

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

**Институт информационных технологий, математики и механики**

**ЛАБОРАТОРНАЯ РАБОТА**

на тему:

**«Аналитические преобразования полиномов от  
нескольких переменных (списки)»**

**Выполнил(а):** студент(ка) группы  
3822Б1ФИ2

\_\_\_\_\_/ Чижов М.А./  
Подпись

**Проверил:** к.т.н, доцент каф. ВВиСП  
\_\_\_\_\_/ Кустикова В.Д./

Подпись

Нижний Новгород  
2024

# Содержание

Введение.....	3
1. Постановка задачи.....	5
2. Руководство пользователя.....	6
2.1. Приложение для демонстрации работы полинома.....	6
3. Руководство программиста.....	8
3.1. Описание алгоритмов.....	8
3.1.1. Линейный односвязный список.....	8
3.1.2. Кольцевой односвязный линейный список.....	11
3.1.3. Моном.....	12
3.1.4. Полином.....	13
3.2. Описание программной реализации.....	16
3.2.1. Описание структуры TNode.....	16
3.2.2. Описание класса TList.....	17
3.2.3. Описание класса TRingList.....	21
3.2.4. Описание класса TMonom.....	23
3.2.5. Описание класса TPolynom.....	24
Заключение.....	32
Литература.....	33
Приложения.....	34
Приложение А. Реализация структуры TNode.....	34
Приложение Б. Реализация класса TList.....	34
Приложение В. Реализация класса TRingList.....	37
Приложение Г. Реализация класса TMonom.....	39
Приложение Д. Реализация класса TPolynom.....	39

## Введение

Линейный список – это структура данных, которая представляет собой последовательность элементов, где каждый элемент содержит ссылку на следующий элемент в списке. Каждый элемент списка называется узлом, а ссылка на следующий элемент называется указателем или ссылкой на следующий узел.

Линейный список может быть пустым, то есть не содержать ни одного элемента, или состоять из одного или более элементов. При этом первый элемент списка называется головой, а последний элемент – хвостом.

Линейный список может быть однонаправленным, когда каждый узел содержит ссылку только на следующий узел, или двунаправленным, когда каждый узел содержит ссылки как на следующий, так и на предыдущий узел.

### Преимущества:

1. Гибкость. Линейные списки позволяют легко добавлять и удалять элементы. Нет необходимости перемещать другие элементы при вставке или удалении элемента из списка.

2. Динамическое выделение памяти. Линейные списки могут быть динамически расширены или сокращены в зависимости от потребностей. Это позволяет эффективно использовать память и избегать неиспользуемых областей памяти.

3. Простота реализации. Линейные списки являются одной из самых простых структур данных для реализации. Они не требуют сложных операций и могут быть реализованы с помощью базовых операций, таких как добавление, удаление и поиск элементов.

4. Удобство использования. Линейные списки обеспечивают удобный доступ к элементам по индексу или значению. Это позволяет легко выполнять операции поиска, сортировки и обхода элементов списка.

### Недостатки:

1. Ограниченная производительность. При использовании линейных списков для хранения большого количества данных может возникнуть проблема производительности. Поиск элемента в списке может занимать много времени, особенно если список содержит миллионы элементов.

2. Потребление памяти. Линейные списки требуют дополнительной памяти для хранения указателей на следующий элемент списка. Это может привести к потреблению большого объема памяти, особенно при хранении больших данных.

3. Ограниченные операции. Линейные списки предоставляют только базовые операции, такие как добавление, удаление и поиск элементов. Другие операции, такие как

сортировка или объединение списков, могут потребовать дополнительных усилий и времени для реализации.

4. Неупорядоченность элементов. Линейные списки не гарантируют упорядоченность элементов. Это означает, что элементы могут быть расположены в произвольном порядке, что может затруднить выполнение некоторых операций, таких как поиск минимального или максимального элемента.

# 1. Постановка задачи

Цель – реализовать структуру данных полином.

Задачи:

1. Реализовать класс для работы с линейным односвязным списком.
2. Реализовать класс для работы с кольцевым линейным односвязным списком.
3. Реализовать класс для работы с полиномами на основе кольцевого линейного односвязного списка.
4. Написать следующие операции для работы с полиномами: сложение, вычитание, умножение, дифференцирование, вычисления значения полинома.
5. Добавить вспомогательную операция считывания полинома.
6. Написать следующие алгоритмы для работы с полиномами: перевод в постфиксную форму, вычисление выражения, записанного в постфиксной форме.

## 2. Руководство пользователя

### 2.1. Приложение для демонстрации работы полинома

1. Запустите приложение с названием sample\_polynom.exe. В результате появится окно, показанное ниже (рис. 1).

```
Enter first polynomial:
```

Рис. 1. Основное окно программы

2. Введите первый полином, а затем второй. Полиномы вводятся без пробелов (рис. 2).

```
Enter first polynomial:
2*x^3*y^5*z-3*x^4*y+5*x
Enter second polynomial:
```

Рис. 2. Ввод полиномов

3. В результате появится следующее окно (рис. 3) и (рис. 4).

```
Enter first polynomial:
2*x^3*y^5*z-3*x^4*y+5*x
Enter second polynomial:
7*x*y^3*z^5+3*x^4*y+4*x*y^5
2*x^3*y^5*z-3*x^4*y+5*x
5 100
2 351
-3 410

7*x*y^3*z^5+3*x^4*y+4*x*y^5
7 135
4 150
3 410

p3 = p2+p1
5*x+7*x*y^3*z^5+4*x*y^5+2*x^3*y^5*z
5 100
7 135
4 150
2 351

p4 = p1-p2
5*x-7*x*y^3*z^5-4*x*y^5+2*x^3*y^5*z-6*x^4*y
5 100
-7 135
-4 150
2 351
-6 410
```

Рис. 3. Результат арифметических выражений

```
p5 = p1*p2
35*x^2*y^3*z^5+2*x^2*y^5+14*x^4*y^8+z^6+8*x^5*z+15*x^5*y-21*x^5*y^4*z^5-12*x^5*y^6+6*x^7*y^6+z-9*x^8*y^2
35 235
20 250
14 486
8 501
15 510
-21 545
-12 560
6 761
-9 820

p6 = p1.dif_x
5+6*x^2*y^5*z-12*x^3*y
5 0
6 251
-12 310


p7 = p2.dif_y
21*x*y^2*z^5+2*x*y^4+3*x^4
21 125
20 140
3 400

p8 = p2.dif_z
35*x*y^3*z^4
35 134

Enter x: |
```

Рис. 4. Результат арифметических выражений

4. Введите значения  $x$ ,  $y$ ,  $z$ , после этого выведется результат вычисления полинома (рис. 5)



```
Enter x: 2  
Enter y: 3  
Enter z: 4  
p3 result:  
404578
```

Рис. 5. Результат вычислений

### 3. Руководство программиста

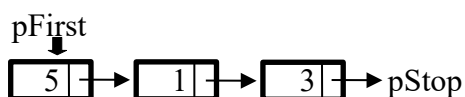
#### 3.1. Описание алгоритмов

##### 3.1.1. Линейный односвязный список

Линейный односвязный список – это динамическая структура данных, которая состоит из звеньев, содержащих значение и указатель на следующий элемент. Структура данных поддерживает операции такие как: вставка в начало, конец списка, вставка до и после элемента. Также присутствуют операции поиска, удаления из списка, проверка на пустоту, полноту.

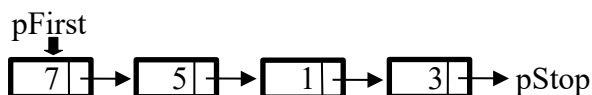
##### Операция вставки в начало

Метод InsertFirst предназначен для добавления нового узла в начало односвязного списка. При вызове создается новый узел с переданными данными. Если список пуст, новый узел становится первым и последним элементом списка. В противном случае новый узел становится первым элементом списка, а указатель на текущий первый элемент становится следующим за новым узлом.



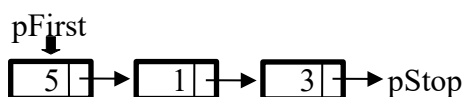
Добавим элемент 7.

Результат:



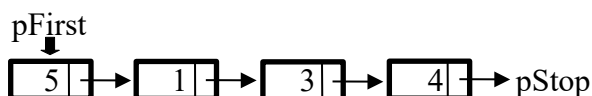
##### Операция вставки в конец

Метод InsertEnd предназначен для добавления нового узла в конец односвязного списка. Если список пуст, новый узел добавляется в начало. В противном случае создается узел с переданными данными и указателем на следующий элемент, указывающим на конец списка. Затем указатель последнего узла обновляется, указывая на новый узел, который становится последним элементом списка.



Добавим элемент 4 в конец.

Результат:



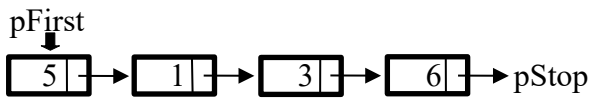
##### Операция поиска

Метод Search ищет узел с определенными данными в односвязном списке. Он начинает поиск с начала списка и двигается по элементам до тех пор, пока не найдет узел



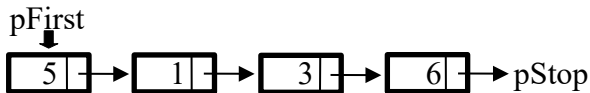
с совпадающими данными или не достигнет конца списка. Если узел с искомыми данными найден, метод возвращает указатель на этот узел. В противном случае возвращается nullptr.

Найдем элемент 6.



Вернется указатель на 6.

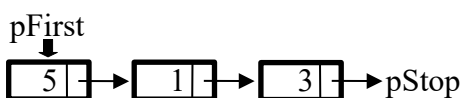
Найдем 7.



