МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

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

**(ННГУ)**

**Институт информационных технологий, математики и механики**

ЛАБОРАТОРНАЯ РАБОТА

на тему:

**«ПОЛИНОМЫ»**

**Выполнил(а):** студент(ка) группы 3822Б1ФИ1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Шпынов Н.А. /

Подпись

**Проверил:** к.т.н., доцент каф. ВВиСП

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Кустикова В.Д. /

Подпись

Нижний Новгород  
2023

**Оглавление**

[Введение 1](#_Toc3715)

[1 Постановка задачи 2](#_Toc15559)

[2 Руководство пользователя 3](#_Toc6449)

[2.1 Приложение для демонстрации работы стеков 3](#_Toc21185)

[2.2 Приложение для демонстрации работы постфиксной формы 4](#_Toc27738)

[3 Руководство программиста 6](#_Toc29860)

[3.1 Использованные алгоритмы 6](#_Toc9401)

[3.2 Описание классов 9](#_Toc8464)

[Заключение 15](#_Toc4085)

[Литература 16](#_Toc14963)

[Приложения 17](#_Toc11430)

# Введе**н**ие

Работу с полиномами можно существенно облегчить с помощью машины. Для реализации представления полиномиальных выражений в коде часто используется циклический список.

Связанный список в C++ - это линейная структура данных, где каждый элемент представляет собой отдельный объект, называемый узлом. Каждый узел состоит из двух частей: данных и ссылки на следующий узел в последовательности.

Циклический список - это связный список, в котором последний узел списка указывает на первый узел, создавая цикл. Это позволяет эффективно обходить список без необходимости доходить до конца и возвращаться к началу.

# Постановка задачи

Цель – реализовать представление полиномов и операций над ними с помощью циклического списка.

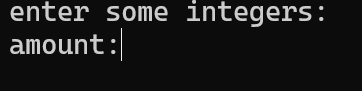
Задачи:

1. Разработка структуры данных связного списка.
2. Разработка структуры данных циклического списка на основе связного.
3. Разработка представления классов и структуры их наследования. Для представления полинома постфиксной форме, следует создать соответствующий класс, вспомогательные структуры и соответствующую систему наследования.
4. Определение операций. Разработать алгоритмы и функции для выполнения операций, таких как сложение полиномов, вычисление результата выражения и т.д.
5. Тестирование и отладка. Как и при любой разработке программного обеспечения, в конце необходимо провести тестирование и отладку кода. Следует обнаружить и исправить возможные ошибки и проблемы, а также убедиться в правильности функционирования.

# Руководство пользователя

## Приложение для демонстрации работы стеков

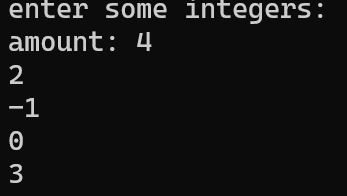
1. Запустить sample\_tlist.exe. В результате появится следующее окно (рис. 1):



1. Основное окно приложения

На этом шаге потребуется ввести количество элементов в будущем списке

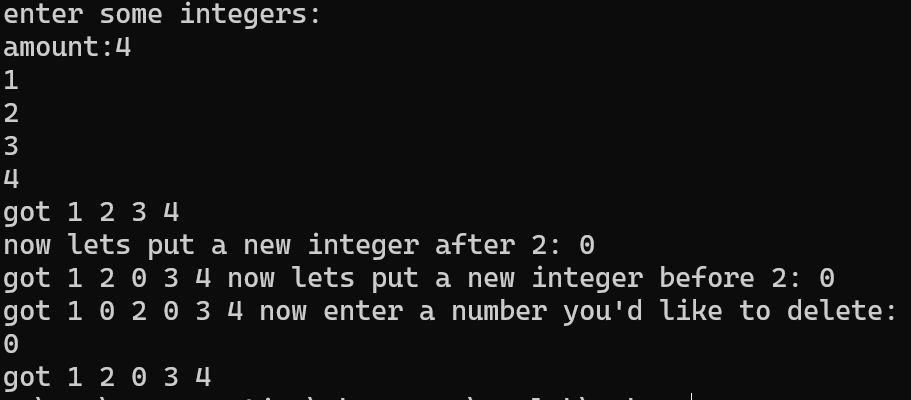
1. После ввода его размера, следует ввести свои числа (рис. 2):



1. Ввод длин

Вводить значения надо по порядку через пробел или Enter.

