

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

**Институт информационных технологий, математики и механики**

**ЛАБОРАТОРНАЯ РАБОТА**

на тему:

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

**Выполнил:** студентка группы  
3822Б1ФИ2

\_\_\_\_\_/Коробейников А. П./  
Подпись

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

Подпись

Нижний Новгород  
2024

# Содержание

Введение.....	3
1 Постановка задачи .....	4
2 Руководство пользователя.....	5
2.1 Приложение для демонстрации работы полиномов .....	5
3 Руководство программиста.....	6
3.1 Описание алгоритмов.....	6
3.1.1 Список.....	6
3.1.2 Кольцевой список с головой .....	12
3.1.3 Моном .....	18
3.1.4 Полином.....	20
3.2 Описание программной реализации .....	22
3.2.1 Описание класса TNode.....	22
3.2.2 Описание класса TList .....	22
3.2.3 Описание класса TRingList .....	26
3.2.4 Описание класса TMonom.....	27
3.2.5 Описание класса TPolynom .....	30
Заключение .....	34
Литература .....	35
Приложения .....	36
Приложение А. Реализация класса TNode .....	36
Приложение Б. Реализация класса TList .....	36
Приложение В. Реализация класса TRingList .....	42
Приложение Г. Реализация класса TMonom.....	44
Приложение Д. Реализация класса TPolinom.....	47
Приложение Е. Sample_tpolinom .....	54

## Введение

Лабораторная работа направлена на изучение обработки полиномов от трёх переменных ( $x$ ,  $y$ ,  $z$ ). Полиномы могут быть использованы для решения многих задач математического анализа, теории вероятностей, линейной алгебры и других областей математики

В данной лабораторной работе студенты будут изучать основные принципы работы алгоритма обработки полиномов и реализовывать его на практике. Это позволит им лучше понять принципы работы связного списка и освоить навыки работы с алгоритмами обработки полиномов.

# 1 Постановка задачи

## **Цель:**

Цель лабораторной работы - разработать библиотеку, которая содержит в себе структуру хранения для кольцевого списка и полиномов, которые реализованы на основе кольцевого списка.

## **Задачи:**

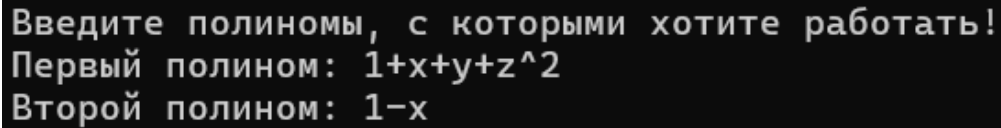
1. Изучить необходимую теорию по работе с кольцевым списком.
2. Написать класс кольцевой список.
3. Написать тесты для этого класса , чтобы убедиться в корректности работы.
4. Изучить необходимую теорию по работе с полиномами.
5. Написать класс, для работы с полиномами.
6. Написать тесты для этого класса, чтобы убедиться в корректности работы.
7. Написать приложение, демонстрирующее работу класса полином.

## 2 Руководство пользователя

### 2.1 Приложение для демонстрации работы полиномов

1. Запустите приложение с названием sample\_tpolynom.exe. В результате появится окно, показанное ниже, вам будет предложено ввести два полинома.

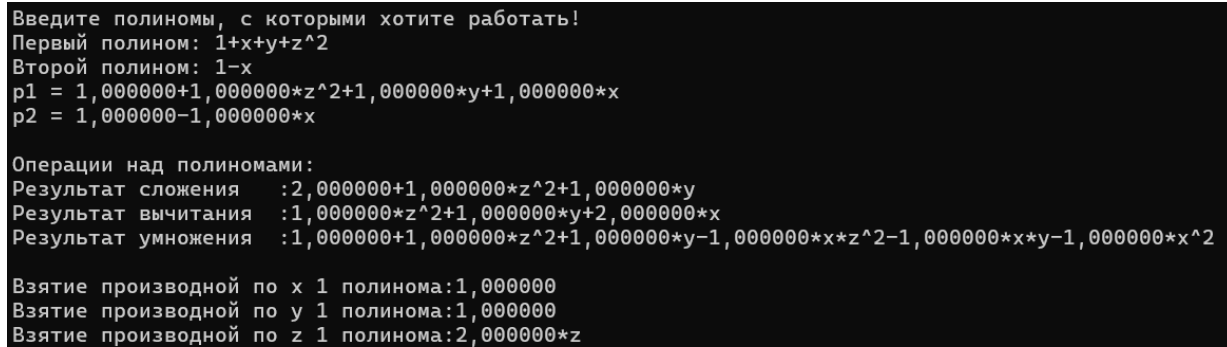
(Рис. 1).



```
Введите полиномы, с которыми хотите работать!  
Первый полином: 1+x+y+z^2  
Второй полином: 1-x
```

Рис. 1. Пример создание полиномов

2. Далее будут выведены результаты операций над полиномами (Error! Reference source not found.).



```
Введите полиномы, с которыми хотите работать!  
Первый полином: 1+x+y+z^2  
Второй полином: 1-x  
p1 = 1,000000+1,000000*z^2+1,000000*y+1,000000*x  
p2 = 1,000000-1,000000*x  
  
Операции над полиномами:  
Результат сложения :2,000000+1,000000*z^2+1,000000*y  
Результат вычитания :1,000000*z^2+1,000000*y+2,000000*x  
Результат умножения :1,000000+1,000000*z^2+1,000000*y-1,000000*x*z^2-1,000000*x*y-1,000000*x^2  
  
Взятие производной по x 1 полинома:1,000000  
Взятие производной по y 1 полинома:1,000000  
Взятие производной по z 1 полинома:2,000000*z
```

Рис. 2. Операции над полиномами

## 3 Руководство программиста

### 3.1 Описание алгоритмов

#### 3.1.1 Список

Список — это динамическая структура данных, которая состоит из звеньев, содержащих данные и ключ-указатель на следующее звено. Структура данных поддерживает операции такие как: вставка в начало, вставка до и после элемента. Также присутствуют операции поиска, удаления из списка, проверка на пустоту, полноту.

**Примечание:** По умолчанию, первый элемент является текущим, в ходе работы со списком указатель на текущий может изменяться.

#### Операция добавления в начало

Операция добавления элемента реализуется при помощи указателя на первый элемент. Если структура хранения пуста, то создаём новый элемент (Рис. 3), иначе создаём новый элемент и переопределяем указатель на начало, он будет указывать на вставленный элемент. Новый элемент будет указывать на следующий, который раньше был первым (Рис. 4 и Рис. 5).

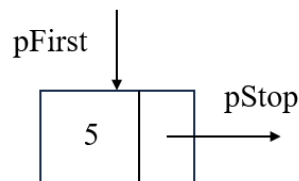


Рис. 3. Добавление в начало, если список пустой

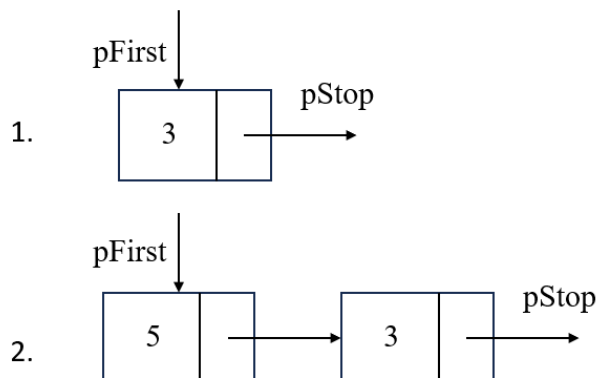


Рис. 4. Добавление в начало, если список не пустой

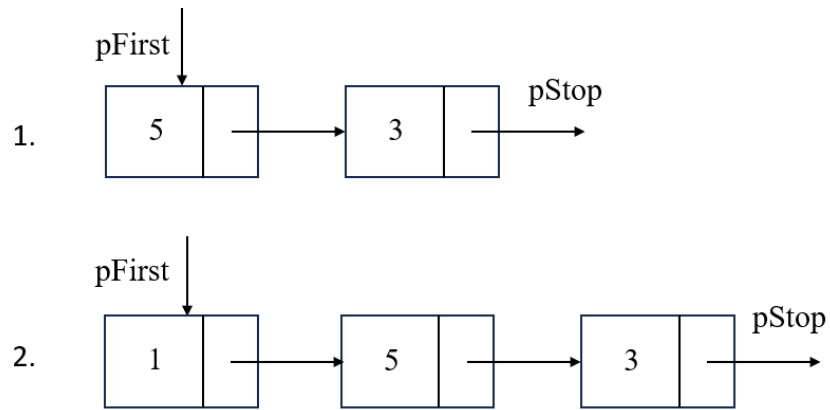


Рис. 5. Операция добавления элемента (1) в начало

### Операция поиска

Операция поиска ищет элемент в списке. Проходимся от начала списка, сравнивая **data** звеньев с указанными пользователем. Если **data** не равна, идём к следующему звену, меняя указатель на текущий **pCur**, найдя нужное звено, возвращаем указатель на найденный элемент. (Рис. 6)

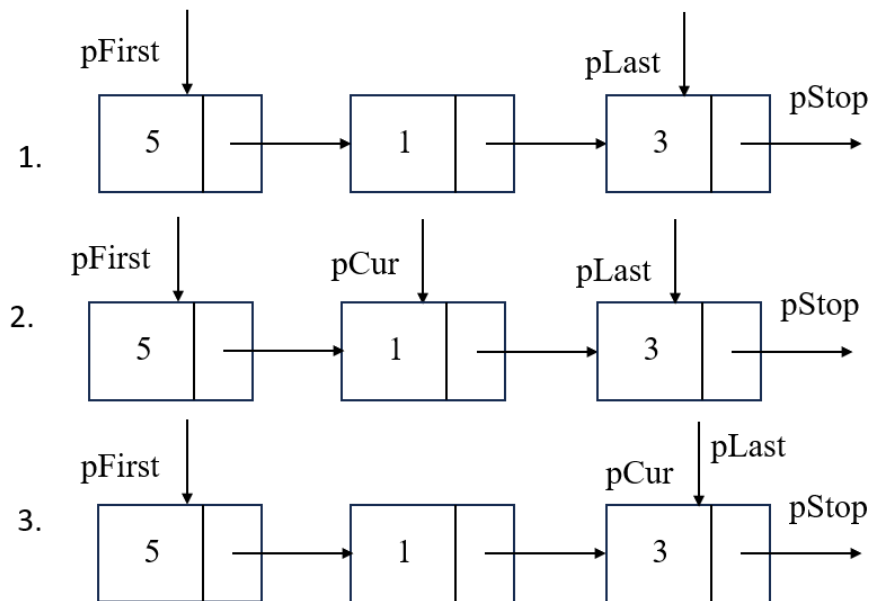


Рис. 6. Операция поиска элемента (3) в списке

Операция поиска, по сравнению с сортированными наборами данных, не может быть оптимизирована, поскольку доступ к элементам только последовательный и бинарный поиск не реализуем.

### Операция добавления после

Находим элемент, после которого хотим вставить, создаём новое звено и сдвигаем указатели. Разберем на примере ниже (Рис. 7 и Рис. 8рис. 7):

- Находим элемент, после которого хотим выполнить операцию вставки с помощью операции поиска (Рис. 6).

