МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования   
**«Национальный исследовательский   
Нижегородский государственный университет им. Н.И. Лобачевского»**

**(ННГУ)**

**Институт информационных технологий, математики и механики**

**ЛАБОРАТОРНАЯ РАБОТА**

на тему:

**«Полиномы»**

**Выполнил(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 4](#__RefHeading___1)

[Постановка задачи 6](#__RefHeading___2)

[Руководство пользователя 7](#__RefHeading___3)

[1.1 Приложение для демонстрации работы полинома 7](#__RefHeading___4)

[Руководство программиста 8](#__RefHeading___5)

[1.2 Описание алгоритмов 8](#__RefHeading___6)

[1.2.1 Список 8](#__RefHeading___7)

[1.2.2 Кольцевой список 11](#__RefHeading___8)

[1.2.3 Моном 12](#__RefHeading___9)

[1.2.4 Полином 14](#__RefHeading___10)

[1.2.5 Стек 15](#__RefHeading___11)

[1.2.6 Арифметические выражения 17](#__RefHeading___12)

[1.3 Описание программной реализации 18](#__RefHeading___13)

[1.3.1 Описание структуры TNode 18](#__RefHeading___14)

[1.3.2 Описание класса TList 19](#__RefHeading___15)

[1.3.3 Описание класса TRingList 23](#__RefHeading___16)

[1.3.4 Описание класса TMonom 25](#__RefHeading___17)

[1.3.5 Описание класса TPolynom 26](#__RefHeading___18)

[1.3.6 Описание класса TStack 33](#__RefHeading___19)

[1.3.7 Описание класса ArithmeticExpression 35](#__RefHeading___20)

[Заключение 38](#__RefHeading___21)

[Литература 39](#__RefHeading___22)

[Приложения 40](#__RefHeading___23)

[Приложение А. Реализация структуры TNode 40](#__RefHeading___24)

[Приложение Б. Реализация класса TList 40](#__RefHeading___25)

[Приложение В. Реализация класса TRingList 43](#__RefHeading___26)

[Приложение Г. Реализация класса TMonom 45](#__RefHeading___27)

[Приложение Д. Реализация класса TPolynom 45](#__RefHeading___28)

[Приложение Е. Реализация класса TStack 57](#__RefHeading___29)

[Приложение Ж. Реализация класса ArithmeticExpression 59](#__RefHeading___30)

