6) Установить ссылку нового узла на начальный узел списка.

- **Удаление узла списка с заданным значением из кольцевого списка**



Для удаления элемента из линейного кольцевого односвязного списка нужно выполнить следующие шаги:

1) Найти элемент, который нужно удалить. Обычно это делается путем обхода списка с помощью указателей и сравнения значений элементов.
2) Изменить указатели таким образом, чтобы элемент был удален из списка. Для этого необходимо изменить указатель предыдущего элемента на указатель следующего элемента.
3) Освободить память, занимаемую удаленным элементом.

- **Очистка списка**

Для очистки линейного кольцевого односвязного списка можно использовать следующий алгоритм:

1) Создать два указателя (указатель текущего элемента и указатель предыдущего элемента).
2) Начать обход списка, начиная с головного элемента.
3) Для каждого элемента списка проверить, не является ли он последним элементом списка (т.е. указывает ли он на головной элемент).

4) Если текущий элемент является последним элементом списка, то удалить его, установив указатель предыдущего элемента на NULL.
5) Если текущий элемент не является последним элементом списка, то удалить его, изменив указатель предыдущего элемента так, чтобы он указывал на следующий за текущим элементом.
6) Перейти к следующему элементу списка и повторить шаги 3-5 до тех пор, пока не будет удален последний элемент списка.
7) Очистить память, выделенную для элементов списка.

Таким образом, данный алгоритм позволяет эффективно очистить кольцевой односвязный список, освободив выделенную для него память и предотвратив утечки памяти.

3.1.3 Полиномы

Под **полиномом** от одной переменной понимается выражение вида:

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

или в более общем виде: $P_n(x) = \sum_{i=0}^{n} a_i x^i$, где n – степень полинома; a_i, $0 \leq i \leq n$ – коэффициенты полинома (действительные или комплексные числа).

Полином можно определить также как выражение из нескольких **термов**, соединенных знаками сложения или вычитания. Терм включает коэффициент и **моном**, содержащий одну или несколько переменных, каждая из которых может иметь **степень**

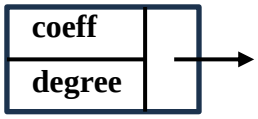
$$P(x, y, z) = \sum_{i,j,k} a_{ijk} x^i y^j z^k.$$

Как пример, полином от трех переменных может иметь вид

$$P(X, Y, Z) = 3x^3 z - 2y^2 z^2 + 3$$

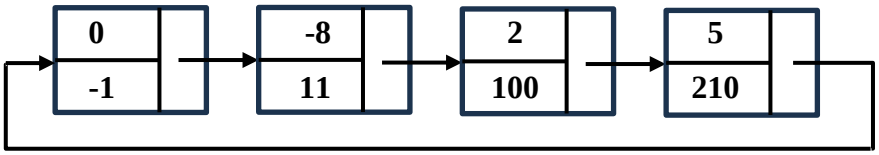
Подобными называют два (или более) мономов, имеющих одинаковые степени при неизвестных.

С точки зрения структуры данных, полином можно хранить как линейный односвязный кольцевой список мономов. В таком случае звено полинома (моном) будет иметь следующий вид:



где **coeff** – коэффициент монома,**degree** – свертка степеней монома

Таким образом, например, полином вида $5x^2y - 8yz + 2x$ будет храниться в виде:



В число основных операций над полиномами входят:

- **Сложение полиномов**

Алгоритм сложения полиномов, хранящихся как кольцевой список мономов, может быть реализован следующим образом:

1) Инициализация указателей на начало каждого из полиномов.
2) Начало обхода первого полинома. Для каждого монома из первого полинома:
- Если моном с такой же степенью уже присутствует во втором полиноме, то сложить их коэффициенты и сохранить результат.
- Если моном с такой степенью отсутствует во втором полиноме, то добавить его в результирующий полином.
3) Продолжить обход второго полинома. Для каждого монома из второго полинома:
- Если моном с такой степенью отсутствует в результирующем полиноме, то добавить его в результирующий полином.

Результатом будет новый полином, являющийся суммой двух исходных полиномов.

- **Вычитание полиномов**

Алгоритм вычитания полиномов, хранящихся как кольцевой список мономов, может быть реализован следующим образом:

1) С помощью оператора унарного минуса получаем вычитаемый полином с противоположными знаками.
2) Прибавляем к полиному полученный «отрицательный» полином.

Результатом будет новый полином, являющийся разностью двух исходных полиномов.

- **Произведение полиномов**

Для умножения полиномов, хранящихся как кольцевой список мономов, можно использовать следующий алгоритм:

1) Инициализировать указатели на начало каждого из полиномов.

2) Создать новый пустой полином, который будет содержать результат умножения исходных полиномов.
3) Начать обход первого полинома. Для каждого монома из первого полинома:

- Умножить его на каждый моном из второго полинома.
- Результат умножения добавить в результирующий полином.

4) Повторить шаг 3 для всех мономов из первого полинома.

Результатом будет новый полином, являющийся произведением двух исходных полиномов.

- **Вычисление значения полинома в точке**

Для вычисления значения полинома в определенной точке, хранящегося как кольцевой список мономов, можно использовать следующий алгоритм:

1) Инициализировать указатель на начало полинома.
2) Инициализировать переменную для хранения значения полинома в точке.
3) Начать обход кольцевого списка мономов. Для каждого монома:

- Вычислить значение монома в заданной точке, учитывая коэффициент и степень монома.
- Добавить значение монома к общему значению полинома.