1. Программа выводит полученный список и попросит произвести несколько операций на списком, чтобы показать исправность своей работы (рис. 3):

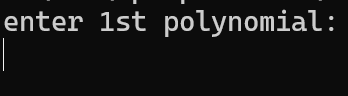


1. Вывод

Операции, которые запросит программа: добавление элемента после заданного, добавление перед ним, удаление элемента по выбору

## Приложение для демонстрации работы постфиксной формы

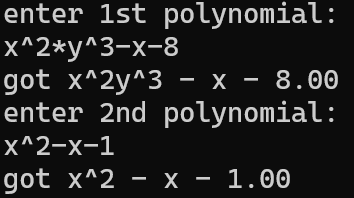
1. Запустить sample\_polynom.exe. В результате появится следующее окно (рис. 4):



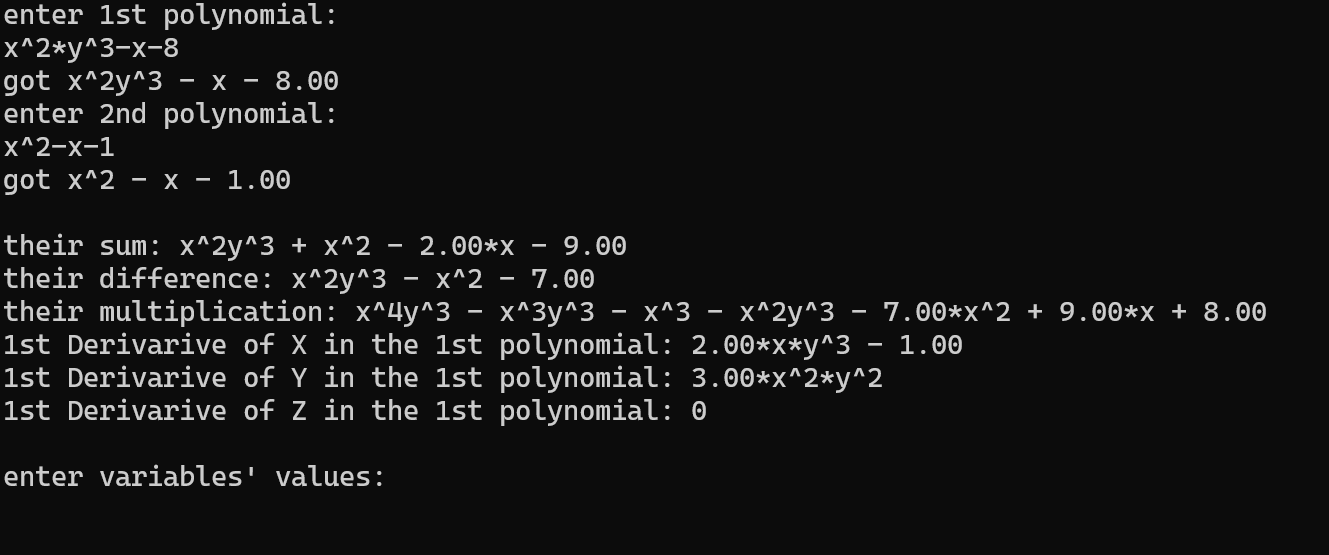
1. Основное окно приложения

Требуется ввести два полинома по очереди. Использовать можно только переменные x, y, z. Для обозначения степени переменной используется «^» (от 0 до 9)

1. После ввода выражений (рис. 5):



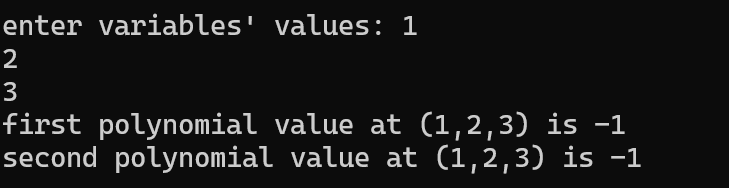
1. Ввод полиномов
2. Далее программа для проверки выведет примеры операций над полиномами и предложит посчитать их значения в конкретной точке (рис. 6):



1. Проверка

Переменные вводятся по очереди, от x до z.