# 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.

# 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.

# Руководство пользователя

## Приложение для демонстрации работы полинома

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



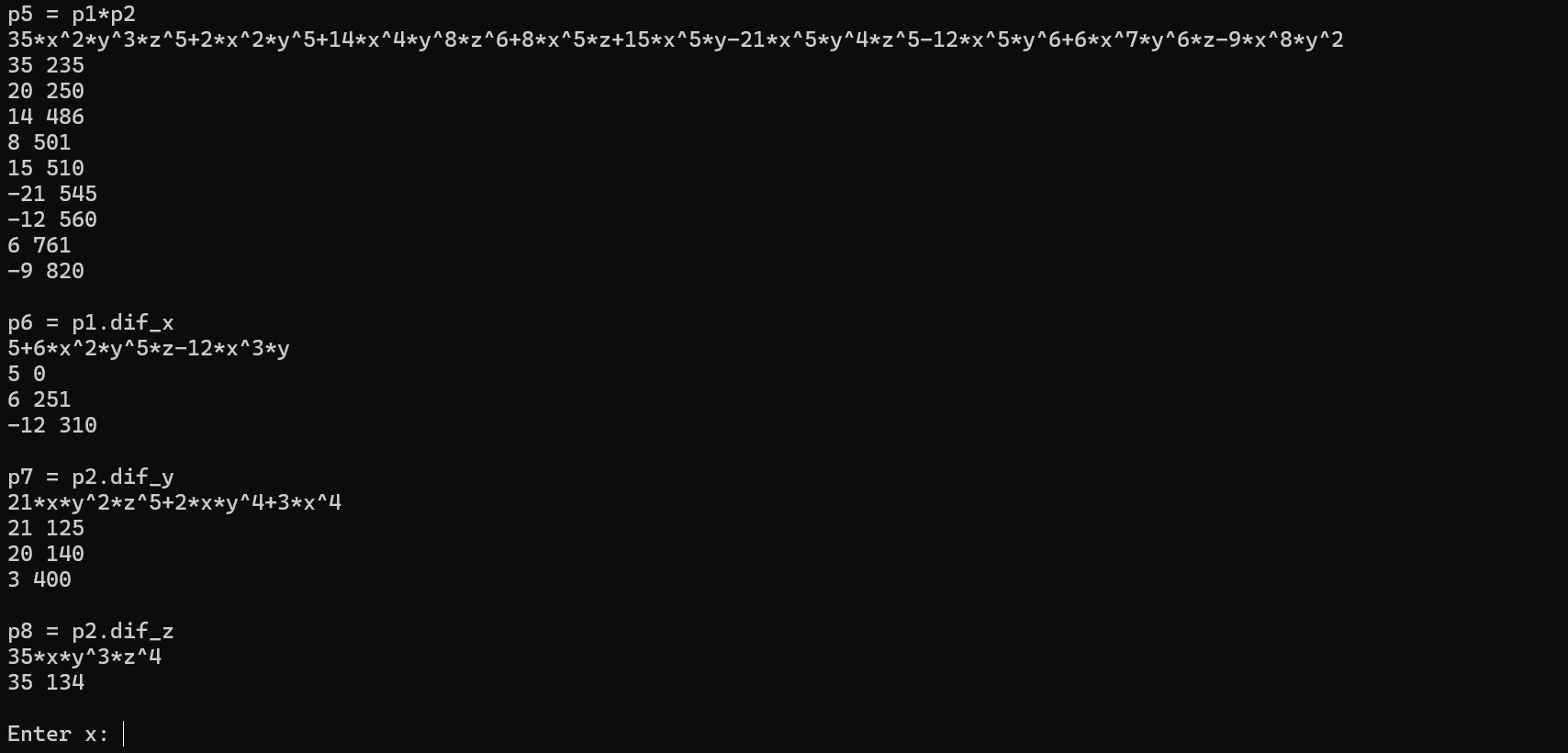
1. Основное окно программы
2. Введите первый полином, а затем второй (Рис. 2).



1. Ввод полиномов
2. В результате появится следующее окно (Рис. 3) и (Рис. 4).



1. Результат арифметических выражений



1. Результат арифметических выражений
2. Введите значения x, y, z, после этого выведется результат вычисления полинома (Рис. 5)



1. Результат вычислений

# Руководство программиста

## Описание алгоритмов

### Список

Структура данных TList представляет собой линейный односвязный список в виде шаблонного класса. Указателем на начало списка является pFirst, указателем на конец pStop, который по умолчанию равен nullptr. 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 вставляет новый узел с заданными данными в начало связанного списка. Если список пустой, создается новый узел и он становится и первым, и последним. Первоначально созданный узел указывает на конечный элемент списка.



Добавим элемент 7.

Результат:



**Операция вставки в конец**

Функция используется для вставки нового узла с заданными данными в конец связанного списка. Если список пустой, метод вызывает операцию вставки в начало для создания первого узла. Затем новый узел создается с указателем на конечный элемент, и текущий последний узел списка указывает на новый узел.



Добавим элемент 4 в конец.

Результат:



**Операция поиска**

Функция используется для поиска узла с заданными данными в связанном списке. Метод последовательно проходит по каждому узлу, сравнивая его данные с заданными. Как только найден узел с заданными данными или достигнут конец списка (узел pStop), метод завершает поиск и возвращает найденный узел или указатель на конечный элемент, если узел с заданными данными не был найден.

Найдем элемент 6.



Вернется указатель на 6.

Найдем 7.



Вернется указатель на pStop(nullptr).

**Операция вставки перед заданным элементом**

Функция вставляет новый узел с заданными данными перед указанным узлом. Если указанный узел не найден, генерируется исключение с сообщением "Element not found!". При успешном поиске новый узел создается и вставляется перед указанным узлом.