4) Продолжить обход по кольцевому списку мономов до конца.

Результатом будет значение полинома в заданной точке.

- **Дифференцирование полинома по переменной x**

Для нахождения производной по переменной x полинома, хранящегося как кольцевой список мономов, можно использовать следующий алгоритм:

1) Инициализировать указатель на начало полинома.
2) Создать новый пустой полином, который будет содержать производную исходного полинома по переменной x.
3) Начать обход кольцевого списка мономов. Для каждого монома:

- Вычислить производную монома по переменной x, умножив коэффициент монома на степень переменной x и уменьшив степень на единицу.
- Добавить новый моном (производную) в результирующий полином.

4) Продолжить обход по кольцевому списку мономов до конца.

Результатом будет новый полином, являющийся производной исходного полинома по переменной x.

- **Дифференцирование полинома по переменной y**

Для нахождения производной по переменной y полинома, хранящегося как кольцевой список мономов, можно использовать следующий алгоритм:

1) Инициализировать указатель на начало полинома.
2) Создать новый пустой полином, который будет содержать производную исходного полинома по переменной y.
3) Начать обход кольцевого списка мономов. Для каждого монома:
- Вычислить производную монома по переменной y, умножив коэффициент монома на степень переменной y и уменьшив степень на единицу.
- Добавить новый моном (производную) в результирующий полином.
4) Продолжить обход по кольцевому списку мономов до конца.

Результатом будет новый полином, являющийся производной исходного полинома по переменной y.

- **Дифференцирование полинома по переменной z**

Для нахождения производной по переменной z полинома, хранящегося как кольцевой список мономов, можно использовать следующий алгоритм:

1) Инициализировать указатель на начало полинома.
2) Создать новый пустой полином, который будет содержать производную исходного полинома по переменной z.
3) Начать обход кольцевого списка мономов. Для каждого монома:
- Вычислить производную монома по переменной z, умножив коэффициент монома на степень переменной z и уменьшив степень на единицу.
- Добавить новый моном (производную) в результирующий полином.
4) Продолжить обход по кольцевому списку мономов до конца.

Результатом будет новый полином, являющийся производной исходного полинома по переменной z.

3.2 Описание классов

3.2.1 Схема наследования классов

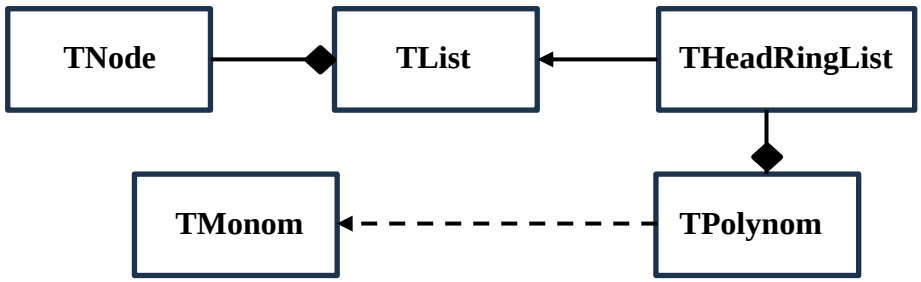


Рис. 17. Схема наследования классов

3.2.2 Структура TNode

Объявление структуры:

```
template <typename ValueType>
struct TNode {
       ValueType data;
       TNode* pNext;
       TNode();
       TNode(const ValueType& d, TNode<ValueType>* Next = nullptr);
};
```

Поля:

data – значение.

pNext – указатель на следующий узел.

Конструкторы:

TNode();

Назначение: установка значений полей класса **TNode** по умолчанию.

Входные данные: отсутствуют.

Выходные данные: отсутствуют.

TNode(const ValueType& d, TNode<ValueType>* Next = nullptr);

Назначение: конструктор с параметрами.

Входные данные: **d** – константная ссылка на значение, **Next** – указатель на следующее звено.

Выходные данные: отсутствуют.

3.2.3 Класс TList

Объявление класса:

```
template <typename ValueType>
class TList {
protected:
       TNode<ValueType>* pFirst;
       TNode<ValueType>* pLast;
       TNode<ValueType>* pCurr;
       TNode<ValueType>* pStop;
public:
       TList();
       TList(const TList<ValueType>& list);
       TList(TNode<ValueType>* _pFirst);
       virtual ~TList();
       TNode<ValueType>* Search(const ValueType& data);
       TNode<ValueType>* GetCurrent() const;
       virtual void InsertFirst(const ValueType& data);
       virtual void InsertLast(const ValueType& data);
       void InsertBefore(const ValueType& who, const ValueType& before_whom);
       void InsertAfter(const ValueType& who, const ValueType& after_whom);
       virtual void Remove(const ValueType& data);
       virtual void Clear();
       void Next();
       void Reset();
       void Sort();
       virtual bool IsEnded() const;
       bool IsEmpty() const;
       bool IsFull() const;
       friend istream& operator>>(istream& in, TList<ValueType>& list);
       friend ostream& operator<<(ostream& out, const TList<ValueType>& list);
};
```

Поля:

pFirst – указатель на первый узел списка.

pLast – указатель на последний узел списка.

pCurr – указатель на текущий узел списка.

pStop – указатель на конечный узел списка.