- У нового звена указатель на следующий элемент равен советуемому указателю элемента, после которого вставляем (красная стрелка на Рис. 8).
- После меняется указатель на следующий элемент у элемента, после которого мы вставляем, он должен указывать на новый, вставленный элемент (фиолетовая стрелка на рис. 8).

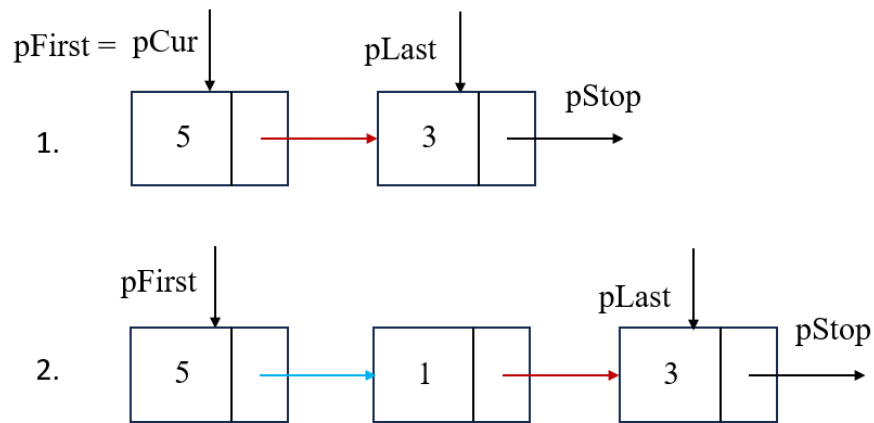


Рис. 7. Операция добавления элемента (1) после текущего. Текущий элемент - первый

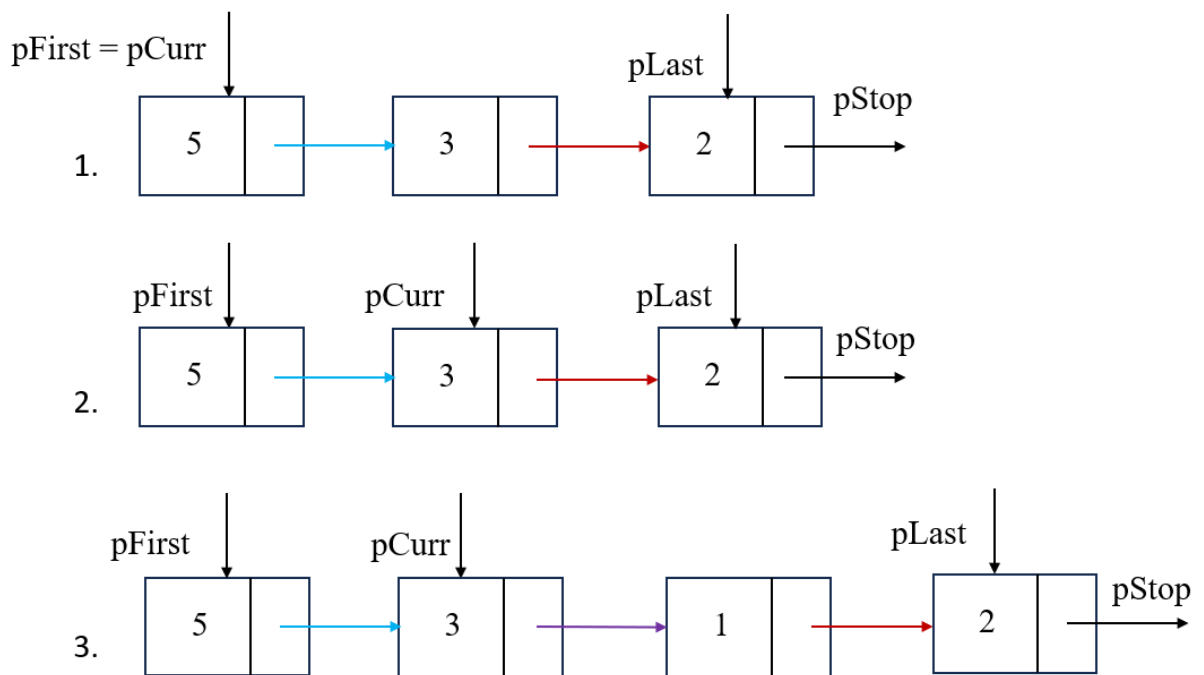


Рис. 8. Операция добавления элемента (1) после текущего (3)

### Операция добавления перед

Находим элемент, перед которым хотим вставить, создаём новое звено и сдвигаем указатели. Разберем на примере ниже (Рис. 9):

- С помощью операции поиска (Рис. 6) находим нужный элемент, перед которым хотим выполнить операцию вставки.



- Создаём новый элемент (1) с указателем на следующий равный указателю на элемент (3), перед которым вставляем (голубая стрелка на Рис. 9). Таким образом, мы не разрываем связь в списке.
- Далее переопределяем указатель у элемента (5), который раньше стоял перед найденным (3), он должен указывать на новый (1) (фиолетовая стрелка на Рис. 9).

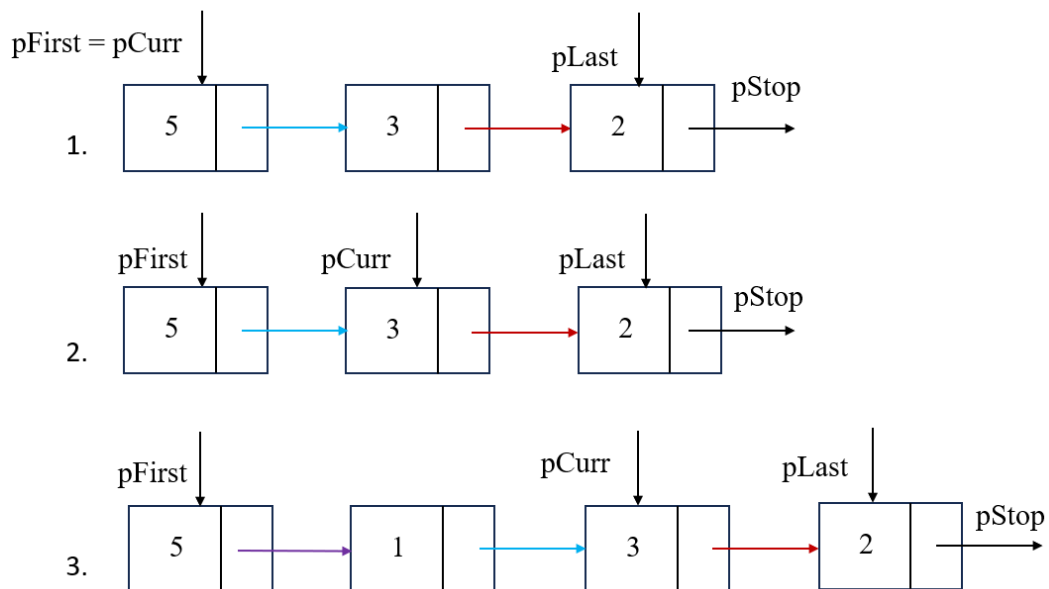


Рис. 9. Операция вставки (1) перед текущим (3)

### Операция удаления первого элемента

Операция удаления элемента реализуется при помощи указателя на первый элемент. Удалив первый элемент, следующий за ним, становится первым, то есть на него указывает  $pFirst$  (указатель на первый элемент в списке) (Рис. 10).

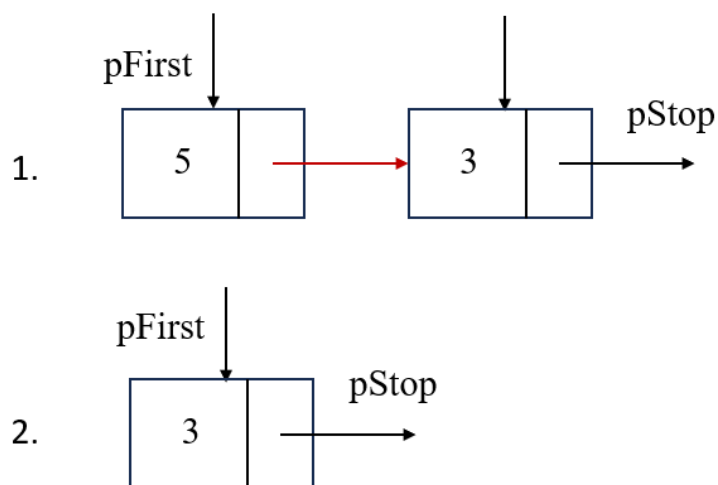


Рис. 10. Операция удаления первого элемента

## Операция удаления текущего элемента

Операция удаления элемента реализуется при помощи указателя на текущий элемент.

Разберем на примере ниже (Рис. 11рис. 7):

- Создаётся вспомогательное звено, с указателем на следующий (3) равным советуемому у того элемента (1), который хотим удалить (красная стрелка на Рис. 11). Таким образом, мы не потеряем связь, удалив элемент
- Удаляем нужный элемент и его указатель на следующий
- У элемента, который стоял перед удаленным (5), переопределяем указатель на следующий, теперь он равен указателю на следующий у вспомогательного звена, теперь (5) указывает на элемент (3), на который раньше указывал удалённый (1)

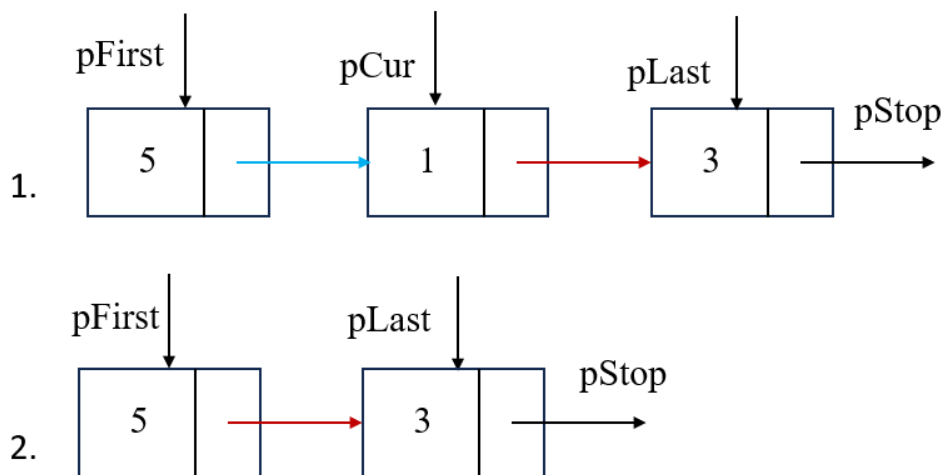


Рис. 11. Операция удаления текущего элемента (1)

## Операция удаления элемента по его данным

Ищем в списке звено с нужными данными с помощью операции поиска (рис. 6Рис. 6) и удаляем, меняя указатели, чтобы не потерять связь.