Добавим 4 перед 1.

Результат:



**Операция вставки после заданного элемента**

Функция вставляет новый узел с заданными данными после указанного узла.. Если указанный узел не найден, генерируется исключение с сообщением "Element not found!". При успешном поиске новый узел создается и вставляется между указанным узлом и следующим за ним узлом.



Добавим 4 после 1.



**Операция удаления**

Функция используется для удаления узла с указанными данными из связанного списка. Сначала проверяется, пуст ли список, и в случае пустого списка, генерируется исключение. Затем происходит поиск узла с указанными данными в списке(pNode). Как только узел найден, он удаляется из списка путем перенаправления указателей pPrev->pNext на pNode->pNext. При удалении первого узла, указатель pFirst переназначается на следующий узел, и удаляемый узел освобождается. Если удаляемый узел последний в списке, указатель pLast переназначается на предыдущий элемент.



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.

### Кольцевой список

Структура данных TRingList представляет собой кольцевой линейный односвязный список в виде шаблонного класса. При создании списка, создается фиктивный элемент pHead, который будет указывать на pFirst. Указателем на начало списка является pFirst, указателем на конец pStop, который по умолчанию равен pHead. 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, pHead->pNext обновляется на pFirst, а указатель pLast->pNext 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 новый указатель pLast->pNext на созданный узел, а затем сам указатель pLast обновляется до нового узла, чтобы поддерживать кольцевую структуру списка.



Добавим элемент 4 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 pLast->pNext на pHead, чтобы поддерживать кольцевую структуру списка.



Удалим 4.

Результат:



### Моном

Тип данных TMonom 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.

### Полином

Структура данных TPolynom представляет собой класс, который содержит строку с полиномом и кольцевой линейный односвязный список с мономами. Структура данных TPolynom поддерживает следующие операции: сложение, вычитание, умножение полиномов, взятие производных. Также поддерживается функция вычисления значения полинома с заданными значениями x, y, и z.

**Операция сложения полиномов**

Функция реализует сложение полиномов. Полиномы разбиваются на мономы, они сравниваются и создают новый полином, путем создания упорядоченного списка мономов. Если степени мономов равны, то складываются их коэффициенты.

Пример:

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

**Операция вычитания полиномов**

Функция реализует вычитание полиномов. Полиномы разбиваются на мономы, они сравниваются и создают новый полином, меняя знаки второго полинома на противоположные, путем создания упорядоченного списка мономов. Если степени мономов равны, то вычитаются их коэффициенты.

Пример:

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

**Операция умножения полиномов**

Функция реализует умножение полиномов. Мономы умножаются каждый с каждым, степени складываются, коэффициенты умножаются, создавая таким образом новый полином, путем создания упорядоченного списка мономов.

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**

Функция высчитывает производную по x. Коэффициент монома умножается на степень x, затем из степени x вычитается единица. Если степень становится равной 0, то переменная x становится равной 1.

Пример:

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

**Операция взятия производной по y**

Функция высчитывает производную по y. Коэффициент монома умножается на степень y, затем из степени y вычитается единица. Если степень становится равной 0, то переменная y становится равной 1.

Пример:

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

**Операция взятия производной по z**

Функция высчитывает производную по z. Коэффициент монома умножается на степень z, затем из степени z вычитается единица. Если степень становится равной 0, то переменная z становится равной 1.

Пример:

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

**Операция вычисления значения полинома**

При помощи перевода в постфиксную форму, функция высчитывает значение полинома с заданными значениями переменных x, y, z.

Пример:

x = 2, y = 3, z = 4

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

### Стек

Структура данных TStack представляет собой шаблонный класс. Для создания экземпляра TStack необходимо указать его размер. По умолчанию размер устанавливается как 10. Структура данных TStack также поддерживает ряд операций, включая добавление на вершину, изъятие с вершины, проверка последнего элемента, проверка на пустоту и полноту. Также присутствуют операции проверки элемента по индексу и размера стека.

