МИНИСТЕРСТВО НАУКИ И ВЫСШЕГООБРАЗОВАНИЯ
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:

«Списки»

Выполнил(а): студент(ка)
группы3822Б1ФИ1
_____ / Чистов А.Д./
Подпись

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

Подпись

Нижний Новгород
2024

Содержание

Введение

Полином — это алгебраическое выражение, состоящее из суммы или разности членов, каждый из которых является произведением переменной (или переменных) на некоторую степень.

Полиномы широко используются в математике и её приложениях. Они играют важную роль в алгебре, анализе, теории чисел, физике, инженерии и других областях. Например, они часто используются при дифференцировании и интегрировании функций. Поскольку производная или интеграл полинома легко вычисляется, это делает этот тип алгебраических выражений удобным инструментом при анализе функций. Таким образом, полиномы представляет собой мощный и универсальный инструмент в математике и науке.

1 Постановка задачи

Цель – реализовать классы THeadRingList и TPolynom.

Задачи:

1. Исследовать тематическую литературу.
2. Реализовать класс THeadRingList.
3. Реализовать класс TPolynom.
4. Провести тестирование разработанных классов для проверки их корректной работы.
5. Сделать выводы о проделанной работе.

2 Руководство пользователя

2.1 Приложение для демонстрации работы полиномов

1. Запустите приложение с названием sample_polynom.exe. В результате появится окно, показанное ниже, где вам нужно будет ввести первый полином (рис. 1).

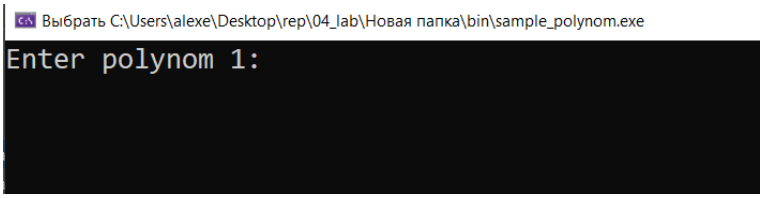


Рис. 1. Основное окно программы

2. Далее вам будет представлены производные по каждой из трех переменных x, y, z, а также значение полинома в точке с координатами (0.1,0.2,0.3) (рис. 2).

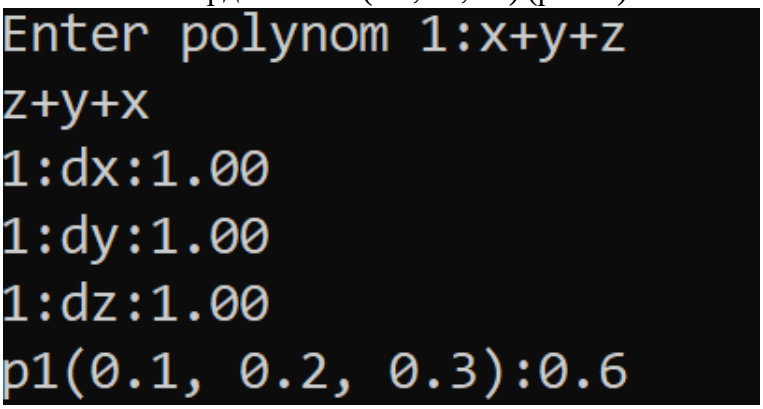


Рис. 2. Результат работы программы

3. Далее вам необходимо будет ввести второй полином (рис. 3).

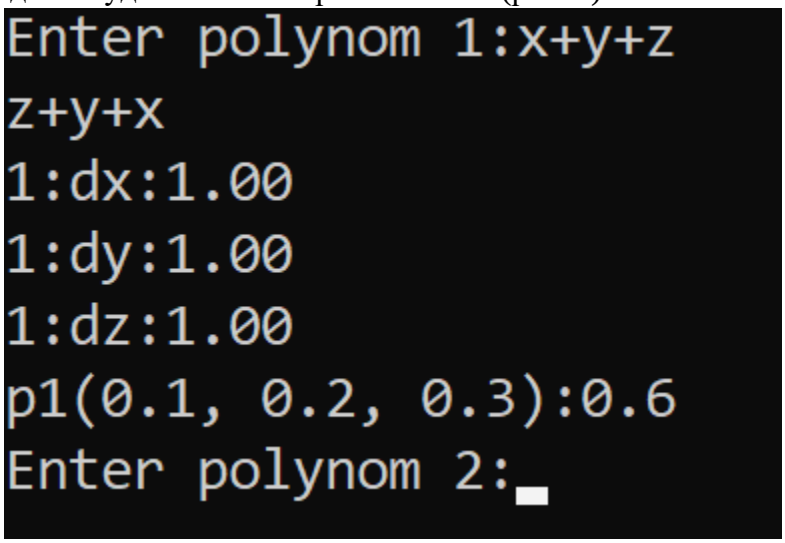


Рис. 3. Ввод второго полинома

4. Далее вам будет представлен результат в виде суммы, разности и произведения двух полиномов (рис. 4).