1. Программа считает результат и выводит его (рис. 7).



1. Итог

# Руководство программиста

## Использованные алгоритмы

### Список

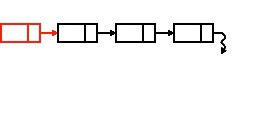
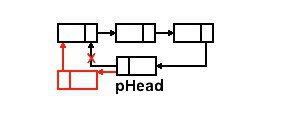
Связанный список в C++ - это линейная структура данных, где каждый элемент представляет собой отдельный объект, называемый узлом. Каждый узел состоит из двух частей: данных и ссылки на следующий узел в последовательности.

Циклический список - это связный список, в котором последний узел списка указывает на первый узел, создавая цикл. Это позволяет эффективно обходить список без необходимости доходить до конца и возвращаться к началу.

Реализованные функции (слева связный список, справа циклический):

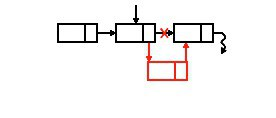
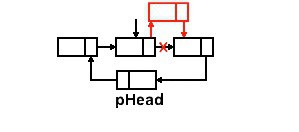
1. Добавление нового звена списка в начало

Добавляет новый элемент в начало списка, переназначая на него pFirst

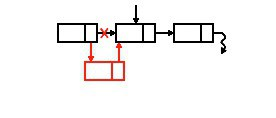
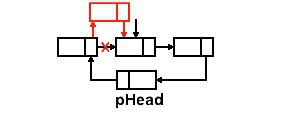
1. Добавление нового звена после заданного

Ищет заданный элемент и производит вставку после него

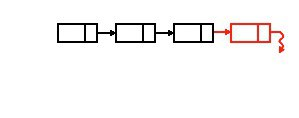
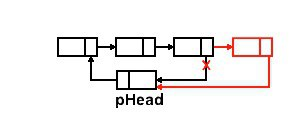
1. Добавление нового звена перед заданным:

Ищет заданный элемент и производит вставку после предыдущего ему элемента (т.е. перед заданным)

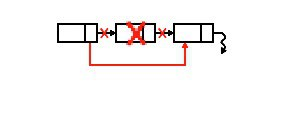
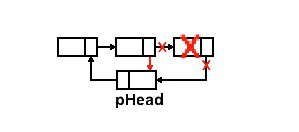
 

1. Добавление нового звена в конец:

Добавляет новый элемент в конец списка, переназначая на него pLast

1. Удаление звена из списка:

### Моном

Моном - выражение в математике, представляющее собой произведение переменных в любой степени и константы.

В программе каждый моном описан его коэффициентом перед переменными и трехзначным числом, характеризующим степени переменных

Реализованные функции:

1. Перевод значения монома в строку  
    представляет коэффициент монома и его степень в виде строки

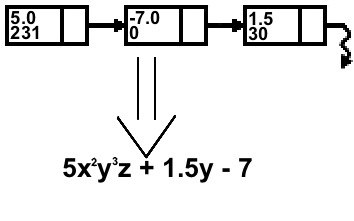
### Полином

Полином - выражение в математике, представляющее собой сумму мономов.

В программе полином представлен как циклический список из мономов.

Реализованные функции:

1. Перевод значения полинома в строку



1. Первая производная по переменной.

Три функции (для каждой переменной по одной), считающие первую производную от этого полинома.

1. Вставка монома в полином.

Получает на вход моном, определяет его место в полиноме и производит туда вставку.

## Описание классов

### Класс TNode

template <typename TData>class TNode {

public:

    TData data;

    TNode\* pNext;

    TNode() {

        data = TData();

        pNext = nullptr;

    }

    TNode(const TData& d, TNode\* pN = nullptr) {

        data = d;

        pNext = pN;

    }

    TNode(const TNode& Node) {

        data = Node.data;

        pNext = Node.pNext;

    }

};

*Поля:*

data – информация, лежащая в узле

pNext – указатель на следующий узел списка

*Конструкторы:*

**TNode()** – конструктор по умолчанию.

TNode(const TData& d, TNode\* pN = nullptr) – конструктор с параметром.

Назначение: создание узла с заданной информацией и указателем.

Входные данные: d – информация, pN – указатель.

TNode(const TNode& Node) – конструктор копирования.

Назначение: создание узла из заданного.

