

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

_____ / Савченко М.П./
Подпись

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

Подпись

Нижний Новгород
2024

Содержание

Введение.....	4
1 Постановка задачи.....	5
2 Руководство пользователя.....	6
2.1 Приложение для демонстрации работы связного списка	6
2.2 Приложение для демонстрации работы полиномов.....	9
3 Руководство программиста	11
3.1 Описание алгоритмов	11
3.1.1 Связный список	11
3.1.2 Полиномы	14
3.2 Описание программной реализации	21
3.2.1 Описание класса TNode	21
3.2.2 Описание класса TList	23
3.2.3 Описание класса TRingList.....	27
3.2.4 Описание класса TMonom.....	31
3.2.5 Описание класса TPolynom	36
Заключение	43
Литература	44
Приложения	45
Приложение А. Реализация класса TNode.....	45
Приложение Б. Реализация класса TList.....	46
Приложение В. Реализация класса TRingList.....	52
Приложение Г. Реализация класса TMonom	56
Приложение Д. Реализация класса TPolynom	61

Введение

Полиномы являются важным математическим объектом, широко применяемым в различных областях науки и техники. Они используются для аппроксимации функций, решения уравнений, моделирования и анализа данных. В данной лабораторной работе мы рассматриваем полиномы от трех переменных x, y и z , которые представляют собой алгебраические выражения, составленные из суммы и произведения степенных функций переменных. Для удобства манипуляции и вычислений с такими полиномами необходимо реализовать соответствующую структуру данных.

В данном отчете представлена реализация класса, описывающего полиномы от трех переменных на языке программирования C++ с использованием связных списков. Класс позволяет удобно создавать, хранить, модифицировать и вычислять значения полиномов. Использование связных списков обеспечивает гибкую структуру хранения и управления полиномами любой степени и сложности.

1 Постановка задачи

Цель лабораторной работы:

Целью данной лабораторной работы является разработка и реализация класса, описывающего полиномы от трех переменных (x , y , z) на основе односвязных списков на языке программирования C++. Разработанный класс должен предоставлять удобные средства для работы с полиномиальными выражениями, включая их создание, модификацию и выполнение операций над ними.

Задачи лабораторной работы:

1. Спроектировать и описать структуру звеньев односвязного списка, которая будет использоваться для представления мономов.
2. Разработать класс односвязного списка, который будет содержать методы для добавления, удаления и обхода элементов списка.
3. Создать класс монома, который будет представлять отдельные члены полинома и содержать информацию о степени переменных и их коэффициенте.
4. Реализовать операции для класса монома, такие как сложение, вычитание и умножение, взятие производной по переменной.
5. Разработать класс полинома, который будет содержать методы для создания, модификации и вычисления полиномиальных выражений.
6. Обеспечить возможность выполнения арифметических операций над полиномами, таких как сложение, вычитание и умножение.
7. Провести тестирование разработанных классов на различных входных данных, чтобы убедиться в их корректности и эффективности.
8. Описать процесс разработки, принятые решения и особенности реализации в отчете по лабораторной работе.

2 Руководство пользователя

2.1 Приложение для демонстрации работы связного списка

1. Запустите приложение с названием `sample_tringlist.exe`. В результате появится окно, показанное ниже (рис. 1). На выбор дается следующие действия: **Insert** – вставка элементов в список; **remove** – удаление элемента из списка; **clear** – удалить все элементы из списка; **next element** – перейти на следующий элемент списка; **reset list** – перейти на первый элемент списка; **EXIT** – выход из программы. Current показывает на каком элементе списка вы сейчас находитесь. Для выбора действия нажмите соответствующую клавишу.

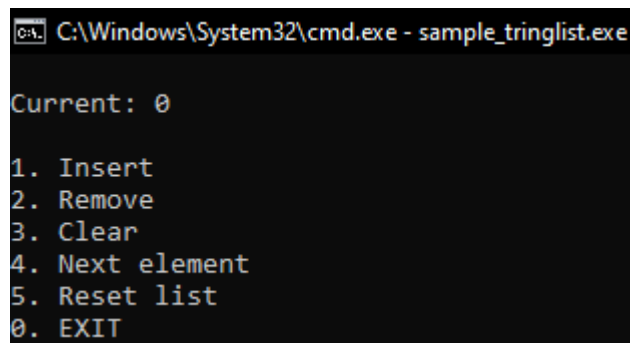


Рис. 1. Основное окно программы.

2. При выборе действия **Insert** на (рис. 1), окно примет вид, показанный ниже (рис. 2). На выбор даются следующие действия: **Insert First** – вставка в начало списка; **Insert Last** – вставка в конец списка; **Insert Before** – вставка перед указанным элементом; **Insert After** – вставка после указанного элемента; **Insert Before Current** – вставка перед текущим элементом; **Insert After Current** – вставка после текущего элемента; **next element** – перейти на следующий элемент списка; **reset list** – перейти на первый элемент списка; **EXIT** – выйти в главное меню. Пример ввода результат после нескольких попыток ввода можете увидеть на (рис. 3) и на (рис. 4) соответственно.

```
C:\Windows\System32\cmd.exe - sample_tringlist.exe

Current: 0

1. Insert First
2. Insert Last
3. Insert Before
4. Insert After
5. Insert Before Current
6. Insert After Current
7. Next element
8. Reset list
0. EXIT
```

Рис. 2. Меню действия Insert.

```
C:\Windows\System32\cmd.exe - sample_tringlist.exe

Current: 0

1. Insert First
2. Insert Last
3. Insert Before
4. Insert After
5. Insert Before Current
6. Insert After Current
7. Next element
8. Reset list
0. EXIT

2

Insert: 1_
```

Рис. 3. Пример добавления элемента в список.

```
C:\Windows\System32\cmd.exe - sample_tringlist.exe

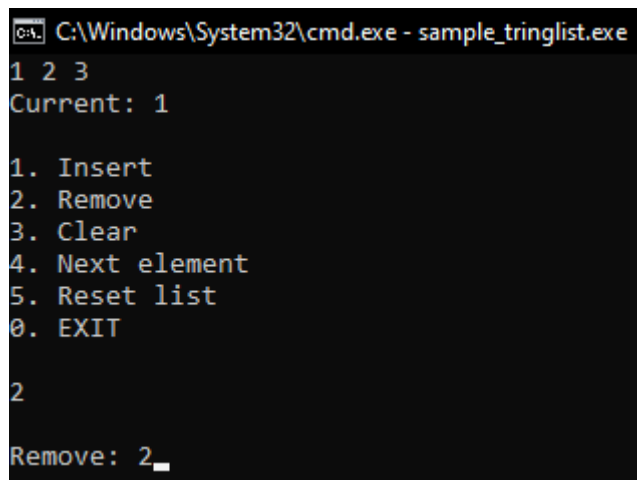
1 2 3
Current: 1

1. Insert
2. Remove
3. Clear
4. Next element
5. Reset list
0. EXIT

_
```

Рис. 4. Пример результата после нескольких добавлений в список.

3. При выборе действия **remove** на (рис. 1) вам будет предложено удалить элемент из списка. Ниже представлены пример ввода и пример результата на (рис. 5) и на (рис. 6) соответственно.



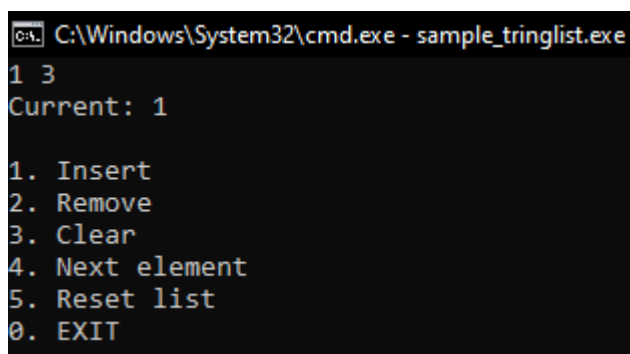
```
C:\Windows\System32\cmd.exe - sample_tringlist.exe
1 2 3
Current: 1

1. Insert
2. Remove
3. Clear
4. Next element
5. Reset list
0. EXIT

2

Remove: 2_
```

Рис. 5. Пример удаления элемента из списка.



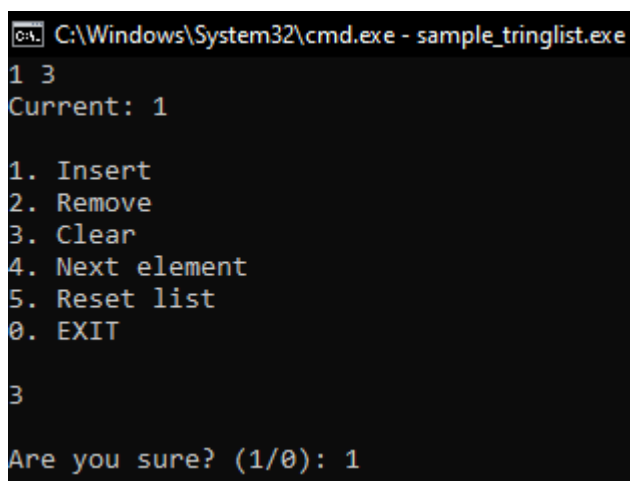
```
C:\Windows\System32\cmd.exe - sample_tringlist.exe
1 3
Current: 1

1. Insert
2. Remove
3. Clear
4. Next element
5. Reset list
0. EXIT

2
```

Рис. 6. Пример результата удаления элемента из списка.

4. При выборе действия **clear** на (рис. 1) вам будет предложено очистить список, удалив все его элементы. Для очистки надо будет подтвердить действие, введя соответствующую клавишу. Пример ввода показан на (рис. 7).



```
C:\Windows\System32\cmd.exe - sample_tringlist.exe
1 3
Current: 1

1. Insert
2. Remove
3. Clear
4. Next element
5. Reset list
0. EXIT

3

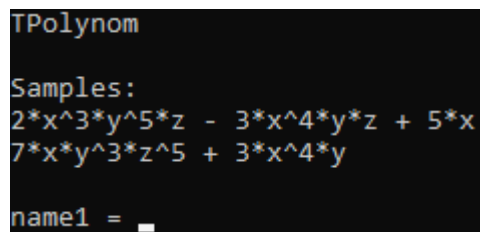
Are you sure? (1/0): 1
```

Рис. 7. Пример подтверждения очистки списка.

5. При выборе действия next element будет произведено изменение текущего элемента на тот, который расположен от него справа. При применении этого действия на последнем элементе, текущим элементом станет первый из списка.
6. При выборе действия next element будет произведено изменение текущего элемента на первый из списка.