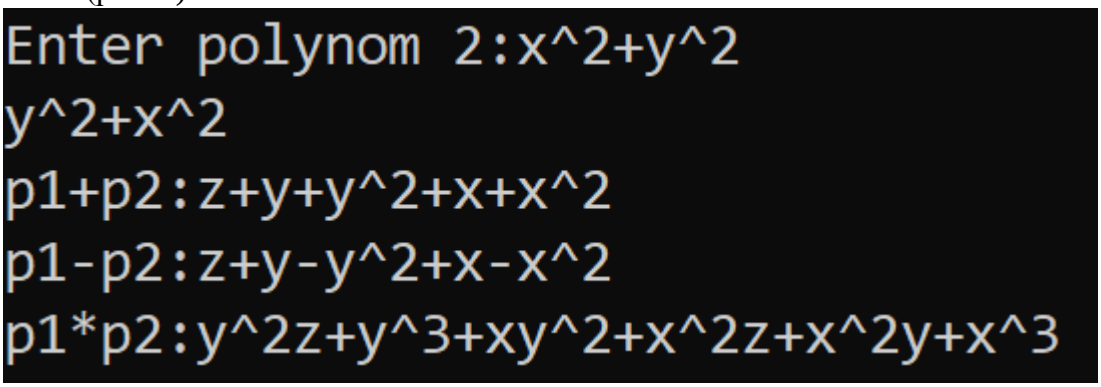


Рис. 4. Результат работы программы

3 Руководство программиста

3.1 Описание алгоритмов

3.1.1 Линейный односвязный список

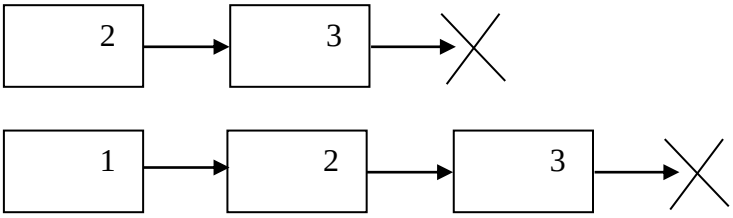
Связный список — базовая динамическая структура данных в информатике, состоящая из узлов, содержащих данные и ссылки на следующий и/или предыдущий узел списка.

Данный класс поддерживает следующие операции:

- **Операция добавления в начало**

1) Создаем узел, который будет являться головой списка.
2) Устанавливаем этот узел в качестве первого элемента списка.

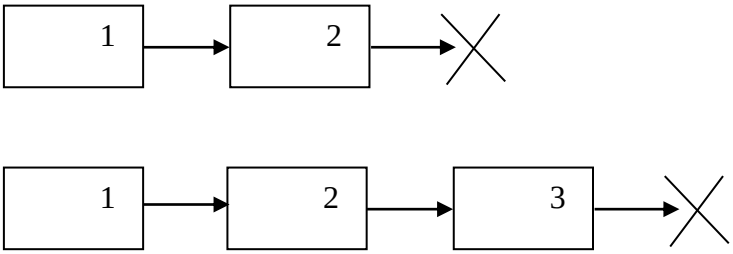
Пример:



- **Операция добавления в конец**

1) Создаем новый узел, который будет указывать на конец списка.
   a) Если список пустой (то есть нет других узлов), то новый узел становится началом списка.
   b) Если список не пустой, начинаем с первого узла и движемся по списку до тех пор, пока не достигнем последнего узла.
2) Когда мы найдем текущий последний узел списка, делаем его указатель указывающим на новый узел, чтобы добавить новый узел в конец списка.

Пример:



- **Операция поиска**

1) Начать с первого узла списка и проверить текущий узел на наличие искомого элемента.

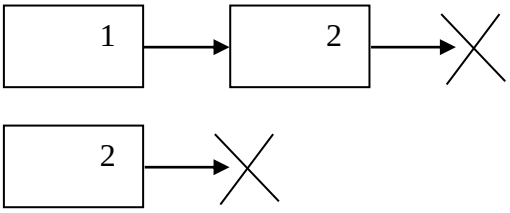
a) Если текущий узел содержит искомый элемент, завершить операцию успешно.
b) Если искомый элемент не найден в текущем узле, перейти к следующему узлу в списке (переход к следующему элементу).
c) Если достигнут конец списка (следующий узел равен nullptr), искомый элемент не найден в списке.

2) Возвращаем элемент или генерируем исключение.

- **Операция удаления элемента**

1) Используется функция поиска, чтобы найти элемент для удаления.
2) Если элемент найден, удаляем его из списка и обновляем указатели элементов.
3) Если элемент не найден при прохождении списка, сгенерируйте исключение или обработайте ситуацию отсутствия искомого элемента в списке.

Пример:



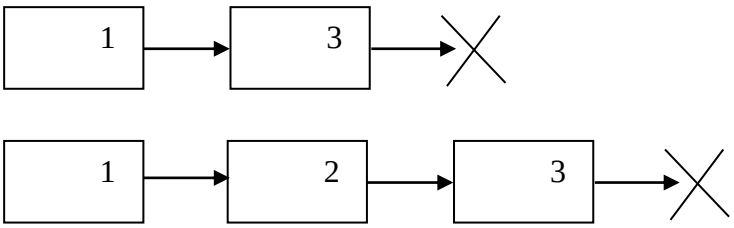
- **Операция очищения**

1) Последовательное удаление узлов.
2) Обновление указателей на начало и конец списка.

- **Операция вставки перед элементом**

1) Поиск узла с указанным значением.
2) Если найденный узел является первым узлом списка, вызывается метод вставки в начало.
3) Создание нового узла.
4) Если элемент не найден генерируем исключение.

Пример вставки перед 3:

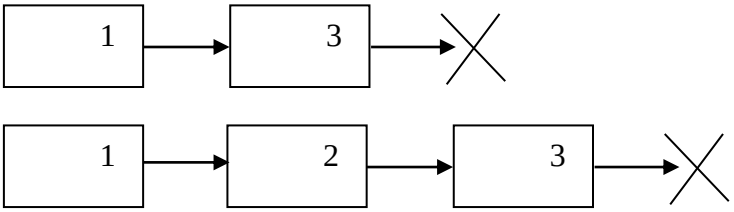


- **Операция вставки после элемента**

1) Поиск узла с указанным значением.

2) Если найденный узел является последним узлом списка, вызывается метод вставки в конец.
3) Создание нового узла.
4) Если элемент не найден генерируем исключение.

Пример вставки после 1:



- **Операция получения текущего элемента**

Метод возвращает указатель на текущий узел списка. Этот узел используется внутри класса для отслеживания текущей позиции при выполнении операций со списком.

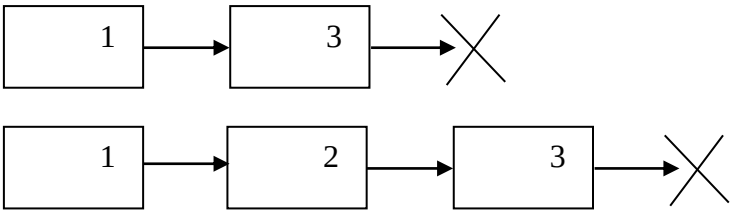
- **Операция получения размера листа**

Метод возвращает количество элементов в списке. Он проходит по всем узлам списка, увеличивая счетчик, пока не достигнет конца списка.