Разберем на примере ниже (Рис. 12рис. 7):

- С помощью операции поиска (Рис. 6) находим элемент, который хотим удалить
- Создается вспомогательное звено, с указателем на следующий (3) равным советуемому у того элемента (1), который хотим удалить (красная стрелка на Рис. 12). Таким образом, мы не потеряем связь, удалив элемент
- Удаляем нужный элемент и его указатель на следующий
- У элемента, который стоял перед удаленным (5), переопределяем указатель на следующий, теперь он равен указателю на следующий у вспомогательного звена, теперь (5) указывает на элемент (3), на который раньше указывал удаленный (1)

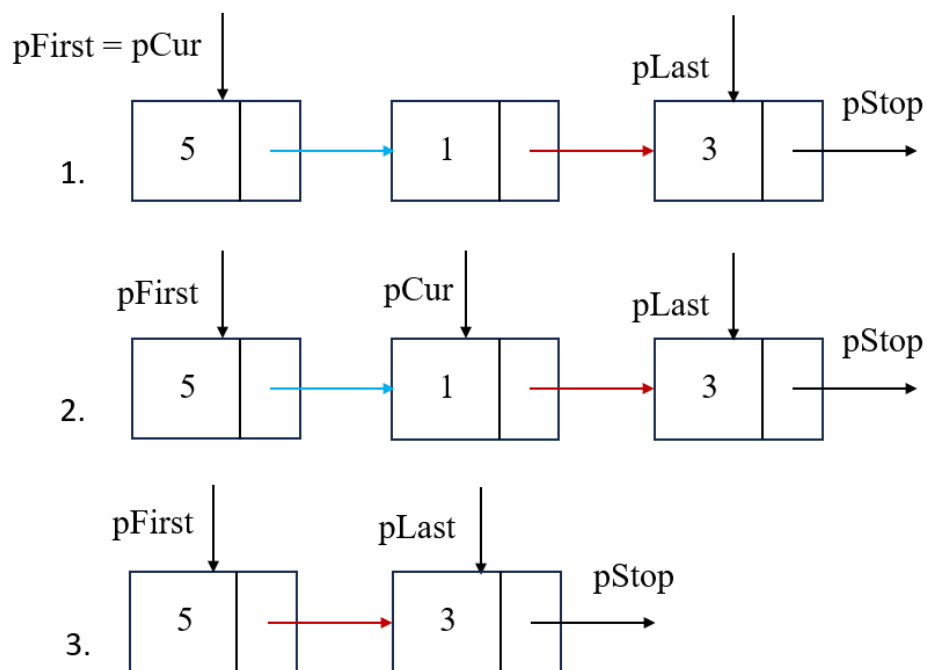


Рис. 12. Операция удаления элемента (1):

### Операция получения текущего элемента

Операция получения текущего элемента реализуется при помощи указателя на текущий элемент. Возвращает указатель на текущий.

Пример:

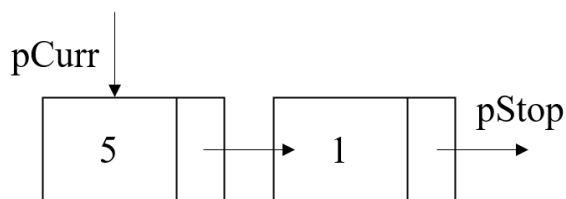


Рис. 13. Операция взятия элемента если текущий по умолчанию

Результат: Указатель на 5 элемент

### Операция проверки на пустоту

Операция проверки на полноту проверяет, есть ли хотя бы один элемент в списке. Также реализуется при помощи указателя на первый элемент.

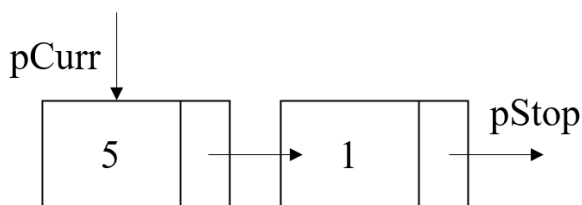


Рис. 14. Операция проверки на полноту

Результат: false

### 3.1.2 Кольцевой список с головой

У кольцевого односвязного списка в отличие от односвязного списка, есть указатель на фиктивную голову. Это позволяет облегчить некоторые операции.

Операции, доступные с данной структурой хранения, следующие: добавление элемента, удаление элемента, взять текущий элемент, проверка на пустоту, сортировка, отчистка списка.

**Примечание:** По умолчанию, первый элемент является текущим, в ходе работы со списком указатель на текущий может изменяться. Столбцы с элементом 0 – это фиктивная голова, для наглядности.

#### Операция добавления в начало

Операция добавления элемента реализуется при помощи указателя на первый элемент.

Если структура хранения пуста, то создаём новый элемент.

Если список не пуст, создаём новый элемент и сдвигаем указатели. Разберем на примере ниже (Рис. 15):

- Создаём новое звено (1), которое указывает на первый элемент (3) в списке.
- Ставим указатель **pFirst** на новый элемент.
- У звена, являющегося фиктивной головой с указателем **pHead**, меняем указатель на следующий так, чтобы он указывал на вставленный элемент.

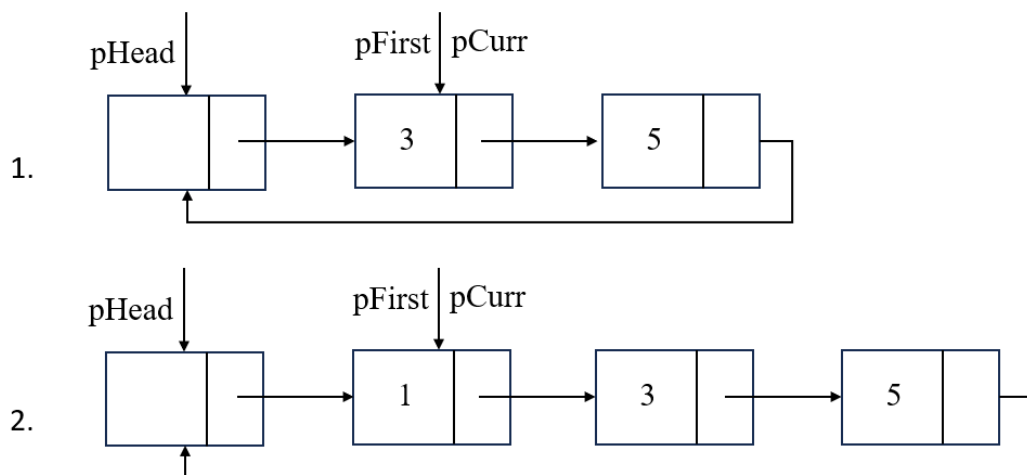


Рис. 15. Операция добавления элемента (1) в начало кольцевого списка с головой

#### Операция добавления в конец

Операция добавления элемента реализуется при помощи указателя на последний элемент.

Если структура хранения пуста, то используем операцию вставки в начало (рис. 15).

Если список не пуст, то создаём новый элемент и сдвигаем указатели. Разберем на примере ниже (Рис. 16):

- Создаем новое звено (1), которое указывает на фиктивную голову.
- У последнего элемента (5) с указателем **pLast**, меняем указатель на следующий так, чтобы он указывал на вставленный (1).
- На новое звено (1) в конце ставим указатель **pLast**, последнего звена.

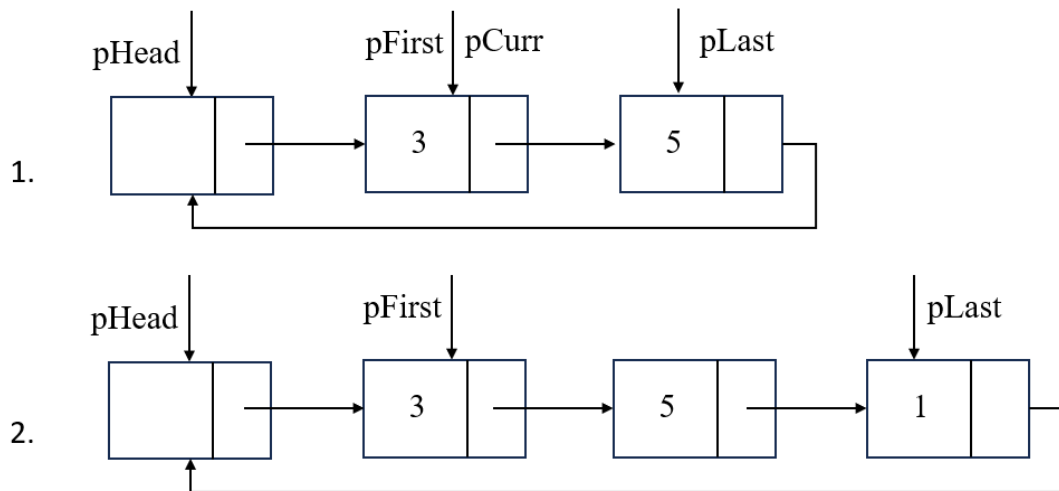


Рис. 16. Операция добавления элемента (1) в конец

## Операция поиска

Операция поиска ищет элемент в списке.

Проходимся от начала списка, сравнивая **data** звеньев с указанными пользователем. Если **data** не равны, идем к следующему звену, меня указатель на текущий **pCurr**, найдя нужное звено, возвращаем указатель на найденный элемент. (Рис. 17)

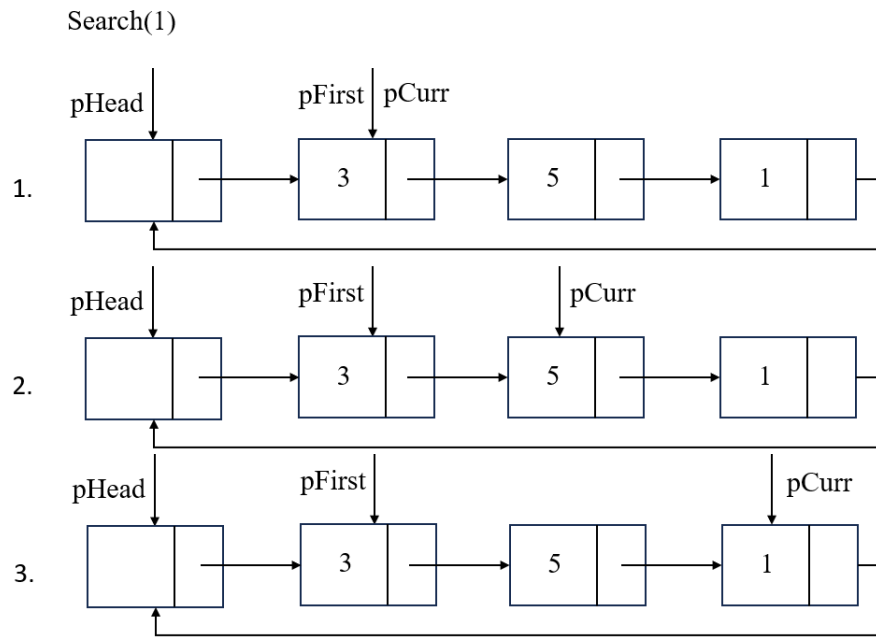


Рис. 17. Операция поиска элемента (1)

### Операция добавления после текущего

Операция добавления элемента реализуется при помощи указателя на текущий элемент.

Разберем на примере ниже (Рис. 18рис. 16):

- Создаем новое звено (7), которое будет указывать на звено, следующее за текущим (красная стрелка на рис. 18) .
- У текущего элемента (5) переопределяем указатель на следующий так, чтобы он указывал на вставленный (7) (синяя стрелка на рис. 18).