2.2 Приложение для демонстрации работы полиномов

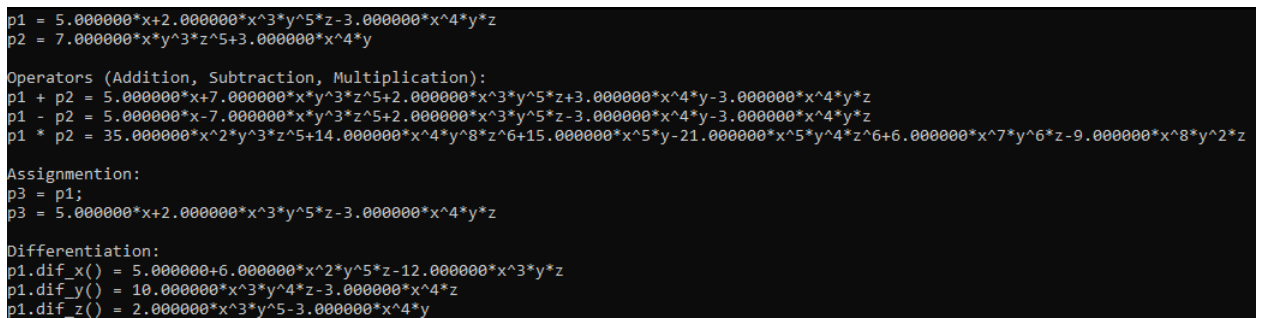
1. Запустите приложение с названием sample_tpolynom.exe. В результате появится окно, показанное ниже (рис. 8). Вам будет предложено ввести два полинома и будет показан пример этих двух полиномов.



```
TPolynom
Samples:
2*x^3*y^5*z - 3*x^4*y*z + 5*x
7*x*y^3*z^5 + 3*x^4*y
name1 = _
```

Рис. 8. Начало программы sample_tpolynom.exe.

2. При вводе двух полиномов будут показаны результаты всех операций над ними, показанные на (рис. 9). При вводе некорректного выражения будет выдана ошибка.



```
p1 = 5.000000*x+2.000000*x^3*y^5*z-3.000000*x^4*y*z
p2 = 7.000000*x*y^3*z^5+3.000000*x^4*y

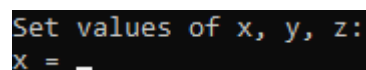
Operators (Addition, Subtraction, Multiplication):
p1 + p2 = 5.000000*x+7.000000*x*y^3*z^5+2.000000*x^3*y^5*z+3.000000*x^4*y-3.000000*x^4*y*z
p1 - p2 = 5.000000*x-7.000000*x*y^3*z^5+2.000000*x^3*y^5*z-3.000000*x^4*y-3.000000*x^4*y*z
p1 * p2 = 35.000000*x^2*y^3*z^5+14.000000*x^4*y^8*z^6+15.000000*x^5*y-21.000000*x^5*y^4*z^6+6.000000*x^7*y^6*z-9.000000*x^8*y^2*z

Assignment:
p3 = p1;
p3 = 5.000000*x+2.000000*x^3*y^5*z-3.000000*x^4*y*z

Differentiation:
p1.dif_x() = 5.000000+6.000000*x^2*y^5*z-12.000000*x^3*y*z
p1.dif_y() = 10.000000*x^3*y^4*z-3.000000*x^4*z
p1.dif_z() = 2.000000*x^3*y^5-3.000000*x^4*y
```

Рис. 9. Пример операций над двумя полиномами.

3. Далее вам будет предложено ввести значения x, y, z, для получения значения полиномов в данной точке (рис. 10). Введите эти значения и ознакомьтесь с результатом (рис. 11).



```
Set values of x, y, z:
x = _
```

Рис. 10. Ввод координат точки.

```
Set values of x, y, z:  
x = 1  
y = 2  
z = 3  
p1(x, y, z) = 179  
p2(x, y, z) = 13614
```

Рис. 11. Значение полиномов в введенной точке.

3 Руководство программиста

3.1 Описание алгоритмов

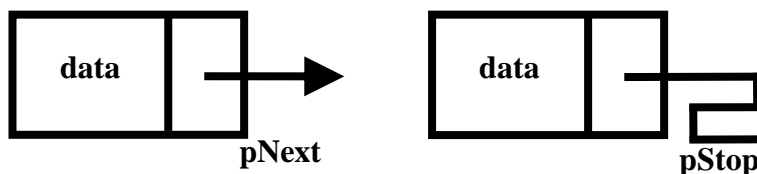
3.1.1 Связный список

Односвязный список (или просто "связанный список") - это базовая структура данных в информатике. Он состоит из: узлов, каждый из которых содержит данные (data) и указатель на следующий узел в списке (pNext); указателя на первый узел списка (pFirst); указателя на адрес остановки (pStop).

Односвязный список поддерживает операции, такие как поиск элемента по значению, добавление элемента в начало или конец списка, добавление элемента перед заданным элементом или после него, удаление элемента из списка.

Узел

Узел состоит из следующих полей: data – хранящиеся данные любого типа, pnext – указатель на следующий. Последний узел списка указывает на адрес остановки pStop, как правило на NULL. Узел и последний узел будем обозначать соответственно следующим образом:



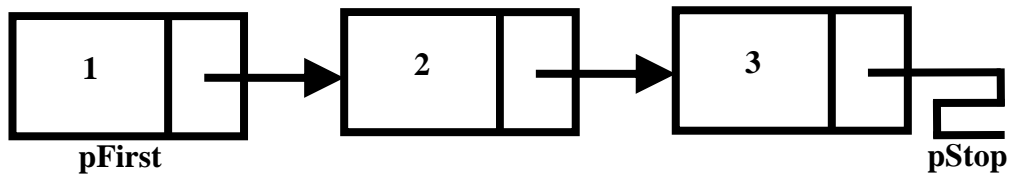
Операция поиска узла по значению

1. Начинаем обход по списку с первого звена (pFirst).
2. Если дошли до конца списка, то возвращаем конец списка (pStop).
3. Если значение рассматриваемого узла равно тому, который ищется, то возвращаем этот узел, иначе рассматриваем следующий узел и начинаем алгоритм с пункта 2.

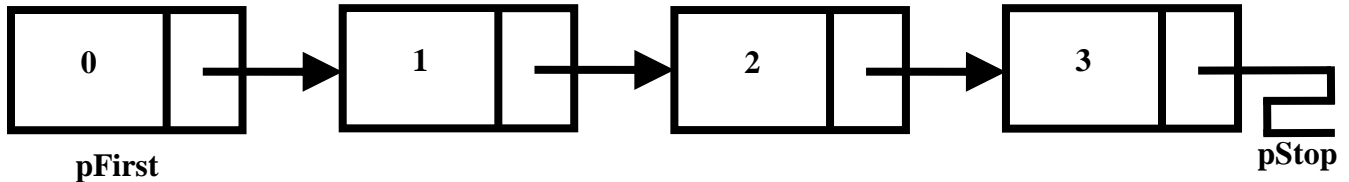
Операция добавления узла в начало списка

1. Создаем узел (pNode), указывающий на первый узел списка (pFirst).
2. Делаем созданный узел первым узлом списка (pFirst).

Пример:



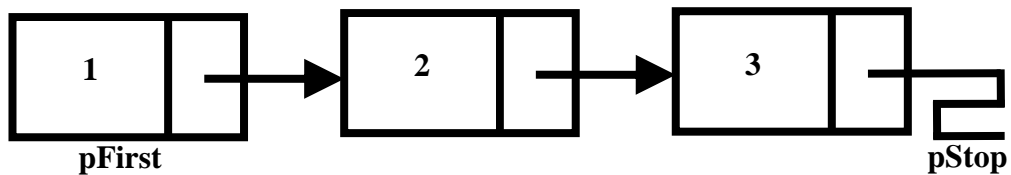
Добавление узла с значением 0 в начало списка:



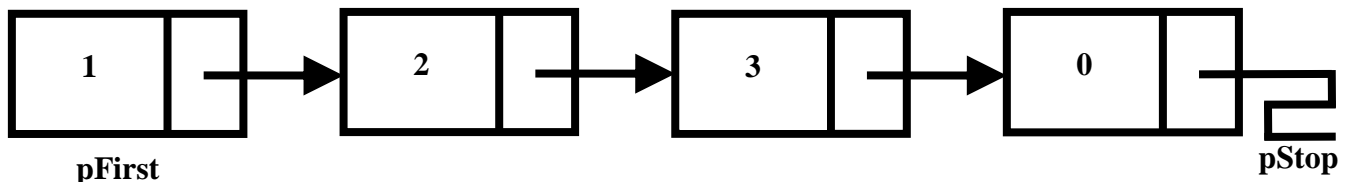
Операция добавления узла в конец списка

1. Создаем узел (pNode), указывающий на конец списка (pStop).
2. Если список пустой, добавляем узел в начало списка.
3. Иначе идем по списку с начала, пока не найдем последний узел в списке.
4. Указываем последний узел на созданный узел (pNode)

Пример:



Добавление узла с значением 0 в начало списка:

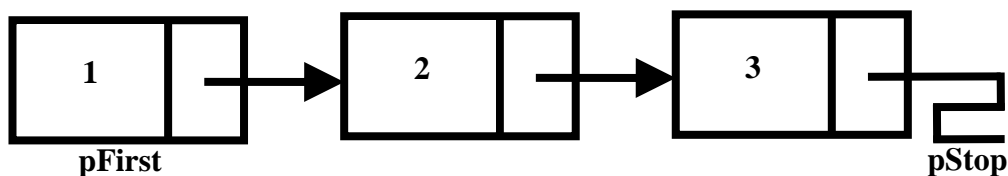


Операция добавления узла перед узлом с заданным значением

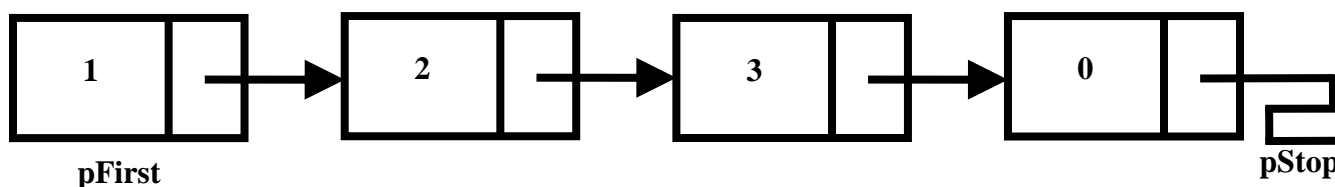
1. Создаем узел (pNode).
2. Если список пустой, заканчиваем операцию (далее КОНЕЦ) или выдаем ошибку.
3. Если первый элемент узла с заданным значением, то вставляем pNode в начало списка, КОНЕЦ.
4. Идем по списку с начала (tmp), рассматривая следующий по списку узел (tmp->pnext).

5. Если значение у $tmp \rightarrow pnext$ совпало с заданным, то $pNode$ указываем на $tmp \rightarrow pnext$, а tmp указываем на $pNode$, КОНЕЦ.
6. Иначе если дошли до конца списка и не нашли заданное значение, то КОНЕЦ или выдаем ошибку.