Вернется указатель на pStop(nullptr).

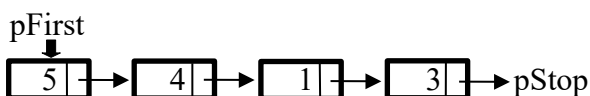
### Операция вставки перед заданным элементом

Метод InsertBefore предназначен для вставки нового узла с определенными данными перед узлом с определенными данными в линейном односвязном списке. При вызове этого метода выполняется проверка наличия элементов в списке: если текущий первый элемент списка pFirst не указывает на pStop и содержит данные, равные nextdata, то вызывается метод вставки в начало InsertFirst с переданными данными, и операция завершается. В противном случае, создается временный указатель tmp и указатель pPrev устанавливается в nullptr. Далее осуществляется проход по списку: пока не будет достигнут конец списка и данные текущего узла не равны nextdata, указатель pPrev обновляется текущим узлом, а tmp перемещается на следующий узел. Если целевой узел с данными nextdata был найден (указатель tmp не равен pStop), создается новый узел pNode с переданными данными. Затем указатель следующего элемента нового узла pNode устанавливается на узел tmp, а указатель следующего элемента pPrev перенаправляется на новый узел pNode, вставляя его между pPrev и tmp. В случае, если целевой узел с данными nextdata не был найден (указатель tmp равен pStop), выбрасывается исключение с сообщением "Element not found!".



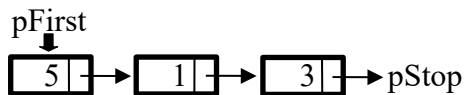
Добавим 4 перед 1.

Результат:

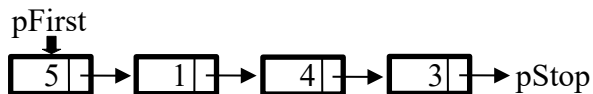


### Операция вставки после заданного элемента

Данный метод InsertAfter предназначен для вставки нового узла с определенными данными после узла с определенными данными в линейном односвязном списке. В начале метода выполняется поиск узла с данными beforedata с помощью вызова функции Search. Если узел найден (указатель pPrev не равен pStop), то происходит проверка: если узел pPrev является последним в списке (т.е. его указатель на следующий элемент равен pStop), то вызывается метод вставки нового узла в конец InsertEnd с переданными данными и завершается операция. Если узел pPrev не является последним в списке, создается новый узел pNode с переданными данными. Затем указатель следующего элемента нового узла pNode устанавливается на элемент, следующий за pPrev, а указатель следующего элемента pPrev перенаправляется на новый узел pNode, вставляя его между pPrev и элементом, который идет после pPrev. В случае, если узел с данным beforedata не был найден (указатель pPrev равен pStop), выбрасывается исключение с сообщением "Element not found!"

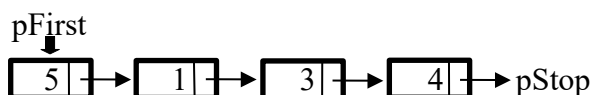


Добавим 4 после 1.



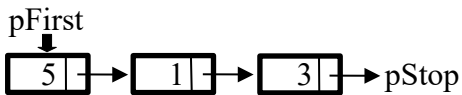
#### Операция удаления

Метод Remove предназначен для удаления узла с определенными данными из линейного односвязного списка. При вызове этого метода происходит проверка наличия элементов в списке: если список пуст, выбрасывается исключение с сообщением "Error". Затем начинается поиск узла с данными data, начиная с первого узла списка. Пока не будет достигнут конец списка и данные текущего узла не равны data, обновляется указатель pPrev текущим узлом, а pNode перемещается на следующий узел. Если достигнут конец списка и узел с данными data не был найден, выбрасывается исключение с сообщением "Error". Если узел с данными data найден и он является первым в списке, то указатель pFirst обновляется исключая найденный узел, затем узел удаляется, и операция завершается. Если узел с данными data найден и не является первым в списке, то указатель следующего элемента предыдущего узла перенаправляется на следующий элемент найденного узла, узел удаляется, и операция завершается. Если найденный узел был последним в списке, указатель pLast обновляется на предыдущий узел перед удалением.



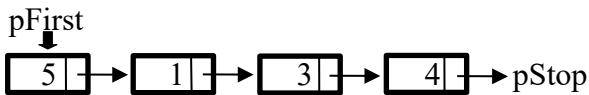
Удалим 1.

Результат:



#### Операция проверка на пустоту

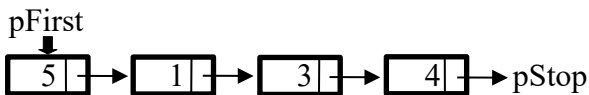
Функция проверяет есть ли элементы в списке. Если указатель pFirst равен nullptr, тогда список пуст, и функция вернет true. В противном случае вернет false.



Результат: false.

#### Операция проверки на полноту

Функция проверяет есть ли элементы в списке. Если указатель pFirst равен nullptr, тогда список пуст, и функция вернет false. В противном случае вернет true.



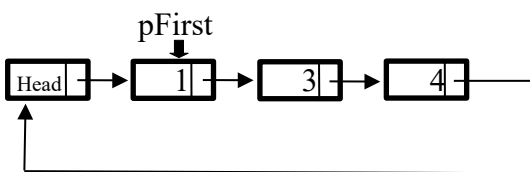
Результат: true.

### 3.1.2. Кольцевой односвязный линейный список

Кольцевой односвязный линейный список - это линейный односвязный список, который имеет фиктивную голову. В таком списке последний элемент указывает на фиктивную голову, а голова, в свою очередь, указывает на первый элемент, создавая таким образом зацикленную структуру. Структура данных поддерживает операции такие как: вставка в начало, конец списка, вставка до и после элемента. Также присутствуют операции поиска, удаления из списка, проверка на пустоту, полноту.

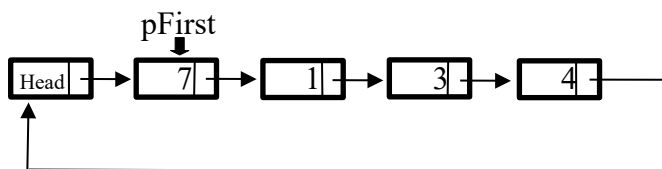
#### Операция вставки в начало

Функция расширяет функциональность родительского метода «вставки в начало» из структуры данных TList. После вызова родительского метода для вставки нового узла в начало списка, указатель следующего элемента после Head обновляется на pFirst, а указатель элемента, следующего после pLast, обновляется на pStop, что обеспечивает кольцевую структуру списка.



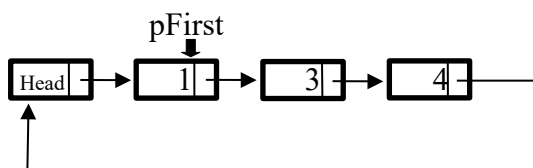
Добавим элемент 7.

Результат:



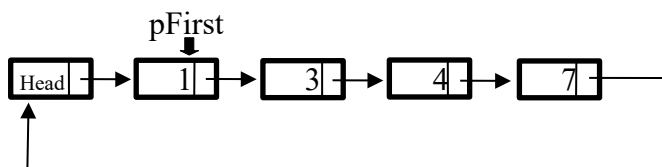
#### Операция вставки в конец

Функция расширяет функциональность родительского метода «вставки в конец» из структуры данных TList. После вызова родительского метода для вставки нового узла в конец списка, устанавливается указатель элемента, следующего после pLast на созданный узел, а затем сам указатель pLast обновляется до нового узла, чтобы поддерживать кольцевую структуру списка.



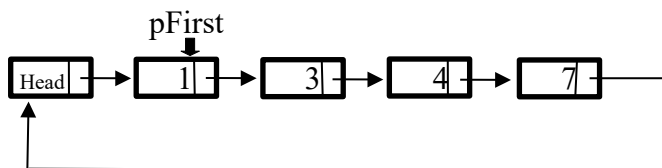
Добавим элемент 7 в конец.

Результат:



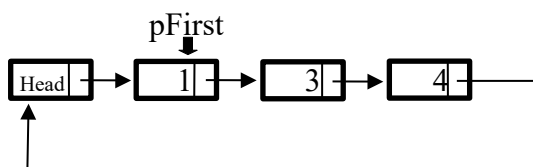
#### Операция удаления

Функция расширяет функциональность родительского метода «удаления» из структуры данных TList. После удаления узла с указанными данными из списка, метод проверяет, не является ли список пустым или единственным узлом. Если это не так, то обновляется указатель элемента, следующего после, pLast на Head, чтобы поддерживать кольцевую структуру списка.



Удалим 7.

Результат:



### 3.1.3. Моном

Тип данных TМоном представляет собой структуру, которая используется для представления мономов в математических выражениях. Моном имеет такие поля, как

коэффициент и степень. Степень может быть представлена в виде целого числа. Структура данных поддерживает операции сравнения.

#### **Операция сравнения больше**

Функция сравнивает два монома по степеням и возвращает true или false.

Пример:

$$2*x^3*y*z > 2*x^2*y^2*z$$

Функция вернет true, так как степень первого монома равна 311, а степень второго 221.

$$2*x^3*y*z > 2*x^4*y^2*z$$

Функция вернет false, так как степень первого монома равна 311, а степень второго 421.

#### **Операция сравнения меньше**

Функция сравнивает два монома по степеням и возвращает true или false.

Пример:

$$2*x^3*y*z < 2*x^2*y^2*z$$

Функция вернет false, так как степень первого монома равна 311, а степень второго 221.

$$2*x^3*y*z < 2*x^4*y^2*z$$

Функция вернет true, так как степень первого монома равна 311, а степень второго 421.