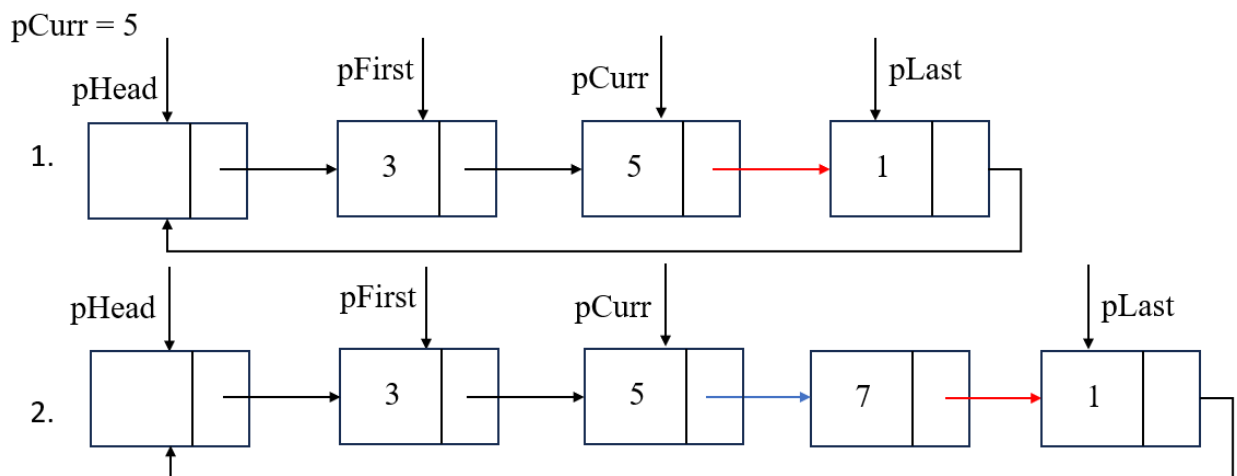


Рис. 18. Операция добавления элемента (7) после текущего (5)

## Операция добавления перед текущим

Операция добавления элемента реализуется при помощи указателя на текущий элемент. Имеем указатель на текущий, создаём новое звено и сдвигаем указатели текущего и нового.

Разберем на примере ниже (Рис. 19):

- Создаём новое звено (7), с указателем на следующий равный указателю на элемент (5), перед которым вставляем (красная стрелка на Рис. 19). Этот указатель берём у элемента, стоящего перед текущим. Таким образом, мы не разрываем связь в списке.
- Далее переопределяем указатель у элемента (3), который раньше стоял перед текущим (5), он должен указывать на новый (7) (синяя стрелка на Рис. 19).

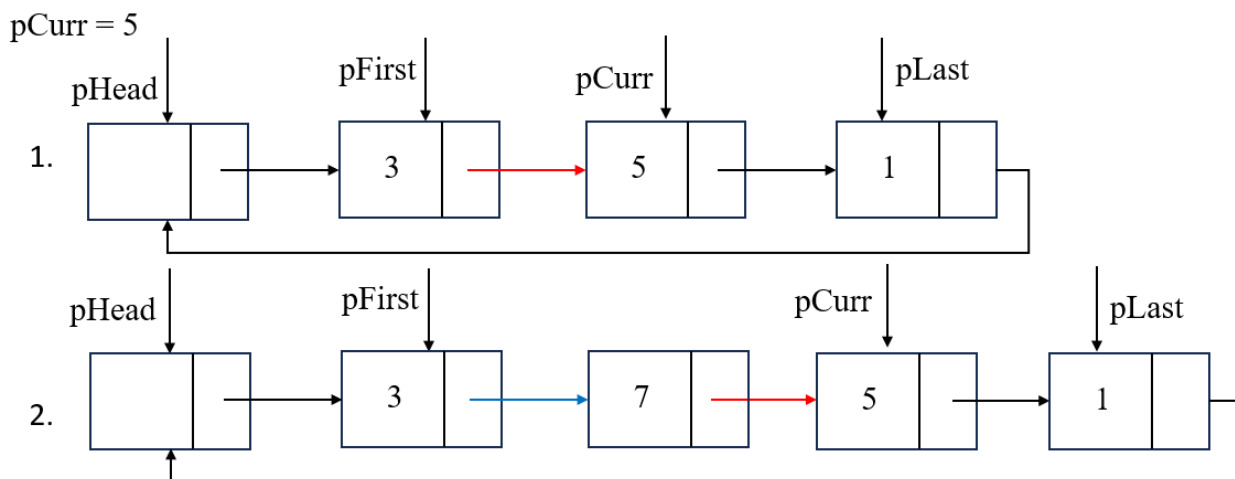


Рис. 19. Операция добавления элемента (7) перед текущим (5)

## Операция удаления первого элемента

Операция удаления элемента реализуется при помощи указателя на первый элемент **pFirst**. Удалив первый элемент, следующий за ним, становится первым. На него теперь будет указывать **pFirst**, и фиктивная голова будет своим указателем на следующий указывать на новый первый элемент. (Рис. 20)

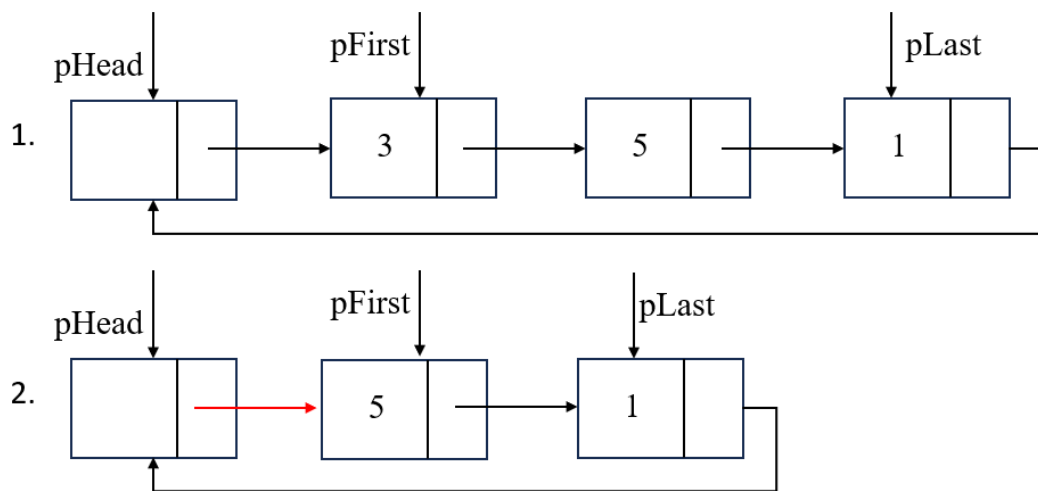


Рис. 20. Операция удаления первого элемента

### Операция удаления текущего элемента

Операция удаления элемента реализуется при помощи указателя на текущий элемент. Разберем на примере ниже (Рис. 21рис. 7):

- Создаётся вспомогательное звено, с указателем на следующий (3) равным советуемому у того элемента (1), который хотим удалить (красная стрелка на рис. 21). Таким образом, мы не потеряем связь, удалив элемент.
- Удаляем нужный элемент и его указатель на следующий
- У элемента, который стоял перед удаленным (5), переопределяем указатель на следующий, теперь он равен указателю на следующий у вспомогательного звена, теперь (5) указывает на элемент (3), на который раньше указывал удалённый (1)

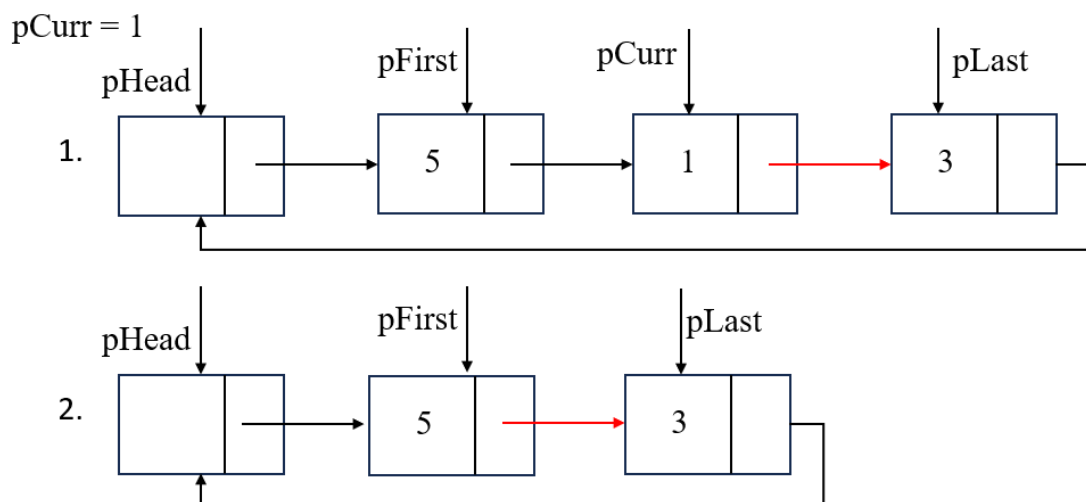




Рис. 21. Операция удаления текущего (1) элемента

### Операция удаления элемента по его данным

Ищем в списке звено с нужными данными с помощью операции поиска (рис. 6) и удаляем, меняя указатели, чтобы не потерять связь.

Разберем на примере ниже (Рис. 22рис. 7):

- С помощью операции поиска (рис. 6Рис. 6) находим элемент, который хотим удалить
- Создается вспомогательное звено, с указателем на следующий (3) равным советуемому у того элемента (1), который хотим удалить (красная стрелка на Рис. 22). Таким образом, мы не потеряем связь, удалив элемент
- Удаляем нужный элемент и его указатель на следующий
- У элемента, который стоял перед удаленным (5), переопределяем указатель на следующий, теперь он равен указателю на следующий у вспомогательного звена, теперь (5) указывает на элемент (3), на который раньше указывал удаленный (1)

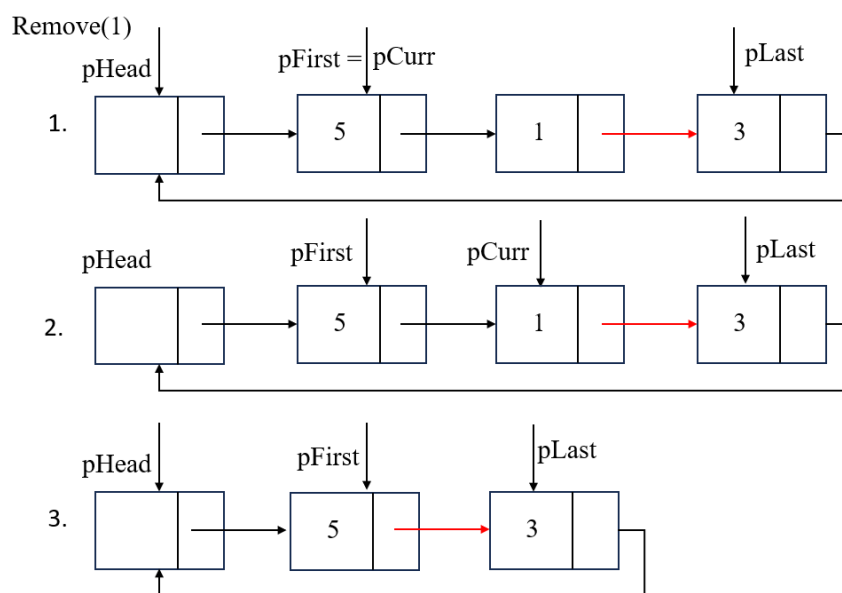


Рис. 22. Операция удаления элемента по его данным (1)

### Операция получения текущего элемента

Операция взятия элемента текущего элемента также реализуется с помощью указателя на текущий элемент. Возвращает указатель на текущий элемент. (Рис. 23)

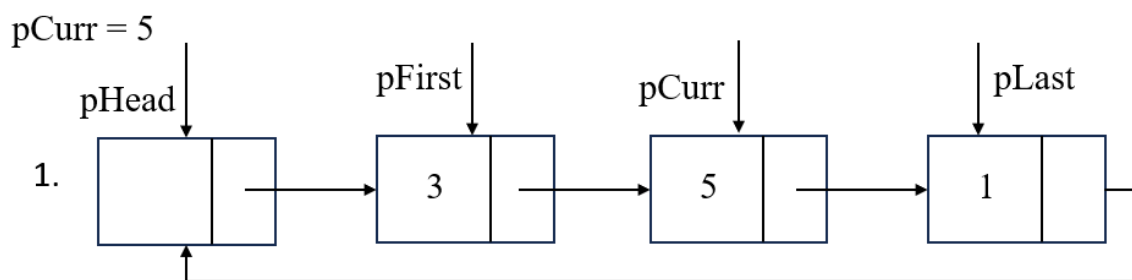


Рис. 23. Операция взятия текущего элемента

Результат: указатель на (5)

### Операция проверки на пустоту.

Операция проверки на полноту проверяет, есть ли хотя бы один элемент в списке.