- **Операция вставки в сортированный список**

1) Проверка списка на пустоту или меньше ли элемент первого элемента. Если да, то вызывается метод вставки в начало и выход из функции.
2) Иначе проходимся по списку и ищем элемент.
3) Если список закончился, вызывается метод вставки в конец и выход из функции.
4) Иначе создается новый узел с значением.

Пример:



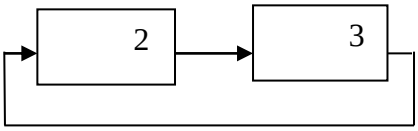
3.1.2 Кольцевой список с головой

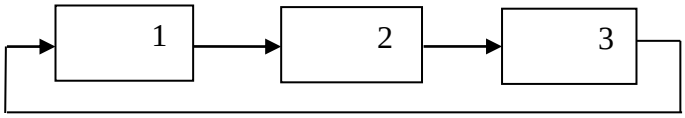
Кольцевые списки могут быть представлены как круг, где последний элемент связан с первым, образуя замкнутую структуру данных. Такой подход обеспечивает удобство в работе с элементами списка и позволяет производить циклические операции без необходимости проверки на достижение конца списка.

Данный класс поддерживает все операции не кольцевого листа, однако некоторые операции переопределяются:

- **Операция вставки в начало**
  1) Вызывает метод вставки в начало из базового класса для вставки элемента.
  2) Обновляется указатель на следующий элемент последнего узла на указатель на голову списка, а указатель на следующий элемент головы списка на первый узел.

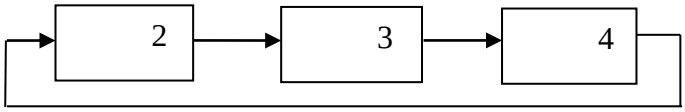
Пример:

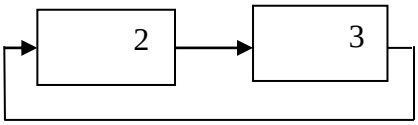




- **Операция удаления элемента**
  1) Начинаем с первого узла и проверяем равно ли значение удаляемого элемента текущему узлу списка.
  2) Если в списке один узел, удаляем его и устанавливаем все указатели в nullptr.
  3) Если указанный элемент первый узел в списке, то удаляем его и сдвигаем указатель на голову и первый узел на следующий.
  4) Иначе вызываем функцию удаления из базового класса.

Пример:





3.1.3 Полином

Полином представляет собой кольцевой список мономов, который получается в результате преобразования входной строки. Полиномы состоят из мономов, которые имеют коэффициент и степень.

Данный класс поддерживает следующие операции:

- **Операция суммирования полиномов**
  1) Создаем новый пустой отсортированный список для хранения суммированных мономов.
  2) Перебираем мономы из двух исходных списков, добавляя их в новый список с учетом сортировки.

3) После завершения операции суммирования удаляем из нового списка все мономы с коэффициентом, равным нулю.

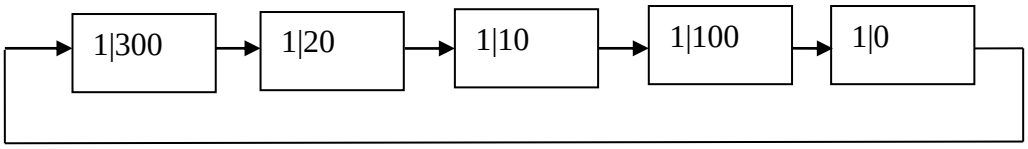
Пример:

Первый полином: `x^3+y^2+y+x+1`

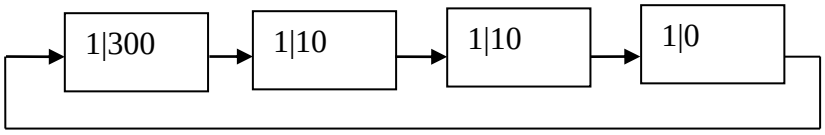
Второй полином: `x^3+y+z+1`

Результат: `2x^3+y^2+2y+x+z+2`

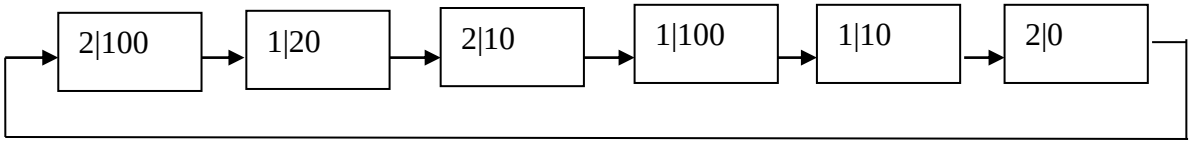
Мономы первого полинома:



Мономы второго полинома:



Результат:



- **Операция разности полиномов**

Операция состоит из двух этапов:

1) Умножение коэффициентов вычитаемого полинома на (-1).
2) Применение операции суммы к данным полиномам.

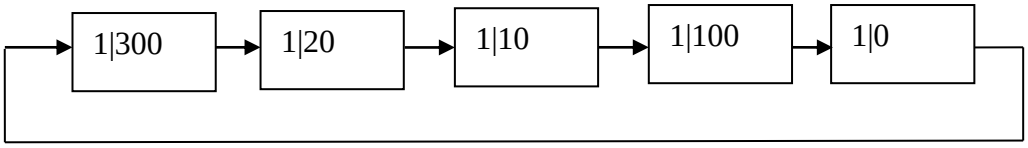
Пример:

Первый полином: `x^3+y^2+y+x+1`

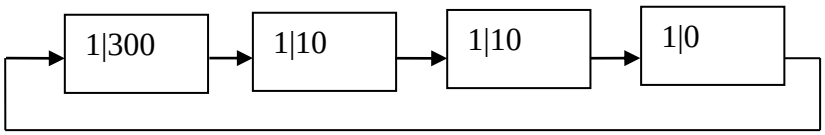
Второй полином: `x^3+y+z+1`

Результат:`x+y^2+-z`

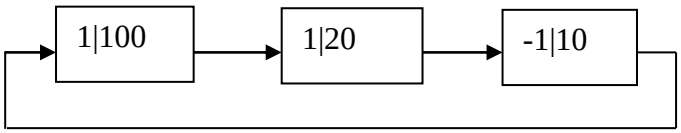
Мономы первого полинома:



Мономы второго полинома:



Результат:



- **Операция произведения полиномов**

1) Создаем новый пустой список для хранения умноженных мономов.
2) Итерируем по мономам первого полинома.
3) Внутри этого цикла итерируем также по мономам второго полинома.
4) Умножаем каждую пару мономов и добавляем ненулевые результаты в новый список.
5) Если в результате умножения нет ненулевых мономов, добавляем нулевой моном.
6) После завершения операции умножения возвращаем новый полином.