#### **Операция сравнения на равенство**

Функция сравнивает по степеням на равенство два монома.

Пример:

$$2*x^3*y*z == 2*x^2*y^2*z$$

Функция вернет false, так как степень первого монома равна 311, а степень второго 221.

$$2*x^4*y^2*z == 2*x^4*y^2*z$$

Функция вернет true, так как степень первого монома равна 421, и степень второго 421.

#### **Операция сравнения на неравенство**

Функция сравнивает по степеням на неравенство два монома.

Пример:

$$2*x^3*y*z != 2*x^2*y^2*z$$

Функция вернет true, так как степень первого монома равна 311, а степень второго 221.

$$2*x^4*y^2*z != 2*x^4*y^2*z$$

Функция вернет false, так как степень первого монома равна 421, и степень второго 421.

### 3.1.4. Полином

Программа предоставляет возможности для работы с полиномами на базе мономов: суммирование, разность, произведение, дифференцирование полиномов. Алгоритм на входе требует строку, которая представляет некоторый полином. Алгоритм допускает наличия трёх независимых переменных, положительные целые степени независимых переменных и вещественные коэффициенты.

#### Операция сложения полиномов

Данная функция реализует операцию сложения двух полиномов. Она создает новый список мономов, в который добавляет мономы из обоих полиномов, суммируя коэффициенты мономов с одинаковыми степенями. Если коэффициент суммы равен нулю, моном не добавляется в список. Затем функция добавляет оставшиеся мономы из обоих полиномов в список. Если в результате сложения получился пустой полином, функция добавляет в список моном с коэффициентом 0 и степенью 0. В конце функция возвращает новый полином, созданный из полученного списка мономов.

Пример:

$$2*x*y+3*z + 2*x^3*z-z = 2*z+2*x*y+2*x^3*z$$

#### Операция вычитания полиномов

Функция возвращает результат сложения текущего полинома со знаком инвертированного полинома, реализованного как умножение вычитаемого полинома на -1 и последующее сложение с текущим полиномом. Таким образом, операция вычитания полиномов сводится к операции сложения с инвертированным полиномом.

Пример:

$$2*x*y+3*z - 2*x^3*z-z = 4*z+2*x*y-2*x^3*z$$

#### Операция умножения полиномов

Эта функция реализует операцию умножения двух полиномов. Она создает новый список мономов и инициализирует флаг "flag" значением 0. Затем функция перебирает все мономы первого полинома и для каждого из них перебирает все мономы второго полинома. Для каждой пары мономов функция вычисляет произведение их коэффициентов и сумму их степеней. Затем функция ищет в списке мономов уже полученный ранее моном с такой же степенью и прибавляет к его коэффициенту полученное произведение коэффициентов. Если полученное произведение коэффициентов равно нулю, функция удаляет из списка моном с такой же степенью. Если такого монома в

списке нет, функция добавляет в список новый моном с полученным произведением коэффициентов и суммой степеней. В конце функция возвращает новый полином, созданный из полученного списка мономов. Флаг "flag" используется для отслеживания того, было ли получено ненулевое произведение коэффициентов в текущей итерации.

Пример:

$$2*x*y+3*z * 2*x^3*z-z = -3*z^2-2*x*y*z+6*x^3*z^2+2*x^4*y*z$$

#### **Операция взятия производной по x**

Эта функция реализует дифференцирование полинома по переменной "x". Она создает новый список мономов и перебирает все мономы исходного полинома. Для каждого монома функция вычисляет новый коэффициент, равный произведению коэффициента текущего монома на степень его переменной x, и новую степень, равную разности степени переменной "x" текущего монома и 1. Затем функция добавляет в список новый моном с полученным коэффициентом и степенью. Если степень переменной "x" текущего монома меньше 100, функция просто пропускает его. В конце функция возвращает новый полином, созданный из полученного списка мономов. Если все мономы в списке имеют степень переменной "x" меньше 1, функция добавляет в список моном с коэффициентом 0 и степенью 0.

Пример:

$$2*x^3*z-z = 6*x^2*z$$

#### **Операция взятия производной по y**

Эта функция реализует дифференцирование полинома по переменной "y". Она создает новый список мономов и перебирает все мономы исходного полинома. Для каждого монома функция вычисляет новый коэффициент, равный произведению коэффициента текущего монома на степень его переменной "y", и новую степень, равную разности степени переменной "y" текущего монома и 1. Затем функция добавляет в список новый моном с полученным коэффициентом и степенью. Если степень переменной y текущего монома меньше 10, функция просто пропускает его. В конце функция возвращает новый полином, созданный из полученного списка мономов. Если все мономы в списке имеют степень переменной y меньше 1, функция добавляет в список моном с коэффициентом 0 и степенью 0.

Пример:

$$2*x^3*y^4*z-6*x*y = 8*x^3*y^3*z-6*x$$

#### **Операция взятия производной по z**

Эта функция реализует дифференцирование полинома по переменной "z". Она создает новый список мономов и перебирает все мономы исходного полинома. Для

каждого монома функция вычисляет новый коэффициент, равный произведению коэффициента текущего монома на степень его переменной "z", и новую степень, равную разности степени переменной "z" текущего монома и 1. Затем функция добавляет в список новый моном с полученным коэффициентом и степенью. Если степень переменной "z" текущего монома меньше 1, функция просто пропускает его. В конце функция возвращает новый полином, созданный из полученного списка мономов. Если все мономы в списке имеют степень переменной "z" меньше 1, функция добавляет в список моном с коэффициентом 0 и степенью 0.

Пример:

$$2*x^3*y^4*z-z = 2*x^3*y^4-1$$

### Операция вычисления значения полинома

Функция принимает три аргумента - значения переменных x, y и z. Внутри функции создается словарь переменных, где ключами являются названия переменных, а значениями - соответствующие аргументам значения. Затем функция преобразует имя многочлена в постфиксную форму. Наконец, функция вычисляет значение многочлена в заданной точке, передавая постфиксную форму и словарь переменных вспомогательной функции Calculate. Результатом функции является вычисленное значение многочлена в заданной точке.

Пример:

$$x = 2, y = 3, z = 4$$

$$2*x^3*y^4*z-z = 5180$$

## 3.2.Описание программной реализации

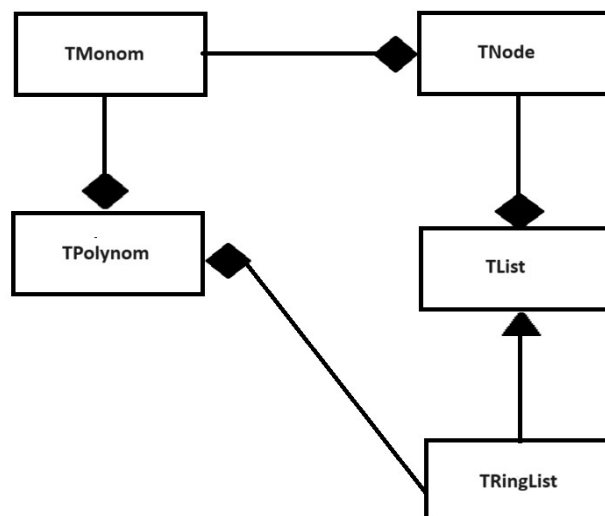


Рис. 6. Схема наследования классов



### 3.2.1. Описание структуры TNode

```
template <typename T>
struct TNode
{
    T data;
    TNode<T>* pNext;
    TNode() : data(), pNext(nullptr) {};
    TNode(const T& data_) : data(data_), pNext(nullptr) {};
    TNode(TNode<T>* pNext_) : data(), pNext(pNext_) {};
    TNode(const T& data_, TNode<T>* pNext_) : data(data_), pNext(pNext_) {};
    ~TNode() {};
};
```

Назначение: представление узла.

Поля:

**data**— указатель на массив типа **T**.

**pNext** — указатель на следующий элемент.

Методы:

**TNode()** ;

Назначение: инициализация значений по умолчанию.

Входные параметры: отсутствуют.

Выходные параметры: новый узел с инициализированными значениями.

**TNode(const T& data\_)** ;

Назначение: инициализация значения **data** узла.

Входные параметры: **data\_** — значение **data**.

Выходные параметры: новый узел с инициализированным значением **data**.

**TNode(TNode<T>\* pNext\_)** ;

Назначение: инициализация указателя на следующий узел.

Входные параметры: **pNext\_** — указатель на следующий узел.

Выходные параметры: новый узел с инициализированным указателем на следующий узел.

**TNode(const T& data\_, TNode<T>\* pNext\_)** ;

Назначение: инициализация значения **data** и указателя на следующий узел.

Входные параметры: **data\_** — значение **data**, **pNext\_** — указатель на следующий узел.

Выходные параметры: новый узел с инициализированными значениями **data** и указателем на следующий узел.

**~TNode()** ;

Назначение: освобождение выделенной памяти, если необходимо.

Входные параметры: отсутствуют.

Выходные параметры: память, выделенная для узла, освобождается.

### 3.2.2. Описание класса TList

```
template <typename T>
class TList
{
protected:
    TNode<T>* pFirst;
    TNode<T>* pPrev;
    TNode<T>* pCurr;
    TNode<T>* pStop;
    TNode<T>* pLast;
public:
    TList();
    TList(TNode<T>* _pFirst);
    TList(const TList<T>& list);
    ~TList();

    TNode<T>* Search(const T& data);
    virtual void InsertFirst(const T& data);
    virtual void InsertEnd(const T& data);
    void InsertAfter(const T& data, const T& beforesdata);
    void InsertBefore(const T& data, const T& nextdata);
    void InsertBeforeCurr(const T& data);
    void InsertAfterCurr(const T& data);
    virtual void Remove(const T& data);
    void Clear();
    int GetSize() const;
    bool IsEmpty() const;
    bool IsFull() const;
    virtual bool IsEnded() const;
    TNode<T>* GetCurrent() const;
    void Next();
    void Reset();
};
```

Назначение: представление списка.