Пример:



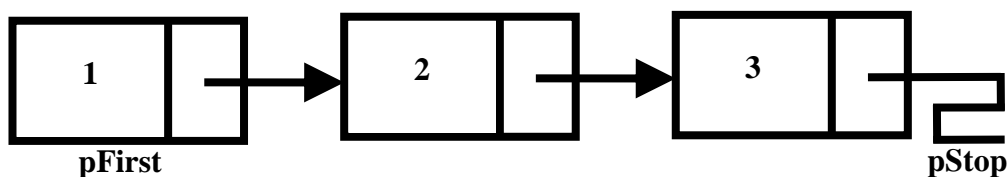
Добавление узла с значением 0 перед узлом с значением 2:



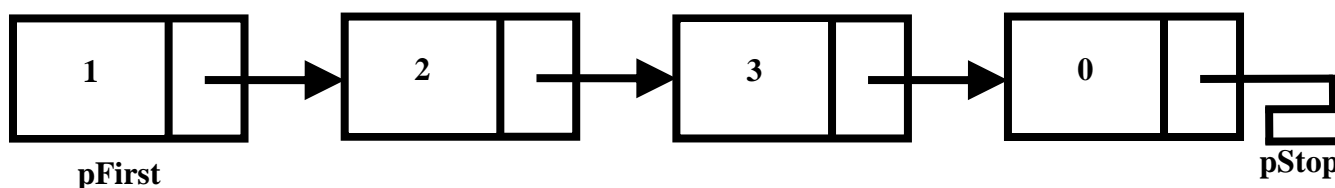
Операция добавления узла после узла с заданным значением

1. Создаем узел ($pNode$).
2. Если список пустой, КОНЕЦ или выдаем ошибку.
3. Идем по списку с начала, рассматривая данный узел (tmp).
4. Если значение tmp совпадает с заданным значением, то $pNode$ указываем на следующий узел ($tmp \rightarrow pnext$), а tmp указываем на $pNode$, КОНЕЦ.
5. Если дошли до конца списка и tmp совпадает с заданным значением, то $pNode$ добавляем в конец списка, КОНЕЦ.
6. Иначе звено не найдено, КОНЕЦ или выдаем ошибку.

Пример:



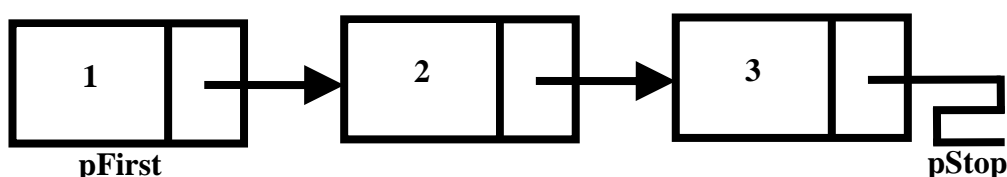
Добавление узла с значением 0 после узла с значением 2:



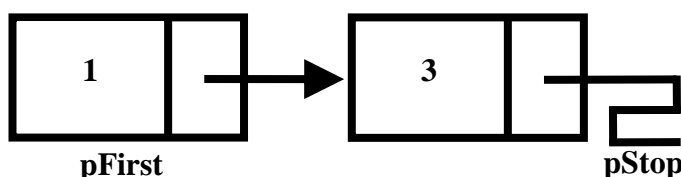
Операция удаления узла из списка по значению

1. Если список пуст, то КОНЕЦ.
2. Если первый элемент из списка (pFirst) равен искомому, то запоминаем следующее звено ($tmp = pFirst \rightarrow pnext$), удаляем pFirst и приравниваем pFirst к tmp, КОНЕЦ.
3. Иначе идем по списку с начала (tmp), рассматривая следующий по списку узел ($tmp \rightarrow pnext$). Если $tmp \rightarrow pnext$ равен искомому, то запоминаем узел $tmp \rightarrow pnext \rightarrow pnext$, удаляем $tmp \rightarrow pnext$, указываем tmp на $tmp \rightarrow pnext \rightarrow pnext$, КОНЕЦ.
4. Если tmp – последний узел списка, то искомый узел не найден, КОНЕЦ.

Пример:



Удаление узла с значением 2:



3.1.2 Полиномы

В данной работе полиномом является связный упорядоченный список из мономов трех переменных. Для начала рассмотрим мономы.

Мономы

Моном – это математическое выражение, состоящее из одного члена. В общем виде имеет вид $A * x^a * y^b * z^c$, где

- A – коэффициент, принимающее любое действительное значение.
- x, y, z – переменные.
- a, b, c – неотрицательные степени переменных, $a, b, c \in [0, 9]$.

Хранятся мономы в следующем виде: $\{coeff, degree\}$, где

- $coeff = A$ – коэффициент, принимающее любое действительное значение.

- $degree = a * 100 + b * 10 + c$ – свертка соответствующих степеней степеней переменных, $degree \in [0, 999]$.

Мономы поддерживают следующие операции:

- Получение степеней переменных x, y, z .
- Сравнение мономов (операции $<, >, \leq, \geq, ==, !=$).
- Операции сложения, разности, произведения мономов.
- Операции взятия производной по x , по y и по z .
- Получение значения монома в точке.

Получение степеней монома

Степень переменной x : свертку степеней делим без остатка на 100.

Степень переменной y : свертку степеней делим без остатка на 10, затем получаем остаток от деления получившегося числа на 10.

Степень переменной z : берем остаток от деления свертки на 10.

Формулы:

- $degX = degree / 100;$
- $degY = (degree / 10) \% 10;$
- $degZ = degree \% 10.$

Пример: Моном $2 * x^2 * y^3 * z^4 \sim \{2, 234\}$

- $degX = 234 / 100 = 2;$
- $degY = (234 / 10) \% 10 = 23 \% 10 = 3;$
- $degZ = 234 \% 10 = 4.$

Пример: Моном $2 * y * z^2 \sim \{2, 12\}$

- $degX = 12 / 100 = 0;$
- $degY = (12 / 10) \% 10 = 1 \% 10 = 1;$
- $degZ = 12 \% 10 = 2.$

Пример: Моном $x * z \sim \{1, 101\}$

- $degX = 101 / 100 = 1;$
- $degY = (101 / 10) \% 10 = 10 \% 10 = 0;$

- $\deg Z = 101 \% 10 = 1$.

Операции сравнения мономов

В этих операциях происходит сравнение сверток степеней мономов как чисел.

Если свертка степеней монома А меньше свертки степеней монома В, то моном А меньше монома В.

Остальные операции сравнения производятся аналогично вышеописанной.

Операция сложения(разности) мономов

Эта операция может производиться, только если показатели степеней полиномов равны. Если это условие выполнено, то результатом сложения(разности) двух полиномов будет полином с тем же показателем степени, а коэффициент будет результатом сложения(разности) коэффициентов полиномов.

Пример:

$$A = 2 * x^2 * y^3 * z^4 \sim \{2, 234\}$$

$$B = 3 * x^2 * y^3 * z^4 \sim \{3, 234\}$$

$$A + B = (2 + 3) * x^2 * y^3 * z^4 = 5 * x^2 * y^3 * z^4 \sim \{2 + 3, 234\} = \{5, 234\}$$

Пример:

$$A = 2 * x^2 * y^3 * z^4 \sim \{2, 234\}$$

$$B = 3 * x^2 * y^3 * z^4 \sim \{3, 234\}$$

$$A - B = (2 - 3) * x^2 * y^3 * z^4 = -1 * x^2 * y^3 * z^4 \sim \{2 - 3, 234\} = \{-1, 234\}$$

Операция произведения мономов

Результатом произведения мономов А и В является моном, коэффициент которого является произведением коэффициентов мономов А и В, а свертка степеней является суммой сверток степеней мономов.

ПРИМЕЧАНИЕ: сумма степеней отдельных переменных не может быть больше 9.

Пример:

$$A = 2 * x^2 * y^3 * z^4 \sim \{2, 234\}$$

$$B = 5 * x^5 * y^4 * z^3 \sim \{5, 543\}$$

$$A * B = (2 * 5) * x^{2+5} * y^{3+4} * z^{4+3} = 10 * x^7 * y^7 * z^7 \sim \{2 * 5, 234 + 543\} = \{10, 777\}$$

Операции взятия производных

Если показатель степени переменной, по которой происходит взятие производной, равен нулю, то получаемый моном является нулевым.

Иначе коэффициент получаемого монома является произведением коэффициента изначального монома на значение степени переменной, по которой берется производная. Свертка степеней получаемого монома является свертка степеней исходного монома, уменьшенная на: 100, если производная по переменной X; 10, если по Y, 1, если по Z.

Пример:

$$A = 2 * x^2 * y^3 * z^4 \sim \{2, 234\}$$

- $A'_x = (2 * 2) * x^{2-1} * y^3 * z^4 = 4 * x * y^3 * z^4 \sim \{2 * 2, 234 - 100\} = \{4, 134\}$
- $A'_y = (2 * 3) * x^2 * y^{3-1} * z^4 = 6 * x^2 * y^2 * z^4 \sim \{2 * 3, 234 - 10\} = \{6, 224\}$
- $A'_z = (2 * 4) * x^2 * y^3 * z^{4-1} = 8 * x^2 * y^3 * z^3 \sim \{2 * 4, 234 - 1\} = \{8, 233\}$

Операция получения значения монома в точке

Получаем значение степеней переменных, возводим значения координат в полученные соответственные степени, перемножаем их и умножаем на коэффициент.

Возведение координаты в степень происходит путем накопления произведения 1 на данную координату в цикле до значения степени.

Пример:

$$A = 2 * x^2 * y^3 * z^4 \sim \{2, 234\}$$

$$A(3,2,1) = 2 * 3^2 * 2^3 * 1^4 = 2 * (3 * 3) * (2 * 2 * 2) * (1 * 1 * 1 * 1) = 2 * 9 * 8 * 1 = 144$$

Полиномы

Полином трёх переменных — это математическое выражение, представляющее собой сумму произведений переменных x, y, z с целыми степенями от 0 до 9 и коэффициентами из множества действительных чисел.

Общая форма полинома трёх переменных выглядит следующим образом:

$$P(x, y, z) = \sum_{i=0}^n \sum_{j=0}^m \sum_{k=0}^k M_{ijk}(x, y, z)$$

Где:

- $M_{ijk}(x, y, z) = a_{ijk}x^i y^j z^k$ – мономы полинома.
- a_{ijk} – коэффициенты полинома, являющиеся действительными числами.
- x, y, z – переменные полинома.
- i, j, k – целые неотрицательные степени переменных x, y, z .
- n, m, l – максимальные степени переменных x, y, z соответственно. Обычно они определяются в зависимости от конкретного полинома.

В данной реализации полиномом является связный упорядоченный список из мономов трех переменных. Список упорядочен из-за того, что каждый моном в списке не меньше всех мономов, его предшествующих.

Поддерживаемые операции:

- Упорядоченное добавление монома в полином (техническая операция)
- Сложение(разность) полиномов
- Произведение полиномов
- Взятие производной по переменным x, y и z
- Получение значения полинома в точке

Упорядоченное добавление монома в полином

1. Если коэффициент добавляемого монома – ноль, то КОНЕЦ.
2. Проходим список с начала, проверяя текущий моном списка.
 - a. Если текущий моном равен добавляемому, то складываем их коэффициенты. Если сумма коэффициентов равна нулю, то удаляем текущий моном из списка. Иначе изменяем коэффициент текущего монома на сумму коэффициентов. КОНЕЦ.
 - b. Иначе если добавляемый моном меньше текущего, то добавляем вставляемый моном перед текущим мономом. КОНЕЦ.
 - c. Иначе переходим на следующий моном.
3. Цикл закончился, и мы дошли до конца списка. Вставляем новый моном в конец списка. КОНЕЦ.