**Операция добавления в стек**

Функция добавляет элемент на вершину стека. Для этого используется флаг top, который показывает, элемент с каким индексом считается вершиной стека. Если стек пуст top=-1. В примере top=2.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 6 | 1 |  |  |

Добавим элемент 7

Результат:

top=3, элемент в стеке

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 6 | 1 | 7 |  |

**Операция изъятия с вершины**

Функция удаляет элемент из стека. Для этого также воспользуемся флагом top=4. Если стек пуст, то будет выбрасываться исключение.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 6 | 1 | 7 | 2 |

Результат:

Флаг top теперь стал равен 3.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 6 | 1 | 7 | 2 |

**Операция проверки на пустоту**

Функция проверяет есть ли в стеке элементы. Если в стеке нет элементов, то функция вернет true, в противном случае false.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 6 | 1 | 7 |  |

Результат:

false – стек не пуст

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |

Результат:

true – стек пуст

**Операция проверки на полноту**

Функция проверяет , достиг ли стек своей максимальной вместимости элементов. Она возвращает true, если количество элементов в стеке соответствует его максимальному размеру, тем самым указывая на то, что добавление дополнительных элементов невозможно. В противном случае вернет false.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 6 | 1 | 7 | 2 |

Результат:

true – стек полон

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 7 | 8 | 3 | 9 |  |

Результат:

false – стек не полон

**Операция проверки последнего элемента**

Функция позволяет получить значение последнего добавленного элемента, не удаляя его из стека.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 6 | 1 | 7 | 2 |

Результат: 2

### Арифметические выражения

**Операция перевода в постфиксную форму**

Функция перевода в обратную польскую запись в стеке выполняет преобразование инфиксного выражения в соответствующее постфиксное выражение.

A+(B-C)\*D-F/(G+H)

Стек 1.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | - |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | / |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | + | + |
|  |  |  |  |  |  |  |  |  |  |  |  |  | H | H | H |
|  |  |  |  |  |  |  |  |  |  |  | G | G | G | G | G |
|  |  |  |  |  |  |  |  | F | F | F | F | F | F | F | F |
|  |  |  |  |  |  |  | + | + | + | + | + | + | + | + | + |
|  |  |  |  |  |  |  | \* | \* | \* | \* | \* | \* | \* | \* | \* |
|  |  |  |  |  |  | D | D | D | D | D | D | D | D | D | D |
|  |  |  |  |  | - | - | - | - | - | - | - | - | - | - | - |
|  |  |  |  | C | C | C | C | C | C | C | C | C | C | C | C |
|  |  | B | B | B | B | B | B | B | B | B | B | B | B | B | B |
| A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |

Стек 2.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  | ) |  |  |
|  |  |  |  | ) |  |  |  |  |  |  |  | + | + |  |  |
|  |  |  | - | - |  |  |  |  |  | ( | ( | ( | ( |  |  |
|  | ( | ( | ( | ( |  | \* |  |  | / | / | / | / | / | / |  |
| + | + | + | + | + | + | + | - | - | - | - | - | - | - | - |  |

**Операция вычисления по постфиксной форме**

Эта функция проходит по каждому токену в постфиксном выражении. Если токен является операндом, он помещается в стек. Если токен является оператором, из стека извлекаются два операнда, над которыми выполняется соответствующая операция. Результат операции помещается обратно в стек. После обработки всех токенов в постфиксном выражении, в стеке остается только один элемент - результат вычисления выражения, который извлекается из стека и возвращается в качестве результата функции.