Поля:

**pFirst** – указатель на первый элемент.

**pStop** – указатель на конец списка.

**pCurr** – указатель на текущий элемент.

**pPrev** – указатель на предыдущий элемент.

**pLast** – указатель на последний элемент.

Методы:

**TList();**

Назначение: создание пустого списка.

Входные параметры: отсутствуют.

Выходные параметры: новый объект класса **TList**.

```
TList(TNode<T>* _pFirst, TNode<T>* _pStop = nullptr) ;
```

Назначение: создание списка с заданным начальным узлом и, при необходимости, конечным узлом.

Входные параметры: **\_pFirst** – указатель на первый узел списка, **\_pStop** – указатель на конечный узел (по умолчанию **nullptr**).

Выходные параметры: новый объект класса **TList**.

```
TList(const TList<T>& list) ;
```

Назначение: создание копии существующего списка.

Входные параметры: **list** – существующий список для копирования.

Выходные параметры: новый объект класса **TList**, являющийся копией списка **list**.

```
~TList() ;
```

Назначение: освобождение памяти списка при удалении объекта.

Входные параметры: отсутствуют.

Выходные параметры: освобожденная память объекта класса **TList**.

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

Назначение: поиск узла с указанным значением.

Входные параметры: **data** – искомое значение.

Выходные параметры: указатель на узел с заданным значением, либо **nullptr**.

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

Назначение: вставляет новый узел с данными в начало списка.

Входные параметры: **data** – данные для нового узла.

Выходные параметры: отсутствуют.

```
virtual void InsertEnd(const T& data) ;
```

Назначение: вставляет новый узел с данными в конец списка.

Входные параметры: **data** – данные для нового узла.

Выходные параметры: отсутствуют.

```
void InsertAfter(const T& data, const T& beforedata) ;
```

Назначение: вставляет новый узел с данными после узла с определенными данными.

Входные параметры: **data** – данные для нового узла, **beforedata** – данные узла, после которого будет вставлен новый узел.

Выходные параметры: отсутствуют.

**void InsertBefore(const T& data, const T& nextdata);**

Назначение: вставляет новый узел с данными перед узлом с определенными данными.

Входные параметры: **data** – данные для нового узла, **nextdata** – данные узла, перед которым будет вставлен новый узел.

Выходные параметры: отсутствуют.

**void InsertBeforeCurr(const T& data);**

Назначение: вставляет новый узел с данными перед текущим узлом.

Входные параметры: **data** – данные для нового узла.

Выходные параметры: отсутствуют.

**void InsertAfterCurr(const T& data);**

Назначение: вставляет новый узел с данными после текущего узла.

Входные параметры: **data** – данные для нового узла.

Выходные параметры: отсутствуют.

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

Назначение: удаляет узел с определенными данными из списка.

Входные параметры: **data** – данные узла для удаления.

Выходные параметры: отсутствуют.

**void Clear();**

Назначение: очищает список, освобождает выделенную память.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

**int GetSize() const;**

Назначение: возвращает текущий размер списка.

Входные параметры: отсутствуют.

Выходные параметры: размер списка (целочисленное значение).

**bool IsEmpty() const;**

Назначение: проверяет, пуст ли список.

Входные параметры: отсутствуют.

Выходные параметры: true – если список пуст, false – в противном случае.

**bool IsFull() const;**

Назначение: проверяет, полон ли список.

Входные параметры: отсутствуют.

Выходные параметры: true – если список полон, false – в противном случае.

**bool IsEnded() const;**

Назначение: проверяет, достигли ли конца списка.

Входные параметры: отсутствуют.

Выходные параметры: true – если достигли, false – в противном случае.

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

Назначение: возвращает указатель на текущий узел.

Входные параметры: отсутствуют.

Выходные параметры: указатель на текущий узел.

`void Next();`

Назначение: переход к следующему узлу.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

`void Reset();`

Назначение: установка текущего узла как первого.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

### 3.2.3. Описание класса TRingList

```
template <typename T>
class TRingList : public TList<T>
{
private:
    TNode<T>* pHead;
public:
    TRingList();
    TRingList(TNode<T>* _pFirst);
    TRingList(const TList<T>& list);
    TRingList(const TRingList<T>& rlist);
    virtual ~TRingList();

    void InsertFirst(const T& data);
    void InsertEnd(const T& data);
    void Remove(const T& data);
    friend ostream& operator<<(ostream& os, const TRingList<T>& rl)
    {
        TNode<T>* cur = rl.pFirst;
        while (cur != rl.pHead)
        {
            os << cur->data << endl;
            cur = cur->pNext;
        }

        return os;
    }
};
```

Назначение: представление списка.

Поля:

**pHead** – указатель на головной элемент.

Методы:

**TRingList()** ;

Назначение: конструктор без параметров, создает пустой кольцевой список.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

**TRingList(TNode<T>\* \_pFirst)** ;

Назначение: создает кольцевой список с указанным начальным узлом.

Входные параметры: **\_pFirst** – указатель на начальный узел.

Выходные параметры: отсутствуют.

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

Назначение: конструктор копирования, создает копию существующего базового списка как кольцевой.

Входные параметры: **list** – ссылка на существующий базовый список.

Выходные параметры: отсутствуют.

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

Назначение: конструктор копирования, создает копию существующего кольцевого списка.

Входные параметры: **rlist** – ссылка на существующий кольцевой список.

Выходные параметры: отсутствуют.

**virtual ~TRingList()** ;

Назначение: виртуальный деструктор, освобождает выделенную память при уничтожении объектов производных классов.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

**void InsertFirst(const T& data)** ;

Назначение: вставляет новый узел с данными в начало списка.

Входные параметры: **data** – данные для нового узла.

Выходные параметры: отсутствуют.

**void InsertEnd(const T& data)** ;

Назначение: вставляет новый узел с данными в конец списка.

Входные параметры: **data** – данные для нового узла.

Выходные параметры: отсутствуют.

**void Remove(const T& data)** ;

Назначение: удаляет узел с определенными данными из списка.

Входные параметры: **data** – данные узла для удаления.

Выходные параметры: отсутствуют.

```
friend ostream& operator<<(ostream& os, const TRingList<T>& rl);
```

Назначение: оператор вывода для класса **TRingList**.

Входные параметры:

**os** – ссылка на объект типа **ostream**, который представляет выходной поток.

**rl** – ссылка на объект типа **TRingList** который будет выводиться.

Выходные параметры: ссылка на объект типа **ostream**.

### 3.2.4. Описание класса **TMonom**

```
struct TMonom
{
    double coef;
    int degree;

    TMonom();
    TMonom(const TMonom& m);
    TMonom(double c, int d);

    virtual bool operator > (const TMonom& d) const;
    virtual bool operator < (const TMonom& d) const;
    virtual bool operator == (const TMonom& d) const;
    virtual bool operator != (const TMonom& d) const;

    friend ostream& operator<<(ostream& os, const TMonom& m)
    {
        os << m.coef << " " << m.degree;
        return os;
    }
};
```

Назначение: представление монома.

Поля:

**coef** – коэффициент монома.

**degree** – степень монома.

Методы:

```
TMonom();
```

Назначение: конструктор по умолчанию, инициализирует объект **TMonom** с коэффициентом и степенью равными нулю.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

```
TMonom(const TMonom& m);
```

Назначение: конструктор копирования, создает копию существующего **TMonom**.

Входные параметры: **m** – ссылка на существующий объект **TMonom**.

Выходные параметры: отсутствуют.

```
TMonom(double c, int d);
```

Назначение: создает объект **TMonom** с указанным коэффициентом и степенью.

Входные параметры: **c** – коэффициент, **d** - степень.

Выходные параметры: отсутствуют.

```
virtual bool operator > (const TMonom& d) const;
```

Назначение: перегруженный оператор "больше". Сравнивает два объекта **TMonom** по убыванию степени.

Входные параметры: **d** – ссылка на объект **TMonom** для сравнения.

Выходные параметры: true, если текущий объект больше, иначе false.

```
virtual bool operator < (const TMonom& d) const;
```

Назначение: перегруженный оператор "меньше". Сравнивает два объекта **TMonom** по убыванию степени.

Входные параметры: **d** – ссылка на объект **TMonom** для сравнения.

Выходные параметры: true, если текущий объект меньше, иначе false.

```
virtual bool operator == (const TMonom& d) const;
```

Назначение: перегруженный оператор "равно". Проверяет равенство коэффициента и степени двух объектов **TMonom**.

Входные параметры: **d** – ссылка на объект **TMonom** для сравнения.

Выходные параметры: true, если объекты равны, иначе false.

```
virtual bool operator != (const TMonom& d) const;
```

Назначение: перегруженный оператор "не равно". Проверяет неравенство коэффициента и степени двух объектов **TMonom**.

Входные параметры: **d** – ссылка на объект **TMonom** для сравнения.

Выходные параметры: true, если объекты не равны, иначе false.

```
friend ostream& operator<<(ostream& os, const TMonom& m);
```

Назначение: оператор вывода для класса **TMonom**.

Входные параметры:

**os** – ссылка на объект типа **ostream**, который представляет выходной поток.

**m** – ссылка на объект типа **TMonom** который будет выводиться.

Выходные параметры: ссылка на объект типа **ostream**.

