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
2023

Содержание

Введение....3

1 Постановка задачи....5

2 Руководство пользователя....6

2.1 Приложение для демонстрации работы стека....6

3 Руководство программиста....7

3.1 Описание алгоритмов....7

3.1.1 Стек....7

3.1.2 Арифметические выражения....8

3.2 Описание программной реализации....10

3.2.1 Описание структуры TNode....10

3.2.2 Описание класса TList....11

3.2.3 Описание класса TRingList....15

3.2.4 Описание класса TMonom....16

3.2.5 Описание класса TPolynom....18

3.2.6 Описание класса TStack....21

3.2.7 Описание класса ArithmeticExpression....23

Заключение....26

Литература....27

Приложения....28

Приложение А. Реализация структуры TNode....28

Приложение Б. Реализация класса TList....28

Приложение В. Реализация класса TRingList....32

Приложение Г. Реализация класса TMonom....33

Приложение Д. Реализация класса TPolynom....41

Приложение Е. Реализация класса TStack....61

Приложение Ж. Реализация класса ArithmeticExpression....64

Введение

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

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

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

Преимущества:

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

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

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

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

Недостатки:

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

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

3. Ограниченные операции: Линейные списки предоставляют только базовые операции, такие как добавление, удаление и поиск элементов. Другие операции, такие как сортировка или объединение списков, могут потребовать дополнительных усилий и времени для реализации.

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

1 Постановка задачи

Цель – реализовать структуру данных полином.

Задачи:

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

2 Руководство пользователя

2.1 Приложение для демонстрации работы стека

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

Рис. 1. Основное окно программы

2. Введите арифметическое выражение. В результате выведется выражение в постфиксной форме (рис. 2).

Рис. 2. Вывод постфиксной формы

3. Введите значения переменных (рис. 3).

Рис. 3. Ввод значений переменных

4. После ввода значений появится результат арифметического выражения (рис. 4).

Рис. 4. Вывод результата

3 Руководство программиста

3.1 Описание алгоритмов

3.1.1 Лист

3.1.2 Моном

3.1.3 Полином

3.1.4 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

3.1.5 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

3.2 Описание программной реализации

3.2.1 Описание структуры TNode

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

Назначение: представление узла.

Поля:

data– указатель на массив типа **`T`**.

pNext – указатель на следующий элемент.

Методы:

TNode()

Назначение: инициализация значений по умолчанию.

Входные параметры: отсутствуют.

Выходные параметры: новый узел с инициализированными значениями.

TNode(const T& data_)

Назначение: инициализация значения data узла.

Входные параметры: **`data_`** – значение **`data`**.

Выходные параметры: новый узел с инициализированным значением data.

TNode(TNode<T>* pNext_)

Назначение: инициализация указателя на следующий узел.

Входные параметры: **pNext_** – указатель на следующий узел.

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

TNode(const T& data_, TNode<T>* pNext_)

Назначение: инициализация значения data и указателя на следующий узел.

Входные параметры: **data_** – значение **data**, **pNext_** – указатель на следующий узел.

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

~TNode()

Назначение: освобождение выделенной памяти, если необходимо.

Входные параметры: отсутствуют.

Выходные параметры: память, выделенная для узла, освобождается.

3.2.2 Описание класса TList

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

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

```
  void InsertFirst(const T& data);
  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);
  void Remove(const T& data);
  void Clear();
  int GetSize() const;
  bool IsEmpty() const;
  bool IsFull() const;
  bool IsEnded() const;
  TNode<T>* GetCurrent();
  TNode<T>* GetStop();
  void SetPStop(TNode<T>* new_pStop);
  void SetPLast(TNode<T>* new_pLast);
  TNode<T>* GetLast();
  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**.

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 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.

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();

Назначение: возвращает указатель на текущий узел.

Входные параметры: отсутствуют.

Выходные параметры: указатель на текущий узел.

TNode<T>* GetStop();

Назначение: возвращает указатель на остановочный узел.

Входные параметры: отсутствуют.

Выходные параметры: указатель на остановочный узел.

void SetPStop(TNode<T>* new_pStop);

Назначение: устанавливает новое значение остановочному узлу.

Входные параметры: **`new_pStop`** – новый указатель на остановочный узел.

Выходные параметры: отсутствуют.

void SetPLast(TNode<T>* new_pLast);

Назначение: устанавливает новое значение последнему узлу.

Входные параметры: **`new_pLast`** – новый указатель на последний узел.

Выходные параметры: отсутствуют.

TNode<T>* GetLast();

Назначение: возвращает указатель на последний узел.

Входные параметры: отсутствуют.

Выходные параметры: указатель на последний узел.

void Next();

Назначение: переход к следующему узлу.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

void Reset();

Назначение: установка текущего узла как первого.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

3.2.3 Описание класса TRingList

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

};
```