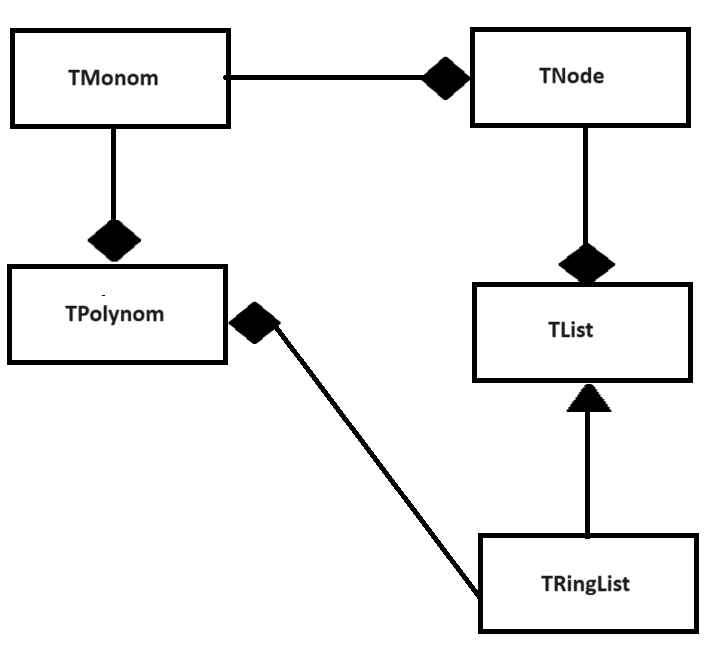
Вычислим значение выражения, представленного выше.

A=0, B=1, C=2, D=-1, F=2, G=0.5, H=0.5

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  | + |  |  |  |  |  |
|  |  |  | - |  | \* |  |  |  |  |  | 0.5 | 0.5 |  | / |  |  |  |
|  |  | 2 | 2 | -1 | -1 |  | + |  |  | 0.5 | 0.5 | 0.5 | 1 | 1 |  | - |  |
|  | 1 | 1 | 1 | -1 | -1 | 1 | 1 |  | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |  |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | -1 |

Результат: -1

## Описание программной реализации



1. Схема наследования классов

### 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.

### 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& beforedata);

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.

### 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**.

### 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**.

### 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:

TPolynom();

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

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

TPolynom(const string s);

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

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

Выходные параметры: отсутствуют.

TPolynom(const TRingList& rlist);

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

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

Выходные параметры: отсутствуют.

TPolynom(const TPolynom& p);

Назначение: конструктор копирования, создает копию существующего объекта TPolynom.

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

Выходные параметры: отсутствуют.

~Tpolynom();

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

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

TPolynom operator+(const TPolynom& p);

Назначение: перегруженный оператор сложения полиномов.

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

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

TPolynom operator-(const TPolynom& p);

Назначение: перегруженный оператор вычитания полиномов.

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

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

TPolynom operator\*(const TPolynom& p);

Назначение: перегруженный оператор умножения полиномов.

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

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

TPolynom operator-() const;

Назначение: перегруженный оператор умножения полиномов на -1.

Входные параметры: отсутствуют.

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

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

Назначение: перегруженный оператор присваивания.

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

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

TPolynom difx() const;

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

Входные параметры: отсутствуют.

Выходные параметры: полином – производная по «x».

TPolynom dify() const;

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

Входные параметры: отсутствуют.

Выходные параметры: полином – производная по «y».

TPolynom 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 «х» 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 «х»

Выходные параметры: отсутствуют.

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

Назначение: обрабатывает степень для переменной «y» 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 «y»

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 «х» 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 «х»,

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

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

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**.

### Описание класса TStack

template <class T>

class TStack

{

private:

int maxSize;

int top;

T\* elems;

public:

TStack(int maxSize = 10);

TStack(const TStack<T>& s);

~TStack();

TStack<T>& operator=(const TStack<T>& s);

bool IsEmpty(void) const;

bool IsFull(void) const;

void ResizeStack();

void ReverseStack();

T Top() const;

void Push(const T& elem);

void Pop();

friend ostream& operator <<(ostream& out, const TStack& s)

{

for (int i = 0; i <= s.top; i++)

out << s.elems[i] << ' ';

return out;

};

};

Назначение: представление вектора.

Поля:

**elems**– указатель на массив типа **T**.

**maxSize** – размер вектора.

**start\_ind** – индекс верхнего элемента в стеке.

Методы:

TStack(int maxSize);

Назначение: конструктор класса, инициализирующий стек заданного размера.

Входные параметры:

**maxSize** – максимальный размер стека, заданный при инициализации.

Выходные параметры: отсутствуют.

TStack(const TStack& s);

Назначение: Конструктор копирования, создающий копию стека.