### 3.2.5. Описание класса **TPolynom**

```
class TPolynom
```

```
{
```

```
private:
```

```
    TRingList<TMonom>* monoms;
```

```
    string name;
```

```
    static map<string, double> symbolDict;
```

```
    void Parse_Polynom(const string& s);
```



```

public:
    TPolynom();
    TPolynom(const string& s);
    TPolynom(const TRingList<TMonom>& rlist);
    TPolynom(const TPolynom& p);
    ~TPolynom();

    TPolynom operator+(const TPolynom& p);
    TPolynom operator-(const TPolynom& p);
    TPolynom operator*(const TPolynom& p);
    TPolynom operator-() const;
    const TPolynom& operator=(const TPolynom& p);

    TPolynom difx() const;
    TPolynom dify() const;
    TPolynom difz() const;

    void HandleX(const string& str, int& i, string& deg);
    void HandleY(const string& str, int& i, string& deg);
    void HandleZ(const string& str, int& i, string& deg);
    void ProcessMonom(const string& numStr, const string& str, int& i, const
string& deg, int& flag, TRingList<TMonom>* monomList);
    void CreateAndInsertMonom(double koef, int degree, TRingList<TMonom>*
monomList);

    string ProcessDegreeZero(const string& str, const string& coef_str,
double& k);
    string ProcessDegreeNonZero(const string& str, const string& coef_str,
double& k, int& f);
    string ProcessDegreeX(const string& str, const string& coef_str, double&
k, int& d, int& flag);
    string ProcessDegreeY(const string& str, const string& coef_str, double&
k, int& d, int& flag);
    string ProcessDegreeZ(const string& str, const string& coef_str, double&
k, int& d, int& flag);

    double operator() (double _x, double _y, double _z);
    string ToString();
    friend ostream& operator<<(ostream& os, const TPolynom& p)
    {
        os << p.name << endl;
        os << *p.monoms << endl;
        return os;
    }

protected:
    string FilteredExpression(const string& s);
    bool isOperand(char c);
    bool isValidExpression(const string& expression);
    int Is_Symbol(string sm);
    bool Is_Number(const string& str);
};

```

Назначение: представление полинома.

Поля:

**monoms** – кольцевой линейный односвязный список.

**name** – строка полинома.

**symbolDict**- словарь значений переменных.

Методы:

**TPolynomial()** ;

Назначение: конструктор по умолчанию, создает объект **TPolynomial** с пустым списком мономов и пустым именем.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

**TPolynomial(const string s)** ;

Назначение: создает объект **TPolynomial** с указанным именем.

Входные параметры: **s** – строка, используемая в качестве имени полинома.

Выходные параметры: отсутствуют.

**TPolynomial(const TRingList& rlist)** ;

Назначение: создает объект **TPolynomial** на основе существующего кольцевого списка мономов.

Входные параметры: **rlist** – ссылка на кольцевой список мономов.

Выходные параметры: отсутствуют.

**TPolynomial(const TPolynomial& p)** ;

Назначение: конструктор копирования, создает копию существующего объекта **TPolynomial**.

Входные параметры: **p** – ссылка на существующий объект **TPolynomial**.

Выходные параметры: отсутствуют.

**~TPolynomial()** ;

Назначение: деструктор, освобождает память при уничтожении объекта **TPolynomial**.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

**TPolynomial operator+(const TPolynomial& p)** ;

Назначение: перегруженный оператор сложения полиномов.

Входные параметры: **p** – ссылка на объект **TPolynomial**.

Выходные параметры: объект **TPolynomial**, который является результатом сложения полиномов.

**TPolynomial operator-(const TPolynomial& p)** ;

Назначение: перегруженный оператор вычитания полиномов.

Входные параметры: **p** – ссылка на объект **TPolynomial**.

Выходные параметры: объект **TPolynomial**, который является результатом вычитания полиномов.

**TPolynomial operator\*(const TPolynomial& p)** ;

Назначение: перегруженный оператор умножения полиномов.

Входные параметры: **p** – ссылка на объект **TPolynomial**.

Выходные параметры: объект **TPolynomial**, который является результатом умножения полиномов.

**TPolynomial operator-() const;**

Назначение: перегруженный оператор умножения полиномов на -1.

Входные параметры: отсутствуют.

Выходные параметры: объект **TPolynomial**, который является результатом умножения полинома на -1.

**const TPolynomial& operator=(const TPolynomial& p);**

Назначение: перегруженный оператор присваивания.

Входные параметры: **p** – ссылка на объект **TPolynomial**, который присваивается текущему объекту.

Выходные параметры: копия объекта **TPolynomial** после присваивания.

**TPolynomial difx() const;**

Назначение: возвращает производную по переменной «x» текущего полинома.

Входные параметры: отсутствуют.

Выходные параметры: полином – производная по «x».

**TPolynomial dify() const;**

Назначение: возвращает производную по переменной «y» текущего полинома.

Входные параметры: отсутствуют.

Выходные параметры: полином – производная по «y».

**TPolynomial difz() const;**

Назначение: возвращает производную по переменной «z» текущего полинома.

Входные параметры: отсутствуют.

Выходные параметры: полином – производная по «z».

**double operator() (double \_x, double \_y, double \_z);**

Назначение: вычисляет значение полинома для заданных значений переменных «x», «y» и «z».

Входные параметры:

**\_x** – значение переменной «x»,

**\_y** – значение переменной «y»,

**\_z** – значение переменной «z».

Выходные параметры: значение полинома с заданными значениями переменных.

**void HandleX(const string& str, int& i, string& deg);**

Назначение: обрабатывает степень для переменной «x» в полиноме.

Входные параметры:

**str** – константная ссылка на строку, содержащую полином

**i** – ссылка на целое число, представляющее текущий индекс в строке

**deg** – ссылка на строку, представляющую степень для переменной «x»

Выходные параметры: отсутствуют.

```
void HandleY(const string& str, int& i, string& deg);
```

Назначение: обрабатывает степень для переменной «у» в полиноме.

Входные параметры:

**str** – константная ссылка на строку, содержащую полином

**i** – ссылка на целое число, представляющее текущий индекс в строке

**deg** – ссылка на строку, представляющую степень для переменной «у»

Выходные параметры: отсутствуют.

```
void HandleZ(const string& str, int& i, string& deg);
```

Назначение: обрабатывает степень для переменной «z» в полиноме.

Входные параметры:

**str** – константная ссылка на строку, содержащую полином,

**i** – ссылка на целое число, представляющее текущий индекс в строке,

**deg** – ссылка на строку, представляющую степень для переменной «z».

Выходные параметры: отсутствуют.

```
void ProcessMonom(const string& numStr, const string& str, int& i, const  
string& deg, int& flag, TRingList<TMonom>* monomList);
```

Назначение: обрабатывает моном в полиноме.

Входные параметры:

**numStr** – константная ссылка на строку, представляющую коэффициент монома,

**str** – константная ссылка на строку, содержащую полином,

**i** – ссылка на целое число, представляющее текущий индекс в строке,

**deg** – константная ссылка на строку, представляющую степень для текущей переменной,

**flag** – ссылка на целое число, представляющее флаг (0 или 1),

**monomList** – указатель на список мономов полинома.

Выходные параметры: отсутствуют.

```
void CreateAndInsertMonom(double koef, int degree, TRingList<TMonom>*  
monomList);
```

Назначение: создает и вставляет моном в список мономов полинома.

Входные параметры:

**koef** – вещественное число, представляющее коэффициент монома,

**degree** – целое число, представляющее степень монома,

**monomList** – указатель на список мономов полинома.

Выходные параметры: отсутствуют.

```
void Parse_Polynom(const string& s);
```

Назначение: разбирает строку, представляющую полином, и создает соответствующий список мономов.

Входные параметры: **s** – строка, представляющая полином.

Выходные параметры: отсутствуют.

```
string ProcessDegreeZero(const string& str, const string& coef_str, double& k);
```

Назначение: обрабатывает случай, когда степень равна нулю.

Входные параметры:

**str** – константная ссылка на строку, представляющую часть полинома без коэффициента,

**coef\_str** – константная ссылка на строку, представляющую коэффициент для данной степени,

**k** – ссылка на вещественное число, представляющее коэффициент.

Выходные параметры: **st** – строка, представляющая обработанный случай со степенью нуля.

```
string ProcessDegreeNonZero(const string& str, const string& coef_str, double& k, int& f);
```

Назначение: обрабатывает случай, когда степень не равна нулю.

Входные параметры:

**str** – константная ссылка на строку, представляющую часть полинома без коэффициента,

**coef\_str** – константная ссылка на строку, представляющую коэффициент для данной степени,

**k** – ссылка на вещественное число, представляющее коэффициент,

**f** – ссылка на целое число, представляющее флаг.

Выходные параметры: **st** – строка, представляющая обработанный случай с ненулевой степенью.

```
string ProcessDegreeX(const string& str, const string& coef_str, double& k, int& d, int& flag);
```

Назначение: обрабатывает степень для переменной «x» в полиноме.

Входные параметры:

**str** – константная ссылка на строку, представляющую часть полинома без коэффициента,

**coef\_str** – константная ссылка на строку, представляющую коэффициент для данной степени,

**k** – ссылка на вещественное число, представляющее коэффициент,

**d** – ссылка на целое число, представляющее степень для переменной «x»,

**flag** – ссылка на целое число, представляющее флаг.

Выходные параметры: **st** – строка, представляющая обработанный случай со степенью переменной «x».

```
string ProcessDegreeY(const string& str, const string& coef_str, double& k, int& d, int& flag);
```

Назначение: обрабатывает степень для переменной «y» в полиноме.