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.

3.2.4 Описание класса TMonom

```
class TMonom
{
private:
  double coef;
  int degree;

public:
  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;
  double GetCoef() { return coef; }
  int GetDegree() { return degree; }
  void SetCoef(double c) { coef = c; }
  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.

double GetCoef();

Назначение: возвращает значение коэффициента объекта.

Входные параметры: отсутствуют.

Выходные параметры: значение коэффициента (вещественное число).

int GetDegree();

Назначение: возвращает значение степени объекта.

Входные параметры: отсутствуют.

Выходные параметры: значение степени (целочисленное число).

void SetCoef(double c);

Назначение: устанавливает новое значение коэффициента объекта.

Входные параметры: **`c`** – новое значение коэффициента.

Выходные параметры: отсутствуют.

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

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

Входные параметры:

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

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

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

3.2.5 Описание класса TPolynom

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

  static map<string, double> symbolDict;

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

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

  double operator() (double _x, double _y, double _z);
  TRingList<TMonom>* GetMonom()const { return monoms; }

  void Parse_Polynom(const string& s);
  void Print_Polynom();
  string ToString();
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.

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.

TRingList<TMonom>* GetMonom() const;

Назначение: возвращает указатель на кольцевой список мономов.

Входные параметры: отсутствуют.

Выходные параметры: указатель на кольцевой список мономов.

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.

void Print_Polynom();

Назначение: выводит полином на экран.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

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.

3.2.6 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`**.

3.2.7 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]

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, TNode<T>* _pStop)
{
  pFirst = _pFirst;
  pStop = _pStop;
  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, list.pFirst->pNext);
  TNode<T>* tmp = pFirst;
  TNode<T>* tmp2 = pFirst;
  tmp = tmp->pNext;
  while (tmp != list.pStop)
  {
            tmp2->pNext = new TNode<T>(tmp->data);
            tmp2 = tmp2->pNext;
            tmp = tmp->pNext;
  }
  pPrev = nullptr;
  pCurr = pFirst;
  pLast = tmp2;
  pStop = pLast->pNext;
}
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)
        pNode->pNext = pFirst;
  pFirst = pNode;
  pLast = 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);
  pLast->pNext = pNode;
  pLast = pNode;
  pLast->pNext = pStop;
}

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;
  }
  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()
{
  return pCurr;
}
template <typename T>
TNode<T>* TList<T>::GetStop()
{
  return pStop;
}
template <typename T>
TNode<T>* TList<T>::GetLast()
{
  return pLast;
}
template<typename T>
```

```
void TList<T>::SetPStop(TNode<T>* new_pStop)
{
  pStop = new_pStop;
}
template<typename T>
void TList<T>::SetPLast(TNode<T>* new_pLast)
{
  pLast = new_pLast;
}
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;
}
template <typename T>
TRingList<T>::TRingList(TNode<T>* _pFirst) : TList<T>(_pFirst)
{
  TNode<T>* pHead = new TNode<T>();
  pHead->pNext = pFirst;
```

```
  SetPStop(pHead);
  if (pFirst != nullptr)
  {
        TNode<T>* tmp = pFirst;
        while (tmp->pNext != nullptr)
              tmp = tmp->pNext;

        tmp->pNext = pHead;
        SetPLast(tmp);
  }
  else
  {
        pFirst = pHead;
        pLast = pHead;
        pCurr = pHead;
  }
}
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)
{
  if (rlist.pFirst == nullptr)
        return;
  pHead = new TNode<T>();
  pFirst = new TNode<T>(rlist.pFirst->data, rlist.pFirst->pNext);
  pHead->pNext = pFirst;
  TNode<T>* tmp = pFirst;
  TNode<T>* tmp2 = pFirst;
```

```
  tmp = tmp->pNext;

  while (tmp->data != pHead->data)
  {
        tmp2->pNext = new TNode<T>(tmp->data);
        tmp2 = tmp2->pNext;
        tmp = tmp->pNext;
  }
  SetPStop(pHead);
  SetPLast(tmp2);
  pLast->pNext = pHead;
  pCurr = pFirst;
}
template <typename T>
TRingList<T>::~TRingList()
{
  if (pHead)
        delete pHead;
}
```