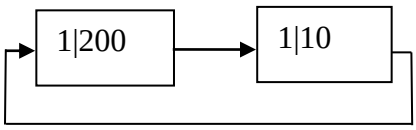
Пример:

Первый полином: `x^2+y`

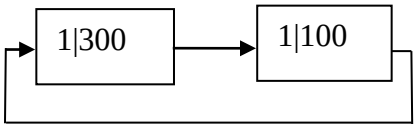
Второй полином:`x^3+x`

Результат:`x^5+x^3+x^3*y+x*y`

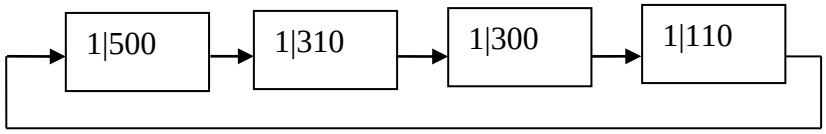
Мономы первого полинома:



Мономы второго полинома:



Результат:



- **Операция дифференцирования полиномов**

1) Создаем новый пустой список для хранения производных мономов.
2) Итерируем по мономам в исходном полиноме.
3) Для каждого монома уменьшаем его степень по указанной переменной и умножаем коэффициент на новую степень.
4) Полученные мономы добавляем в новый список.
5) После завершения операции по всем мономам, возвращаем новый полином, который является производной исходного по указанной переменной. Если в результате нет ненулевых мономов, добавляется нулевой моном.

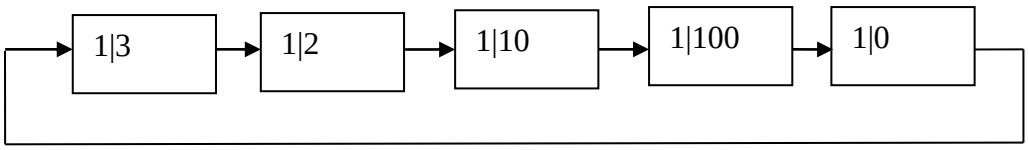
Пример:

Полином: `z^3+z^2+y+x+1`

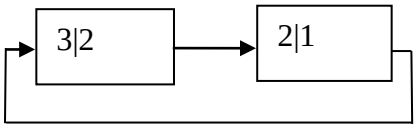
Результат после дифференцирования функции по z:`3z^2+2z`

Мономы полинома до операции:

Мономы первого полинома:



Результат:



3.2 Описание программной реализации

3.2.1 Схема наследования классов

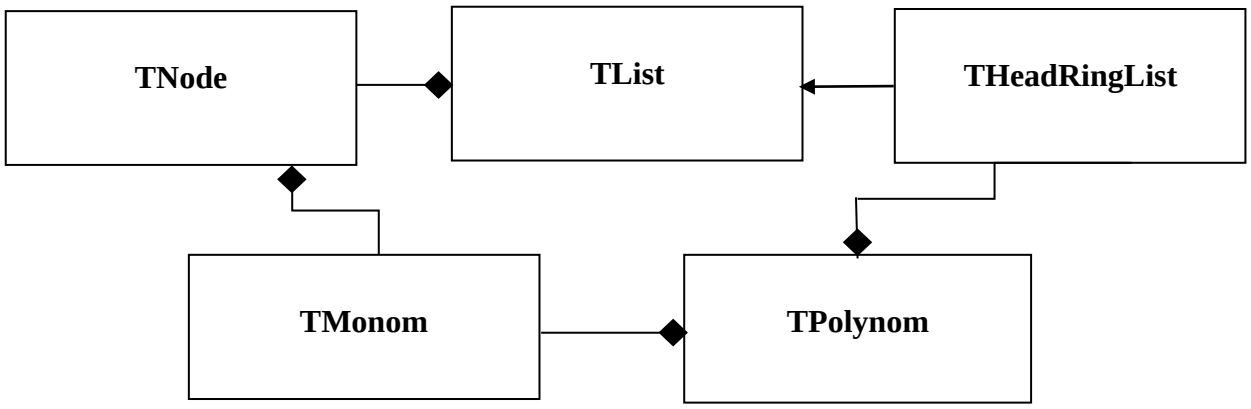


Рис. 5. Схема наследования классов

На рис. 5 показаны отношения между классами:

- обычными стрелками показаны отношения наследования (базовый класс – производный класс),
- ромбовидными стрелками – отношения ассоциации (класс-владелец – класс-компонент).

3.2.2 Описание класса 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) {};
};
```

Поля:

- **data** – данные, хранящиеся в звене.
- **pNext** – указатель на следующее звено.

Методы:

- **TNode();**

Назначение: конструктор по умолчанию.

Входные параметры отсутствуют:

Выходные параметры: отсутствуют.

- **TNode(constT&data);**

Назначение: конструктор с параметром.

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

Выходные параметры: отсутствуют.

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

Назначение: конструктор с параметром.

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

Выходные параметры: отсутствуют.

- **TNode(constT&data, TNode<T>* _pNext);**

Назначение: конструктор с параметрами.

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

Выходные параметры: отсутствуют.

3.2.3 Описание класса TList