Конструкторы:

TList();

Назначение: установка значений полей класса **TList** по умолчанию.

Входные данные: отсутствуют.

Выходные данные: отсутствуют.

TList(const TList<ValueType>& list);

Назначение: создание копии списка.

Входные данные: **list** – константная ссылка на копируемый список.

Выходные данные: отсутствуют.

```
TList(TNode<ValueType>* _pFirst);
```

Назначение: создание списка с заданным первым узлом.

Входные данные: **_pFirst** – указатель на первый узел списка.

Выходные данные: отсутствуют.

Деструктор:

```
~TList();
```

Назначение: освобождение памяти, занимаемой динамическими полями класса **TList**.

Входные данные: отсутствуют.

Выходные данные: отсутствуют.

Методы:

```
TNode<ValueType>* Search(const ValueType& data);
```

Назначение: поиск узла с заданным значением.

Входные данные: **data** – константная ссылка на заданное значение.

Выходные данные: указатель на узел с заданным значением.

```
TNode<ValueType>* GetCurrent() const;
```

Назначение: получение текущего узла списка.

Входные данные: отсутствуют.

Выходные данные: указатель на текущий узел списка.

```
virtual void InsertFirst(const ValueType& data);
```

Назначение: вставка узла с заданным значением в начало списка.

Входные данные: **data** – константная ссылка на заданное значение.

Выходные данные: отсутствуют.

```
virtual void InsertLast(const ValueType& data);
```

Назначение: вставка узла с заданным значением в конец списка.

Входные данные: **data** – константная ссылка на заданное значение.

Выходные данные: отсутствуют.

```
void InsertBefore(const ValueType& who, const ValueType& before_whom);
```

Назначение: вставка узла с заданным значением до определенного узла списка.

Входные данные: **who** – константная ссылка на значение вставляемого узла, **before_whom** – константная ссылка на значение узла, до которого вставляем.

Выходные данные: отсутствуют.

`void InsertAfter(const ValueType& who, const ValueType& after_whom);`

Назначение: вставка узла с заданным значением после определенного узла списка.

Входные данные: **`who`** – константная ссылка на значение вставляемого узла, **`after_whom`** – константная ссылка на значение узла, после которого вставляем.

Выходные данные: отсутствуют.

`virtual void Remove(const ValueType& data);`

Назначение: удаление узла списка с заданным значением.

Входные данные: **`data`** – константная ссылка на заданное значение.

Выходные данные: отсутствуют.

`virtual void Clear();`

Назначение: очистка списка.

Входные данные: отсутствуют.

Выходные данные: отсутствуют.

`void Next();`

Назначение: переход к следующему узлу списка.

Входные данные: отсутствуют.

Выходные данные: отсутствуют.

`void Reset();`

Назначение: переход в начало списка.

Входные данные: отсутствуют.

Выходные данные: отсутствуют.

`void Sort();`

Назначение: упорядочивание списка.

Входные данные: отсутствуют.

Выходные данные: отсутствуют.

`virtual bool IsEnded() const;`

Назначение: проверка, достигли ли конца списка.

Входные данные: отсутствуют.

Выходные данные: результат проверки (**`true`** – достигли, **`false`** – не достигли).

`bool IsEmpty() const;`

Назначение: проверка, является ли список пустым.

Входные данные: отсутствуют.

Выходные данные: результат проверки (`true` – пустой, `false` – не пустой).

`bool IsFull() const;`

Назначение: проверка, является ли список полным.

Входные данные: отсутствуют.

Выходные данные: результат проверки (`true` – полный, `false` – не полный).

`friend istream& operator>>(istream& in, TList<ValueType>& list);`

Назначение: ввод списка.

Входные данные: `in` – ссылка на стандартный поток ввода, `list` – ссылка на вводимый список.

Выходные данные: ссылка на поток ввода.

`friend ostream& operator<<(ostream& out, const TList<ValueType>& list);`

Назначение: вывод списка.

Входные данные: `out` – ссылка на стандартный поток вывода, `list` – константная ссылка на выводимый список.

Выходные данные: ссылка на поток вывода.

3.2.4 Класс THeadRingList

Объявление класса:

```
template <typename ValueType>
class THeadRingList : public TList<ValueType> {
protected:
       TNode<ValueType>* pHead;
public:
       THeadRingList();
       THeadRingList(const THeadRingList<ValueType>& ringlist);
       ~THeadRingList();
       void InsertFirst(const ValueType& data);
       void InsertLast(const ValueType& data);
       void Remove(const ValueType& data);
       void Clear();
       bool IsEnded() const;
};
```

Поля:

`pHead` – указатель на головной узел списка.

Конструкторы:

`THeadRingList();`

Назначение: установка значений полей класса **`THeadRingList`** по умолчанию.

Входные данные: отсутствуют.

Выходные данные: отсутствуют.

`THeadRingList(const THeadRingList<ValueType>& ringlist);`

Назначение: создание копии кольцевого списка.

Входные данные: ring**`list`** – константная ссылка на копируемый кольцевой список.

Выходные данные: отсутствуют.