Входные данные: Node – узел.

### Класс TList

template <typename TData>class TList {

protected:

    TNode<TData>\* pFirst;

    TNode<TData>\* pLast;

    TNode<TData>\* pStop;

    TNode<TData>\* pPrev;

    TNode<TData>\* pCurrent;

public:

    TList() {

        pFirst = nullptr;

        pLast = nullptr;

        pCurrent = nullptr;

        pStop = nullptr;

        pPrev = nullptr;

    }

    TList(const TList<TData>&);

    TList(TNode<TData>\*);

    virtual ~TList() {clear();}

    virtual void InsertFirst(const TData&);

    void InsertLast(const TData&);

    void InsertBefore(const TData& after, const TData& ND);

    void InsertAfter(const TData& previous, const TData& ND);

    void DeleteNode(TNode<TData>\* node);

    void DeleteCurrentNode() {DeleteNode(pCurrent);}

    bool operator==(const TList<TData>& l) const;

    TNode<TData>\* GetCurrent() {return pCurrent;}

    bool IsEnded() const { return pCurrent == pStop; }

    bool IsEmpty() const {return pFirst == nullptr;}

    TNode<TData>\* Search(const TData& d);

    TNode<TData>\* Search(const TNode<TData>\* node);

    void next() {

        if ((pCurrent == pStop) ||(pCurrent == nullptr))

            throw;

        pPrev = pCurrent;

        pCurrent = pCurrent->pNext;

    }

    void reset() {

    pCurrent = pFirst;

    pPrev = nullptr;

    }

    virtual void clear();

};

*Поля*:

pFirst – указатель на первое звено списка.

pLast – указатель на последнее звено списка.

pStop – указатель на конец списка.

pPrev – указатель на звено, идущее перед текущим.

pCurrent – указатель на текущее звено.

*Конструкторы*:

**TList(const TList<TData>& l)** – конструктор копирования.

Назначение: создание элемента класса **TList**, равного данному.

Входные данные: l – список.

**TList(const TList<TData>& l)** – конструктор c параметром.

Назначение: создание элемента класса **TList** из узла.

Входные данные: l – узел.

*Методы*:

virtual void InsertFirst(const TData& ND) – вставка узла в начало списка.

Входные данные: **ND** – содержимое вставляемого узла.

Выходные данные: нет.

    void InsertLast(const TData& **ND**)– вставка в конец списка.

Входные данные: **ND** – содержимое вставляемого узла.

Выходные данные: нет.

    void InsertBefore(const TData& after, const TData& ND) – вставка перед заданным узлом.

Входные данные: **after** – узел, перед которым нужно произвести вставку, **ND** – содержимое вставляемого узла.

Выходные данные: нет.

    void InsertAfter(const TData& previous, const TData& ND)– вставка после заданного узла.

Входные данные: **previous** – узел, после которого нужно произвести вставку, **ND** – содержимое вставляемого узла.

Выходные данные: нет.

    void DeleteNode(TNode<TData>\* node) – удаление заданного узла.

Входные данные: **node** – заданный узел.

Выходные данные: нет.

    void DeleteCurrentNode() – удаление текущего узла.

Входные данные: нет.

Выходные данные: нет.

    TNode<TData>\* GetCurrent() – получение текущего элемента.

Входные данные: нет.

Выходные данные: текущий элемент.

    bool IsEnded() const – проверяет, дошел ли указатель на текущий элемент до конца списка.

Входные данные: нет.

Выходные данные: **true** если дошел, **false** если нет.

    bool IsEmpty() const – проверка списка на заполненность.

Входные данные: нет.

Выходные данные: **true** если пуст, **false** если нет.

    TNode<TData>\* Search(const TData& d) – поиск узла с заданным содержимым.

Входные данные: **d** – содержимое искомого узла.

Выходные данные: найденный узел.

    TNode<TData>\* Search(const TNode<TData>\* node) – поиск по списку. узла, одинакового с данным.

Входные данные: **node** – узел для поиска.

Выходные данные: науденный узел.

    void next()– переходит к следующему элементу.

Входные данные: нет.

Выходные данные: нет.

    void reset() – ставит указатель на текущий элемент в начало списка.

Входные данные: нет.

Выходные данные: нет.

**virtual void clear(**) – очищает список.

Входные данные: нет.