Операция сложения(вычитания) полиномов

При сложении полинома А с полиномом В (разности полинома А с полиномом В), создаем полином С – копию полинома А, и в полином С упорядоченно добавляем каждый моном (умноженный на -1) из полинома В.

Пример:

$$A = -3xyz + 5xy^4z^2 + 2x^3y^2z$$

$$B = 7z + 2xy^4z^2 + 4x^3y^2z$$

$$\begin{aligned} A + B &= (-3xyz + 5xy^4z^2 + 2x^3y^2z) + (7z + 2xy^4z^2 + 4x^3y^2z) = \\ &= 7z - 3xyz + 7xy^4z^2 + 6x^3y^2z \end{aligned}$$

$$\begin{aligned} A - B &= (-3xyz + 5xy^4z^2 + 2x^3y^2z) - (7z + 2xy^4z^2 + 4x^3y^2z) = \\ &= -7z - 3xyz + 3xy^4z^2 - 2x^3y^2z \end{aligned}$$

Операция произведения полиномов

При умножении полинома А на полином В, создаем пустой моном С, каждый моном полинома А умножаем на каждый моном полинома В и результаты умножения мономов упорядоченно добавляем в моном С.

Пример:

$$A = 4 - 3xyz + 2x^2y^3$$

$$B = 6 - 7z^3 + 5xy^2z$$

$$\begin{aligned} A * B &= (4 - 3xyz + 2x^2y^3) * (6 - 7z^3 + 5xy^2z) = \\ &= 24 - 21z^4 + 30y^2z + 24xy^2z - 28yz^3 + 20x^2y^2z - 18xyz \\ &\quad + 21xyz^4 - 15xy^3z^4 + 12x^2y^3z - 14x^2y^3z^4 + 10x^3y^5z^2 \end{aligned}$$

Операции взятия производных

Проходим по списку, применяя для каждого монома функцию взятия производной монома по переменной, и упорядоченно добавляя в новый полином.

Пример:

$$P = -3xz + 5xy^4z^2 + 2x^3y^2z$$

- $P'_x = (-3xz + 5xy^4z^2 + 2x^3y^2z)'_x = -3z + 5y^4z^2 + 6x^2y^2z$
- $P'_y = (-3xz + 5xy^4z^2 + 2x^3y^2z)'_y = 20xy^3z^2 + 4x^3yz$
- $P'_z = (-3xz + 5xy^4z^2 + 2x^3y^2z)'_z = -3x + 10xy^4z + 2x^3y^2$

Операция получения значения полинома в точке

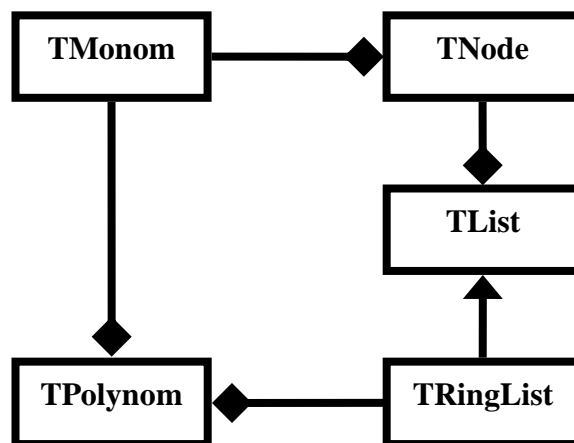
Проходим по списку, применяя для каждого монома функцию получения значения монома в точке, и суммируем все полученные значения.

Пример:

$$P(x, y, z) = -3xz + 5xy^4z^2 + 2x^3y^2z$$

$$P(1, 2, 3) = -3 * 1 * 3 + 5 * 1 * 2^4 * 3^2 + 2 * 1^3 * 2^2 * 3 = -9 + 720 + 24 = 735$$

3.2 Описание программной реализации



3.2.1 Описание класса TNode

```
template <typename T>
struct TNode {
    T key;
    TNode<T>* pnext;

    TNode() : key(), pnext(nullptr) {};
    TNode(const T& data) : key(data), pnext(nullptr) {};
    TNode(TNode<T>* _pnext) : key(), pnext(_pnext) {};
    TNode(const T& data, TNode<T>* _pnext) : key(data), pnext(_pnext) {};
};
```

Назначение: представление звена списка.

Поля:

key – данные, хранящиеся в звене.

pnext – указатель на следующее звено.

Конструкторы:

`TNode();`

Назначение: конструктор по умолчанию.

`TNode(const T& data);`

Назначение: конструктор с параметром.

Входные параметры: **data** – данные для хранения в звене.

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

Назначение: конструктор с параметром.

Входные параметры: **_pNext** – указатель на следующее звено.

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

Назначение: конструктор с параметрами.

Входные параметры: **data** – данные для хранения в звене, **_pNext** – указатель на следующее звено.

3.2.2 Описание класса TList

```
template <typename T>
class TList {
protected:
    TNode<T>* pFirst;
    TNode<T>* pLast;
    TNode<T>* pCurr;
    TNode<T>* pStop;

public:
    TList();
    TList(TNode<T>* _pFirst);
    TList(const TList<T>& obj);
    virtual ~TList();

    virtual TNode<T>* find(const T& data);
    virtual void insert_first(const T& data);
    virtual void insert_last(const T& data);
    virtual void insert_before(const T& data, const T& before);
    virtual void insert_after(const T& data, const T& after);
    virtual void insert_before(const T& data);
    virtual void insert_after(const T& data);
    virtual void remove(const T& data);
    virtual void clear();

    virtual size_t size() const;
    bool full() const;
    virtual bool empty() const;

    virtual void reset();
    TNode<T>* get_curr() const;
    virtual void next(const int count = 1);
    virtual bool ended() const;

    TNode<T>* first() const;
    TNode<T>* last() const;
};
```

Назначение: структура данных «Односвязный список».

Поля:

pFirst – указатель на первый элемент.

pLast – указатель на последний элемент.

pCurr – указатель на текущий элемент.

pStop – указатель на конец списка.

Конструкторы:

TList();

Назначение: конструктор по умолчанию.

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

Назначение: конструктор преобразования типа.

Входные параметры: **_pFirst** – указатель на первое звено.

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

Назначение: конструктор копирования.

Входные параметры: **obj** – копируемый односвязный список.

```
virtual ~TList();
```

Назначение: деструктор.

Методы:

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

Назначение: поиск звена в списке по ключю.

Входные параметры: **data** – искомый ключ.

Выходные параметры: указатель на найденное звено в списке.

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

Назначение: вставка в начало списка.

Входные параметры: **data** – вставляемые данные.

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

Назначение: вставка в конец списка.

Входные параметры: **data** – вставляемые данные.

```
virtual void insert_before(const T& data, const T& before);
```

Назначение: вставка перед заданным элементом.

Входные параметры: **data** – вставляемые данные, **before** – элемент, перед которым должна произойти вставка.

```
virtual void insert_after(const T& data, const T& after);
```

Назначение: вставка после заданного элемента.

Входные параметры: **data** – вставляемые данные, **before** – элемент, после которого должна произойти вставка.

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

Назначение: вставка перед текущим элементом.

Входные параметры: **data** – вставляемые данные.

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

Назначение: вставка после текущего элемента.

Входные параметры: **data** – вставляемые данные.

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

Назначение: удаление элемента из списка.

Входные параметры: **data** – удаляемые данные.

```
virtual void clear();
```

Назначение: удаление всех звеньев списка.

```
virtual size_t size() const;
```

Назначение: получение количества звеньев в списке.

Выходные параметры: количества звеньев в списка.

```
bool full() const;
```

Назначение: проверка на заполненность памяти.

Выходные параметры: **true** или **false**.

```
virtual bool empty() const;
```

Назначение: проверка, пуст ли список.

Выходные параметры: **true** или **false**.

```
virtual void reset();
```

Назначение: переход в начало списка.

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

Назначение: получение текущего звена списка.

Выходные параметры: ссылка на текущее звено списка.

```
virtual void next(const int count = 1);
```

Назначение: переход на следующее звено списка.

Входные параметры: **count** – количество переходов.

```
virtual bool ended() const;
```

Назначение: проверка, является ли текущее звено концом списка.

Выходные параметры: **true** или **false**.

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

Назначение: получение указателя на первое звено списка.

Выходные параметры: указатель на первое звено списка.

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

Назначение: получение указателя на последнее звено списка.

Выходные параметры: указатель на последнее звено списка.

3.2.3 Описание класса TRingList

```
template <typename T>
class TRingList : public TList<T> {
protected:
    TNode<T>* pHead;    // фиктивное звено

public:
    TRingList();
    TRingList(TNode<T>* _pFirst);
    TRingList(const TRingList<T>& list);
    virtual ~TRingList();

    TNode<T>* find(const T& data) override;
    void insert_first(const T& data) override;
    void insert_last(const T& data) override;
    void insert_before(const T& data) override;
    void insert_after(const T& data) override;
    void insert_before(const T& data, const T& before) override;
    void insert_after(const T& data, const T& after) override;
    void remove(const T& data) override;
    void clear() override;

    bool empty() const override;

    void reset() override;
    void next(const int count = 1) override;
    bool ended() const override;

    TNode<T>* head() const { return pHead };
};
```

Назначение: структура данных «Односвязный список закольцованный».

Поля:

pHead – фиктивное звено для закальцовывания списка.

Конструкторы:

TRingList();

Назначение: конструктор по умолчанию

TRingList(TNode<T>* _pFirst);

Назначение: конструктор преобразования типа.

Входные параметры: **_pFirst** – указатель на первое звено.

TRingList(const TRingList<T>& list);

Назначение: конструктор копирования.

Входные параметры: **obj** – копируемый односвязный список.

```
virtual ~TRingList();
```

Назначение: деструктор.

Методы:

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

Назначение: поиск звена в списке по ключю.

Входные параметры: **data** – искомый ключ.

Выходные параметры: указатель на найденное звено в списке.

```
void insert_first(const T& data) override;
```

Назначение: вставка в начало списка.

Входные параметры: **data** – вставляемые данные.

```
void insert_last(const T& data) override;
```

Назначение: вставка в конец списка.

Входные параметры: **data** – вставляемые данные.

```
void insert_before(const T& data) override;
```

Назначение: вставка перед текущим элементом.

Входные параметры: **data** – вставляемые данные.

```
void insert_after(const T& data) override;
```

Назначение: вставка после текущего элемента.

Входные параметры: **data** – вставляемые данные.

```
void insert_before(const T& data, const T& before) override;
```

Назначение: вставка перед заданным элементом.

Входные параметры: **data** – вставляемые данные, **before** – элемент, перед которым должна произойти вставка.

```
void insert_after(const T& data, const T& after) override;
```

Назначение: вставка после заданного элемента.

Входные параметры: **data** – вставляемые данные, **before** – элемент, после которого должна произойти вставка.

```
void remove(const T& data) override;
```

Назначение: удаление элемента из списка.