Деструктор:

`~THeadRingList();`

Назначение: освобождение памяти, занимаемой динамическими полями класса **`THeadRingList`**.

Входные данные: отсутствуют.

Выходные данные: отсутствуют.

Методы:

`void InsertFirst(const ValueType& data);`

Назначение: вставка узла с заданным значением в начало кольцевого списка.

Входные данные: **`data`** – константная ссылка на заданное значение.

Выходные данные: отсутствуют.

`void InsertLast(const ValueType& data);`

Назначение: вставка узла с заданным значением в конец кольцевого списка.

Входные данные: **`data`** – константная ссылка на заданное значение.

Выходные данные: отсутствуют.

`void Remove(const ValueType& data);`

Назначение: удаление узла кольцевого списка с заданным значением.

Входные данные: **`data`** – константная ссылка на заданное значение.

Выходные данные: отсутствуют.

```
void Clear();
```

Назначение: очистка кольцевого списка.

Входные данные: отсутствуют.

Выходные данные: отсутствуют.

```
bool IsEnded() const;
```

Назначение: проверка, достигли ли конца кольцевого списка.

Входные данные: отсутствуют.

Выходные данные: результат проверки (**true** – достигли, **false** – не достигли).

3.2.5 Класс TMonom

Объявление класса:

```
class TMonom {
public:
       double coeff;
       int degree;

       TMonom(const TMonom& monom);
       TMonom(double coef = 0, int deg = -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;
       bool operator==(const TMonom& monom)const;
       bool operator!=(const TMonom& monom)const;
       friend ostream& operator<<(ostream& out, const TMonom& monom);
}
```

Поля:

coeff – коэффициент монома.

degree – свертка степеней монома.

Конструкторы:

```
TMonom(const TMonom& monom);
```

Назначение: создание копии монома.

Входные данные: **monom** – константная ссылка на копируемый моном.

Выходные данные: отсутствуют.

```
TMonom(double coef = 0, int deg = -1);
```

Назначение: инициализация полей класса **TMonom**.

Входные данные: **coef** – коэффициент монома, **deg** – свертка степеней монома.

Выходные данные: отсутствуют.

Методы:

bool operator<(const TMonom& monom)const;

Назначение: сравнение мономов.

Входные данные: **monom** – константная ссылка на сравниваемый моном.

Выходные данные: результат сравнения (**true** – моном меньше **monom**, **false** – в противном случае).

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

Назначение: сравнение мономов.

Входные данные: **monom** – константная ссылка на сравниваемый моном.

Выходные данные: результат сравнения (**true** – моном меньше или равен **monom**, **false** – в противном случае).

bool operator>(const TMonom& monom)const;

Назначение: сравнение мономов.

Входные данные: **monom** – константная ссылка на сравниваемый моном.

Выходные данные: результат сравнения (**true** – моном больше **monom**, **false** – в противном случае).

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

Назначение: сравнение мономов.

Входные данные: **monom** – константная ссылка на сравниваемый моном.

Выходные данные: результат сравнения (**true** – моном больше или равен **monom**, **false** – в противном случае).

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

Назначение: сравнение мономов.

Входные данные: **monom** – константная ссылка на сравниваемый моном.

Выходные данные: результат сравнения (**true** – моном равен **monom**, **false** – в противном случае).

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

Назначение: сравнение мономов.

Входные данные: **monom** – константная ссылка на сравниваемый моном.

Выходные данные: результат сравнения (**true** – моном не равен **monom**, **false** – в противном случае).

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

Назначение: вывод монома.

Входные данные: **out** – ссылка на стандартный поток вывода, **monom** – константная ссылка на выводимый моном.

Выходные данные: ссылка на поток вывода.

3.2.6 Класс TPolynom

Объявление класса:

```
class TPolynom {
private:
       THeadRingList<TMonom> monoms;
       string expr;

       void Check(const string& expr);
       void Parse(const string& expr);
       void Cancellation();
public:
       TPolynom();
       TPolynom(const string& expr);
       TPolynom(const THeadRingList<TMonom>& monomlist);
       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 operator-(const TPolynom& polynom);
       TPolynom operator*(const TPolynom& polynom);
       double operator()(double x, double y, double z)const;
       TPolynom dx()const;
       TPolynom dy()const;
       TPolynom dz()const;
       friend ostream& operator<<(ostream& out, TPolynom& polynom);
}
```

Поля:

monoms – кольцевой линейный односвязный список мономов.

expr – строка с полиномом.

Конструкторы:

```
TPolynom();
```

Назначение: установка значений полей класса **TPolynom** по умолчанию.

Входные данные: отсутствуют.

Выходные данные: отсутствуют.

```
TPolynom(const string& expr);
```

Назначение: формирование объекта класса **TPolynom** с указанным выражением.

Входные данные: **expr** – константная ссылка на строку с полиномом.

Выходные данные: отсутствуют.

`TPolynom(const THeadRingList<TMonom>& monomlist);`

Назначение: формирование объекта класса **`TPolynom`** на основе кольцевого списка мономов.

Входные данные: **`monomlist`** – константная ссылка на кольцевой список мономов.

Выходные данные: отсутствуют.

`TPolynom(const TPolynom& polynom);`

Назначение: создание копии полинома.

Входные данные: **`polynom`** – константная ссылка на копируемый полином.

Выходные данные: отсутствуют.