Выходные данные: нет.

*Операторы:*

**bool operator==(const TList<TData>& l) const** – оператор сравнения.

Входные данные: **l** – список, с которым идет сравнение.

Выходные данные: **true** если равны, **false** если нет.

### Класс THeadRingList

template <typename TData> class THeadRingList : public TList<TData> {

public:

    TNode<TData>\* pHead;

    THeadRingList() :TList<TData>() {

        pHead = new TNode<TData>();

        pHead->pNext = pHead;

        pStop = pHead;

    }

    THeadRingList(const THeadRingList<TData>& l) :TList<TData>(l) {

        pHead = new TNode<TData>(l.pHead->data, pFirst);

        if (!l.IsEmpty())

            pLast->pNext = pHead;

        pStop = pHead;

    }

    ~THeadRingList() { delete pHead;}

    void InsertFirst(const TData& d);

    void clear();

  const THeadRingList<TData>& operator=(const THeadRingList<TData>& l);

};

*Поля*:

**pHead**– указатель на заглавный узел.

*Конструкторы*:

**THeadRingList()** – конструктор по умолчанию.

Назначение: создание базового элемента класса **THeadRingList**.

Входные данные: l – список.

**THeadRingList(const THeadRingList<TData>& l)** – конструктор копирования.

Назначение: создание элемента класса **THeadRingList**, равного данному.

Входные данные: l – список.

*Методы*:

void InsertFirst(const TData& d) – вставка в начало списка.

Входные данные: нет.

Выходные данные: **d** – содержимое для вставляемого узла.

    void clear() – очищение списка

Входные данные: нет.

Выходные данные: нет.

*Операторы:*

**const THeadRingList<TData>& operator=(const THeadRingList<TData>& l)** – оператор присваивания.

Входные данные: **l** – исходный узел.

Выходные данные: возвращает новый узел.

### Класс TMonomial

class TMonomial {

public:

    double coef;

    short int degr;

    TMonomial() {

        coef = 0;

        degr = -1;

    };

    TMonomial(double d, short int i) {

        coef = d;

        degr = i;

    };

    bool operator<(const TMonomial& m) const {return degr < m.degr;};

    bool operator>(const TMonomial& m) const {return degr > m.degr;}

    bool operator<=(const TMonomial& m) const {return degr <= m.degr;}

    bool operator>=(const TMonomial& m) const {return degr >= m.degr;}

    bool operator==(const TMonomial& m) const { return degr == m.degr;}

    bool operator!=(const TMonomial& m) const { return degr != m.degr;}

    string ToString() const;

    friend ostream& operator<<(ostream& out, const TMonomial& m) {

        out << m.ToString();

        return out;

    }

};

*Поля*:

coef – коэффициент перед переменными.

degr – степени переменных.

*Конструкторы:*

**TMonomial()** – конструктор по умолчанию.

Назначение: создание базового элемента класса **TMonomial**.

Входные данные: нет.

**TMonomial(double d, short int i)** – конструктор с параметром.

Назначение: создание элемента класса **TMonomial** с заданными коэффициентом и степенями.

Входные данные: **d** – коэффициент, **i** – степени.

*Методы*:

**string ToString() const** – вывод данных из монома в качестве строки.

Входные данные: нет.

Выходные данные: строка с данными.

*Операторы:*

**bool operator<(const TMonomial& m) const**

**bool operator<(const TMonomial& m) const**

**bool operator<(const TMonomial& m) const**

**bool operator<(const TMonomial& m) const**

**bool operator<(const TMonomial& m) const**

**bool operator<(const TMonomial& m) const**

– семейство операторов сравнения.

Входные данные: **m** – моном, с которым идет сравнение.

Выходные данные: **true** если условие оператора выполнено, **false** если нет.