Приложение Г. Реализация класса TMonom

```
TMonom::TMonom()
{
  coef = 0.0;
  degree = 0;
};
TMonom::TMonom(double c, int d)
{
  coef = c;
  degree = d;
};
TMonom::TMonom(const TMonom& m)
{
  coef = m.coef;
```

```
  degree = m.degree;
};
bool TMonom::operator<(const TMonom& m) const
{
  if (degree < m.degree)
        return true;
  else
        return false;
};
bool TMonom::operator>(const TMonom& m) const
{
  if (degree > m.degree)
        return true;
  else
        return false;
};
bool TMonom::operator==(const TMonom& m) const
{
  if (degree == m.degree)
        return true;
  else
        return false;
};
bool TMonom::operator!=(const TMonom& m) const
{
  if (degree != m.degree)
        return true;
  else
        return false;
}
```

Приложение Д. Реализация класса TPolynom

```
TPolynom::TPolynom()
{
```

```
  monoms = new TRingList<TMonom>();
  name = "";
}

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

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

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

TPolynom::~TPolynom()
{
  delete monoms;
  name = "";
}
TPolynom TPolynom::operator+(const TPolynom& p)
{
  TList<TMonom>* list = new TList<TMonom>();
  monoms->Reset();
  p.monoms->Reset();
  while (monoms->GetCurrent() != monoms->GetStop())
  {
          list->InsertEnd(monoms->GetCurrent()->data);
```

```
        monoms->Next();
    }
    while (p.monoms->GetCurrent() != p.monoms->GetStop())
    {
        monoms->Reset();
        int flag = 1;
        TMonom m2 = p.monoms->GetCurrent()->data;

        while (monoms->GetCurrent() != monoms->GetStop())
        {
            TMonom m1 = monoms->GetCurrent()->data;
            if (m1 == m2)
            {
                double k = m1.GetCoef();
                double k2 = m2.GetCoef();
                double k3 = k + k2;
                if (k3 == 0)
                    flag = 3;
                else
                {
                    m2.SetCoef(k3);
                    flag = 2;
                }
                break;
            }
            monoms->Next();
        }
        if (flag == 1)
            list->InsertEnd(p.monoms->GetCurrent()->data);
        else if (flag == 2)
        {
            TMonom search = list->Search(monoms->GetCurrent()->data)->data;
            list->InsertAfter(m2, search);
            list->Remove(monoms->GetCurrent()->data);
        }
```

```
            else if (flag == 3)
                list->Remove(monoms->GetCurrent()->data);
            p.monoms->Next();
    }
    TRingList<TMonom>* res = new TRingList<TMonom>(*list);
    TPolynom result;
    result.monoms = res;
    result.name = result.ToString();
    return result;
}
TPolynom TPolynom::operator-(const TPolynom& p)
{
    TList<TMonom>* list = new TList<TMonom>();
    monoms->Reset();
    p.monoms->Reset();
    while (monoms->GetCurrent() != monoms->GetStop())
    {
            list->InsertEnd(monoms->GetCurrent()->data);
            monoms->Next();
    }
    while (p.monoms->GetCurrent() != p.monoms->GetStop())
    {
            monoms->Reset();
            int flag = 1;
            TMonom m2 = p.monoms->GetCurrent()->data;

            while (monoms->GetCurrent() != monoms->GetStop())
            {
                TMonom m1 = monoms->GetCurrent()->data;
                if (m1 == m2)
                {
                    double k = m1.GetCoef();
                    double k2 = m2.GetCoef();
                    double k3 = k - k2;
                    if (k3 == 0)
```

```
                        flag = 3;
                    else
                    {
                        m2.SetCoef(k3);
                        flag = 2;
                    }
                    break;
                }
                monoms->Next();
            }
            if (flag == 1)
            {
                double t = m2.GetCoef();
                int deg = m2.GetDegree();
                double t2 = -t;
                TMonom mon(t2, deg);