Методы:

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

Назначение: присваивание полинома.

Входные данные: **`polynom`** – константная ссылка на присваиваемый полином.

Выходные данные: ссылка на поток вывода.

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

Назначение: проверка полиномов на равенство.

Входные данные: **`polynom`** – константная ссылка на сравниваемый полином.

Выходные данные: результат сравнения (**`true`** – полиномы равны, **`false`** – полиномы не равны).

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

Назначение: сложение полиномов.

Входные данные: **`polynom`** – константная ссылка на прибавляемый полином.

Выходные данные: сумма полиномов.

`TPolynom operator-()const;`

Назначение: унарный минус.

Входные данные: отсутствуют.

Выходные данные: полином с противоположными знаками.

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

Назначение: вычитание полиномов.

Входные данные: **polynom** – константная ссылка на вычитаемый полином.

Выходные данные: разность полиномов.

TPolynom operator*(const TPolynom& polynom);

Назначение: умножение полиномов.

Входные данные: **polynom** – константная ссылка на умножаемый полином.

Выходные данные: произведение полиномов.

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

Назначение: вычисление значения полинома в заданной точке.

Входные данные: **x, y, z** – координаты точки.

Выходные данные: значение полинома в точке.

TPolynom dx()const;

Назначение: дифференцирование полинома по переменной «x».

Входные данные: отсутствуют.

Выходные данные: первая производная полинома по переменной «x».

TPolynom dy()const;

Назначение: дифференцирование полинома по переменной «y».

Входные данные: отсутствуют.

Выходные данные: первая производная полинома по переменной «y».

TPolynom dz()const;

Назначение: дифференцирование полинома по переменной «z».

Входные данные: отсутствуют.

Выходные данные: первая производная полинома по переменной «z».

friend ostream& operator<<(ostream& out, TPolynom& polynom);

Назначение: вывод полинома.

Входные данные: **out** – ссылка на стандартный поток вывода, **polynom** – константная ссылка на выводимый полином.

Выходные данные: ссылка на поток вывода.

Заключение

В ходе лабораторной работы были изучены основные термины и понятия, связанные со списками, а также наиболее эффективные способы их представления (хранения). Были изучены понятия о полиномах.

На основе подготовленной теоретической базы, были реализованы классы для представления списков TList и работы с полиномами TPolynom со всеми необходимыми операциями. Для проверки работоспособности и эффективности реализации перечисленных выше классов были написаны приложения sample_tlist и sample_tpolynom, а также модульные тесты.

Литература

1. Барышева И.В. Учебно-методическое пособие. – Нижний Новгород: Нижегородский госуниверситет, 2017 – 105 с.
2. Односвязный линейный список [https://prog-cpp.ru/data-ols/]
3. Кольцевой односвязный список [https://metanit.com/sharp/algoritm/2.7.php]

Приложения

Приложение А. Реализация структуры TNode

```
template <typename ValueType>
TNode<ValueType>::TNode() {
	data = {};
	pNext = nullptr;
}

template <typename ValueType>
TNode<ValueType>::TNode(const ValueType& d, TNode<ValueType>* Next) {
	data = d;
	pNext = Next;
}
```

Приложение Б. Реализация класса TList

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

template <typename ValueType>
TList<ValueType>::TList(const TList<ValueType>& list) {
	if (list.IsEmpty()) {
		pFirst = nullptr;
		pLast = nullptr;
		pCurr = nullptr;
		pStop = nullptr;
		return;
	}
	pFirst = new TNode<ValueType>(list.pFirst->data);
	TNode<ValueType>* tmp = pFirst;
	TNode<ValueType>* ltmp = list.pFirst->pNext;
	while (ltmp != list.pStop) {
		tmp->pNext = new TNode<ValueType>(ltmp->data);
		tmp = tmp->pNext;
		ltmp = ltmp->pNext;
	}
	pLast = tmp;
	pCurr = pFirst;
	pStop = nullptr;
}

template <typename ValueType>
TList<ValueType>::TList(TNode<ValueType>* pNode) {
	pFirst = pNode;
	TNode<ValueType>* tmp = pNode;
	while (tmp->pNext != nullptr)
		tmp = tmp->pNext;
	pLast = tmp;
	pCurr = pFirst;
	pStop = nullptr;
}

template <typename ValueType>
TList<ValueType>::~TList() {
	Clear();
}

template <typename ValueType>
TNode<ValueType>* TList<ValueType>::Search(const ValueType& data) {
	TNode<ValueType>* curr = pFirst;
```