Входные параметры: **data** – удаляемые данные.

```
void clear() override;
```

Назначение: удаление всех звеньев списка.

```
bool empty() const override;
```

Назначение: проверка, пуст ли список.

Выходные параметры: **true** или **false**.

```
void reset() override;
```

Назначение: переход в начало списка.

```
void next(const int count = 1) override;
```

Назначение: переход на следующее звено списка.

Входные параметры: **count** – количество переходов.

```
bool ended() const override;
```

Назначение: проверка, является ли текущее звено концом списка.

Выходные параметры: **true** или **false**.

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

Назначение: получение указателя фиктивного звена.

Выходные параметры: указатель на фиктивное звено

3.2.4 Описание класса TMonom

```
class TMonom{
private:
    double coeff;
    int degree;
public:
    TMonom();
    TMonom(const TMonom& monom);
    TMonom(double _coeff, int _degree);
    TMonom(double _coeff, int degX, int degY, int degZ);

    double get_coeff() const;
    int get_degree() const;

    int get_degX() const;
    int get_degY() const;
    int get_degZ() const;

    void set_coeff(double _coeff);
    void set_degree(int _degree);
    void set_degree(int degX, int degY, int degZ);

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

    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;

    const TMonom& operator=(const TMonom& data);
    TMonom operator*(const TMonom& monom) const;

    TMonom dif_x() const;
    TMonom dif_y() const;
    TMonom dif_z() const;

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

Назначение: представление монома.

Поля:

coeff – коэффициент монома.

degree – свертка степеней монома.

Конструкторы:

TMonom();

Назначение: конструктор по умолчанию.

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

Назначение: конструктор копирования.

Входные параметры: **monom** – копируемый моном.

```
TMonom(double _coeff, int _degree);
```

Назначение: конструктор с параметрами.

Входные параметры: **_coeff** – коэффициент монома, **_degree** – свертка степеней монома.

```
TMonom(double _coeff, int degX, int degY, int degZ);
```

Назначение: конструктор с параметрами.

Входные параметры: **_coeff** – коэффициент монома, **degX, degY, degZ** – степени соответствующих переменных.

Методы:

```
double get_coeff() const;
```

Назначение: получение значения коэффициента монома.

Выходные параметры: значение коэффициента монома.

```
int get_degree() const;
```

Назначение: получение значения свертки степеней монома.

Выходные параметры: значение свертки степеней монома

```
int get_degX() const;
```

Назначение: получение значения степени переменной X.

Выходные параметры: значение степени переменной X.

```
int get_degY() const;
```

Назначение: получение значения степени переменной Y.

Выходные параметры: значение степени переменной Y.

```
int get_degZ() const;
```

Назначение: получение значения степени переменной Z.

Выходные параметры: значение степени переменной Z.

```
void set_coeff(double _coeff);
```

Назначение: задание коэффициента монома.

Входные параметры: `_coeff` - коэффициент монома.

```
void set_degree(int _degree);
```

Назначение: задание свертки степеней монома.

Входные параметры: `_degree` – свертка степени монома.

```
void set_degree(int degX, int degY, int degZ);
```

Назначение: задание свертки степеней монома.

Входные параметры: `degX, degY, degZ` – степени соответствующих переменных.

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

Назначение: получение значения монома в точке.

Входные параметры: `x, y, z` – координаты точки.

Выходные параметры: значение монома в точке

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

Назначение: операция сравнения “<”.

Входные параметры: `monom` – сравниваемый моном.

Выходные параметры: `true` или `false`.

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

Назначение: операция сравнения “>”.

Входные параметры: **monom** – сравниваемый моном.

Выходные параметры: **true** или **false**.

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

Назначение: операция сравнения “<=”.

Входные параметры: **monom** – сравниваемый моном.

Выходные параметры: **true** или **false**.

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

Назначение: операция сравнения “>=”.

Входные параметры: **monom** – сравниваемый моном.

Выходные параметры: **true** или **false**.

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

Назначение: операция сравнения “==”.

Входные параметры: **monom** – сравниваемый моном.

Выходные параметры: **true** или **false**.

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

Назначение: операция сравнения “!=”.

Входные параметры: **monom** – сравниваемый моном.

Выходные параметры: **true** или **false**.

```
const TMonom& operator=(const TMonom& data);
```

Назначение: оператор присваивания монома.

Входные параметры: **data** – присваиваемый моном.

Выходные параметры: ссылка на себя.

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

Назначение: оператор умножения мономов.

Входные параметры: `monom` – моном, на который происходит умножение.

Выходные параметры: результат умножения мономов.

```
TMonom dif_x() const;
```

Назначение: взятие производной монома по X.

Выходные параметры: производная монома по X.

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

Назначение: взятие производной монома по Y.

Выходные параметры: производная монома по Y.

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

Назначение: взятие производной монома по Z.

Выходные параметры: производная монома по Z.

```
string get_string() const;
```

Назначение: преобразование монома в строку.

Выходные параметры: сформированная строка монома.

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

Назначение: вывод строки монома.

Входные параметры: `out` – поток вывода, `monom` – выводящийся моном.

Выходные параметры: поток вывода.

3.2.5 Описание класса TPolynom

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

    static map<string, int> priority;

    void parse(string& name);
    void to_monoms(vector<string>& lexems);
    void add_monom(const TMonom& m);
public:
    TPolynom();
    TPolynom(const string& _name);
    TPolynom(const TRingList<TMonom>& ringlist);
    TPolynom(const TPolynom& polynom);
    ~TPolynom();

    bool operator<(TPolynom& polynom);
    bool operator>(TPolynom& polynom);
    bool operator<=(TPolynom& polynom);
    bool operator>=(TPolynom& polynom);
    bool operator==(TPolynom& polynom);
    bool operator!=(TPolynom& polynom);

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

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

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

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

private:
    bool is_operator(const string& isopr) const;
    bool is_const(const string& isopd) const;
    bool is_variable(const string& str) const;

    int find_operator(const string& name, int pos = 0) const;

    void convert_infix(string& name);
    void correctness_check(const string& name) const;
};
```

Назначение: представление полинома.

Поля:

monoms – указатель на односвязный кольцевой список мономов.

priority – список арифметических операций с их приоритетами.

Конструкторы:

```
TPolynomial();
```

Назначение: конструктор по умолчанию

```
TPolynomial(const string& _name);
```

Назначение: конструктор с параметром.

Входные параметры: **_name** – строка полинома.

```
TPolynomial(const TRingList<TMonom>& ringlist);
```

Назначение: конструктор преобразования типа

Входные параметры: **ringlist** - односвязный кольцевой список мономов.

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

Назначение: конструктор копирования.

Входные параметры: **polynom** – копируемый полином.

```
~TPolynomial();
```

Назначение: деструктор.

Методы:

```
bool operator<(TPolynomial& polynom);
```

Назначение: операция сравнения “<”.

Входные параметры: **polynom** – сравниваемый полином.

Выходные параметры: **true** или **false**.

```
bool operator>(TPolynomial& polynom);
```

Назначение: операция сравнения “>”.

Входные параметры: **polynom** – сравниваемый полином.

Выходные параметры: **true** или **false**.

```
bool operator<=(TPolynomial& polynomial);
```

Назначение: операция сравнения “<=”.

Входные параметры: **polynomial** – сравниваемый полином.

Выходные параметры: **true** или **false**.

```
bool operator>=(TPolynomial& polynomial);
```

Назначение: операция сравнения “>=”.

Входные параметры: **polynomial** – сравниваемый полином.

Выходные параметры: **true** или **false**.

```
bool operator==(TPolynomial& polynomial);
```

Назначение: операция сравнения “==”.

Входные параметры: **polynomial** – сравниваемый полином.

Выходные параметры: **true** или **false**.

```
bool operator!=(TPolynomial& polynomial);
```

Назначение: операция сравнения “!=”.

Входные параметры: **polynomial** – сравниваемый полином.

Выходные параметры: **true** или **false**.

```
const TPolynomial& operator=(const TPolynomial& polynomial);
```

Назначение: оператор присваивания полинома.

Входные параметры: **polynomial** – присваиваемый полином.

Выходные параметры: ссылка на себя.

```
const TPolynomial operator+(const TPolynomial& polynomial) const;
```

Назначение: оператор сложения полиномов.

Входные параметры: **polynomial** – прибавляемый полином.

Выходные параметры: результат сложения полиномов.

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

Назначение: оператор унарного минуса.

Выходные параметры: результат унарного минуса.

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

Назначение: оператор вычитания полиномов.

Входные параметры: **polynom** – вычитаемый полином.

Выходные параметры: результат вычитания полиномов

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

Назначение: оператор произведения полиномов.

Входные параметры: **polynom** – полином, на который умножают.

Выходные параметры: результат произведения полиномов

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

Назначение: получение значения полинома в точке.

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

Выходные параметры: значение полинома в точке

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

Назначение: взятие производной полинома по X.

Выходные параметры: производная полинома по X.

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

Назначение: взятие производной полинома по Y.

Выходные параметры: производная полинома по Y.

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

Назначение: взятие производной полинома по Z .

Выходные параметры: производная полинома по Z .

```
string get_string() const;
```

Назначение: преобразование полинома в строку.

Выходные параметры: сформированная строка полинома.

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

Назначение: вывод строки полинома.

Входные параметры: `out` – поток вывода, `monom` – выводящийся полином.

Выходные параметры: поток вывода.

Служебные методы:

```
void parse(string& name);
```

Назначение: анализ строки инфиксной формы арифметического выражения.

Входные параметры: `name` – строка инфиксной формы арифметического выражения.

```
void to_monoms(vector<string>& lexems);
```

Назначение: анализ массива лексем инфиксной формы арифметического выражения для выделения отдельных мономов.

Входные параметры: `lexems` - лексемы инфиксной формы арифметического выражения.

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

Назначение: упорядоченное добавление монома в полином.

Входные параметры: `m` – добавляемый моном.

```
bool is_operator(const string& isopr) const;
```

Назначение: проверка строки, является ли она доступным оператором.

Входные параметры: **isopr** – проверяемая строка.

Выходные параметры: **true** или **false**.

```
bool is_const(const string& isopd) const;
```

Назначение: проверка строки, является ли она константой, но не операндом.

Входные параметры: **isopd** – проверяемая строка.

Выходные параметры: **true** или **false**.

```
bool is_variable(const string& str) const;
```

Назначение: проверка строки, является ли она доступной переменной.

Входные параметры: **str** – проверяемая строка.

Выходные параметры: **true** или **false**.

```
int find_operator(const string& name, int pos = 0) const;
```

Назначение: поиск первого встречного оператора.

Входные параметры: **name** – строка, в которой происходит поиск, **pos** – начальная позиция поиска.

Выходные параметры: индекс первого встречного оператора.

```
void convert_infix(string& name);
```

Назначение: корректирование строки инфиксной формы арифметического выражения (избавление от пробелов; добавление отсутствующих операторов умножения на скобки; избавление от сокращенной записи действительных констант).

Входные параметры: **name** – корректируемая строка.

```
void correctness_check(const string& name) const;
```

Назначение: проверка инфиксной формы арифметического выражения на наличие критических ошибок в ее записи.