Также реализуется при помощи указателя на первый элемент. (Рис. 24)

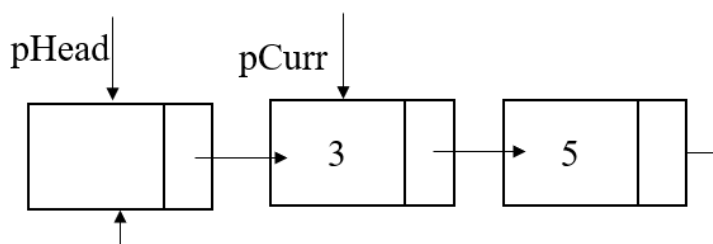


Рис. 24. Операция проверки на полноту

Результат: false

### 3.1.3 Моном

Моном представляет собой структуру данных и характеризуется, как коэффициент и свёртку степеней. Структура данных поддерживает операции сравнения. В свёртке степеней разряд сотен – это степень  $x$ , десятков – степень  $y$ , единиц – степень  $z$ .

#### Операция сравнения больше

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

Пример 1:  $2 \cdot x^3 \cdot y \cdot z > 2 \cdot x^2 \cdot y^2 \cdot z$

Функция вернёт **true**, так как свёртка степеней первого монома равна 311, а свёртка степеней второго 221.

Пример 2:  $2 \cdot x^3 \cdot y \cdot z > 2 \cdot x^4 \cdot y^2 \cdot z$

Функция вернёт **false**, так как свёртка степеней первого монома равна 311, а свёртка степеней второго 421.

#### Операция сравнения меньше

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

Пример 1:  $2*x^3*y*z < 2*x^2*y^2*z$

Функция вернёт **false**, так как свёртка степеней первого монома равна 311, а свёртка степеней второго 221.

Пример 2:  $2*x^3*y*z < 2*x^4*y^2*z$

Функция вернёт **true**, так как свёртка степеней первого монома равна 311, а свёртка степеней второго 421.

### Операция сравнения на равенство

Функция сравнивает по степеням на равенство два монома.

Пример 1:  $2*x^3*y*z == 2*x^2*y^2*z$

Функция вернёт **false**, так как свёртка степеней первого монома равна 311, а свёртка степеней второго 221.

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

Функция вернёт **true**, так как свёртка степеней первого монома равна 421, и свёртка степеней второго 421.

### Операция сравнения на неравенство

Функция сравнивает по степеням на неравенство два монома.

Пример 1:  $2*x^3*y*z != 2*x^2*y^2*z$

Функция вернёт **true**, так как свёртка степеней первого монома равна 311, а свёртка степеней второго 221.

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

Функция вернёт **false**, так как свёртка степеней первого монома равна 421, и свёртка степеней второго 421.

### Операция сложения мономов

Если правый и левый мономы имеют одинаковую свёртку степеней, то создаётся результирующий моном со степенью мономов, которые складываем, и с коэффициентом, равным сумме коэффициентов левого и правого мономов.

Пример:  $(x^2y) + (3x^2y)$

Результат:  $4x^2y$

### Операция умножения мономов

Если сумма свёртки степеней левого и правого мономов не меньше 0 и не больше 999, то создаётся результирующий моном со степенью, равной сумме свёрток степеней мономов. Коэффициент результирующего монома равен произведению коэффициентов левого и правого мономов.

Пример:  $(8y^2) * (5y^2z)$

Результат:  $40y^4z$

### **Операция вычитания мономов**

Если правый и левый мономы имеют одинаковую свёртку степеней, то создаётся результирующий моном со степенью мономов, которые складываем, и с коэффициентом, равным разности коэффициентов левого и правого мономов.

Пример:  $(x^2y) - (3x^2y)$

Результат:  $-2x^2y$

### **Операция дифференцирования мономов**

Для дифференцирования по  $x$ , разряд сотен в свёртке монома уменьшается на 1 и коэффициент монома умножается на первоначальную степень  $x$ .

Для дифференцирования по  $y$ , разряд десятков в свёртке монома уменьшается на 1 и коэффициент монома умножается на первоначальную степень  $y$ .

Для дифференцирования по  $z$ , разряд единиц в свёртке монома уменьшается на 1 и коэффициент монома умножается на первоначальную степень  $z$ .

Пример:

$$x^4y^5z^3$$

Результат дифференцирования по  $x$ :  $4x^3y^5z^3$

Результат дифференцирования по  $y$ :  $5x^4y^4z^3$

Результат дифференцирования по  $z$ :  $3x^4y^5z^2$

## **3.1.4 Полином**

Программа предоставляет возможности для работы с полиномами на базе мономов: суммирование, разность, произведение, дифференцирование полиномов.

Алгоритм на входе требует строку, которая представляет некоторый полином. Алгоритм допускает наличия трёх независимых переменных, положительные целые степени независимых переменных и вещественные коэффициенты.

### **Операция суммирования полиномов**

Полиномы разбиваются на мономы, они сравниваются и создают результирующий полином, путём создания упорядоченного списка мономов. Если свёртки мономов равны, то складываются их коэффициенты.

Пример:

$$(+2.0*x^2-yz-x) + (+3.0*x^2+2.0*x^3+xyz+4.0*x)$$

Результат:

$$+2.0*x^3+5.0*x^2+xyz+3.0*x-yz$$

### **Операция вычитания полиномов**

Используется операция суммирования полиномов, левого полинома и правого полинома, умноженного на -1.

Пример:

$$(+x^4+3.0*x+yz+3.0*y^3z+x^2yz^5) - (+x^3+2.0*x-yz)$$

Результат:

$$+x^4-x^3+x^2yz^5+x+3.0*y^3z+2.0*yz$$

### **Операция произведения полиномов**

Мономы умножаются каждый с каждым, степени складываются, коэффициенты умножаются, создавая таким образом новый полином, путём создания упорядоченного списка мономов.

Пример:

$$(-2x^2 + 3x*y*z + 1) * (3x^2+1)$$

Результат:

$$-6x^4 + x^2 + 9x^2yz + 3xyz + 3x*y*z + 2$$

### **Операция дифференцирования полиномов**

Операция дифференцирования полинома согласно математическим правилам. Возможно дифференцирование по независимым переменным x, y или z. Полином разбивается на мономы и к каждому моному применяется операция дифференцирования по одной из трех переменных.

Пример:

$$+x^4+3.0*x+yz+3.0*y^3z+x^2yz^5$$

Результат дифференцирования по x:  $+4.0*x^3+2.0*xyz^5+3.0$

Результат дифференцирования по y:  $+x^2z^5+9.0*y^2z+z$

Результат дифференцирования по z:  $+5.0*x^2yz^4+3.0*y^3+y$

## 3.2 Описание программной реализации

### 3.2.1 Описание класса TNode

```
template<class T>
class TNode {
public:
    T data;
    TNode* pNext;
    TNode() : data(), pNext(nullptr) {};
    TNode(const T& data_, TNode* pNext_ = nullptr) : data(data_),
pNext(pNext_) { return; };
};
```

Назначение: представление звена списка.

Поля:

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

**pNext** – указатель на следующий элемент.

Методы:

**TNode()** ;

Назначение: конструктор по умолчанию.

Входные параметры: отсутствуют.

Выходные параметры: новое звено с инициализированными значениями.

**TNode(const T& data\_, const TNode\* pNext\_ = nullptr);**

Назначение: инициализация значения **data** и указателя на следующее звено.

Входные параметры: **data\_** – значение **data**, **pNext\_** – указатель на следующее звено.

Выходные параметры: новое звено с инициализированными значениями **data** и указателем на следующее звено.

**~TNode()** ;

Назначение: освобождение выделенной памяти.

### 3.2.2 Описание класса TList

```
template<class T>
class TList {
protected:
    TNode<T>* pFirst = nullptr;
    TNode<T>* pLast = nullptr;
    TNode<T>* pStop = nullptr;
    TNode<T>* pCurr = nullptr;
public:
    TList();
    TList(TNode<T>* pFirst_);
    TList(const TList<T>& obj);
    virtual ~TList();
```

```

void Clear();

TNode<T>* Search(const T& data_);
virtual void InsertFirst(const T& data_);
void InsertBefore(const T& data_, const T& NextData);
void InsertAfter(const T& data_, const T& BeforeData);
void InsertBefore(const T& data_); //текущего
void InsertAfter(const T& data_); //текущего
virtual void Remove(const T& data_);

int GetSize() const;
bool IsEmpty() const;
bool IsFull() const;
bool IsEnded() const;
bool IsLast() const;

void Reset();
void Next(const int count = 1);
TNode<T>* GetCurrent() const;
};

```

Назначение: представление списка.

Поля:

**pFirst** – указатель на первый элемент списка.

**pLast** – указатель на последний элемент списка.

**pStop** – указатель на конец списка.

**pCurr** – указатель на текущий элемент списка (по умолчанию равен указателю на первый элемент).

Методы:

**TList();**

Назначение: конструктор по умолчанию.

Входные параметры отсутствуют:

Выходные параметры: отсутствуют.

**TList(const TList& List);**

Назначение: конструктор копирования.

Входные параметры:

**List** – список, на основе которого создаем новый список.

Выходные параметры: отсутствуют.

**TList(const TNode<T>\* pFirst\_);**

Назначение: конструктор с параметрами.

Входные параметры:

**node** – звено, на основе которого создаем новый список.

Выходные параметры: отсутствуют.

**virtual void Clear();**

Назначение: очистка списка.

Входные параметры отсутствуют.

Выходные параметры отсутствуют.

**virtual ~TList();**

Назначение: деструктор.

Входные параметры отсутствуют:

Выходные параметры: отсутствуют.

**bool IsFull() const;**

Назначение: проверка списка на полноту.