```
	while (curr->pNext != pStop && curr->data != data) {
		curr = curr->pNext;
	}
	if (curr->pNext == pStop && curr->data != data)
		return nullptr;
	return curr;
}

template <typename ValueType>
TNode<ValueType>* TList<ValueType>::GetCurrent() const {
	return pCurr;
}

template <typename ValueType>
void TList<ValueType>::InsertFirst(const ValueType& data) {
	TNode<ValueType>* new_first = new TNode<ValueType>(data, pFirst);
	pFirst = new_first;
	if (pLast == nullptr) {
		pLast = pFirst;
	}
	pCurr = pFirst;
}

template <typename ValueType>
void TList<ValueType>::InsertLast(const ValueType& data) {
	if (IsEmpty()) {
		InsertFirst(data);
		return;
	}
	TNode<ValueType>* new_last = new TNode<ValueType>(data, pStop);
	pLast->pNext = new_last;
	pLast = new_last;
	pCurr = new_last;
}

template <typename ValueType>
void TList<ValueType>::InsertBefore(const ValueType& who, const ValueType&
before_whom) {
	TNode<ValueType>* prev = nullptr;
	TNode<ValueType>* curr = pFirst;
	while (curr != pStop && curr->data != before_whom) {
		prev = curr;
		curr = curr->pNext;
	}
	if (curr == pStop) {
		throw exception("no such element");
	}
	if (prev == nullptr) {
		InsertFirst(who);
		return;
	}
	TNode<ValueType>* new_node = new TNode<ValueType>(who, curr);
	prev->pNext = new_node;
}

template <typename ValueType>
void TList<ValueType>::InsertAfter(const ValueType& who, const ValueType& after_whom)
{
	TNode<ValueType>* pWhere = Search(after_whom);
	if (pWhere == nullptr) {
		throw exception("no such element");
	}
	if (pWhere == pLast) {
		InsertLast(who);
		return;
	}
	TNode<ValueType>* new_node = new TNode<ValueType>(who, pWhere->pNext);
	pWhere->pNext = new_node;
}
```

```
template <typename ValueType>
void TList<ValueType>::Remove(const ValueType& data) {
	TNode<ValueType>* prev = pStop;
	TNode<ValueType>* curr = pFirst;
	while (curr != pStop && curr->data != data) {
		prev = curr;
		curr = curr->pNext;
	}
	if (curr == pStop) {
		throw exception("no such element");
	}
	if (curr == pFirst) {
		pFirst = pFirst->pNext;
		delete curr;
		return;
	}
	if (curr == pLast) {
		prev->pNext = pStop;
		delete curr;
		return;
	}
	prev->pNext = curr->pNext;
	curr->pNext = nullptr;
	delete curr;
}

template <typename ValueType>
void TList<ValueType>::Clear() {
	if (pFirst == nullptr)
		return;
	TNode<ValueType>* curr = pFirst;
	TNode<ValueType>* next = pFirst->pNext;
	while (next != pStop) {
		delete curr;
		curr = next;
		next = curr->pNext;
	}
	delete curr;
	pCurr = pStop;
	pFirst = nullptr;
	pLast = nullptr;
}

template <typename ValueType>
void TList<ValueType>::Next() {
	if (pCurr == pStop)
		throw exception("end of the list");
	pCurr = pCurr->pNext;
}

template <typename ValueType>
void TList< ValueType>::Reset() {
	pCurr = pFirst;
}

template <typename ValueType>
void TList< ValueType>::Sort() {
	TNode<ValueType>* node1 = pFirst;
	while (node1->pNext != pStop) {
		TNode<ValueType>* node2 = node1->pNext;
		while (node2 != pStop) {
			if (node1->data < node2->data) {
				ValueType tmp = node1->data;
				node1->data = node2->data;
				node2->data = tmp;
			}
			node2 = node2->pNext;
		}
		node1 = node1->pNext;
	}
```

```
}

template <typename ValueType>
bool TList<ValueType>::IsEmpty()const {
	return pFirst == nullptr;
}

template <typename ValueType>
bool TList<ValueType>::IsFull()const {
	TNode<ValueType>* tmp = new TNode<ValueType>();
	if (tmp == nullptr)
		return true;
	delete tmp;
	return false;
}

template <typename ValueType>
bool TList<ValueType>::IsEnded()const {
	return pCurr == pStop;
}
```

Приложение В. Реализация класса THeadRingList

```
template <typename ValueType>
THeadRingList<ValueType>::THeadRingList() : TList<ValueType>() {
	pHead = new TNode<ValueType>();
	pHead->pNext = pHead;
	pStop = pHead;
}

template <typename ValueType>
THeadRingList<ValueType>::THeadRingList(const THeadRingList<ValueType>& ringlist) :
TList<ValueType>(ringlist) {
	pHead = new TNode<ValueType>();
	pHead->pNext = pFirst;
	if (!ringlist.IsEmpty())
		pLast->pNext = pHead;
	pStop = pHead;
}

template <typename ValueType>
THeadRingList<ValueType>::~THeadRingList() {
	delete pHead;
}

template <typename ValueType>
void THeadRingList<ValueType>::InsertFirst(const ValueType& data) {
	TList<ValueType>::InsertFirst(data);
	pHead->pNext = pFirst;
	pLast->pNext = pHead;
}

template <typename ValueType>
void THeadRingList<ValueType>::InsertLast(const ValueType& data) {
	TList<ValueType>::InsertLast(data);
	pLast->pNext = pHead;
	if (pFirst == pLast)
	{
		pHead->pNext = pFirst;
	}
}

template <typename ValueType>
void THeadRingList<ValueType>::Remove(const ValueType& data) {
	TList<ValueType>::Remove(data);
	pHead->pNext = pFirst;
	pLast->pNext = pHead;
}

template <typename ValueType>
```

```
void THeadRingList<ValueType>::Clear() {
	TList<ValueType>::Clear();
	pHead->pNext = pHead;
}

template <typename TData>
bool THeadRingList<TData>::IsEnded() const {
	if (IsEmpty())
		return true;
	return pCurr == pStop;
}
```

Приложение Г. Реализация класса TMonom