Входные параметры **s** – объект класса **TStack**, который нужно скопировать.

Выходные параметры: отсутствуют.

~TStack();

Назначение: освобождает выделенную память для стека.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

bool IsEmpty() const;

Назначение: Метод, проверяющий, пуст ли стек.

Входные параметры: отсутствуют.

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

bool IsFull() const;

Назначение: Метод, проверяющий, заполнен ли стек.

Входные параметры: отсутствуют.

Выходные параметры: true – заполнен, false в противном случае.

void ResizeStack();

Назначение: Метод, увеличивающий размер стека при его переполнении.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

const TStack<T>& operator=(const TStack<T>& s);

Назначение: присваивание одного стека другому.

Входные параметры: **s** – объект класса **TStack**, который нужно присвоить.

Выходные параметры: ссылка на текущий объект класса **TStack**.

void ReverseStack();

Назначение: изменение порядка элементов в стеке на обратный.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

T Top() const;

Назначение: Метод, возвращающий верхний элемент стека.

Входные параметры: отсутствуют.

Выходные параметры: верхний элемент стека.

void Push(const T& elem);

Назначение: Метод, помещающий элемент на вершину стека.

Входные параметры: **elem** – элемент, который требуется добавить на вершину стека.

Выходные параметры: отсутствуют.

void Pop();

Назначение: Метод, удаляющий верхний элемент стека.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

friend ostream& operator <<(ostream& out, const TStack& s);

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

Входные параметры:

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

**s** – ссылка на объект типа **TStack,** который будет выводиться.

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

### Описание класса ArithmeticExpression

class ArithmeticExpression

{

public:

static TStack<string> Postfix\_Form(const string& s);

static double Calculate(TStack<string>& st, const map<string, double>& values);

static int Is\_Symbol(string sm);

static int Get\_Priority(const string symbol);

static void Add\_to\_Stack1(TStack<string>& st1, TStack<string>& st2, string s);

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

static bool isValidExpression(const string& expression);

static bool isOperand(char c);

static string FilteredExpression(const string& s);

static map<string, double> GetVariables(TStack<string>& postfixExpression);

};

Методы:

TStack<string> Postfix\_Form(const string& s);

Назначение: преобразование инфиксного выражения в постфиксное.

Входные параметры: **s** – инфиксное выражение.

Выходные параметры: стек, представляющий постфиксное выражение.

double Calculate(TStack<string>& st, const map<string, double>& values);

Назначение: вычисление значения постфиксного выражения.

Входные параметры:

**st** – постфиксное выражение.

**values** – словарь переменных и их значений.

Выходные параметры: значение выражения.

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.

int Get\_Priority(const string symbol);

Назначение: получение приоритета оператора.

Входные параметры: **symbol** – оператор.

Выходные параметры: целое число – приоритет оператора.

void Add\_to\_Stack1(TStack<string>& st1, TStack<string>& st2, string s);

Назначение: добавление элемента в стек **st1** и удаление элемента из стека **st2**

Входные параметры: стек **st1**, стек **st2**, **s** – элемент для добавления в стек **st1**

Выходные параметры: отсутствуют.

bool Is\_Number(const string& str);

Назначение: проверка, является ли переданная строка числом.

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

Выходные параметры: 0 – не является числом, 1 – число.

bool isValidExpression(const string& expression);

Назначение: проверка правильности математического выражения.

Входные параметры: **expression** – математическое выражение для проверки.

Выходные параметры: 1 – выражение введено верно, 0 – в противном случае.

bool isOperand(char c);

Назначение: проверка, является ли переданный символ операндом.

Входные параметры: символ **c**.

Выходные параметры: 1 – операнд, 0 – в противном случае.

string FilteredExpression(const string& s);

Назначение: удаление пробелов из переданной строки.

Входные параметры: строка **s** – выражение для фильтрации.

Выходные параметры: отфильтрованная строка.

map<string, double> GetVariables(TStack<string>& postfixExpression);

Назначение: получение значения переменных из постфиксного выражения.

Входные параметры: стек **postfixExpression** – постфиксное выражение.

Выходные параметры: словарь с уникальными переменными и их значениями.