```
template <typename T>
class TList {
protected:
      TNode<T>* pFirst;
      TNode<T>* pCurr;
      TNode<T>* pStop;
      TNode<T>* pLast;
public:
      TList();
      TList(TNode<T>* _pFirst);
      TList(const TList<T>& list);
      virtual ~TList();
      virtual void insert_first(const T& data);
      virtual void remove(const T& data);
      virtual void insert_before(const T& data, const T&nextdata);
      void insert_last(const T& data);
      void insert_after(const T& data, const T&beforedata);
      TNode<T>* search(const T& data);
      TNode<T>* GetCurrent() const;
      void clear();
      int GetSize() const;
      bool IsEmpty() const;
      bool IsFull() const;
      bool IsEnded()const;
      void next();
      void reset();
      void insert_sort(const T& data);
};
```

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);`**

Назначение: создание списка с заданным начальным узлом

Входные параметры: **`_pFirst`** – указатель на первый узел списка.

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`**.

- **`virtual ~TList();`**

Назначение: освобождение памяти списка при удалении объекта.

Входные параметры: отсутствуют.

Выходные параметры: освобожденная память объекта класса TList.

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

Назначение: вставляет новый узел с данными в начало списка.

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

Выходные параметры: отсутствуют.

- **`virtual void remove(constT&data);`**

Назначение: удаляет узел с определенными данными из списка.

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

Выходные параметры: отсутствуют.

- **`void insert_before(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 insert_last(constT&data);`**

Назначение: вставляет новый узел с данными в конец списка.

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

Выходные параметры: отсутствуют.

- **`void insert_after(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.

- **`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.

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

Назначение: возвращает указатель на текущий узел.

Входные параметры: отсутствуют.

Выходные параметры: указатель на текущий узел.

- **`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.

- **`void next();`**

Назначение: переход к следующему узлу.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

- **`virtual void reset();`**

Назначение: установка текущего узла как первого.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

- **`void insert_sort(const T& data);`**

Назначение: вставляет новый узел с данными в отсортированный список.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

3.2.4 Описание класса THeadRingList

```
template <typename T>
class THeadRingList : public TList<T> {
private:
TNode<T>* pHead;
public:
THeadRingList();
THeadRingList(const THeadRingList&ringL);
    virtual ~THeadRingList();
    void insert_first(const T& data);
    void insert_before(const T& who, const T& where);
};
```

Поля:

pHead – указатель на головной элемент.

Методы:

- **THeadRingList();**

Назначение: конструктор без параметров, создает пустой кольцевой список.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

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

Назначение: конструктор копирования, создает копию существующего кольцевого списка.

Входные параметры: rlist – ссылка на существующий кольцевой список.

Выходные параметры: отсутствуют.

- **virtual ~THeadRingList();**

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

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

- **void insert_first(const T& data);**

Назначение: вставляет новый узел с данными в начало списка.

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

Выходные параметры: отсутствуют.

- **void insert_before(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.

3.2.5 Описание классаTMonom

```
class TMonom  {
 public:
      double coeff;
      int degree;

      TMonom();
      TMonom(const TMonom&monom);
      TMonom(double _coeff, int _degree);
      bool operator ==(const TMonom& data)const;
      bool operator !=(const TMonom& data)const;
      bool operator <(const TMonom& data)const;
      bool operator <=(const TMonom& data)const;
      TMonom operator*(const TMonom&monom)const;
      TMonom operator+(const TMonom&monom)const;
};
```

Назначение: представление монома.

Поля:

coeff– 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&monom);**

Назначение: конструктор копирования, создает копию существующего TMonom.

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

Выходные параметры: отсутствуют.

- **TMonom(double _coeff, int _degree);**

Назначение: создает объект TMonom с указанным коэффициентом и степенью.

Входные параметры: **_coeff** – коэффициент, **_degree** - степень.

Выходные параметры: отсутствуют.

- **bool operator == (const TMonom& data) const;**

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

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

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

- **bool operator != (const TMonom& data) const;**

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

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

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

- **bool operator < (const TMonom& data) const;**

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

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

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

- **bool operator <= (const TMonom& data) const;**

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

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

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

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

Назначение: произведение мономов: коэффиценты умножаются, а степени складываются.

Входные параметры: **monom**– ссылка на объект TMonom.

Выходные параметры: моном, который получился в результате операции.

- **TMonom operator+(const TMonom&monom)const;**

Назначение: произведение мономов: коэффиценты складываются, а степени не изменяются.

Входные параметры: **monom** – ссылка на объект TMonom.

Выходные параметры: моном, который получился в результате операции.

3.2.6 Описание класса TPolynom

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

      void ParseMonoms();
      void conversion(string& str) const;
public:
      TPolynom();
      TPolynom(const string& _name);
      TPolynom(const THeadRingList<TMonom>* m);
      TPolynom(const TPolynom& p);
      ~TPolynom();
      TPolynom operator +(const TPolynom& p);
      TPolynom operator -(const TPolynom& p);
      TPolynom operator-() const;
      TPolynom operator *(const TPolynom& p);
      const TPolynom& operator =(const TPolynom& p);
      double operator ()(double x, double y, double z);
      TPolynomdx() const;
      TPolynomdy() const;
      TPolynomdz() const;
      string ToString()const;
      bool operator==(const TPolynom&p) const;
      bool operator!=(const TPolynom& p) const;
```

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

Назначение: представление полинома.

Поля:

name – строка полинома.

monoms – кольцевой линейный односвязный список.

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& name);**

Назначение: создает объект TPolynom с указанным именем.

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

Выходные параметры: отсутствуют.

- **TPolynom(const THeadRingList<TMonom>* m);**

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

Входные параметры: **THeadRingList <TMonom>* m**– 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-();**

Назначение: получение полинома с противоположными по знаку коэффициентами.

Входные параметры: отсутствуют.

Выходные параметры: объект TPolynom с противоположными знаками.

- **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.

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

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

Входные параметры: x, y, z.

Выходные параметры: значение полинома с заданными значениями переменных.

- **TPolynom dx() const;**

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

Входные параметры: отсутствуют.

Выходные параметры: полином – производная по «x».

- **TPolynom dy() const;**

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

Входные параметры: отсутствуют.

Выходные параметры: полином – производная по «y».

- **TPolynom dz() const;**

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

Входные параметры: отсутствуют.

Выходные параметры: полином – производная по «z».

- **String ToString()const;**

Назначение: возвращает строковое представление полинома.