```
TMonom::TMonom(const TMonom& monom) {
	coeff = monom.coeff;
	degree = monom.degree;
}

TMonom::TMonom(double coef, int deg) {
	if (deg > 999)
		throw exception("degree out of range");
	coeff = coef;
	degree = deg;
}

bool TMonom::operator<(const TMonom& monom)const {
	return degree < monom.degree || degree == monom.degree && coeff < monom.coeff;
}

bool TMonom::operator<=(const TMonom& monom)const {
	return degree < monom.degree || degree == monom.degree && coeff < monom.coeff ||
*this == monom;
}

bool TMonom::operator>(const TMonom& monom)const {
	return degree > monom.degree || degree == monom.degree && coeff > monom.coeff;
}

bool TMonom::operator>=(const TMonom& monom)const {
	return degree > monom.degree || degree == monom.degree && coeff > monom.coeff ||
*this == monom;
}

bool TMonom::operator==(const TMonom& monom)const {
	return degree == monom.degree && coeff == monom.coeff;
}

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

Приложение Д. Реализация класса TPolynom

```
TPolynom::TPolynom() : monoms() {
}

TPolynom::TPolynom(const string& expr) : monoms() {
	this->expr = expr;
	Check(expr);
	Parse(expr);
}

TPolynom::TPolynom(const THeadRingList<TMonom>& monomlist) : monoms(monomlist) {
}

TPolynom::TPolynom(const TPolynom& p) : monoms(p.monoms), expr(p.expr) {}
```

```
void TPolynom::Check(const string& expr) {
	string items = "0123456789xyz.*^+-";
	for (int i = 0; i < expr.size(); i++) {
		if (items.find(expr[i]) == string::npos)
			throw exception("an expression contains invalid characters");
		if (i != expr.size() && !isdigit(expr[i]) && expr[i] == expr[i + 1])
			throw exception("an expression contains repetitive operators");
	}
}

void TPolynom::Parse(const string& expr) {
	string str = expr;
	while (!str.empty()) {
		double curr_coeff = -1;
		int curr_degree = 0;
		string curr_monom = str.substr(0, str.find_first_of("+-", 1));
		str.erase(0, str.find_first_of("+-", 1));
		string coef = "";
		int i = 0;
		while (i < curr_monom.size() && !isalpha(curr_monom[i])) {
			coef += curr_monom[i];
			i++;
		}
		if (coef == "+" || coef == "")
			curr_coeff = 1;
		else if (coef == "-")
			curr_coeff = -1;
		else
			curr_coeff = stod(coef);
		for (i; i < curr_monom.size(); i++) {
			if (i != curr_monom.size() && isalpha(curr_monom[i])) {
				int deg = 1;
				if (curr_monom[i + 1] == '^')
					deg = curr_monom[i + 2] - '0';
				switch (curr_monom[i]) {
				case 'x':
					curr_degree += deg * 100;
					break;
				case 'y':
					curr_degree += deg * 10;
					break;
				case 'z':
					curr_degree += deg * 1;
					break;
				}
			}
		}
		TMonom monom(curr_coeff, curr_degree);
		monoms.InsertLast(monom);
	}
	this->Cancellation();
}

void TPolynom::Cancellation() {
	TPolynom tmp(*this);
	this->monoms.Clear();
	tmp.monoms.Sort();
	tmp.monoms.Reset();
	while (!tmp.monoms.IsEnded()) {
		TMonom mn = tmp.monoms.GetCurrent()->data;
		double coef = mn.coeff;
		tmp.monoms.Next();
		while (!tmp.monoms.IsEnded() && tmp.monoms.GetCurrent()->data.degree ==
mn.degree) {
			coef += tmp.monoms.GetCurrent()->data.coeff;
			tmp.monoms.Next();
		}
		if (coef != 0) {
			TMonom monom(coef, mn.degree);
```

```
                this->monoms.InsertLast(monom);
            }
        }
        if (this->monoms.IsEmpty()) {
            TMonom mon(0, 0);
            this->monoms.InsertLast(mon);
        }
}

const TPolynom& TPolynom::operator=(const TPolynom& p) {
        if (this == &p)
            return *this;
        monoms = p.monoms;
        expr = p.expr;
        return *this;
}

bool TPolynom::operator==(const TPolynom& p)const {
        TPolynom polynom(*this);
        TPolynom tmp(p);
        while (!polynom.monoms.IsEnded() && !tmp.monoms.IsEnded()) {
            if (polynom.monoms.GetCurrent()->data != tmp.monoms.GetCurrent()->data)
                return false;
            polynom.monoms.Next();
            tmp.monoms.Next();
        }
        return true;
}

TPolynom TPolynom::operator+(const TPolynom& p) {
        TPolynom p1(*this);
        TPolynom p2(p);
        TPolynom sum;
        p1.monoms.Reset();
        p2.monoms.Reset();
        while (!p1.monoms.IsEnded() && !p2.monoms.IsEnded()) {
            if (p1.monoms.GetCurrent()->data.degree > p2.monoms.GetCurrent()-