# 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)

{

name = s;

monoms = new TRingList<TMonom>();

Parse\_Polynom(name);

}

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())

{

if (deg == list.GetCurrent()->data.degree)

{

TMonom m\_curr = list.GetCurrent()->data;

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);

list.InsertBeforeCurr(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, c);

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 == l - 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;

string x = to\_string(n);

deg += x;

i++;

}

else

deg = "100";

}

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;

if (k2 < k)

{

monomList->InsertBeforeCurr(monom);

break;

}

else

{

monomList->InsertAfterCurr(monom);

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);

else if (str[i] == 'y')

HandleY(str, i, deg);

else

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 вывода

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;

}

## Приложение Е. Реализация класса TStack

template <class T>

TStack<T>::TStack<T>(int maxSize)

{

if (maxSize <= 0)

{

string msg = "Error";

throw msg;

}

this->maxSize = maxSize;

top = -1;

elems = new T[maxSize];

}

template <class T>

TStack<T>::TStack<T>(const TStack<T>& s)

{

maxSize = s.maxSize;

top = s.top;

elems = new T[maxSize];

for (int i = 0; i <= top; i++)

elems[i] = s.elems[i];

}

template <class T>

TStack<T>::~TStack()

{

delete[] elems;

}

template <class T>

const TStack<T>& TStack<T>::operator=(const TStack<T>& s)

{

if (this == &s) {

return \*this;

}

delete[] elems;

maxSize = s.maxSize;

top = s.top;

elems = new T[maxSize];

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

elems[i] = s.elems[i];

}

return \*this;

}

template <class T>

bool TStack<T>::IsEmpty(void) const

{

return (top == -1);

}

template <class T>

bool TStack<T>::IsFull(void) const

{

if (maxSize - 1 == top)

return true;

return false;

}

template <class T>

void TStack<T>::ResizeStack()

{

int newMaxSize = maxSize \* 5;

T\* newElems = new T[newMaxSize];

for (int i = 0; i <= top; i++)

newElems[i] = elems[i];

delete[] elems;

elems = newElems;

maxSize = newMaxSize;

}

template <typename T>

void TStack<T>::ReverseStack()

{

TStack<T> tempStack;

while (!IsEmpty())

{

T element = Top();

Pop();

tempStack.Push(element);

}

\*this = tempStack;

}

template <class T>

T TStack<T>::Top() const

{

if (top == -1)

{

string msg = "Error: stack is empty";

throw msg;

}

return elems[top];

}

template <class T>

void TStack<T>::Push(const T& elem)

{

if (IsFull())

ResizeStack();

elems[++top] = elem;

}

template <class T>

void TStack<T>::Pop()

{

if (IsEmpty())

{

string msg = "Error: stack is empty";

throw msg;

}

top -= 1;

}

## Приложение Ж. Реализация класса ArithmeticExpression

map<string, double> ArithmeticSymbol::symbolDict = {

{"\*", 3},

{"/", 3},

{"+", 2},

{"-", 2},

};

int ArithmeticExpression::Is\_Symbol(string sm)

{

if (ArithmeticSymbol::symbolDict.find(sm) != ArithmeticSymbol::symbolDict.end())

return 1;

else if (sm == "(" || sm == ")")

return 2;

return 0;

}

int ArithmeticExpression::Get\_Priority(const string symbol)

{

int priority = ArithmeticSymbol::symbolDict[symbol];

if (symbol == "(" || symbol == ")")

priority = 1;

return priority;

}

void ArithmeticExpression::Add\_to\_Stack1(TStack<string>& st1, TStack<string>& st2, string s)

{

st1.Push(s);

st2.Pop();

}

bool ArithmeticExpression::Is\_Number(const string& str)

{

for (int i = 0; i< str.length(); i++)

{

char c = str[i];

if (!isdigit(c)) {

return false; }

}

return true;

}

bool ArithmeticExpression::isOperand(char c)

{

return ((c >= '0' && c <= '9') || (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'));

}

string ArithmeticExpression::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 ArithmeticExpression::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 (Is\_Symbol(s)==1)

return false;

if (s == "(" || s == ")")