                list->InsertEnd(mon);
            }
            else if (flag == 2)
            {
                TMonom search = list->Search(monoms->GetCurrent()->data)->data;
                list->InsertAfter(m2, search);
                list->Remove(monoms->GetCurrent()->data);
            }
            else
                list->Remove(monoms->GetCurrent()->data);
            p.monoms->Next();
        }
        TRingList<TMonom>* res = new TRingList<TMonom>(*list);
        TPolynom result;
        result.monoms = res;
        result.name = result.ToString();
        return result;
    }
    TPolynom TPolynom::operator*(const TPolynom& p)
```

```
{
  TList<TMonom>* list = new TList<TMonom>();
  monoms->Reset();
  p.monoms->Reset();
  while (monoms->GetCurrent() != monoms->GetStop())
  {
        TMonom m = monoms->GetCurrent()->data;
        p.monoms->Reset();
        while (p.monoms->GetCurrent() != p.monoms->GetStop())
        {
              TMonom m2 = p.monoms->GetCurrent()->data;
              double k = m.GetCoef();
              double k2 = m2.GetCoef();
              double k3 = k * k2;
              int d = m.GetDegree();
              int d2 = m2.GetDegree();
              int deg = d + d2;
              TMonom mon(k3, deg);
              list->InsertEnd(mon);
              p.monoms->Next();
        }
        monoms->Next();
  }
  TRingList<TMonom>* res = new TRingList<TMonom>(*list);
  TPolynom result;
  result.monoms = res;
  result.name = result.ToString();
  return result;
}
const TPolynom& TPolynom::operator=(const TPolynom& p)
{
  name = p.name;
  TList<TMonom>* list = new TList<TMonom>();
  p.monoms->Reset();
  while (p.monoms->GetCurrent() != p.monoms->GetStop())
```

```
  {
        list->InsertEnd(p.monoms->GetCurrent()->data);
        p.monoms->Next();
  }
  TRingList<TMonom>* res = new TRingList<TMonom>(*list);
  delete monoms;
  this->monoms = res;
  return *this;
}
TPolynom TPolynom::difx() const
{
  TList<TMonom>* list = new TList<TMonom>();
  monoms->Reset();
  while (monoms->GetCurrent() != monoms->GetStop())
  {
        TMonom m = monoms->GetCurrent()->data;
        double k = m.GetCoef();
        int d = m.GetDegree();
        if (d >= 100)
        {
              int d0 = d / 100;
              d -= 100;
              k *= d0;
              TMonom newM(k, d);
              list->InsertEnd(newM);
        }
        monoms->Next();
  }
  TRingList<TMonom>* res = new TRingList<TMonom>(*list);
  TPolynom result;
  result.monoms = res;
  result.name = result.ToString();
  return result;
}
TPolynom TPolynom::dify() const
```

```
{
  TList<TMonom>* list = new TList<TMonom>();
  monoms->Reset();
  while (monoms->GetCurrent() != monoms->GetStop())
  {
        TMonom m = monoms->GetCurrent()->data;
        double k = m.GetCoef();
        int d = m.GetDegree();
        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();
  }
  TRingList<TMonom>* res = new TRingList<TMonom>(*list);
  TPolynom result;
  result.monoms = res;
  result.name = result.ToString();
  return result;
}
TPolynom TPolynom::difz() const
{
  TList<TMonom>* list = new TList<TMonom>();
  monoms->Reset();
  while (monoms->GetCurrent() != monoms->GetStop())
  {
        TMonom m = monoms->GetCurrent()->data;
        double k = m.GetCoef();
```

```
            int d = m.GetDegree();
            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();
  }
  TRingList<TMonom>* res = new TRingList<TMonom>(*list);
  TPolynom result;
  result.monoms = res;
  result.name = result.ToString();
  result.name = result.ToString();
  return result;
}
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(5);
  st = ArithmeticExpression::Postfix_Form(pol_name);
  //cout << st << endl;
  double result = ArithmeticExpression::Calculate(st, variableDict);
  return result;
}
```

```
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::Parse_Polynom(const string& s)
{
  string str = FilteredExpression(s);
  if (!isValidExpression(str))