Входные параметры отсутствуют:

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

**void Next();**

Назначение: переход к следующему звену.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

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

Назначение: поиск звена с указанным значением.

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

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

**virtual void InsertFirst(const T& data\_);**

Назначение: вставляет новое звено с данными в начало списка.

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

Выходные параметры: отсутствуют.

**void InsertBefore(const T& data\_, const T& NextData);**

Назначение: вставляет новое звено с данными перед звеном с определёнными данными.

Входные параметры: **data\_** – данные для нового звена, **NextData** – данные звена, перед которым будет вставлен новое звено.

Выходные параметры: отсутствуют.



```
void InsertAfter(const T& data_, const T& PrevData);
```

Назначение: вставляет новое звено с данными перед звеном с определёнными данными.

Входные параметры: **data\_** – данные для нового звена, **NextData** – данные звена, перед которым будет вставлен новое звено.

Выходные параметры: отсутствуют.

```
virtual bool IsEmpty() const;
```

Назначение: проверка на пустоту.

Входные параметры: отсутствуют.

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

```
void Reset();
```

Назначение: установка текущего звена на первый.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

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

Назначение: возвращает указатель на текущее звено.

Входные параметры: отсутствуют.

Выходные параметры: указатель на текущее звено.

```
const TList<T>& operator=(const TList<T>& pList);
```

Назначение: операция присваивания.

Входные параметры: **pList** – список, на основе которого создаем новый список.

Выходные параметры: ссылка на присвоенный список.

```
void InsertBefore(const T& data_);
```

Назначение: вставляет новое звено перед текущим звеном.

Входные параметры: **data\_** – данные нового звена.

Выходные параметры: отсутствуют.

```
void InsertAfter(const T& data_);
```

Назначение: вставляет новое звено после текущего звена.

Входные параметры: **data\_** – данные нового звена.

Выходные параметры: отсутствуют.

```
void Remove(const T& data_);
```

Назначение: удаляет звено с определенными данными из списка.

Входные параметры: **data\_** – данные звена для удаления.

Выходные параметры: отсутствуют.

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

Назначение: проверяет, завершен ли список.

Входные параметры: отсутствуют.

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

### 3.2.3 Описание класса TRingList

```
template <class T>  
class TRingList : public TList<T> {  
protected:  
    TNode<T>* pHead;  
public:  
    TRingList();  
    TRingList(TNode<T>* pFirst_);  
    TRingList(const TList<T>& obj);  
    TRingList(const TRingList<T>& obj);  
  
    virtual void InsertFirst(const T& data_);  
    virtual ~TRingList();  
    virtual void Remove(const T& data_);  
};
```

Назначение: представление кольцевого списка.

Поля:

**pHead** – указатель на головной элемент.

Методы:

```
TRingList();
```

Назначение: конструктор по умолчанию.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

```
TRingList(const TNode<T>* pFirst);
```

Назначение: конструктор с параметром.

Входные параметры: **pFirst** – указатель на начальное звено.

Выходные параметры: отсутствуют.

```
TRingList(const TRingList<T>& ringList);
```

Назначение: конструктор копирования

Входные параметры: **ringList** – ссылка на существующий кольцевой список, на основе которого будет создан новый.

Выходные параметры: отсутствуют.

```
void InsertFirst(const T& data_);
```

Назначение: вставляет новое звено с данными в начало списка.

Входные параметры: **data\_** – данные для нового звена.

Выходные параметры: отсутствуют.

```
void Remove(const T& data_);
```

Назначение: удаляет звено списка с заданным ключом.

Входные параметры: **data\_** – данные звена, которое нужно удалить.

Выходные параметры: отсутствуют.

### 3.2.4 Описание класса TMonom

```
class TMonom {
private:
    int st;
    double koef;
public:
    TMonom();
    TMonom(int st_, double koef_);
    TMonom(int degX, int degY, int degZ, double koef_);
    TMonom(const TMonom& obj);

    double get_koef() const;
    int get_degree() const;

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

    void set_koef(double _koef);
    void set_degree(int _degree);
    void set_degree(int degX, int degY, int degZ);

    TMonom operator*(const TMonom& monom) const;

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

    string get_string() const;

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

Назначение: представление монома.

Поля:

**koef** – коэффициент монома.

**st** – степень монома.

Методы:

**TMonom()** ;

Назначение: конструктор по умолчанию.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

**TMonom(int st\_, double koef\_)** ;

Назначение: конструктор с параметрами.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

**TMonom(const TMonom& obj)** ;

Назначение: конструктор копирования.

Входные параметры: отсутствуют

Выходные параметры: отсутствуют.

**string getString() const** ;

Назначение: получение строкового представления монома.

Входные параметры: отсутствуют.

Выходные параметры: моном в виде строки.

**void set\_koef(double \_koef)** ;

Назначение: устанавливает значение коэффициента.

Входные параметры: **\_koef** – новое значение коэффициента.

Выходные параметры: отсутствуют.

**void set\_degree(int \_degree)** ;

Назначение: устанавливает значение степени.

Входные параметры: **\_degree** – новое значение степени.

Выходные параметры: отсутствуют.

**double get\_koef()** ;

Назначение: возвращает значение коэффициента.

Входные параметры: отсутствуют.

Выходные параметры: значение коэффициента.

**int get\_degree()** ;

Назначение: возвращает значение степени.

Входные параметры: отсутствуют.

Выходные параметры: значение степени.

**bool operator<(const TMonom& obj) const;**

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

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

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

**bool operator>(const TMonom& obj) const;**

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

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

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

**bool operator==(const TMonom& obj) const;**

Назначение: операция равенства. Проверяет равенство коэффициента и степени двух объектов **TMonom**.

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

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

**bool operator!=(const TMonom& obj) const;**

Назначение: операция неравенства. Проверяет неравенство коэффициента и степени двух объектов **TMonom**.

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

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

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

Назначение: умножение мономов.

Входные параметры: **monom** – моном, на который будем умножать.

Выходные параметры: произведение мономов.

**TMonom dif\_x();**

Назначение: дифференцирование монома по переменной **x**.

Входные параметры: отсутствуют.

Выходные параметры: дифференциал монома по **x**.

**TMonom dif\_y();**

Назначение: дифференцирование монома по переменной **y**.

Входные параметры: отсутствуют.

Выходные параметры: дифференциал монома по **y**.

**TMonom def\_z()** ;

Назначение: дифференцирование монома по переменной **z**.

Входные параметры: отсутствуют.

Выходные параметры: дифференциал монома по **z**.

### 3.2.5 Описание класса **TPolynom**

```
class TPolinom {
private:
    TArithmeticExpression polynom;
    TRingList<TMonom>* monoms;

    void GetRingListFromString();
    string GetStringFromRingList();
    void AddMonom(TMonom& m);

public:
    TPolynom();
    TPolynom(const string &name);
    TPolynom(TRingList<TMonom>* obj);
    TPolynom(const TPolynom& obj);
    ~TPolynom();

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

    TPolynom operator + (const TPolynom& obj) const;
    TPolynom operator - (const TPolynom& obj) const;
    TPolynom operator - () const;
    TPolynom operator * (const TPolynom& obj) const;

    TPolynom dif_x() const;
    TPolynom dif_y() const;
    TPolynom dif_z() const;
    double operator () (double x, double y, double z);

    TNode<TMonom>* GetCurrent() const;
    string GetString() const {
        return polynom.GetInfix();
    }
};
```

Назначение: работа с полиномами

Поля:

**monoms** — список мономов.

**polynom** — полином представленный в виде арифметического выражения

Методы:

**TPolynomial()** ;

Назначение: конструктор по умолчанию.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

**TPolynomial(const string &name)** ;

Назначение: конструктор с параметром строка.

Входные параметры отсутствуют:

Выходные параметры: отсутствуют.

**TPolynomial(TRingList<TMonom>\* obj)** ;

Назначение: конструктор с параметром кольцевой список.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

**TPolynomial(const TPolynom& obj)** ;

Назначение: конструктор копирования.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

**~TPolynomial()** ;

Назначение: деструктор.

Входные параметры: отсутствуют. Выходные параметры: отсутствуют.

**const TPolynom& operator=(const TPolynom& obj)** ;

Назначение: оператор присваивания.

Входные параметры: **obj** – полином, который будем присваивать.

Выходные параметры: Полученный полином.

**TPolynomial operator + (const TPolynom& obj) const** ;

Назначение: оператор сложения.

Входные параметры: **obj** – полином, с которым будем складывать.

Выходные параметры: Полученный полином.

**TPolynomial operator - (const TPolynom& obj) const** ;

Назначение: оператор вычитания.

Входные параметры: **obj** – полином, который будем вычитать.

Выходные параметры: Полученный полином.

**TPolynomial operator -() const;**

Назначение: оператор унарного минуса.

Входные параметры: отсутствуют.

Выходные параметры: Полученный полином.

**TPolynomial operator \*(const TPolynom& obj) const;**

Назначение: оператор умножения.

Входные параметры: полином, на который будем умножать.

Выходные параметры: Полученный полином.

**TPolynomial dif\_x() const;**

Назначение: дифференцировать полином по переменной x.

Входные параметры: отсутствуют.

Выходные параметры: Дифференцированный по переменной x полином.

**TPolynomial dif\_y() const;**

Назначение: дифференцировать полином по переменной y.

Входные параметры: отсутствуют.

Выходные параметры: Дифференцированный по переменной y полином.

**TPolynomial dif\_z() const;**

Назначение: дифференцировать полином по переменной z.

Входные параметры: отсутствуют.

Выходные параметры: Дифференцированный по переменной z полином.

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

Назначение: находить значение полинома при указанных параметрах.

Входные параметры: значения переменных:

Выходные параметры: значение полинома.

**TNode<TMonom>\* GetCurrent() const;**

Назначение: доступ к текущему моному.

Входные параметры: отсутствуют.

Выходные параметры: указатель на текущий моном.

**string GetString() const;**

Назначение: доступ к полиному в виде строки.

Входные параметры: отсутствуют.

Выходные параметры: полином в виде строки.





## Заключение

В рамках данной лабораторной работы была разработана и реализована библиотека, которая содержит в себе структуру хранения для кольцевого списка и полиномов, которые реализованы на основе кольцевого списка.

Были созданы классы **TNode**, **TList** и **TRingList**, предоставляющие функционал для работы с звеном списка, односвязным списком и кольцевым односвязным списком соответственно.

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 мономов по степени.

Класс **TPolynom** осуществляет работу с полиномами через кольцевой список мономов. Реализованы операторы сложения, вычитания и умножения полиномов, а также оператор присваивания. Класс также предоставляет функционал для вычисления значения полинома для заданных значений переменных, а также позволяет дифференцировать полиномы по трём переменным по отдельности.

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. Связный список [[https://ru.wikipedia.org/wiki/Связный\\_список](https://ru.wikipedia.org/wiki/Связный_список)].
2. Полином [<https://ru.wikipedia.org/wiki/Многочлен>]
3. Линейные списки: эффективное и удобное хранение данных  
[<https://nauchniestati.ru/spravka/hranenie-dannyh-s-ispolzovaniem-linejnyh-spiskov/?ysclid=ltwvp5uwda145425180>]

# Приложения

## Приложение А. Реализация класса TNode