Входные параметры: отсутствуют.

Выходные параметры: строковое представление полинома.

- **`void ParseMonoms();`**

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

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

Выходные параметры: отсутствуют.

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

Назначение: оператор вывода для класса TPolynom.

Входные параметры:

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

`p` – ссылка на объект типа TPolynom который будет выводиться.

Выходные параметры: ссылка на объект типа ostream.

- **`void conversion(string& str) const;`**

Назначение: удаляет из строки пробелы

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

Выходныепараметры: отсутствуют.

- **`bool operator == (const TPolynom&p) const;`**

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

Входные параметры: **`p`** – ссылка на объект TPolynom для сравнения.

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

- **`bool operator == (const TPolynom&p) const;`**

Назначение: перегруженный оператор "не равно". Проверяет равенство коэффициента и степени двух объектов TPolynom.

Входные параметры: **`p`** – ссылка на объект TPolynom для сравнения.

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

Заключение

В результате данной лабораторной работе удалось изучить и реализовать алгоритм обработки полиномов на основе связных списков. Была освоена работа с мономами и полиномами, а также разработали функционал для выполнения операций над ними: сложение, вычитание, умножение, и вычисление значений при заданных переменных. Это обеспечивает удобный и эффективный способ решения математических задач, требующих работы с полиномами.

Литература

1. Кольцевой односвязный список [Структуры данных в C# | Кольцевой односвязный список (metanit.com)].
2. Лекция «Списковые структуры хранения» Сысоев А. В.
   [https://cloud.unn.ru/s/6g44ey6HFB4ncDy].

Приложения

Приложение А. Реализация класса TList

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

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

	pFirst = new TNode<T>(l.pFirst->data);
	TNode<T>* tmp = pFirst;
	TNode<T>* ltmp = l.pFirst->pNext;
	while (ltmp != l.pStop)
	{
		tmp->pNext = new TNode<T>(ltmp->data);
		tmp = tmp->pNext;
		ltmp = ltmp->pNext;
	}
	pLast = tmp;
	pCurr = pFirst;
	pStop = nullptr;
}

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

template <typename T>
void TList<T>::clear() {
	if (pFirst == nullptr) return;
	TNode<T>* curr = pFirst;
	while (curr != pStop) {
		TNode<T>* next = curr->pNext;
		delete curr;
		curr = next;
	}
	pCurr = nullptr;
```

```
	pFirst = nullptr;
	pLast = nullptr;
}

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

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

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

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

template <typename T>
TNode<T>* TList<T>::search(const T& data) const {
	TNode<T>* curr = pFirst;
	while (curr != pStop && curr->data != data) {
		curr = curr->pNext;
	}
	if (curr == pStop) {
		throw ("Element not found!");
	}
	return curr;
}

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

template <typename T>
void TList<T>::insert_last(const T& data) {
	if (IsEmpty()) {
		insert_first(data);
		return;
	}
	TNode<T>* new_last = new TNode<T>(data, pStop);
	pLast->pNext = new_last;
	pLast = new_last;
	pCurr = new_last;
}

template <typename T>
void TList<T>::insert_before(const T& who, const T& where) {
```

```
	TNode<T>* pWhere = search(where);
	if (pWhere == pFirst) {
		insert_first(who);
		return;
	}
	TNode<T>* pPrev = pFirst;
	while (pPrev->pNext != pWhere) {
		pPrev = pPrev->pNext;
	}
	TNode<T>* new_node = new TNode<T>(who, pWhere);
	pPrev->pNext = new_node;
}

template <typename T>
void TList<T>::insert_after(const T& who, const T& where) {
	TNode<T>* pWhere = search(where);
	if (pWhere == pLast) {
		insert_last(who);
		return;
	}
	TNode<T>* new_node = new TNode<T>(who, pWhere->pNext);
	pWhere->pNext = new_node;
}

template <typename T>
void TList<T>::remove(const T& data_) {
	if (pFirst == nullptr) throw "List is empty!";
	TNode<T>* tmp = pFirst;
	TNode<T>* pPrev = nullptr;
	while (tmp != pStop && tmp->data != data_)
	{
		pPrev = tmp;
		tmp = tmp->pNext;
	}
	if (tmp == pFirst) {
		pFirst = pFirst->pNext;
		delete tmp;
		return;
	}
	if (tmp == pStop)throw "Data not found!";
	pPrev->pNext = tmp->pNext;
	delete tmp;
}

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

template <typename T>
void TList<T>::next() {
	if (pCurr == pStop) throw("List is ended");
	pCurr = pCurr->pNext;
}

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

```
	return size;
}

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

template <typename T>
void TList<T>::insert_sort(const T& data) {
	if (IsEmpty() || data < pFirst->data) {
		insert_first(data);
		return;
	}

	TNode<T>* prev = pFirst;
	TNode<T>* current = pFirst->pNext;

	while (current != pStop && current->data <= data) {
		prev = current;
		current = current->pNext;
	}

	if (current == pStop) {insert_last(data); }
	else {
		TNode<T>* newNode = new TNode<T>(data);
		prev->pNext = newNode;
		newNode->pNext = current;
	}
}

template <typename T>
const TList<T>& TList<T>::operator=(const TList<T>& other)
{
	if (this == &other){return *this;}
	clear();
	TNode<T>* otherCurr = other.pFirst;
	while (otherCurr != nullptr)
	{
		insert_last(otherCurr->data);
		otherCurr = otherCurr->pNext;
	}
	return *this;
}
```

Приложение Б. Реализация классаTHeadRingList

```
template <typename T>
THeadRingList<T>::THeadRingList() : TList<T>() {
    pHead = new TNode<T>();
    pStop = pHead;
}

template <typename T>
THeadRingList<T>::THeadRingList(const THeadRingList<T>& ringL) :
TList<T>(ringL) {
    pHead = new TNode<T>(ringL.pHead->data, pFirst);
    if (!ringL.IsEmpty()) {
        pLast->pNext = pHead;
    }
    pStop = pHead;
}
```

```
template <typename T>
THeadRingList<T>::~THeadRingList() {
    delete pHead;
}

template <typename T>
void THeadRingList<T>::insert_first(const T& data) {
    TList<T>::insert_first(data);
    pHead->pNext = pFirst;
    pStop = pHead;
    pLast->pNext = pHead;
}

template <typename T>
void THeadRingList<T>::remove(const T& data) {
    TNode<T>* curr = pFirst;
    if (curr->data == data) {
        if (curr->pNext == pHead) {
            delete curr;
            pFirst = nullptr;
            pCurr = nullptr;
            pLast = nullptr;
            pHead->pNext = nullptr;
            return;
        }
        pFirst = pFirst->pNext;
        pHead->pNext = pFirst;
        delete curr;
        return;
    }
    TList<T>::remove(data);
}
```