### Класс TPolynomial.

class TPolynomial {

private:

    void PutNewMonomialInPlace(TMonomial& m);

    void DividePolynomialName(const string& name);

protected:

    string name;

    THeadRingList<TMonomial> monomials;

public:

    TPolynomial() : monomials(){}

    TPolynomial(const string& n) : monomials() {

        name = n;

        DividePolynomialName(n);

    };

    TPolynomial(const THeadRingList<TMonomial>& l){ monomials = l;}

    TPolynomial(const TPolynomial& p){

        name = p.name;

        monomials = p.monomials;

    }

    string ToString() const;

    ~TPolynomial() {};

    bool operator==(const TPolynomial& p) const;

    TPolynomial operator+(const TPolynomial&);

    TPolynomial operator-(const TPolynomial&);

    TPolynomial operator\*(const TPolynomial&);

    TPolynomial operator\*(const double&);

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

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

    TPolynomial DerivativeX() const;

    TPolynomial DerivativeY() const;

    TPolynomial DerivativeZ() const;

    friend ostream& operator<<(ostream& out, TPolynomial& p) {

        out << p.ToString();

        return out;

    }

};

*Поля*:

name – собственное имя полинома.

monomials – список мономов полинома.

*Конструкторы:*

**TPolynomial()** – конструктор по умолчанию.

Назначение: создание базового элемента класса **TPolynomial**.

Входные данные: нет.

**TPolynomial(const string& n)** – конструктор с параметром.

Назначение: создание элемента класса **TPolynomial** из заданной строки.

Входные данные: **n** – строка.

TPolynomial(const THeadRingList<TMonomial>& l)– конструктор с параметром.

Назначение: создание элемента класса **TPolynomial** из заданного списка мономов.

Входные данные: **l** – список мономов.

TPolynomial(const TPolynomial& p)– конструктор копирования.

Назначение: создание элемента класса **TPolynomial** из заданного элемента класса **TPolynomial**.

Входные данные: **p** – полином.

*Методы*:

**void PutNewMonomialInPlace(TMonomial& m)** – вставка нового монома в полином.

Входные данные: **m** – моном для вставки.

Выходные данные: нет.

**void DividePolynomialName(const string& name)** – перевод строки в данные, хранящиеся в классе полинома.

Входные данные: **name** – строка, являющаяся полиномом.

Выходные данные: нет.

**string ToString() const** – перевод данных из полинома в строку.

Входные данные: нет.

Выходные данные: строка, являющаяся полиномом.

**TPolynomial DerivativeX() const**

**TPolynomial DerivativeY() const**

**TPolynomial DerivativeZ() const**

– семейство методов, расчитывающее производные соответствующих переменных от полинома.

Входные данные: нет.

Выходные данные: полином, являющийся производной по соответствующей переменной от исходного.

*Операторы:*

**bool operator==(const TPolynomial& p) const** – оператор сравнения.

Входные данные: **p** – полином для сравнения.

Выходные данные:**true** если полиномы равны, **false** если нет.

**TPolynomial operator+(const TPolynomial& p)** – оператор сложения.

Входные данные: **p** – полином для сложения.

Выходные данные:результат от сложения полиномов.

**TPolynomial operator-(const TPolynomial& p)** – оператор умножения.

Входные данные: **p** – полином для вычитания.

Выходные данные: результат от вычитания полинома из исходного.

**TPolynomial operator\*(const TPolynomial& p)** – оператор умножения.

Входные данные: **p** – полином для умножения.

Выходные данные: результат от перемножения полиномов.

**TPolynomial operator\*(const double& a)** – оператор умножения на константу.

Входные данные: **a** – константа

Выходные данные: результат от умножения полинома на константу.

**const TPolynomial& operator=(const TPolynomial& p)** – оператор присваивания.

Входные данные: **p** – полином на присванивание.

Выходные данные: новый полином.

**double operator()(const double x, const double y, const double z) const** – арифметический оператор.

Входные данные: **x, y, z** – значения соответствующих переменных.

Выходные данные: значение полинома в заданной точке.

friend ostream& operator<<(ostream& out, TPolynomial& p) – оператор вывода

Входные данные: out – поток вывода, p – полином для вывода.

Выходные данные: полином в потоке вывода.

# Заключение

Реализация работы с полиномами при помощи списков оказалась эффективным подходом, позволяющим удобно хранить, оперировать ими, считать производные и частные значения. Этот метод легко был создан на основе циклиеских, функционал которых оказался крайне полезным для выполнения задачи.