```
#ifndef __NODE_H__
#define __NODE_H__

#include <iostream>

template <typename T>
class TNode {
public:
    T data;
    TNode<T>* pNext;
    TNode() : data(), pNext(nullptr) {};
    TNode(const T& data_, TNode* pNext_ = nullptr) : data(data_),
pNext(pNext_) { return; };
};
#endif
```

## Приложение Б. Реализация класса TList

```
#ifndef __TLIST_H__
#define __TLIST_H__

#include "tnode.h"

template <typename T>
class TList {
protected:
    TNode<T>* pFirst;
    TNode<T>* pCurr;
    TNode<T>* pLast;
    TNode<T>* pStop;
public:
    TList();
    TList(TNode<T>* pFirst_);
    TList(const TList<T>& obj);
    virtual ~TList();
    void Clear();

    TNode<T>* Search(const T& data_);
    virtual void InsertFirst(const T& data_);
    void InsertBefore(const T& data_, const T& NextData);
    void InsertAfter(const T& data_, const T& BeforeData);
    void InsertBefore(const T& data_); //текущего
    void InsertAfter(const T& data_);  //текущего
    virtual void Remove(const T& data_);

    int GetSize() const;
    bool IsEmpty() const;
    bool IsFull() const;
    bool IsEnded() const;
    bool IsLast() const;

    void Reset();
    void Next(const int count = 1);
    TNode<T>* GetCurrent() const;
```

```

};

//конструкторы

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

template<typename T>
TList<T>::TList(TNode<T>* pFirst_) {

    if (pFirst_ == nullptr) {
        pFirst = nullptr;
        pCurr = nullptr;
        pLast = nullptr;
        return;
    }
    pFirst = new TNode<T>(pFirst_>data);
    pStop = nullptr;
    TNode<T>* tmp = pFirst_;
    tmp = tmp->pNext;
    TNode<T>* tmp1 = pFirst;
    while (tmp != pStop) {
        tmp1->pNext = new TNode<T>(tmp->data);
        tmp1 = tmp1->pNext;
        tmp = tmp->pNext;
    }
    pLast = tmp1;
    pLast->pNext = pStop;
    pCurr = pFirst;
}

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

```

```

}

//деструктор и метод Clear для него

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

template<typename T>
void TList<T>::Clear() {

    if (IsEmpty()) {
        return;
    }
    while (pFirst != pStop) {
        TNode<T>* tmp = pFirst;
        pFirst = pFirst->pNext;
        delete tmp;
    }
}

//функции класса

template<typename T>
TNode<T>* TList<T>::Search(const T& data_) {
    TNode<T>* tmp = pFirst;
    while ((tmp != pStop) && (tmp->data != data_)) {
        tmp = tmp->pNext;
    }
    if ((tmp == pStop)) { throw "Not found"; }
    return tmp;
}

template<typename T>
void TList<T>::InsertFirst(const T& data_) {
    if (IsFull()) {
        throw "List is full";
    }
    if (IsEmpty()) {
        pFirst = new TNode<T>(data_, pStop);
        pLast = pFirst;
        pCurr = pFirst;
        return;
    }

    TNode<T>* tmp = new TNode<T>(data_, pFirst);
    pFirst = tmp;
    Reset();
}

template<typename T>
void TList<T>::InsertBefore(const T& data_, const T& NextData) {
    if (IsEmpty()) {
        throw "List is empty";
    }
    if (IsFull()) {
        throw "List is full";
    }
    TNode<T>* tmp = pFirst, * pPrev = nullptr;

```

```

while ((tmp->pNext != pStop) && (tmp->data != NextData)) {
    pPrev = tmp;
    tmp = tmp->pNext;
}
if ((tmp->pNext == pStop) && (tmp->data != NextData)) {
    throw "NextData not is find";
}
TNode<T>* tmp1 = new TNode<T>(data_, tmp);
if (tmp == pFirst) {
    pFirst = tmp1;
    Reset();
    return;
}
pPrev->pNext = tmp1;
Reset();
}

template<typename T>
void TList<T>::InsertAfter(const T& data_, const T& BeforeData) {
    if (IsEmpty()) {
        throw "Node is empty";
    }
    if (IsFull()) {
        throw "List is full";
    }
    TNode<T>* tmp = pFirst->pNext, * pPrev = pFirst;
    while ((tmp != pStop) && (pPrev->data != BeforeData)) {
        pPrev = tmp;
        tmp = tmp->pNext;
    }
    if ((tmp == pStop) && (pPrev->data != BeforeData)) {
        throw "BeforeData not is find";
    }
    TNode<T>* tmp1 = new TNode<T>(data_, tmp);
    pPrev->pNext = tmp1;
    Reset();
}

template<typename T>
void TList<T>::InsertBefore(const T& data_){
    if (IsEmpty()) {
        throw "Node is empty";
    }
    if (IsFull()) {
        throw "List is full";
    }
    TNode<T>* tmp = pFirst, * pPrev = nullptr;
    while ((tmp->pNext != pStop) && (tmp->data != pCurr->data)) {
        pPrev = tmp;
        tmp = tmp->pNext;
    }
    if ((tmp->pNext == pStop) && (tmp->data != pCurr->data)) {
        throw "Current not is find";
    }
    TNode<T>* tmp1 = new TNode<T>(data_, tmp);
    if (tmp == pFirst) {
        pFirst = tmp1;
        Reset();
        return;
    }
    pPrev->pNext = tmp1;
    Reset();
}

```

```

}

template<typename T>
void TList<T>::InsertAfter(const T& data_) {
    if (IsEmpty()) {
        throw "Node is empty";
    }
    if (IsFull()) {
        throw "List is full";
    }
    TNode<T>* tmp = pFirst->pNext, * pPrev = pFirst;
    while ((tmp != pStop) && (pPrev->data != pCurr->data)) {
        pPrev = tmp;
        tmp = tmp->pNext;
    }
    if ((tmp == pStop) && (pPrev->data != pCurr->data)) {
        throw "Current not is find";
    }

    if (tmp == pStop) {
        TNode<T>* tmp1 = new TNode<T>(data_, tmp);
        pPrev->pNext = tmp1;
        pLast = pPrev->pNext;
        Reset();
        return;
    }

    TNode<T>* tmp1 = new TNode<T>(data_, tmp);
    pPrev->pNext = tmp1;

    Reset();
}

template<typename T>
void TList<T>::Remove(const T& data_) {
    if (pFirst == nullptr) {
        throw "Node is empty";
    }
    TNode<T>* tmp = pFirst, * pPrev = nullptr;
    while ((tmp->pNext != pStop) && (tmp->data != data_)) {
        pPrev = tmp;
        tmp = tmp->pNext;
    }
    if ((tmp->pNext == pStop) && (tmp->data != data_)) {
        throw "Node with this data not is find";
    }
    if (tmp == pFirst) {
        if (IsLast()) {
            delete pFirst;
            pFirst = nullptr;
            pCurr = nullptr;
            pLast = nullptr;
        }
        else {
            pFirst = tmp->pNext;
            delete tmp;
            pCurr = pFirst;
            return;
        }
    }
    if (tmp == pLast) {
        pPrev->pNext = pStop;
    }
}

```



```

        delete tmp;
        pLast = pPrev;
        return;
    }
    pPrev->pNext = tmp->pNext;
    delete tmp;
}

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

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

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

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

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

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

```

```

        return true;
    }
    else {
        return false;
    }
}

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

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

    for (int i = 0; i < count; i++) {
        //if (!IsEnded() || pCurr != pStop) pCurr = pCurr->pNext;
        if (!IsEnded()) pCurr = pCurr->pNext;
        else Reset();
    }
}

#endif

```

## Приложение В. Реализация класса TRingList

```

#ifndef __TRINGLIST_H__
#define __TRINGLIST_H__

#include "tlist.h"

template <typename T>
class TRingList : public TList<T> {
protected:
    TNode<T>* pHead;
public:
    TRingList();
    TRingList(TNode<T>* pFirst_);
    TRingList(const TList<T>& obj);
    TRingList(const TRingList<T>& obj);

    virtual void InsertFirst(const T& data_);
    virtual ~TRingList();
    virtual void Remove(const T& data_);
};

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

```

```

}

template <typename T>
TRingList<T>::TRingList(TNode<T>* pFirst_) : TList(pFirst_) {
    pHead = new TNode<T>();
    pHead->pNext = pFirst;
    pStop = pHead;
    pLast->pNext = pHead;

    if (pFirst == nullptr) {
        pHead->pNext = pHead;
    }
}

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

    if (pFirst == nullptr) {
        pHead->pNext = pHead;
    }
    else {
        pHead->pNext = pFirst;
        pLast->pNext = pHead;
    }
}

}

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

    if (obj.pFirst == nullptr) {
        pHead->pNext = pHead;
    }

    else {
        pHead->pNext = pFirst;
        pLast->pNext = pHead;
    }
}

template <typename T>
TRingList<T>::~~TRingList() {
    TList<T>::Clear();
    if (pStop) delete pStop;
}

template <typename T>
void TRingList<T>::InsertFirst(const T& data_) {
    TList<T>::InsertFirst(data_);
    pHead->pNext = pFirst;
}

template<typename T>
void TRingList<T>::Remove(const T& data_) {
    if (pFirst == nullptr) {
        throw "Node is empty";
    }
    TNode<T>* tmp = pFirst, * pPrev = nullptr;
    while ((tmp->pNext != pStop) && (tmp->data != data_)) {
        pPrev = tmp;
        tmp = tmp->pNext;
    }
}

```

```

        if ((tmp->pNext == pStop) && (tmp->data != data_)) {
            throw "Node with this data not is find";
        }
        if (tmp == pFirst) {
            if (IsLast()) {
                delete pFirst;
                pFirst = nullptr;
                pCurr = nullptr;
                pLast = nullptr;
                pHead->pNext = pHead;
                return;
            }
            else {
                pFirst = tmp->pNext;
                delete tmp;
                pCurr = pFirst;
                return;
            }
        }
        if (tmp == pLast) {
            pPrev->pNext = pStop;
            delete tmp;
            pLast = pPrev;
            return;
        }
        pPrev->pNext = tmp->pNext;
        delete tmp;
    }

#endif

```

## Приложение Г. Реализация класса TMonom

```

#include "tmonom.h"
TMonom::TMonom() {
    st = int();
    koef = double();
}
TMonom::TMonom(int st_, double koef_) {
    if ((st_ < 0) || (st_ > 999)) {
        throw "Incorrect st";
    }

    this->st = st_;
    this->koef = koef_;
}
TMonom::TMonom(int degX, int degY, int degZ, double koef_) {
    int tmp_st = degX * 100 + degY * 10 + degZ;
    if (tmp_st < 0 || tmp_st > 999) {
        throw "Error: st incorrect";
    }
    st = tmp_st;
    koef = koef_;
}
TMonom::TMonom(const TMonom& obj) {
    st = obj.st;
    koef = obj.koef;
}

bool TMonom::operator == (const TMonom& obj) {
    if (obj.st == st) {
        return true;
    }
}

```

```

        return false;
    }
    bool TMonom::operator != (const TMonom& obj) {
        if (obj.st != st || obj.koef != koef) {
            return true;
        }
        return false;
    }
    bool TMonom::operator < (const TMonom& obj) {
        if (st < obj.st) {
            return true;
        }
        return false;
    }
    bool TMonom::operator > (const TMonom& obj) {
        if (st > obj.st) {
            return true;
        }
        return false;
    }
    TMonom TMonom::operator*(const TMonom& monom) const {
        int degZ1 = get_degZ();
        int degY1 = get_degY();
        int degX1 = get_degX();