Входные параметры: **name** – проверяемая строка.

Заключение

В результате выполнения лабораторной работы был разработан и реализован класс, описывающий полиномы от трех переменных (x , y , z) на основе односвязных списков на языке программирования C++. Этот класс предоставляет удобные средства для работы с полиномиальными выражениями, включая создание, модификацию и выполнение операций над ними.

В процессе работы были успешно выполнены все поставленные задачи. Была спроектирована и описана структура звеньев односвязного списка, разработан класс односвязного списка с необходимыми методами, создан класс монома для представления отдельных членов полинома, а также реализован класс полинома с возможностью выполнения арифметических операций над ними.

Тестирование разработанных классов показало их корректность и эффективность при обработке различных входных данных.

Данная лабораторная работа позволила углубить понимание структур данных и алгоритмов, а также приобрести практические навыки в разработке классов и их применении для решения задач вычислительной математики.

В будущем возможно расширение функциональности разработанных классов, включая поддержку дополнительных операций над полиномами и оптимизацию алгоритмов для улучшения производительности.

Литература

1. Лекция «Списковые структуры хранения» Сысоева А.В.
<https://cloud.unn.ru/s/x33MEa9on8HgNgw>
2. Лекция «Списки в динамической памяти» Сысоева А.В.
<https://cloud.unn.ru/s/rCiKGSX33SSGPi4>
3. Лекция «Полиномы» Сысоева А.В. <https://cloud.unn.ru/s/t6o9kp5g9bpf2yz>

Приложения

Приложение А. Реализация класса TNode

```
#include <iostream>

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) {};
};
```

Приложение Б. Реализация класса TList

```
#include "tnode.h"

template <typename T>
class TList {
protected:
    TNode<T>* pFirst;
    TNode<T>* pLast;
    TNode<T>* pCurr;
    TNode<T>* pStop;

public:
    TList();
    TList(TNode<T>* _pFirst);
    TList(const TList<T>& list);
    virtual ~TList();

    virtual TNode<T>* find(const T& data);
    virtual void insert_first(const T& data);
    virtual void insert_last(const T& data);
    virtual void insert_before(const T& data);
    virtual void insert_after(const T& data);
    virtual void insert_before(const T& data, const T& before);
    virtual void insert_after(const T& data, const T& after);
    virtual void remove(const T& data);
    virtual void clear();

    virtual size_t size() const;
    bool full() const;
    virtual bool empty() const;

    virtual void reset();
    TNode<T>* get_curr() const;
    virtual void next(const int count = 1);
    virtual bool ended() const;
    bool IsLast() const;

    TNode<T>* first() const;
    TNode<T>* last() const;
};

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

    reset();
}

template <typename T>
TList<T>::TList(TNode<T>* _pFirst) {
    pStop = nullptr;
    pFirst = _pFirst;
    if (pFirst == pStop) {
        pLast = pStop;
        pCurr = pStop;

        reset();
        return;
    }
}
```

```

        TNode<T>* tmp = pFirst;
        while (tmp->pnext != pStop)
            tmp = tmp->pnext;
        pLast = tmp;

        reset();
    }

template <typename T>
TList<T>::TList(const TList<T>& obj) {
    if (obj.pFirst == obj.pStop) {
        pStop = nullptr;
        pFirst = pStop;
        pLast = pStop;
        pCurr = pFirst;

        reset();
        return;
    }

    pFirst = new TNode<T>(obj.pFirst->data);
    TNode<T>* tmp = obj.pFirst;
    TNode<T>* pNode = pFirst;

    while (tmp->pnext != obj.pStop) {
        pNode->pnext = new TNode<T>(tmp->pnext->data);
        pNode = pNode->pnext;
        tmp = tmp->pnext;
    }
    pLast = pNode;
    pStop = pLast->pnext;
    pCurr = pFirst;

    reset();
}

template <typename T>
TList<T>::~~TList() {
    clear();
}

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

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

    reset();
}

template <typename T>

```

```

void TList<T>::insert_last(const T& data) {
    if (pFirst == pStop) {
        insert_first(data);
        return;
    }
    TNode<T>* tmp = new TNode<T>(data, pStop);
    pLast->pnext = tmp;
    pLast = pLast->pnext;

    reset();
}

////////////////////////////////////
template <typename T>
void TList<T>::insert_before(const T& data) {
    if (pCurr == nullptr) {
        std::string exp = "Error: data not found";
        throw exp;
    }

    if (pCurr == pFirst) {
        insert_first(data);
        return;
    }

    TNode<T>* before = pFirst;
    while (before->pnext != pCurr)
        before = before->pnext;

    TNode<T>* tmp = new TNode<T>(data, pCurr);
    before->pnext = tmp;

    reset();
}

template <typename T>
void TList<T>::insert_after(const T& data) {
    if (pCurr == nullptr) {
        std::string exp = "Error: data not found";
        throw exp;
    }

    if (ended()) {
        insert_last(data);
        return;
    }

    TNode<T>* tmp = new TNode<T>(data, pCurr->pnext);
    pCurr->pnext = tmp;

    reset();
}

////////////////////////////////////

template <typename T>
void TList<T>::insert_before(const T& data, const T& before) {
    pCurr = find(before);

    insert_before(data);

    reset();
}

```



```

template <typename T>
void TList<T>::insert_after(const T& data, const T& after) {
    pCurr = find(after);

    insert_after(data);

    reset();
}

```

```

////////////////////////////////////

```

```

template <typename T>
void TList<T>::remove(const T& data) {
    if (pFirst == pStop) {
        std::string exp = "Error: list is empty";
        throw exp;
    }

    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) {
        std::string exp = "Error: data not found";
        throw exp;
    }

    if (pCurr == pNode) reset();

    if (pPrev == nullptr) {
        pFirst = pNode->pnext;
        delete pNode;
        return;
    }

    pPrev->pnext = pNode->pnext;
    delete pNode;
}

```

```

template <typename T>
void TList<T>::clear() {
    TNode<T>* tmp = pFirst;
    while (pFirst != pStop) {
        pFirst = tmp->pnext;
        delete tmp;
        tmp = pFirst;
    }
    pLast = pStop;
    pCurr = pStop;
}

```

```

template <typename T>
size_t TList<T>::size() const {
    size_t size = 0;
    TNode<T>* tmp = pFirst;
    while (tmp != pStop) {
        tmp = tmp->pnext;
    }
}

```

```

        ++size;
    }
    return size;
}

template <typename T>
bool TList<T>::full() const {
    TNode<T>* tmp = new TNode<T>;
    if (tmp != pStop) {
        return false;
    }
    else {
        return true;
    }
}

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

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

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

template <typename T>
void TList<T>::next(const int count) {
    if (count <= 0) {
        std::string exp = "Error: count can't be less than 0";
        throw exp;
    }

    for (int i = 0; i < count; i++) {
        if (!ended()) pCurr = pCurr->pnext;
        else reset();
    }
}

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

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

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

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

```

}

Приложение В. Реализация класса TRingList

```
#include "tlist.h"

template <typename T>
class TRingList : public TList<T> {
protected:
    TNode<T>* pHead; // фиктивное звено

public:
    TRingList();
    TRingList(TNode<T>* _pFirst);
    TRingList(const TRingList<T>& list);
    virtual ~TRingList();

    TNode<T>* find(const T& data) override;
    void insert_first(const T& data) override;
    void insert_last(const T& data) override;
    void insert_before(const T& data) override;
    void insert_after(const T& data) override;
    void insert_before(const T& data, const T& before) override;
    void insert_after(const T& data, const T& after) override;
    void remove(const T& data) override;
    void clear() override;

    bool empty() const override;

    void reset() override;
    void next(const int count = 1) override;
    bool ended() const override;

    TNode<T>* head() const { return pHead };
};

template <typename T>
TRingList<T>::TRingList() {
    pHead = new TNode<T>();
    pHead->pnext = pHead;
    pFirst = pHead;
    pLast = pHead;
    pStop = pHead;
    pCurr = pHead;
}

template <typename T>
TRingList<T>::TRingList(TNode<T>* _pFirst) {
    pHead = new TNode<T>();
    pHead->pnext = pHead;
    pFirst = pHead;
    pLast = pHead;
    pStop = pHead;
    pCurr = pHead;

    if (_pFirst != nullptr) {
        pFirst = _pFirst;
        TNode<T>* tmp = _pFirst;

        // Найдем последний узел в переданном списке
        while (tmp->pnext != nullptr) {
            tmp = tmp->pnext;
        }

        tmp->pnext = pHead; // Сделаем список кольцевым
    }
}
```

```

        pLast = tmp;
    }

    reset();
}

template <typename T>
TRingList<T>::TRingList(const TRingList<T>& obj) : TList<T>(obj) {
    pHead = new TNode<T>();
    pHead->pnext = pHead;
    pFirst = pHead;
    pLast = pHead;
    pStop = pHead;
    pCurr = pHead;

    TNode<T>* tmp = obj.pFirst;
    while (tmp != obj.pStop) {
        insert_last(tmp->data);
        tmp = tmp->pnext;
    }
}

template <typename T>
TRingList<T>::~~TRingList() {
    clear();
    delete pHead;
}

template <typename T>
TNode<T>* TRingList<T>::find(const T& data) {
    TNode<T>* tmp = pFirst;
    while (tmp != pHead) {
        if (tmp->data == data) {
            return tmp;
        }
        tmp = tmp->pnext;
    }
    return nullptr; // Элемент не найден
}

template <typename T>
void TRingList<T>::insert_first(const T& data) {
    TNode<T>* tmp = new TNode<T>(data, pHead->pnext);
    pHead->pnext = tmp;
    if (pFirst == pHead) {
        pLast = tmp;
    }
    pFirst = tmp;
    reset();
}

template <typename T>
void TRingList<T>::insert_last(const T& data) {
    TNode<T>* tmp = new TNode<T>(data, pHead);
    pLast->pnext = tmp;
    pLast = tmp;
    if (pFirst == pHead) {
        pFirst = tmp;
    }
    reset();
}

```

```

template <typename T>
void TRingList<T>::insert_before(const T& data) {
    if (pCurr == pHead) {
        insert_last(data);
        return;
    }

    TNode<T>* before = pHead;
    while (before->pnext != pCurr) {
        before = before->pnext;
    }

    TNode<T>* tmp = new TNode<T>(data, pCurr);
    before->pnext = tmp;

    if (pCurr == pFirst) {
        pFirst = tmp;
    }

    reset();
}

template <typename T>
void TRingList<T>::insert_after(const T& data) {
    TNode<T>* tmp = new TNode<T>(data, pCurr->pnext);
    pCurr->pnext = tmp;
    if (pCurr == pLast) {
        pLast = tmp;
    }
    reset();
}

template <typename T>
void TRingList<T>::insert_before(const T& data, const T& before) {
    pCurr = find(before);
    if (pCurr == nullptr) {
        std::string exp = "Error: Element not found";
        throw exp;
    }

    insert_before(data);
    reset();
}

template <typename T>
void TRingList<T>::insert_after(const T& data, const T& after) {
    pCurr = find(after);
    if (pCurr == nullptr) {
        std::string exp = "Error: Element not found";
        throw exp;
    }

    insert_after(data);
    reset();
}

template <typename T>
void TRingList<T>::remove(const T& data) {
    TNode<T>* pNode = pHead->pnext;
    TNode<T>* pPrev = pHead;

    while (pNode != pHead && pNode->data != data) {
        pPrev = pNode;
    }
}