# Литература

1. Списки в c++

[https://pvs-studio.ru/ru/blog/terms/6684/#:~:text=%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA%20%E2%80%94%20%D1%8D%D1%82%D0%BE%20%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0%20%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85%2C%20%D1%85%D1%80%D0%B0%D0%BD%D1%8F%D1%89%D0%B0%D1%8F,%D1%8D%D0%BB%D0%B5%D0%BC%D0%B5%D0%BD%D1%82%D1%8B%20%D0%B2%20%D0%BB%D1%8E%D0%B1%D0%BE%D0%BC%20%D0%BC%D0%B5%D1%81%D1%82%D0%B5%20%D0%BF%D0%BE%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D0%B8.].

1. Алгоритм преобразования в постфиксную форму

[https://ru.stackoverflow.com/questions/465356/%D0%A6%D0%B8%D0%BA%D0%BB%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9-%D1%81%D0%BF%D0%B8%D1%81%D0%BE%D0%BA-%D0%B4%D0%BB%D1%8F-%D0%BF%D1%80%D0%B5%D0%B4%D1%81%D1%82%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F-%D0%BB%D1%8E%D0%B4%D0%B5%D0%B9]

1. Справочник по С++ [[https://prog-cpp.ru/cpp](https://prog-cpp.ru/cpp/)].
2. Справочник по С++ [[https://en.cppreference.com/w](https://en.cppreference.com/w/)].

# Приложения

Реализация методов из TList:

template <typename TData> TList<TData>::TList(const TList<TData>& l) {

    if (l.IsEmpty())

    {

        pFirst = nullptr;

        pLast = nullptr;

        pCurrent = nullptr;

        pStop = nullptr;

        return;

    }

    pFirst = new TNode<TData>(\*l.pFirst);

    pCurrent = pFirst;

    pStop = nullptr;

    TNode<TData>\* tmpNode = pFirst;

    TNode<TData>\* tmpNode1 = pFirst->pNext;

    while (tmpNode1 != nullptr) {

        tmpNode->pNext = new TNode<TData>(tmpNode1->data);

        tmpNode = tmpNode->pNext;

        tmpNode1 = tmpNode1->pNext;

    }

    pLast = tmpNode;

}

template <typename TData> TList<TData>::TList(TNode<TData>\* d) {

    pFirst = d;

    TNode<TData>\* tmpNode = d;

    while (tmpNode->pNext != nullptr) {

        tmpNode = tmpNode->pNext;

    }

    pLast = tmpNode;

    pCurrent = pFirst;

    pStop = nullptr;

}

template <typename TData> TNode<TData>\* TList<TData>::Search(const TData& d) {

    reset();

    while ((pCurrent != pStop) && (pCurrent->data != d))

        next();

    return (pCurrent == pStop) ? nullptr : pCurrent;

}

template <typename TData> TNode<TData>\* TList<TData>::Search(const TNode<TData>\* node) {

    return Search(node->data);

}

template <typename TData> void TList<TData>::InsertFirst(const TData& d) {

    TNode<TData>\* nFirst = new TNode<TData>(d, pFirst);

    pFirst = nFirst;

    if (pLast == nullptr) {

        pLast = pFirst;

    }

    pCurrent = pFirst;

}

template <typename TData> void TList<TData>::InsertLast(const TData& d) {

    if (IsEmpty()) {

        InsertFirst(d);

        return;

    }

    TNode<TData>\* nLast = new TNode<TData>(d, pStop);

    pLast->pNext = nLast;

    pLast = nLast;

    pCurrent = nLast;

}

template <typename TData> void TList<TData>::InsertBefore(const TData& after, const TData& ND) {

    TNode<TData>\* TNow = Search(after);

    if (TNow == nullptr) {

        throw std::exception("Element not found");

    }

    pPrev->pNext = new TNode<TData>(ND, TNow);

    return;

}

template <typename TData> void TList<TData>::InsertAfter(const TData& prev, const TData& ND) {

    TNode<TData>\* TNow = Search(prev);

    if (TNow->pNext == pStop) {

        throw std::exception("Element not found");

    }

    TNode<TData>\* nNode = new TNode<TData>(ND, TNow->pNext);

    TNow->pNext = nNode;

}

template <typename TData> void TList<TData>::DeleteNode(TNode<TData>\* node) {

    if (pFirst == nullptr)

        throw std::exception("Element not found");

    if (pFirst == node)

    {

        pFirst = pFirst->pNext;

        delete node;

        return;

    }

    Search(node);

    pPrev->pNext = pCurrent->pNext;

    if (pCurrent->pNext == pStop) {

        pLast = pPrev;