Приложение В. Реализация класса TMonom

```
TMonom::TMonom() : coeff(0.0), degree(-1) {};

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

TMonom::TMonom(double _coeff,  int _degree){
      if (_degree < 0 || _degree > 999){throw ("Degree must be from 0 to
999");}
      coeff = _coeff;
      degree = _degree;
}

bool TMonom::operator==(const TMonom& data) const {
      return  (degree == data.degree);
}

bool TMonom::operator!=(const TMonom& data) const {
      return !(*this == data);
}

bool TMonom::operator<(const TMonom& data) const {
      return (degree < data.degree);
}

bool TMonom::operator<=(const TMonom& data) const {
      return (degree <= data.degree);
```

```
}

TMonom TMonom::operator*(const TMonom& monom) const {
	if ((degree + monom.degree) <= 999 && (degree + monom.degree) >= 0) {
		return TMonom(coeff * monom.coeff, degree + monom.degree);
	}
	else {throw ("Degree must be from 0 to 999");}
}

TMonom TMonom::operator+(const TMonom& monom) const {
	return TMonom(coeff + monom.coeff, degree);
}
```

Приложение Г. Реализация класса TPolynom

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

TPolynom::TPolynom(const string& _name) {
	monoms = new THeadRingList<TMonom>();
	ParseMonoms(_name);
}

TPolynom::TPolynom(const THeadRingList<TMonom>* list) {
	monoms = new THeadRingList<TMonom>();
	TNode<TMonom>* current = list->GetCurrent();
	int n= list->GetSize();
	for (int i = 0; i < n; i++) {
		TMonom curr = current->data;
		if (curr.coeff != 0) {
			monoms->insert_sort(curr);
		}
		current = current->pNext;
	}
	similar();
	name = ToString();
}

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

TPolynom::~TPolynom() {
	if (monoms != nullptr) {
		delete monoms;
	}
}

void TPolynom::convert_string(string& str) const {
	str.erase(remove(str.begin(), str.end(), ' '), str.end());
	transform(str.begin(), str.end(), str.begin(), ::tolower);
}

string TPolynom::ToString() const {
	string str;
	TPolynom p(*this);
	if (p.monoms->IsEmpty()) {return "";}
	bool firstTerm = true;
	p.monoms->reset();
```

```
	if (p.monoms->GetCurrent()->data.coeff == 0&&p.monoms->GetCurrent()-
>data.degree == 0) {
		return "0";
	}
	while (!p.monoms->IsEnded()) {
		int deg = p.monoms->GetCurrent()->data.degree;
		double coeff = p.monoms->GetCurrent()->data.coeff;
		int x = deg / 100;
		int y = (deg % 100) / 10;
		int z = deg % 10;
			if (!firstTerm) {
				str += ((coeff > 0) ? "+" : "-");
			}
			else {
				if (coeff < 0) str += '-';
				firstTerm = false;
			}
			if (abs(coeff) != 1 || deg == 0) {
				char tmp[10];
				sprintf(tmp, "%.2f", abs(coeff));
				str += string(tmp);
			}
			string mul_symbol = ((abs(coeff) == 1) ? "" : "*");
			if (x != 0) {
				str += (mul_symbol + "x") + ((x != 1) ? "^" +
std::to_string(x) : "");
			}
			if (y != 0) {
				mul_symbol = (x == 0) ? mul_symbol : "*";
				str += (mul_symbol + "y") + ((y != 1) ? "^" +
std::to_string(y) : "");
			}
			if (z != 0) {
				mul_symbol = (x == 0 && y == 0) ? mul_symbol : "*";
				str += (mul_symbol + "z") + ((z != 1) ? "^" +
std::to_string(z) : "");
			}
		p.monoms->next();
	}
	return str;
}


void TPolynom::similar() {
	monoms->reset();
	while (!monoms->IsEnded() && !monoms->IsEmpty()) {
		TNode<TMonom>* current = monoms->GetCurrent();
		TNode<TMonom>* next = monoms->GetCurrent()->pNext;

		if (current->data == next->data) {
			next->data.coeff += current->data.coeff;
			if (next->data.coeff == 0)
			{
				monoms->next();
				monoms->remove(current->data);
				monoms->next();
				monoms->remove(next->data);
			}
			else {
				monoms->next();
				monoms->remove(current->data);
			}
		}
		else {
```

```
                monoms->next();
            }
        }
        if (monoms->IsEmpty()) { monoms->insert_first(TMonom(0, 0)); }
}

void TPolynom::ParseMonoms(const string& _name) {
        string str = _name;
        convert_string(str);
        if (_name == "0") {
            monoms->insert_first(TMonom(0, 0));
            return;
        }
        while (!str.empty()) {
            TMonom tmp;
            int degree = 0;
            size_t j = str.find_first_of("+-", 1);
            string monom = str.substr(0, j);
            str.erase(0, j);
            string coefficient = monom.substr(0, monom.find_first_of("xyz"));
            tmp.coeff = ((coefficient.empty() || coefficient == "+") ? 1 :
                (coefficient == "-") ? -1 : stod(coefficient));
            monom.erase(0, monom.find_first_of("xyz"));
            for (size_t i = 0; i < monom.size(); ++i) {
                if (isalpha(monom[i])) {
                    int exp = 1;
                    if (monom[i + 1] == '^') {
                        size_t exp_start = i + 2;
                        while (isdigit(monom[exp_start])) {
                            exp_start++;
                        }
                        exp = stoi(monom.substr(i + 2, exp_start - i -
2));
                    }
                    switch (monom[i]) {
                    case 'x':
                        degree += exp * 100;
                        break;
                    case 'y':
                        degree += exp * 10;
                        break;
                    case 'z':
                        degree += exp * 1;
                        break;
                    default:
                        throw ("Invalid monom format");
                        break;
                    }
                }
            }
            tmp.degree = degree;
            if (tmp.coeff != 0) {
                monoms->insert_sort(tmp);
            }
        }
        similar();
        name = ToString();
}

double TPolynom::operator()(double x,double y,double z) const {
        TArithmeticExpression expression(name);
        vector<double> xyz ={x,y,z};
        expression.ToPostfix();
        expression.SetValues(xyz);
```