Входные параметры:

**str** – константная ссылка на строку, представляющую часть полинома без коэффициента,

**coef\_str** – константная ссылка на строку, представляющую коэффициент для данной степени,

**k** – ссылка на вещественное число, представляющее коэффициент,

**d** – ссылка на целое число, представляющее степень для переменной «y»,

**flag** – ссылка на целое число, представляющее флаг.

Выходные параметры: **st** – строка, представляющая обработанный случай со степенью переменной «y».

```
string ProcessDegreeZ(const string& str, const string& coef_str, double& k, int& d, int& flag);
```

Назначение: обрабатывает степень для переменной «z» в полиноме.

Входные параметры:

**str** – константная ссылка на строку, представляющую часть полинома без коэффициента,

**coef\_str** – константная ссылка на строку, представляющую коэффициент для данной степени,

**k** – ссылка на вещественное число, представляющее коэффициент,

**d** – ссылка на целое число, представляющее степень для переменной «z»,

**flag** – ссылка на целое число, представляющее флаг.

Выходные параметры: **st** – строка, представляющая обработанный случай со степенью переменной «z».

```
string ToString();
```

Назначение: возвращает строковое представление полинома.

Входные параметры: отсутствуют.

Выходные параметры: строковое представление полинома.

```
string FilteredExpression(const string& s);
```

Назначение: возвращает отфильтрованное выражение.

Входные параметры: **s** – исходная строка выражения.

Выходные параметры: отфильтрованное выражение.

**bool isOperand(char c);**

Назначение: проверяет, является ли символ операндом.

Входные параметры: **c** – символ для проверки.

Выходные параметры: логическое значение, указывающее, является ли символ операндом.

**bool isValidExpression(const string& expression);**

Назначение: проверяет, является ли строка допустимым математическим выражением.

Входные параметры: **expression** – строковое выражение.

Выходные параметры: логическое значение, указывающее, является ли выражение допустимым.

**int Is\_Symbol(string sm);**

Назначение: определяет, является ли переданный символ оператором.

Входные параметры: **sm** – символ для проверки.

Выходные параметры: 1 – символ является оператором, 2 – скобка, иначе 0.

**bool Is\_Number(const string& str);**

Назначение: проверяет, является ли строка числом.

Входные параметры: **str** – строка для проверки.

Выходные параметры: логическое значение, указывающее, является ли строка числом.

**friend ostream& operator<<(ostream& os, const TPolynom& p);**

Назначение: оператор вывода для класса **TPolynom**.

Входные параметры:

**os** – ссылка на объект типа **ostream**, который представляет выходной поток.

**m** – ссылка на объект типа **TPolynom** который будет выводиться.

Выходные параметры: ссылка на объект типа **ostream**.

## Заключение

В рамках данной лабораторной работы была разработана и реализована структура данных для работы с полиномами. Были созданы классы `TMonom` и `TPolynom`, предоставляющие функционал для работы с мономами и полиномами соответственно.

Класс `TMonom` содержит информацию о коэффициенте и степени монома, а также определены операторы сравнения для сравнения мономов по степени.

Класс `TPolynom` осуществляет работу с полиномами через список мономов. Реализованы операторы сложения, вычитания и умножения полиномов, а также оператор присваивания. Кроме того, класс предоставляет функционал для вычисления значения полинома для заданных значений переменных, а также для нахождения частных производных по каждой из переменных.

Таким образом, результатом выполнения лабораторной работы стала реализация структуры данных для работы с полиномами, позволяющей удобно и эффективно выполнять различные операции над ними, а также проводить анализ и вычисления, необходимые в контексте математических вычислений.



## Литература

1. Линейные списки: эффективное и удобное хранение данных  
[<https://nauchniestati.ru/spravka/hranenie-dannyh-s-ispolzovaniem-linejnyh-spiskov/?ysclid=ltwvp5uwda145425180>]
2. Лекция «Списковые структуры хранения» Сысоева А.В.  
[<https://cloud.unn.ru/s/x33MEa9on8HgNgw>]
3. Лекция «Списки в динамической памяти» Сысоева А.В.  
[<https://cloud.unn.ru/s/rCiKGSX33SSGPi4>]
4. Лекция «Полиномы» Сысоева А.В. [ <https://cloud.unn.ru/s/t6o9kp5g9bpf2yz> ]

## Приложения

### Приложение А. Реализация структуры TNode

```
template <typename T>
struct TNode
{
    T data;
    TNode<T>* pNext;
    TNode() : data(), pNext(nullptr) {};
    TNode(const T& data_) : data(data_), pNext(nullptr) {};
    TNode(TNode<T>* pNext_) : data(), pNext(pNext_) {};
    TNode(const T& data_, TNode<T>* pNext_) : data(data_), pNext(pNext_) {};
    ~TNode() {};
};
```

### Приложение Б. Реализация класса TList

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

template <typename T>
TList<T>::TList(TNode<T>* _pFirst)
{
    pFirst = _pFirst;
    pStop = nullptr;
    pPrev = nullptr;
    if (pFirst == pStop)
    {
        pLast = pStop;
        pCurr = pStop;
        return;
    }
    pCurr = pFirst;
    TNode<T>* tmp = pFirst;
    while (tmp->pNext != pStop)
        tmp = tmp->pNext;
    pLast = tmp;
    pStop = pLast->pNext;
}

template <typename T>
TList<T>::TList(const TList<T>& list)
{
    if (list.pFirst == nullptr)
        return;
    pFirst = new TNode<T>(list.pFirst->data);
    TNode<T>* tmp = list.pFirst->pNext;
    TNode<T>* tmp2 = pFirst;
    while (tmp != list.pStop)
    {
        tmp2->pNext = new TNode<T>(tmp->data);
        tmp2 = tmp2->pNext;
        tmp = tmp->pNext;
    }
    pPrev = nullptr;
    pCurr = pFirst;
    pLast = tmp2;
}
```

```

        pStop = nullptr;
    }
    template <typename T>
    void TList<T>::Clear()
    {
        TNode<T>* tmp = pFirst;
        while (tmp != pStop)
        {
            pFirst = pFirst->pNext;
            delete tmp;
            tmp = pFirst;
        }
        pCurr = pPrev = pStop = pLast = nullptr;
    }
    template <typename T>
    TList<T>::~~TList()
    {
        Clear();
    }
    template <typename T>
    TNode<T>* TList<T>::Search(const T& data)
    {
        TNode<T>* tmp = pFirst;
        while (tmp != pStop && tmp->data != data)
            tmp = tmp->pNext;
        return tmp;
    }
    template <typename T>
    void TList<T>::InsertFirst(const T& data)
    {
        TNode<T>* pNode = new TNode<T>(data);
        if (pFirst == pStop)
            pLast = pNode;
        pNode->pNext = pFirst;
        pFirst = pNode;
        pCurr = pFirst;
    }
    template <typename T>
    void TList<T>::InsertEnd(const T& data)
    {
        if (pFirst == pStop)
        {
            InsertFirst(data);
            return;
        }
        TNode<T>* pNode = new TNode<T>(data, pStop);
        pLast->pNext = pNode;
        pLast = pNode;
    }

    template <typename T>
    void TList<T>::InsertAfter(const T& data, const T& beforedata)
    {
        TNode<T>* pPrev = Search(beforedata);
        if (pPrev != pStop)
        {
            if (pPrev->pNext == pStop)
            {
                InsertEnd(data);
                return;
            }
            TNode<T>* pNode = new TNode<T>(data);
            pNode->pNext = pPrev->pNext;

```

```

        pPrev->pNext = pNode;
    }
    else
    {
        string msg = "Element not found!";
        throw msg;
    }
}

template <typename T>
void TList<T>::InsertBefore(const T& data, const T& nextdata)
{
    if (pFirst != pStop && pFirst->data == nextdata)
        InsertFirst(data);
    else
    {
        TNode<T>* tmp = pFirst;
        pPrev = nullptr;
        while (tmp != pStop && tmp->data != nextdata)
        {
            pPrev = tmp;
            tmp = tmp->pNext;
        }
        if (tmp != pStop)
        {
            TNode<T>* pNode = new TNode<T>(data);
            pNode->pNext = tmp;
            pPrev->pNext = pNode;
        }
        else
        {
            string msg = "Element not found!";
            throw msg;
        }
    }
}

template <typename T>
void TList<T>::InsertBeforeCurr(const T& data)
{
    InsertBefore(data, pCurr->data);
}

template <typename T>
void TList<T>::InsertAfterCurr(const T& data)
{
    InsertAfter(data, pCurr->data);
}

template <typename T>
void TList<T>::Remove(const T& data)
{
    if (IsEmpty())
    {
        string msg = "Error";
        throw msg;
    }
    TNode<T>* pNode = pFirst;
    pPrev = nullptr;
    while (pNode->pNext != pStop && pNode->data != data)
    {
        pPrev = pNode;
        pNode = pNode->pNext;
    }
    if (pNode->pNext == pStop && pNode->data != data)
    {
        string msg = "Error";
    }
}

```

```

        throw msg;
    }
    if (pPrev == nullptr)
    {
        pFirst = pFirst->pNext;
        pCurr = pFirst;
        delete pNode;
        return;
    }
    if (pNode->pNext == pStop)
    {
        pLast = pPrev;
    }
    pPrev->pNext = pNode->pNext;
    delete pNode;
}

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

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

template <typename T>
bool TList<T>::IsFull() const
{
    return !IsEmpty();
}

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

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

template <typename T>
void TList<T>::Next()
{
    if (pCurr != pStop)
        pCurr = pCurr->pNext;
}

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

```

## Приложение В. Реализация класса TRingList