{

if (s == "(")

k1++;

else

{

k2++;

char c1 = expression[i + 1];

string s1(1, c);

if (isOperand(c1) || c1=='(')

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 == l - 1)

return false;

}

if ((isOperand(c)) && (s1 == "("))

return false;

continue;

}

else

return false;

}

if (k1 != k2)

return false;

return true;

}

map<string, double> ArithmeticExpression::GetVariables(TStack<string>& postfixExpression) {

map<string, double> uniqueVariables;

for (int i = 0; i < postfixExpression.Length(); i++) {

string token = postfixExpression.GetElement(i);

if ((Is\_Symbol(token) != 0) || Is\_Number(token))

continue;

else

{

if (uniqueVariables.find(token) == uniqueVariables.end()) {

double value;

cout << "Enter the value for variable " << token << ": ";

cin >> value;

uniqueVariables[token] = value;

}

}

}

return uniqueVariables;

}

TStack<string> ArithmeticExpression::Postfix\_Form(const string& s)

{

string str = FilteredExpression(s);

if (!isValidExpression(str))

{

string msg = "Input error";

throw msg;

}

TStack<string> st1(5);

TStack<string> st2(5);

string numStr;

string varStr;

for (int i = 0; i < str.length(); i++)

{

char s1 = str[i];

string s(1,s1);

if (isdigit(s1))

if (!varStr.empty())

varStr += s;

else

numStr += s;

else

{

if (!numStr.empty())

{

st1.Push(numStr);

numStr.clear();

}

if (!varStr.empty())

{

st1.Push(varStr);

varStr.clear();

}

if (Is\_Symbol(s) != 0)

{

if (!st2.IsEmpty())

{

if (s == ")")

{

string sm = st2.Top();

while (sm != "(")

{

Add\_to\_Stack1(st1, st2, sm);

sm = st2.Top();

}

st2.Pop();

}

else if (s == "(")

st2.Push(s);

else

{

string priveous = st2.Top();

int pr1 = Get\_Priority(priveous);

int pr2 = Get\_Priority(s);

if (pr2 != 1)

{

while (pr1 >= pr2)

{

string sm1 = st2.Top();

Add\_to\_Stack1(st1, st2, sm1);

if (st2.IsEmpty())

break;

else

{

priveous = st2.Top();

pr1 = Get\_Priority(priveous);

}

}

}

st2.Push(s);

}

}

else

st2.Push(s);

}

else

varStr += s;

}

}

if (!numStr.empty())

{

st1.Push(numStr);

numStr.clear();

}

if (!varStr.empty())

{

st1.Push(varStr);

varStr.clear();

}

while (!st2.IsEmpty())

{

string a = st2.Top();

Add\_to\_Stack1(st1, st2, a);

}

return st1;

}

double ArithmeticExpression::Calculate(TStack<string>& st, const map<string, double>& values)

{

TStack<double> stack(20);

double rightOp, leftOp, resOp;

for (int i = 0; i < st.Length(); i++)

{

string c = st.GetElement(i);

if (Is\_Symbol(c) == 1)

{

if (c == "+")

{

rightOp = stack.Top();

stack.Pop();

leftOp = stack.Top();

stack.Pop();

resOp = leftOp + rightOp;

stack.Push(resOp);

}

if (c == "-")

{

rightOp = stack.Top();

stack.Pop();

leftOp = stack.Top();

stack.Pop();

resOp = leftOp - rightOp;

stack.Push(resOp);

}

if (c == "\*")

{

rightOp = stack.Top();

stack.Pop();

leftOp = stack.Top();

stack.Pop();

resOp = leftOp \* rightOp;

stack.Push(resOp);

}

if (c == "/")

{

rightOp = stack.Top();

stack.Pop();

leftOp = stack.Top();

stack.Pop();

resOp = leftOp / rightOp;

stack.Push(resOp);

}

}

else

{

if (Is\_Number(c))

stack.Push(stod(c));

else

stack.Push(values.at(c));

}

}

double r = stack.Top();

stack.Pop();

return r;

}