  {
            string msg = "Input error";
            throw msg;
  }
  else
  {
            TList<TMonom>* monomList = new TList<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')
	{
		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";
	}
	else if (str[i] == 'y')
	{
		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;
                                }
                        }
                        else
                        {
                                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;
                                }
                        }
                }
                if (str[i] == '*' || str[i] == '/')
                        i++;


        }
}
int degree = stoi(deg);
```

```
double koef = 0.0;
if (numStr == "")
	koef = 1.0;
else
	koef = stod(numStr);
if (flag == 0)
	koef = -koef;
TMonom monom(koef, degree);
if (monomList->IsEmpty())
	monomList->InsertFirst(monom);
else
{
	monomList->Reset();
	while (monomList->GetCurrent() != monomList->GetStop())
	{
		TMonom m = monomList->GetCurrent()->data;
		int deg = m.GetDegree();
		int deg2 = monom.GetDegree();
		if (deg2 < deg)
		{
			monomList->InsertBeforeCurr(monom);
			break;
		}
		else if (deg2 == deg)
		{
			double k = m.GetCoef();
			double k2 = monom.GetCoef();
			if (k2 < k)
			{
				monomList->InsertBeforeCurr(monom);
				break;
			}
			else
			{
				monomList->InsertAfterCurr(monom);
```

```
                                        break;
                                }
                        }
                        monomList->Next();
                }
                if (monomList->GetCurrent() == monomList->GetStop())
                        monomList->InsertEnd(monom);
        }

        if (str[i] == '-')
                flag = 0;
        else
                flag = 1;
    }
    TRingList<TMonom>* polynomList = new TRingList<TMonom>(*monomList);
    this->monoms = polynomList;
  }
}

void TPolynom::Print_Polynom()
{
  cout << name << endl;
  cout << "Monoms:" << endl;
  while (monoms->GetCurrent() != monoms->GetStop())
  {
        cout << monoms->GetCurrent()->data << endl;
        monoms->Next();
  }
}

string TPolynom::ToString()
{
  string st = "";
  monoms->Reset();
  while (monoms->GetCurrent() != monoms->GetStop())
```

```
{
        TMonom m = monoms->GetCurrent()->data;
        double k = m.GetCoef();
        int d = m.GetDegree();
        int flag = 0;
        if (d >= 100)
        {
                if (st == "")
                        st += to_string(k);
                else
                        if(k>0)
                                st += "+" + to_string(k);
                        else
                                st += to_string(k);

                int deg_x = d / 100;
                if (deg_x == 1)
                        st += "*x";
                else
                        st += "*x^" + to_string(deg_x);
                d = d % 100;
                flag++;
        }
        if (d >= 10)
        {
                if (flag == 0)
                {
                        if (st == "")
                                st += to_string(k);
                        else
                                if (k > 0)
                                        st += "+" + to_string(k);
                                else
                                        st += to_string(k);
                }
```

```
            int deg_y = d / 10;
            if (deg_y == 1)
                st += "*y";
            else
                st += "*y^" + to_string(deg_y);
            d = d % 10;
        }
        if (d >= 1)
        {
            if (flag == 0)
            {
                if (st == "")
                    st += to_string(k);
                else
                    if (k > 0)
                        st += "+" + to_string(k);
                    else
                        st += to_string(k);
            }

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