```

template <typename T>
TRingList<T>::TRingList() : TList<T>()
{

```

```

        pHead = new TNode<T>();
        pHead->pNext = pHead;
        pFirst = nullptr;
        pStop = nullptr;
    }
    template <typename T>
    TRingList<T>::TRingList(TNode<T>* _pFirst) : TList<T>(_pFirst)
    {
        pHead = new TNode<T>();
        pHead->pNext = pFirst; // todo: вызов конструктора есть
        pStop = pHead;
        pLast->pNext = pStop;
    }
    template <typename T>
    TRingList<T>::TRingList(const TList<T>& list) : TList<T>(list)
    {
        pHead = new TNode<T>();
        pHead->pNext = pFirst;
        pLast->pNext = pHead;
        pStop = pHead;
    }
    template <typename T>
    TRingList<T>::TRingList(const TRingList<T>& rlist) : TList<T>(rlist)
    {
        if (rlist.pFirst == nullptr)
            return;
        pHead = new TNode<T>();
        pHead->pNext = pFirst;
        pLast->pNext = pHead;
        pStop = pHead;
    }
    template <typename T>
    TRingList<T>::~~TRingList()
    {
        if (pStop)
            delete pStop;
    }
    template <typename T>
    void TRingList<T>::InsertFirst(const T& data)
    {
        TList<T>::InsertFirst(data);
        pHead->pNext = pFirst;
        pStop = pHead;
        pLast->pNext = pStop;
    }
    template <typename T>
    void TRingList<T>::InsertEnd(const T& data)
    {
        if (pFirst == pStop)
        {
            InsertFirst(data);
            return;
        }
        pLast->pNext = new TNode<T>(data, pHead);
        pLast = pLast->pNext;
    }
    template <typename T>
    void TRingList<T>::Remove(const T& data)
    {
        TList<T>::Remove(data);
        if (pFirst != pStop && pFirst != nullptr)
        {

```

```

        pLast->pNext = pHead;
    }
}

```

## Приложение Г. Реализация класса TMonom

```

TMonom::TMonom()
{
    coef = 0.0;
    degree = -1;
};
TMonom::TMonom(double c, int d)
{
    coef = c;
    degree = d;
};
TMonom::TMonom(const TMonom& m)
{
    coef = m.coef;
    degree = m.degree;
};
bool TMonom::operator<(const TMonom& m) const
{
    if (degree < m.degree)
        return true;
    else
        return false;
};
bool TMonom::operator>(const TMonom& m) const
{
    if (degree > m.degree)
        return true;
    else
        return false;
};
bool TMonom::operator==(const TMonom& m) const
{
    if (degree == m.degree)
        return true;
    else
        return false;
};
bool TMonom::operator!=(const TMonom& m) const
{
    if (degree != m.degree)
        return true;
    else
        return false;
}

```

## Приложение Д. Реализация класса TPolynom

```

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

TPolynom::TPolynom(const string& s)
{
    monoms = new TRingList<TMonom>();
    Parse_Polynom(s);
}

```

```

        name = ToString();
    }

TPolynom::TPolynom(const TRingList<TMonom>& rlist)
{
    monoms = new TRingList<TMonom>(rlist);
    name = ToString();
}

TPolynom::TPolynom(const TPolynom& p)
{
    name = p.name;
    monoms = new TRingList<TMonom>(*p.monoms);
}

TPolynom::~~TPolynom()
{
    delete monoms;
    name = "";
}

TPolynom TPolynom::operator-() const
{
    TRingList<TMonom> list;
    monoms->Reset();
    while (!monoms->IsEnded())
    {
        TMonom m = monoms->GetCurrent()->data;
        m.coef *= -1;
        list.InsertEnd(m);
        monoms->Next();
    }
    return TPolynom(list);
}

TPolynom TPolynom::operator+(const TPolynom& p)
{
    TRingList<TMonom> list;
    monoms->Reset();
    p.monoms->Reset();

    while (!monoms->IsEnded() && !p.monoms->IsEnded())
    {
        TMonom m1 = monoms->GetCurrent()->data;
        TMonom m2 = p.monoms->GetCurrent()->data;

        if (m1 == m2)
        {
            double k = m1.coef;
            double k2 = m2.coef;
            double k3 = k + k2;

            if (k3 != 0)
            {
                m2.coef = k3;
                list.InsertEnd(m2);
            }

            monoms->Next();
            p.monoms->Next();
        }
        else if (m1 > m2)
        {
            list.InsertEnd(m2);

```



```

        p.monoms->Next();
    }
    else
    {
        list.InsertEnd(m1);
        monoms->Next();
    }
}
while (!monoms->IsEnded())
{
    list.InsertEnd(monoms->GetCurrent()->data);
    monoms->Next();
}
while (!p.monoms->IsEnded())
{
    list.InsertEnd(p.monoms->GetCurrent()->data);
    p.monoms->Next();
}
if (list.IsEmpty())
{
    TMonom m(0, 0);
    list.InsertEnd(m);
}
return TPolynom(list);
}
TPolynom TPolynom::operator-(const TPolynom& p)
{
    return *this + (-p);
}
TPolynom TPolynom::operator*(const TPolynom& p)
{
    TRingList<TMonom> list;
    int flag = 0;
    monoms->Reset();
    p.monoms->Reset();
    while (!monoms->IsEnded())
    {
        TMonom m = monoms->GetCurrent()->data;
        p.monoms->Reset();
        while (!p.monoms->IsEnded())
        {
            TMonom m2 = p.monoms->GetCurrent()->data;
            double k = m.coef;
            double k2 = m2.coef;
            double k3 = k * k2;
            int d = m.degree;
            int d2 = m2.degree;
            int deg = d + d2;
            TMonom mon(k3, deg);
            bool inserted = false;
            list.Reset();

            while (!list.IsEnded())
            {
                TMonom m_curr = list.GetCurrent()->data;
                int curr_d = m_curr.degree;
                if (deg == curr_d)
                {
                    double k_curr = m_curr.coef;
                    k3 = k_curr + k3;
                    if (k3 != 0)
                        flag = 1;
                    else

```

```

        {
            list.Remove(m_curr);
            inserted = true;
            break;
        }
    }
    if (deg <= list.GetCurrent()->data.degree)
    {
        if (flag == 1)
        {
            TMonom mon_new(k3, deg);
            TMonom m_del = list.GetCurrent()->data;
            list.Remove(m_del);
            if (list.GetSize() == 0)
                list.InsertFirst(mon_new);
            else
                list.InsertAfterCurr(mon_new);
        }
        else
            list.InsertBeforeCurr(mon);
        inserted = true;
        break;
    }
    list.Next();
}
if (!inserted)
    list.InsertEnd(mon);

p.monoms->Next();
}
monoms->Next();
}
return TPolynom(list);
}
const TPolynom& TPolynom::operator=(const TPolynom& p)
{
    if (this == &p)
        return *this;
    name = p.name;
    TRingList<TMonom>* list = new TRingList<TMonom>();
    p.monoms->Reset();
    while (!p.monoms->IsEnded())
    {
        list->InsertEnd(p.monoms->GetCurrent()->data);
        p.monoms->Next();
    }
    delete monoms;
    this->monoms = list;
    return *this;
}
TPolynom TPolynom::difx() const
{
    TRingList<TMonom> list;
    monoms->Reset();
    while (!monoms->IsEnded())
    {
        TMonom m = monoms->GetCurrent()->data;
        double k = m.coef;
        int d = m.degree;
        if (d >= 100)
        {
            int d0 = d / 100;
            d -= 100;

```

```

        k *= d0;
        TMonom newM(k, d);
        list.InsertEnd(newM);
    }
    monoms->Next();
}
if (list.IsEmpty())
{
    TMonom m(0, 0);
    list.InsertEnd(m);
}
return TPolynom(list);
}
TPolynom TPolynom::dify() const
{
    TRingList<TMonom> list;
    monoms->Reset();
    while (!monoms->IsEnded())
    {
        TMonom m = monoms->GetCurrent()->data;
        double k = m.coef;
        int d = m.degree;
        int intdeg = d / 100;
        d = d % 100;
        if (d >= 10)
        {
            int d0 = d / 10;
            d -= 10;
            k *= d0;
            d += intdeg * 100;
            TMonom newM(k, d);
            list.InsertEnd(newM);
        }
        monoms->Next();
    }
    if (list.IsEmpty())
    {
        TMonom m(0, 0);
        list.InsertEnd(m);
    }
    return TPolynom(list);
}
TPolynom TPolynom::difz() const
{
    TRingList<TMonom> list;
    monoms->Reset();
    while (!monoms->IsEnded())
    {
        TMonom m = monoms->GetCurrent()->data;
        double k = m.coef;
        int d = m.degree;
        int intdeg = d / 10;
        d = d % 10;
        if (d >= 1)
        {
            int d0 = d;
            d -= 1;
            k *= d0;
            d += intdeg * 10;
            TMonom newM(k, d);
            list.InsertEnd(newM);
        }
        monoms->Next();
    }
}

```

```

    }
    if (list.IsEmpty())
    {
        TMonom m(0, 0);
        list.InsertEnd(m);
    }
    return TPolynom(list);
}
double TPolynom::operator()(double _x, double _y, double _z)
{
    string pol_name = name;
    map<string, double> variableDict = {
        {"x", _x},
        {"y", _y},
        {"z", _z},
    };
    TStack<string> st = ArithmeticExpression::Postfix_Form(pol_name);
    return ArithmeticExpression::Calculate(st, variableDict);
}
string TPolynom::FilteredExpression(const string& s)
{
    string filteredExpression = "";
    int l = s.length();
    for (int i = 0; i < l; i++)
    {
        char c = s[i];
        if (c != ' ')
        {
            filteredExpression += c;
        }
    }
    return filteredExpression;
}
bool TPolynom::isOperand(char c)
{
    return ((c >= 'x' && c <= 'z') || (c >= '0' && c <= '9'));
}
bool TPolynom::Is_Number(const string& str)
{
    for (int i = 0; i < str.length(); i++)
    {
        char c = str[i];
        if (!isdigit(c)) {
            return false; // если встречен не цифровой символ,
возвращаем false
        }
    }
    return true;
}
int TPolynom::Is_Symbol(string sm)
{
    if (TPolynom::symbolDict.find(sm) != TPolynom::symbolDict.end())
        return 1;
    else if (sm == "(" || sm == ")")
        return 2;
    return 0;
}
bool TPolynom::IsValidExpression(const string& expression)
{
    int k1 = 0, k2 = 0;
    int l = expression.length();
    for (int i = 0; i < l; i++)
    {

```