>data.degree) {
                sum.monoms.InsertLast(p1.monoms.GetCurrent()->data);
                p1.monoms.Next();
            }
            else if (p1.monoms.GetCurrent()->data.degree < p2.monoms.GetCurrent()-
>data.degree) {
                sum.monoms.InsertLast(p2.monoms.GetCurrent()->data);
                p2.monoms.Next();
            }
            else {
                p1.monoms.GetCurrent()->data.coeff += p2.monoms.GetCurrent()-
>data.coeff;
                if (p1.monoms.GetCurrent()->data.coeff != 0)
                    sum.monoms.InsertLast(p1.monoms.GetCurrent()->data);
                p1.monoms.Next();
                p2.monoms.Next();
            }
        }
        while (!p1.monoms.IsEnded()) {
            sum.monoms.InsertLast(p1.monoms.GetCurrent()->data);
            p1.monoms.Next();
        }
        while (!p2.monoms.IsEnded()) {
            sum.monoms.InsertLast(p2.monoms.GetCurrent()->data);
            p2.monoms.Next();
        }
        if (sum.monoms.IsEmpty()) {
            TMonom mon(0, 0);
            sum.monoms.InsertLast(mon);
        }
        return sum;
}
```

```
TPolynom TPolynom::operator-()const {
	TPolynom polynom(*this);
	polynom.monoms.Reset();
	while (!polynom.monoms.IsEnded()) {
		polynom.monoms.GetCurrent()->data.coeff = polynom.monoms.GetCurrent()-
>data.coeff * (-1);
		polynom.monoms.Next();
	}
	return polynom;
}

TPolynom TPolynom::operator-(const TPolynom& p) {
	TPolynom dif = (*this) + (-p);
	return dif;
}

TPolynom TPolynom::operator*(const TPolynom& p) {
	TPolynom prd;
	TPolynom tmp(p);
	monoms.Reset();
	while (!monoms.IsEnded()) {
		tmp.monoms.Reset();
		while (!tmp.monoms.IsEnded()) {
			TMonom mon1 = monoms.GetCurrent()->data;
			TMonom mon2 = tmp.monoms.GetCurrent()->data;
			double new_coeff = mon1.coeff * mon2.coeff;
			int new_degree = mon1.degree + mon2.degree;
			if (new_degree > 999)
				throw exception("degree out of range");
			TMonom monom(new_coeff, new_degree);
			prd.monoms.InsertLast(monom);
			tmp.monoms.Next();
		}
		monoms.Next();
	}
	prd.Cancellation();
	return prd;
}

double TPolynom::operator()(double x, double y, double z)const {
	double result = 0;
	TPolynom polynom(*this);
	while (!polynom.monoms.IsEnded()) {
		double mon = polynom.monoms.GetCurrent()->data.coeff;
		mon *= pow(x, polynom.monoms.GetCurrent()->data.degree / 100);
		mon *= pow(y, polynom.monoms.GetCurrent()->data.degree / 10 % 10);
		mon *= pow(z, polynom.monoms.GetCurrent()->data.degree % 10);
		result += mon;
		polynom.monoms.Next();
	}
	return result;
}

TPolynom TPolynom::dx()const {
	TPolynom polynom(*this);
	TPolynom dx_polynom;
	while (!polynom.monoms.IsEnded()) {
		double new_coeff = polynom.monoms.GetCurrent()->data.coeff *
(polynom.monoms.GetCurrent()->data.degree / 100);
		int new_degree = polynom.monoms.GetCurrent()->data.degree - 100;
		if (new_coeff != 0) {
			TMonom monom(new_coeff, new_degree);
			dx_polynom.monoms.InsertLast(monom);
		}
		polynom.monoms.Next();
	}
	if (dx_polynom.monoms.IsEmpty()) {
		TMonom mon(0, 0);
		dx_polynom.monoms.InsertLast(mon);
	}
```

```
	return dx_polynom;
}

TPolynom TPolynom::dy()const {
	TPolynom polynom(*this);
	TPolynom dy_polynom;
	while (!polynom.monoms.IsEnded()) {
		double new_coeff = polynom.monoms.GetCurrent()->data.coeff *
(polynom.monoms.GetCurrent()->data.degree / 10 % 10);
		int new_degree = polynom.monoms.GetCurrent()->data.degree - 10;
		if (new_coeff != 0) {
			TMonom monom(new_coeff, new_degree);
			dy_polynom.monoms.InsertLast(monom);
		}
		polynom.monoms.Next();
	}
	if (dy_polynom.monoms.IsEmpty()) {
		TMonom mon(0, 0);
		dy_polynom.monoms.InsertLast(mon);
	}
	return dy_polynom;
}

TPolynom TPolynom::dz()const {
	TPolynom polynom(*this);
	TPolynom dz_polynom;
	while (!polynom.monoms.IsEnded()) {
		double new_coeff = polynom.monoms.GetCurrent()->data.coeff *
(polynom.monoms.GetCurrent()->data.degree % 10);
		int new_degree = polynom.monoms.GetCurrent()->data.degree - 1;
		if (new_coeff != 0) {
			TMonom monom(new_coeff, new_degree);
			dz_polynom.monoms.InsertLast(monom);
		}
		polynom.monoms.Next();
	}
	if (dz_polynom.monoms.IsEmpty()) {
		TMonom mon(0, 0);
		dz_polynom.monoms.InsertLast(mon);
	}
	return dz_polynom;
}
```