```
	return (expression.Calculate());
}

const TPolynom& TPolynom::operator=(const TPolynom& p)  {
	if (this != &p) {
		name = p.name;
		delete monoms;
		monoms = new THeadRingList<TMonom>(*(p.monoms));
	}
	return (*this);
}

TPolynom TPolynom::operator+(const TPolynom& p){
	TPolynom result;
	monoms->reset();
	p.monoms->reset();
	if (name == "0.00" || p.name == "0.00") { return (name == "0.00") ? p :
(*this); }
	while (!monoms->IsEnded() && !p.monoms->IsEnded()){
		TMonom m1 = monoms->GetCurrent()->data;
		TMonom m2 = p.monoms->GetCurrent()->data;
		if (m1 == m2){
			TMonom m3 = m1 + m2;
			if (m3.coeff != 0) {result.monoms->insert_last(m3);}
			monoms->next();
			p.monoms->next();
		}
		else if (m2<m1){
			result.monoms->insert_last(m2);
			p.monoms->next();
		}
		else{
			result.monoms->insert_last(m1);
			monoms->next();
		}
	}
	while (!monoms->IsEnded()) {
		TMonom m1 = monoms->GetCurrent()->data;
		result.monoms->insert_last(m1);
		monoms->next();
	}
	while (!p.monoms->IsEnded()) {
		TMonom m2 = p.monoms->GetCurrent()->data;
		result.monoms->insert_last(m2);
		p.monoms->next();
	}
	if (result.monoms->IsEmpty()) { result.monoms->insert_first(TMonom(0,
0)); }
	result.name = result.ToString();
	return result;
}

TPolynom TPolynom::operator-(const TPolynom& p) {
	TPolynom result(*this);
	result = result+(-p);
	result.name = result.ToString();
	return result;
}

TPolynom TPolynom::operator-() const {
	TPolynom result(*this);
	result.monoms->reset();
	while (!result.monoms->IsEnded()) {
		result.monoms->GetCurrent()->data.coeff *= -1;
```

```
            result.monoms->next();
      }
      result.name = result.ToString();
      return result;
}

TPolynom TPolynom::operator*(const TPolynom& p) {
      TPolynom result;
      monoms->reset();
      bool not_null =false;
      while (!monoms->IsEnded()) {
            p.monoms->reset();
            while (!p.monoms->IsEnded()) {
                  TMonom m1 = monoms->GetCurrent()->data;
                  TMonom m2 = p.monoms->GetCurrent()->data;
                  TMonom m3 = m1 * m2;
                  if (m3.coeff != 0) {
                        result.monoms->insert_sort(m3);
                        not_null = true;
                  }
                  p.monoms->next();
            }
            monoms->next();
      }
      if (!not_null) {result.monoms->insert_first(TMonom(0, 0));}
      result.similar();
      result.name = result.ToString();
      return result;
}

TPolynom TPolynom::dx() const {
      TPolynom result;
      bool not_null = false;
      monoms->reset();
      while (!monoms->IsEnded()) {
            TMonom m = monoms->GetCurrent()->data;
            if (m.degree >= 100) {
                  int new_degree = m.degree - 100;
                  double new_coeff = m.coeff * (m.degree / 100);
                  TMonom new_monom(new_coeff, new_degree);
                  result.monoms->insert_last(new_monom);
                  not_null = true;
            }
            monoms->next();
      }
      if (!not_null) {result.monoms->insert_first(TMonom(0, 0));}
      result.name = result.ToString();
      return result;
}

TPolynom TPolynom::dy() const {
      TPolynom result;
      bool not_null = false;
      monoms->reset();
      while (!monoms->IsEnded()) {
            TMonom m = monoms->GetCurrent()->data;
            int deg = m.degree;
            int y = (deg % 100) / 10;
            if (y >= 1) {
                  int new_degree = m.degree - 10;
                  double new_coeff = m.coeff * y;
                  TMonom new_monom(new_coeff, new_degree);
                  result.monoms->insert_last(new_monom);
                  not_null = true;
```

```
            }
            monoms->next();
      }
      if (!not_null) {result.monoms->insert_first(TMonom(0, 0));}
      result.name = result.ToString();
      return result;
}

TPolynom TPolynom::dz() const {
      TPolynom result;
      monoms->reset();
      bool not_null = false;
      while (!monoms->IsEnded()) {
            TMonom m = monoms->GetCurrent()->data;
            int deg = m.degree;
            int z = deg % 10;
            if (z >= 1) {
                  int new_degree = m.degree - 1;
                  double new_coeff = m.coeff * z;
                  TMonom new_monom(new_coeff, new_degree);
                  result.monoms->insert_last(new_monom);
                  not_null = true;
            }
            monoms->next();
      }
      if (!not_null) {result.monoms->insert_first(TMonom(0, 0));}
      result.name=result.ToString();
      return result;
}

bool TPolynom::operator==(const TPolynom& p) const {
      monoms->reset();
      p.monoms->reset();
      while (!monoms->IsEnded() && !p.monoms->IsEnded()) {
            if (monoms->GetCurrent()->data != p.monoms->GetCurrent()->data) {
                  return false;
            }
            monoms->next();
            p.monoms->next();
      }
      return (monoms->IsEnded() && p.monoms->IsEnded());
}

bool TPolynom::operator!=(const TPolynom& p) const {
      return !(*this == p);
}

ostream& operator<<(ostream& out, const TPolynom& p) {
      out << p.ToString() << endl;
      return out;
}
```