```

char c = expression[i];
string s(1, c);
if (i == 0)
{
    if (s == "-")
    {
        string num = "";
        char c1 = expression[i + 1];
        if (isOperand(c1))
            continue;
        else
            return false;
    }
    else if (s == "+" || s == "*" || s == "/")
        return false;
}
if (s == "^")
{
    char c2 = expression[i + 1];
    string s2(1, c2);
    if (c2 == 'x' || c2 == 'y' || c2 == 'z' || Is_Symbol(s2))
        return false;
}
if (s == "(" || s == ")")
{
    if (s == "(")
        k1++;
    else
    {
        k2++;
        char c1 = expression[i + 1];
        string s1(1, c1);
        if (isOperand(c1) || c1 == '(')
            return false;
    }
}
if (s == ".")
{
    char c2 = expression[i + 1];
    string s2(1, c2);
    if (!Is_Number(s2))
        return false;
}
else if ((Is_Symbol(s) != 0) || isOperand(c))
{
    char c1 = expression[i + 1];
    string s1(1, c1);
    if (Is_Symbol(s) == 1)
    {
        if (Is_Symbol(s1) == 1)
            return false;
        if (i == 1 - 1)
            return false;
    }
    if ((isOperand(c)) && (s1 == "("))
        return false;
    continue;
}
else
    return false; // обнаружен недопустимый символ
}
if (k1 != k2)
    return false;

```

```

        return true;
    }
    void TPolynom::HandleX(const string& str, int& i, string& deg)
    {
        i++;
        if (str[i] == '^')
        {
            i++;
            char k = str[i];
            int n = k - '0';
            n = n * 100;
            int N = stoi(deg);
            n += N;
            deg = to_string(n);
            i++;
        }
        else
        {
            int N = stoi(deg);
            N += 100;
            deg = to_string(N);
        }
    }
    void TPolynom::HandleY(const string& str, int& i, string& deg)
    {
        i++;
        if (str[i] == '^')
        {
            i++;
            char k = str[i];
            int n = k - '0';
            int N = stoi(deg);
            N += n * 10;
            string y = to_string(N);
            deg = y;
            i++;
        }
        else
        {
            int N = stoi(deg);
            N += 10;
            string y = to_string(N);
            deg = y;
        }
    }
    void TPolynom::HandleZ(const string& str, int& i, string& deg)
    {
        i++;
        if (str[i] == '^')
        {
            i++;
            char k = str[i];
            int n = k - '0';
            int N = stoi(deg);
            N += n;
            string z = to_string(N);
            deg = z;
            i++;
        }
        else
        {
            int N = stoi(deg);
            N += 1;
        }
    }

```

```

        string z = to_string(N);
        deg = z;
    }
}

void TPolynom::CreateAndInsertMonom(double koef, int degree,
TRingList<TMonom>* monomList) {
    TMonom monom(koef, degree);
    if (monomList->IsEmpty())
        monomList->InsertFirst(monom);
    else
    {
        monomList->Reset();
        while (!monomList->IsEnded())
        {
            TMonom m = monomList->GetCurrent()->data;
            int deg = m.degree;
            int deg2 = monom.degree;
            if (deg2 < deg)
            {
                monomList->InsertBeforeCurr(monom);
                break;
            }
            else if (deg2 == deg)
            {
                double k = m.coef;
                double k2 = monom.coef;
                double k3 = k + k2;
                if (k3 == 0)
                    return;
                TMonom new_m(k3, deg);
                monomList->InsertAfterCurr(new_m);
                monomList->Remove(m);
                break;
            }
            monomList->Next();
        }
        if (monomList->IsEnded())
            monomList->InsertEnd(monom);
    }
}

void TPolynom::ProcessMonom(const string& numStr, const string& str, int& i,
const string& deg, int& flag, TRingList<TMonom>* monomList)
{
    int degree = stoi(deg);
    double koef = 0.0;
    if (numStr == "")
        koef = 1.0;
    else
        koef = stod(numStr);
    if (flag == 0)
        koef = -koef;
    if (koef != 0.0)
        CreateAndInsertMonom(koef, degree, monomList);
    if (str[i] == '-')
        flag = 0;
    else
        flag = 1;
}

void TPolynom::Parse_Polynom(const string& s) // ??? разбить на функции
{
    string str = FilteredExpression(s);
    if (!isValidExpression(str))
    {

```

```

        string msg = "Input error";
        throw msg;
    }
    TRingList<TMonom>* monomList = new TRingList<TMonom>();
    int flag = 1;
    for (int i = 0; i < str.length(); i++)
    {
        string numStr = "";
        string deg = "0";
        if (str[i] == '-')
        {
            flag = 0;
            i++;
        }
        while (str[i] != '+' && str[i] != '-' && i != str.length())
        {
            char s1 = str[i];
            string s(1, s1);
            if (isdigit(s1) || s == ".")
            {
                numStr += s;
                i++;
            }
            else
            {
                if (isOperand(str[i]))
                {
                    if (str[i] == 'x')
                        HandleX(str, i, deg);
                    if (str[i] == 'y')
                        HandleY(str, i, deg);
                    if (str[i] == 'z')
                        HandleZ(str, i, deg);
                }
                if (str[i] == '*' || str[i] == '/')
                    i++;
            }
        }
        ProcessMonom(numStr, str, i, deg, flag, monomList);
    }
    this->monoms = monomList;
}

string TPolynom::ProcessDegreeZero(const string& str, const string& coef_str,
double& k)
{
    string st = str;
    if (st == "")
        st += coef_str;
    else
        if (k > 0)
            st += "+" + coef_str;
        else
            st += coef_str;
    return st;
}

string TPolynom::ProcessDegreeNonZero(const string& str, const string&
coef_str, double& k, int& f)
{
    string st = str;
    if (coef_str == "1" || coef_str == "-1")
        f = 1;
    if (k > 0)
    {

```



```

        if (coef_str == "1")
        {
            if (st != "")
                st += "+";
        }
        else
        {
            if (st != "")
                st += "+" + coef_str;
            else
                st += coef_str;
        }
    }
    else
    {
        if (coef_str == "-1")
            st += "-";
        else
            st += coef_str;
    }
    return st;
}

string TPolynom::ProcessDegreeX(const string& str, const string& coef_str,
double& k, int& d, int& flag)
{
    int f = 0;
    string st = ProcessDegreeNonZero(str,coef_str,k, f);

    int deg_x = d / 100;
    if (f == 0)
    {
        if (deg_x == 1)
            st += "*x";
        else
            st += "*x^" + to_string(deg_x);
    }
    else
    {
        if (deg_x == 1)
            st += "x";
        else
            st += "x^" + to_string(deg_x);
    }
    d = d % 100;
    flag++;
    return st;
}

string TPolynom::ProcessDegreeY(const string& str, const string& coef_str,
double& k, int& d, int& flag)
{
    string st = str;
    int f = 0;
    if (flag == 0)
        st = ProcessDegreeNonZero(str, coef_str, k, f);

    int deg_y = d / 10;
    if (f == 0)
    {
        if (deg_y == 1)
            st += "*y";
        else
            st += "*y^" + to_string(deg_y);
    }
}

```

```

    }
    else
    {
        if (deg_y == 1)
            st += "y";
        else
            st += "y^" + to_string(deg_y);
    }
    d = d % 10;
    flag++;
    return st;
}

string TPolynom::ProcessDegreeZ(const string& str, const string& coef_str,
double& k, int& d, int& flag)
{
    string st = str;
    int f = 0;
    if (flag == 0)
        st = ProcessDegreeNonZero(str, coef_str, k, f);

    int deg_z = d;
    if (f == 0)
    {
        if (deg_z == 1)
            st += "*z";
        else
            st += "*z^" + to_string(deg_z);
    }
    else
    {
        if (deg_z == 1)
            st += "z";
        else
            st += "z^" + to_string(deg_z);
    }
    d = 0;
    return st;
}

string TPolynom::ToString()
{
    string st = "";
    monoms->Reset();
    while (!monoms->IsEnded())
    {
        TMonom m = monoms->GetCurrent()->data;
        double k = m.coef;
        int d = m.degree;
        int flag = 0;

        ostringstream oss;
        oss << fixed << setprecision(8) << k; // Устанавливаем точность
        string coef_str = oss.str();
        coef_str.erase(coef_str.find_last_not_of('0') + 1,
string::npos); // Удаляем конечные нули
        if (coef_str.back() == '.') {
            coef_str.pop_back(); // Удаляем десятичную точку, если она
осталась в конце
        }
    }
}

```

```

        if (d == 0)
        {
            st = ProcessDegreeZero(st, coef_str, k);
        }
        if (d >= 100)
        {
            st = ProcessDegreeX(st, coef_str, k, d, flag);
        }
        if (d >= 10)
        {
            st = ProcessDegreeY(st, coef_str, k, d, flag);
        }
        if (d >= 1)
        {
            st = ProcessDegreeZ(st, coef_str, k, d, flag);
        }

        monoms->Next();
    }
    return st;
}

```