```

```

        pNode = pNode->pnext;
    }

    if (pNode == pHead) {
        std::string exp = "Error: data not found";
        throw exp;
    }

    pPrev->pnext = pNode->pnext;

    if (pNode == pLast) {
        pLast = pPrev;
    }

    if (pNode == pFirst) {
        pFirst = pNode->pnext;
    }

    delete pNode;
    reset();
}

template <typename T>
void TRingList<T>::clear() {
    TNode<T>* tmp = pHead->pnext;
    while (tmp != pHead) {
        TNode<T>* next = tmp->pnext;
        delete tmp;
        tmp = next;
    }
    pHead->pnext = pHead;
    pFirst = pHead;
    pLast = pHead;
    reset();
}

template <typename T>
bool TRingList<T>::empty() const {
    return (pHead->pnext == pHead);
}

template <typename T>
void TRingList<T>::reset() {
    pCurr = pHead->pnext;
}

template <typename T>
void TRingList<T>::next(const int count) {
    for (int i = 0; i < count; ++i) {
        if (pCurr != pHead) {
            pCurr = pCurr->pnext;
        }
    }
}

template <typename T>
bool TRingList<T>::ended() const {
    return (pCurr == pHead);
}

```

Приложение Г. Реализация класса TMonom

```
#include <iostream>
#include <string>

class TMonom{
private:
    double coeff;
    int degree;
public:
    TMonom();
    TMonom(const TMonom& monom);
    TMonom(double _coeff, int _degree);
    TMonom(double _coeff, int degX, int degY, int degZ);

    double get_coeff() const;
    int get_degree() const;

    int get_degX() const;
    int get_degY() const;
    int get_degZ() const;

    void set_coeff(double _coeff);
    void set_degree(int _degree);
    void set_degree(int degX, int degY, int degZ);

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

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

    const TMonom& operator=(const TMonom& data);
    TMonom operator*(const TMonom& monom) const;

    TMonom dif_x() const;
    TMonom dif_y() const;
    TMonom dif_z() const;

    std::string get_string() const;
    friend std::ostream& operator<<(std::ostream& out, const TMonom& monom);
};

TMonom::TMonom() {
    coeff = 0;
    degree = -1;
}

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

TMonom::TMonom(double _coeff, int _degree) {
    if (_degree > 999 || _degree < 0) {
        std::string exp = "Error: degree must be in [0, 999]";
        throw exp;
    }

    coeff = _coeff;
    degree = _degree;
}
```



```

TMonom::TMonom(double _coeff, int degX, int degY, int degZ) {
    if (degX < 0 || degY < 0 || degZ < 0 || degX > 9 || degY > 9 || degZ >
9) {
        std::string exp = "Error: x,y,z must be in [0, 9]";
        throw exp;
    }

    coeff = _coeff;
    degree = 100 * degX + 10 * degY + degZ;
}

double TMonom::get_coeff() const {
    return coeff;
}
int TMonom::get_degree() const {
    return degree;
}

int TMonom::get_degX() const {
    int degX = degree / 100;
    return degX;
}
int TMonom::get_degY() const {
    int degY = (degree / 10) % 10;
    return degY;
}
int TMonom::get_degZ() const {
    int degZ = degree % 10;
    return degZ;
}

void TMonom::set_coeff(double _coeff) {
    coeff = _coeff;
}
void TMonom::set_degree(int _degree) {
    if (_degree > 999 || _degree < 0) {
        std::string exp = "Error: degree must be in [0, 999]";
        throw exp;
    }

    degree = _degree;
}
void TMonom::set_degree(int degX, int degY, int degZ) {
    if (degX < 0 || degY < 0 || degZ < 0 || degX > 9 || degY > 9 || degZ >
9) {
        std::string exp = "Error: degX,degY,degZ must be in [0, 9]";
        throw exp;
    }

    degree = 100 * degX + 10 * degY + degZ;
}

double TMonom::operator()(double x, double y, double z) const {
    double resX = 1, resY = 1, resZ = 1;

    int degZ = get_degZ();
    int degY = get_degY();
    int degX = get_degX();

    for (int i = 0; i < degZ; i++) {

```

```

        resZ *= z;
    }
    for (int i = 0; i < degY; i++) {
        resY *= y;
    }
    for (int i = 0; i < degX; i++) {
        resX *= x;
    }

    return coeff * resX * resY * resZ;
}

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

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

    return (*this);
}

TMonom TMonom::operator*(const TMonom& monom) const {
    int degZ1 = get_degZ();
    int degY1 = get_degY();
    int degX1 = get_degX();

    int degZ2 = monom.get_degZ();
    int degY2 = monom.get_degY();
    int degX2 = monom.get_degX();

    if (degZ1 + degZ2 > 9 || degY1 + degY2 > 9 || degX1 + degX2 > 9 ||
        degZ1 + degZ2 < 0 || degY1 + degY2 < 0 || degX1 + degX2 < 0) {
        std::string exp = "Error: res_degrees must be in [0, 9]";
        throw exp;
    }

    int res_degree = degree + monom.degree;
    double res_coeff = coeff * monom.coeff;

    TMonom res(res_coeff, res_degree);
    return res;
}

```

```

TMonom TMonom::dif_x() const {
    int degX = get_degX();

    if (degX == 0) {
        TMonom res(0, 0);
        return res;
    }
    else {
        double _coeff = coeff * degX;
        int _degree = degree - 100;

        TMonom res(_coeff, _degree);
        return res;
    }
}

TMonom TMonom::dif_y() const {
    int degY = get_degY();

    if (degY == 0) {
        TMonom res(0, 0);
        return res;
    }
    else {
        double _coeff = coeff * degY;
        int _degree = degree - 10;

        TMonom res(_coeff, _degree);
        return res;
    }
}

TMonom TMonom::dif_z() const {
    int degZ = get_degZ();

    if (degZ == 0) {
        TMonom res(0, 0);
        return res;
    }
    else {
        double _coeff = coeff * degZ;
        int _degree = degree - 1;

        TMonom res(_coeff, _degree);
        return res;
    }
}

std::string TMonom::get_string() const {
    int degZ = get_degZ();
    int degY = get_degY();
    int degX = get_degX();
    std::string res = "";

    res += std::to_string(coeff);

    if (degX == 1) res += "x";
    else if (degX > 1) res += "x^" + std::to_string(degX);

    if (degY == 1) res += "y";
    else if (degY > 1) res += "y^" + std::to_string(degY);

    if (degZ == 1) res += "z";

```

```

        else if (degZ > 1)      res += "*z^" + std::to_string(degZ);

        return res;
    }
    std::ostream& operator<<(std::ostream& out, const TMonom& monom) {
        return out << monom.get_string();
    }

```

Приложение Д. Реализация класса TPolynom

```
#include <iostream>
#include <string>
#include <map>
#include <vector>

#include "tringlist.h"
#include "tmonom.h"

using namespace std;

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

    static map<string, int> priority;

    void parse(string& name);
    void to_monoms(vector<string>& lexems);
    void add_monom(const TMonom& m);
public:
    TPolynom();
    TPolynom(const string& _name);
    TPolynom(TRingList<TMonom>& ringlist);
    TPolynom(TPolynom& polynom);
    ~TPolynom();

    bool operator<(TPolynom& polynom);
    bool operator>(TPolynom& polynom);
    bool operator<=(TPolynom& polynom);
    bool operator>=(TPolynom& polynom);
    bool operator==(TPolynom& polynom);
    bool operator!=(TPolynom& polynom);

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

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

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

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

private:
    bool is_operator(const string& isopr) const;
    bool is_const(const string& isopd) const;
    bool is_variable(const string& str) const;

    int find_operator(const string& name, int pos = 0) const;

    void convert_infix(string& name);
    void correctness_check(const string& name) const;

    void reset();
    void next();
```

```

    bool ended();
    TNode<TMonom>* get_curr();
    TMonom get_current_monom();
};

map<string, int> TPolynom::priority = {
    {"^", 4},
    {"*", 3},
    {"/", 3},
    {"+", 2},
    {"-", 2},
    {"(", 1},
    {")", 1}
};

bool TPolynom::is_operator(const string& isopr) const { // opr : priority
    can't be char
    bool flag = false;
    for (const auto& opr : priority) {
        if (isopr == opr.first) {
            flag = true;
            break;
        }
    }
    return flag;
}

bool TPolynom::is_const(const string& isopd) const {
    bool flag = true;
    for (int i = 0; i < isopd.size(); i++)
        if (isopd[i] < '0' || isopd[i] > '9') {
            if (isopd[i] != '.')
                flag = false;
            break;
        }
    return flag;
}

bool TPolynom::is_variable(const string& str) const {
    return (str == "x" || str == "y" || str == "z");
}

// If found any operator, returns index. Else returns -1.
int TPolynom::find_operator(const string& name, int pos) const {
    if (pos < 0 || pos >= name.size()) return -1;

    int ind = -1;
    for (int i = pos; i < name.size(); i++) {
        string isopr;
        isopr += name[i];

        if (is_operator(isopr)) {
            ind = i;
            break;
        }
    }
    return ind;
}

void TPolynom::convert_infix(string& name) {
    string nospaces;
    for (int i = 0; i < name.size(); i++) {
        if (name[i] != ' ')
            nospaces += name[i];
    }
}

```

```

name = nospaces;

string tmp;
if (name[0] == '-') tmp += "0-";
else if (name[0] == '.') tmp += "0.";
else if (name[0] == '(') tmp += '(';
else tmp += name[0];

for (int i = 1; i < name.size() - 1; i++) {
    char elem = name[i];
    if (elem == ' ') {
        continue;
    }
    else if (elem == '-') {
        if (name[i - 1] == '(')
            tmp += '0';
        tmp += '-';
    }
    else if (elem == '(') {
        if (name[i - 1] == ')' || name[i - 1] == '.' || (name[i - 1]
>= '0' && name[i - 1] <= '9'))
            tmp += '*';
        tmp += '(';
    }
    else if (elem == '.') {
        if (name[i - 1] < '0' || name[i - 1] > '9') {
            if (name[i - 1] == ')')
                tmp += '*';
            tmp += '0';
        }
        tmp += '.';
        if (name[i + 1] < '0' || name[i + 1] > '9') {
            tmp += '0';
            if (name[i + 1] == '(')
                tmp += '*';
        }
    }
    else if (elem >= '0' && elem <= '9') {
        if (name[i - 1] == ')')
            tmp += '*';
        tmp += elem;
    }
    else {
        tmp += elem;
    }
}
if (name[name.size() - 1] == '.') tmp += ".0";
else if (name[name.size() - 1] == ')') tmp += ')';
//else tmp += name[name.size() - 1];
else if (tmp.size() != 1) tmp += name[name.size() - 1];
name = tmp;
}

void TPolynom::correctness_check(const string& name) const {
    int op_bracket = 0;
    int cl_bracket = 0;
    int dot = 0;

    string exp = "Incorrect arithmetic expression";

    if (name[0] == '+' || name[0] == '*' || name[0] == '/' ||
        name[0] == '^') throw exp;
    if (name[name.size() - 1] == '+' || name[name.size() - 1] == '-' ||