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

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

        int res_degree = st + monom.st;
        double res_coeff = koef * monom.koef;

        TMonom res(res_degree, res_coeff);
        return res;
    }
    string TMonom::get_string() const {
        int degZ = st % 10;
        int degY = (st % 100) / 10;
        int degX = st / 100;
        string res = "";

        res += to_string(koef);

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

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

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

        return res;
    }
}

```

```

int TMonom::get_degX() const {
    int degX = st / 100;
    return degX;
}
int TMonom::get_degY() const {
    int degY = (st / 10) % 10;
    return degY;
}
int TMonom::get_degZ() const {
    int degZ = st % 10;
    return degZ;
}
double TMonom::get_koef() const {
    return koef;
}
}
TMonom TMonom::dif_x() const {
    int degX = get_degX();

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

        TMonom res(_degree, _coeff);
        return res;
    }
}
TMonom TMonom::dif_y() const {
    int degY = get_degY();

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

        TMonom res(_degree, _coeff);
        return res;
    }
}
TMonom TMonom::dif_z() const {
    int degZ = get_degZ();

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

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

```

## Приложение Д. Реализация класса TPolinom

```
#include "tpolinom.h"
TPolinom::TPolinom(): polynom("\n") {
    monoms = new TRingList<TMonom>();
}

TPolinom::TPolinom(const string &name): polynom(name) {
    GetRingListFromString();
    polynom = TArithmeticExpression(GetStringFromRingList());
}

TPolinom::TPolinom(TRingList<TMonom>* obj) {
    monoms = new TRingList<TMonom>;

    obj->Reset();
    int flag = 1;
    while(!obj->IsEnded()) {
        if (flag == 1) {
            monoms->InsertFirst(obj->GetCurrent()->data);
            obj->Next();
            flag = 0;
        }
        else {
            AddMonom(obj->GetCurrent()->data);
            obj->Next();
        }
    }

    obj->Reset();

    polynom = TArithmeticExpression(GetStringFromRingList());
}

TPolinom::TPolinom(const TPolinom& obj) {
    monoms = new TRingList<TMonom>(*obj.monoms);
    polynom = TArithmeticExpression(obj.polynom.GetInfix());
}

TPolinom::~TPolinom() {
    if (monoms) delete monoms;
}

void TPolinom::GetRingListFromString() {
    vector<string> lexems_from_polymon = polynom.GetLexems();
    double coeff = 1;
    int degX = 0, degY = 0, degZ = 0;
    int next_const_sign = 1;
    monoms = new TRingList<TMonom>();

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

    int flag = 1;
    for (int i = 0; i < lexems.size(); i++) {

        if (lexems[i] == "+" || lexems[i] == "-") {

            int tmp_st = 100 * degX + 10 * degY + degZ;
            TMonom monom(tmp_st, next_const_sign * coeff);
```

```

//TMonom monom(degX, degY, degZ, next_const_sign * coeff);
if (flag == 1) {
    monoms->InsertFirst(monom);
    flag = 0;
}
else {
    AddMonom(monom);
}

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

```



```

        degZ += 1;
    }
}

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

}

int tmp_st = 100 * degX + 10 * degY + degZ;
TMonom monom(tmp_st, next_const_sign * coeff);
//TMonom monom(degX, degY, degZ, next_const_sign * coeff);
if (flag == 1) {
    monoms->InsertFirst(monom);
    flag = 0;
}
else {
    AddMonom(monom);
}
}

void TPolynom::AddMonom(TMonom& m) {
    if (m.get_koef() == 0) return;

    while (!monoms->IsLast()) {
        TNode<TMonom>* curr = monoms->GetCurrent();

        if (m == curr->data) {
            double coeff = m.get_koef() + curr->data.get_koef();
            if (coeff == 0.0f) {
                monoms->Remove(curr->data);
                monoms->Reset();
                return;
            }
            else {
                curr->data.set_koef(coeff);
                monoms->Reset();
                return;
            }
        }

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

        else {
            monoms->Next();
        }
    }

    TNode<TMonom>* curr = monoms->GetCurrent();

    if (m == curr->data) {
        double coeff = m.get_koef() + curr->data.get_koef();
        if (coeff == 0.0f) {
            monoms->Remove(curr->data);
            monoms->Reset();
        }
    }
}

```

```

        return;
    }
    else {
        curr->data.set_koef(coeff);
        monoms->Reset();
        return;
    }
}

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

else {
    monoms->InsertAfter(m);
}

}

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

    string res = "";
    monoms->Reset();
    int flag = 0;
    if (monoms->GetCurrent() == nullptr) {
        return res;
    }
    else {
        res += monoms->GetCurrent()->data.get_string();
        monoms->Next();
        flag = 1;
        while (!monoms->IsEnded()) {
            string monom = monoms->GetCurrent()->data.get_string();
            if (monom[0] == '-') res += monom;
            else res += "+" + monom;
            flag = 0;
            monoms->Next();
        }
    }

    monoms->Reset();

    return res;
}

TNode<TMonom>* TPolynom::GetCurrent() const {
    return monoms->GetCurrent();
}

//операторы

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

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

    return (*this);
}

```

```

TPolynom TPolynom:: operator + (const TPolynom& obj) const{
    TPolynom res(*this);

    obj.monoms->Reset();
    while (!obj.monoms->IsEnded()) {
        TMonom curr_monom = obj.GetCurrent()->data;
        if (res.monoms->IsEmpty()) {
            res.monoms->InsertFirst(curr_monom);
        }
        else {
            res.AddMonom(curr_monom);
        }
        obj.monoms->Next();
    }

    obj.monoms->Reset();

    res.polynom = TArithmeticExpression(res.GetStringFromRingList());
    return res;
}

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

    res.monoms->Reset();
    while (!res.monoms->IsEnded()) {
        TNode<TMonom>* curr = res.GetCurrent();
        double koef_ = (-1) * curr->data.get_koef();
        curr->data.set_koef(koef_);
        res.monoms->Next();
    }

    res.monoms->Reset();
    res.polynom = TArithmeticExpression(res.GetStringFromRingList());

    return res;
}

TPolynom TPolynom::operator-(const TPolynom& obj) const{
    TPolynom res(-obj + (*this));

    return res;
}

TPolynom TPolynom::operator*(const TPolynom& obj) const{
    TPolynom res;
    int flag = 1;
    monoms->Reset();
    while (!monoms->IsEnded()) {
        TMonom curr1 = monoms->GetCurrent()->data;

        obj.monoms->Reset();
        while (!obj.monoms->IsEnded()) {
            if (flag == 1) {
                TMonom curr2 = obj.GetCurrent()->data;
                res.monoms->InsertFirst(curr1 * curr2);
                flag = 0;
                obj.monoms->Next();
            }
            else {
                TMonom curr2 = obj.GetCurrent()->data;
                res.AddMonom(curr1 * curr2);
                obj.monoms->Next();
            }
        }
    }
}

```

```

        }
    }

    monoms->Next();
}

obj.monoms->Reset();
monoms->Reset();
res.polynom = TArithmeticExpression(res.GetStringFromRingList());

return res;
}
//дифференцирование
TPolynom TPolynom::dif_x() const {
    TPolynom res;
    int flag = 1;
    monoms->Reset();
    while (!monoms->IsEnded()) {
        if (flag == 1) {
            TMonom tmp = GetCurrent()->data.dif_x();
            if (tmp.get_koef() != 0) {
                res.monoms->InsertFirst(tmp);
                monoms->Next();
                flag = 0;
            }
            else {
                monoms->Next();
            }
        }
        else {
            TMonom tmp = GetCurrent()->data.dif_x();
            if (tmp.get_koef() != 0) {
                res.AddMonom(tmp);
                monoms->Next();
            }
            else {
                monoms->Next();
            }
        }
    }

    monoms->Reset();
    res.polynom = TArithmeticExpression(res.GetStringFromRingList());
    return res;
}

TPolynom TPolynom::dif_y() const {
    TPolynom res;
    int flag = 1;
    monoms->Reset();
    while (!monoms->IsEnded()) {
        if (flag == 1) {
            TMonom tmp = GetCurrent()->data.dif_y();
            if (tmp.get_koef() != 0) {
                res.monoms->InsertFirst(tmp);
                monoms->Next();
                flag = 0;
            }
            else {
                monoms->Next();
            }
        }
        else {

```

```

        TMonom tmp = GetCurrent()->data.dif_y();
        if (tmp.get_koef() != 0) {
            res.AddMonom(tmp);
            monoms->Next();
        }
        else {
            monoms->Next();
        }
    }

    monoms->Reset();
    res.polynom = TArithmeticExpression(res.GetStringFromRingList());
    return res;
}

```

```

TPolynom TPolynom::dif_z() const {
    TPolynom res;
    int flag = 1;
    monoms->Reset();
    while (!monoms->IsEnded()) {
        if (flag == 1) {
            TMonom tmp = GetCurrent()->data.dif_z();
            if (tmp.get_koef() != 0) {
                res.monoms->InsertFirst(tmp);
                monoms->Next();
                flag = 0;
            }
            else {
                monoms->Next();
            }
        }
        else {
            TMonom tmp = GetCurrent()->data.dif_z();
            if (tmp.get_koef() != 0) {
                res.AddMonom(tmp);
                monoms->Next();
            }
            else {
                monoms->Next();
            }
        }
    }

    monoms->Reset();
    res.polynom = TArithmeticExpression(res.GetStringFromRingList());
    return res;
}

```

```

double TPolynom::operator () (double x, double y, double z) {
    map<string, double> map_tmp;
    map_tmp["x"] = x;
    map_tmp["y"] = y;
    map_tmp["z"] = z;

    string tmp = GetString();
    polynom = TArithmeticExpression(tmp, map_tmp);

    return polynom.Calculate();
}

```

## Приложение E. Sample\_tpolinom

```
#include "tpolynom.h"
#include<string>
#include<Windows.h>
using namespace std;

int main()
{
    setlocale(LC_ALL, "Russian");

    try {

        string name1;
        string name2;

        cout << "Введите полиномы, с которыми хотите работать!\n";

        cout << "Первый полином: ";
        cin >> name1;
        cout << "Второй полином: ";
        cin >> name2;

        TPolynom p1(name1);
        TPolynom p2(name2);
        cout << "p1 = " << p1.GetString() << endl;
        cout << "p2 = " << p2.GetString() << endl << endl;

        cout << "Операции над полиномами:" << endl;
        cout << "Результат сложения      :" << (p1 + p2).GetString() << endl;
        cout << "Результат вычитания     :" << (p1 - p2).GetString() << endl;
        cout << "Результат умножения    :" << (p1 * p2).GetString() << endl <<
endl;

        cout << "Взятие производной по x 1 полинома:" <<
(p1.dif_x()).GetString() << endl;
        cout << "Взятие производной по y 1 полинома:" <<
(p1.dif_y()).GetString() << endl;
        cout << "Взятие производной по z 1 полинома:" <<
(p1.dif_z()).GetString() << endl;
    }
    catch (string exp) {
        cout << exp << endl;
    }
}
```