```

```

        name[name.size() - 1] == '*' || name[name.size() - 1] == '/' ||
        name[name.size() - 1] == '(' || name[name.size() - 1] == '^')
throw exp;

    if (name[0] == '(') op_bracket++;
    else if (name[0] == '.') dot++;
    if (name[name.size() - 1] == ')') cl_bracket++;
    else if (name[name.size() - 1] == '.') dot++;

    for (int i = 1; i < name.size() - 1; i++) {
        char elem = name[i];
        if (elem == '(')
            op_bracket++;
        else if (elem == ')') {
            if (name[i - 1] == '(' || name[i - 1] == '+' || name[i - 1]
== '-' ||
            name[i - 1] == '*' || name[i - 1] == '/' || name[i -
1] == '^') throw exp;
            cl_bracket++;
        }
        else if (elem == '.')
            dot++;
        else if (elem == '+' || elem == '-' || elem == '*' || elem == '/'
|| elem == '^') {
            if (dot > 1) throw exp;
            dot = 0;
            if (name[i - 1] == '+' || name[i - 1] == '-' || name[i - 1]
== '*' ||
            name[i - 1] == '/' || name[i - 1] == '(' || name[i -
1] == '^') throw exp;
        }
        if (op_bracket != cl_bracket) throw exp;
    }

void TPolynom::parse(string& name) {
    convert_infix(name);
    correctness_check(name);

    vector<string> lexems;
    int firstind, secondind;
    string token;

    firstind = find_operator(name);
    secondind = find_operator(name, firstind + 1);
    if (firstind == -1) {
        lexems.push_back(name);
        to_monoms(lexems);
        return;
    }
    if (firstind > 0) {
        lexems.push_back(name.substr(0, firstind));
    }
    while (secondind != -1) {
        string opr = name.substr(firstind, 1);
        string opd = name.substr(firstind + 1, secondind - firstind - 1);

        lexems.push_back(opr);
        if (opd != "")
            lexems.push_back(opd);
        firstind = secondind;
        secondind = find_operator(name, firstind + 1);
    }
}

```



```

        lexems.push_back(name.substr(firstind, 1));
        if (firstind != name.size() - 1)
            lexems.push_back(name.substr(firstind + 1));

        to_monoms(lexems);
    }

// ===== //

void TPolynom::to_monoms(vector<string>& _lexems) {
    double coeff = 1;
    int degX = 0, degY = 0, degZ = 0;
    int next_const_sign = 1;

    vector<string> lexems;
    for (const string& token : _lexems) {
        if (token != "*" && token != "^") {
            lexems.push_back(token);
        }
    }

    for (int i = 0; i < lexems.size(); i++) {
        if (lexems[i] == "+" || lexems[i] == "-") {
            TMonom monom(next_const_sign * coeff, degX, degY, degZ);
            add_monom(monom);

            if (lexems[i] == "+") next_const_sign = 1;
            else next_const_sign = -1;
            coeff = 1;
            degX = 0;
            degY = 0;
            degZ = 0;
        }
        else {
            if (lexems[i] == "x") {
                if (i < lexems.size() - 1) {
                    if (!is_variable(lexems[i + 1]) &&
                        !is_operator(lexems[i + 1])) {
                        degX += stoi(lexems[i + 1]);
                        i++;
                    }
                    else {
                        degX += 1;
                    }
                }
                else {
                    degX += 1;
                }
            }
            else if (lexems[i] == "y") {
                if (i < lexems.size() - 1) {
                    if (!is_variable(lexems[i + 1]) &&
                        !is_operator(lexems[i + 1])) {
                        degY += stoi(lexems[i + 1]);
                        i++;
                    }
                    else {
                        degY += 1;
                    }
                }
            }
        }
    }
}

```

```

        else {
            degY += 1;
        }
    }

    else if (lexems[i] == "z") {
        if (i < lexems.size() - 1) {
            if (!is_variable(lexems[i + 1])) &&
!is_operator(lexems[i + 1])) {
                degZ += stoi(lexems[i + 1]);
                i++;
            }
            else {
                degZ += 1;
            }
        }
        else {
            degZ += 1;
        }
    }

    else {
        if (!is_const(lexems[i])) {
            string exp = "Error: not valid operand";
            throw exp;
        }
        coeff *= stod(lexems[i]);
    }
}

}

TMonom monom(next_const_sign * coeff, degX, degY, degZ);
add_monom(monom);
}

void TPolynom::add_monom(const TMonom& m) {
    if (m.get_coeff() == 0) return;

    while (!monoms->ended()) {
        TNode<TMonom>* curr = monoms->get_curr();

        if (m == curr->data) {
            double coeff = m.get_coeff() + curr->data.get_coeff();
            if (coeff == 0.0f) {
                monoms->remove(curr->data);
                return;
            }
            else {
                curr->data.set_coeff(coeff);
                return;
            }
        }

        else if (m < curr->data) {
            monoms->insert_before(m, curr->data);
            return;
        }

        else {
            monoms->next();
        }
    }
}

```

```

        monoms->insert_last(m);
    }

TPolynom::TPolynom() {
    monoms = new TRingList<TMonom>;
}
TPolynom::TPolynom(const string& _name) {
    monoms = new TRingList<TMonom>;
    string name = _name;
    if (name.size() == 0) name += "0";

    parse(name);
}
TPolynom::TPolynom(TRingList<TMonom>& ringlist) {
    monoms = new TRingList<TMonom>;

    ringlist.reset();
    while (!ringlist.ended()) {
        add_monom(ringlist.get_curr()->data);
        ringlist.next();
    }
    ringlist.reset();
}
TPolynom::TPolynom(TPolynom& polynom) {
    monoms = new TRingList<TMonom>(*polynom.monoms);
}
TPolynom::~TPolynom() {
    if (monoms) delete monoms;
}

bool TPolynom::operator<(TPolynom& polynom) {
    reset();
    while (!ended()) {
        polynom.reset();
        while (!polynom.ended()) {
            if (!(get_current_monom() < polynom.get_current_monom())) {
                return false;
            }
            polynom.next();
        }
        next();
    }
    reset();
    polynom.reset();
    return true;
}
bool TPolynom::operator>(TPolynom& polynom) {
    reset();
    while (!ended()) {
        polynom.reset();
        while (!polynom.ended()) {
            if (!(get_current_monom() >= polynom.get_current_monom())) {
                return false;
            }
            polynom.next();
        }
        next();
    }
    reset();
    polynom.reset();
}

```

```

        return true;
    }
    bool TPolynom::operator<=(TPolynom& polynom) {
        reset();
        while (!ended()) {
            polynom.reset();
            while (!polynom.ended()) {
                if (!(get_current_monom() <= polynom.get_current_monom())) {
                    return false;
                }
                polynom.next();
            }
            next();
        }
        reset();
        polynom.reset();
        return true;
    }
    bool TPolynom::operator>=(TPolynom& polynom) {
        reset();
        while (!ended()) {
            polynom.reset();
            while (!polynom.ended()) {
                if (!(get_current_monom() >= polynom.get_current_monom())) {
                    return false;
                }
                polynom.next();
            }
            next();
        }
        reset();
        polynom.reset();
        return true;
    }
    bool TPolynom::operator==(TPolynom& polynom) {
        reset();
        while (!ended()) {
            polynom.reset();
            while (!polynom.ended()) {
                if (!(get_current_monom() == polynom.get_current_monom())) {
                    return false;
                }
                polynom.next();
            }
            next();
        }
        reset();
        polynom.reset();
        return true;
    }
    bool TPolynom::operator!=(TPolynom& polynom) {
        return !((*this) == polynom);
    }

    const TPolynom& TPolynom::operator=(const TPolynom& polynom) {
        if (this == &polynom) return (*this);

        if (monoms) delete monoms;
        monoms = new TRingList<TMonom>(*polynom.monoms);

        return (*this);
    }

```

```

TPolynomial TPolynom::operator+(TPolynomial& polynom) {
    TPolynom res(*this);

    polynom.reset();
    while (!polynom.ended()) {
        TMonom curr_monom = polynom.get_current_monom();
        res.add_monom(curr_monom);
        polynom.next();
    }
    polynom.reset();

    return res;
}

TPolynomial TPolynom::operator-() {
    TPolynom res(*this);

    res.reset();
    while (!res.ended()) {
        TNode<TMonom>* curr = res.get_curr();
        double coeff = (-1) * curr->data.get_coeff();
        curr->data.set_coeff(coeff);
        res.next();
    }
    res.reset();

    return res;
}

TPolynomial TPolynom::operator-(TPolynomial& polynom) {
    TPolynom res(-polynom + (*this));

    return res;
}

TPolynomial TPolynom::operator*(TPolynomial& polynom) {
    TPolynom res;

    reset();
    while (!ended()) {
        TMonom curr1 = get_current_monom();

        polynom.reset();
        while (!polynom.ended()) {
            TMonom curr2 = polynom.get_current_monom();
            res.add_monom(curr1 * curr2);
            polynom.next();
        }

        next();
    }

    polynom.reset();
    reset();

    return res;
}

double TPolynom::operator()(double x, double y, double z) {
    double res = 0;

    reset();
    while (!ended()) {
        res += get_current_monom()(x, y, z);
        next();
    }
}

```

```

    }
    reset();

    return res;
}

TPolynom TPolynom::dif_x() {
    TPolynom res;

    reset();
    while (!ended()) {
        res.add_monom(get_current_monom().dif_x());
        next();
    }
    reset();

    return res;
}

TPolynom TPolynom::dif_y() {
    TPolynom res;

    reset();
    while (!ended()) {
        res.add_monom(get_current_monom().dif_y());
        next();
    }
    reset();

    return res;
}

TPolynom TPolynom::dif_z() {
    TPolynom res;

    reset();
    while (!ended()) {
        res.add_monom(get_current_monom().dif_z());
        next();
    }
    reset();

    return res;
}

string TPolynom::get_string() {
    if (monoms->size() == 0) return "0.000000";

    string res = "";
    reset();
    if (get_curr() == nullptr) {
        return res;
    }
    else {
        res += get_current_monom().get_string();
        next();
        while (!ended()) {
            string monom = get_current_monom().get_string();
            if (monom[0] == '-') res += monom;
            else res += "+" + monom;
            next();
        }
    }
}

```

```

        reset();

        return res;
    }
    ostream& operator<<(ostream& out, TPolynom& monom) {
        return out << monom.get_string();
    }

void TPolynom::reset() {
    monoms->reset();
}
void TPolynom::next() {
    monoms->next();
}
bool TPolynom::ended() {
    return monoms->ended();
}
TNode<TMonom>* TPolynom::get_curr() {
    return monoms->get_curr();
}
TMonom TPolynom::get_current_monom() {
    return monoms->get_curr()->data;
}

```