    }

    delete pCurrent;

    pCurrent = pPrev->pNext;

}

template <typename TData> void TList<TData>::clear() {

    if (pFirst == nullptr)

        return;

    TNode<TData>\* TNow = pFirst;

    TNode<TData>\* TNext = pFirst->pNext;

    while (TNext != pStop) {

        delete TNow;

        TNow = TNext;

        TNext = TNow->pNext;

    }

    pFirst = nullptr;

    pLast = nullptr;

    pCurrent = nullptr;

    pStop = nullptr;

}

template <typename TData> bool TList<TData>::operator==(const TList<TData>& l) const {

    TList<TData> tmpThis = TList<TData>(\*this);

    TList<TData> tmpL = TList<TData>(l);

    tmpThis.reset();

    tmpL.reset();

    while ((!tmpThis.IsEnded()) && (!tmpL.IsEnded())) {

        if (tmpThis.GetCurrent()->data != tmpL.GetCurrent()->data)

            return false;

        tmpThis.next();

        tmpL.next();

    }

    if (!tmpThis.IsEnded() || !tmpL.IsEnded())

        return false;

    return true;

}

Реализация методов из THeadRingList:

template <typename TData> void THeadRingList<TData>::InsertFirst(const TData& d){

    TList<TData>::InsertFirst(d);

    pLast->pNext = pHead;

    pHead->pNext = pFirst;

}

template <typename TData>

const THeadRingList<TData>& THeadRingList<TData>::operator=(const THeadRingList<TData>& l) {

    if (this == &l)

        return \*this;

    pFirst = new TNode<TData>(\*l.pFirst);

    pHead = new TNode<TData>();

    pHead->pNext = pFirst;

    pStop = pHead;

    pCurrent = pFirst;

    TNode<TData>\* tmp = l.pFirst->pNext;

    while (tmp != l.pStop) {

        pCurrent->pNext = new TNode<TData>(\*tmp);

        pCurrent = pCurrent->pNext;

        tmp = tmp->pNext;

    }

    pCurrent->pNext = pStop;

    pLast = pCurrent;

    pCurrent = pFirst;

    return (\*this);

}

template <typename TData> void THeadRingList<TData>::clear() {

    TList<TData>::clear();

    pHead->pNext = pHead;

    pStop = pHead;

}

Реализация методов из TMonomial:

string TMonomial::ToString() const {

    stringstream S;

    double tmp = coef;

    if (tmp == 0) {

        S << fixed << setprecision(2) << tmp;

        return S.str();

    }

    if (tmp < 0)

    {

        tmp \*= -1;

    }

    int x = degr / 100;

    int y = (degr % 100) / 10;

    int z = degr % 10;

    if ((tmp != 1) && (degr != 0)) {

        S << fixed << setprecision(2) << tmp;

        if ((x != 0) && (x != 1)) {

            S<<"\*x^";

            S<<(to\_string(x));

        }

        else if (x == 1) S << "\*x";

        if ((y != 0) && (y != 1)) {

            S << ("\*y^");

            S << (to\_string(y));

        }

        else if (y == 1) S << "\*y";

        if ((z != 0) && (z != 1)) {

            S << ("\*z^");

            S << (to\_string(z));

        }

        else if (z == 1) S << "\*z";

        return S.str();

    }

    else if (degr != 0) {

        bool tmp1 = false;

        if (tmp1)

            S << "\*";

        if ((x != 0) && (x != 1)) {

            S<<"x^";

            S<<(to\_string(x));

            tmp1 = true;

        }

        else if (x == 1) {

            S << "x";

            tmp1 = true;

        }

        if ((y != 0) && (y != 1)) {

            S << "y^";

            S<<(to\_string(y));

            tmp1 = true;

        }

        else if (y == 1) {

            S << "y";

            tmp1 = true;

        }

        if ((z != 0) && (z != 1)) {

            S << "z^";

            S << to\_string(z);

            tmp1 = true;

        }

        else if (z == 1) {

            S << "z";

            tmp1 = true;

        }

    }

    else if (degr == 0) {

        S << fixed << setprecision(2) << coef;

    }

    return S.str();

}

Реализация методов из TPolynomial:

bool TPolynomial::operator==(const TPolynomial& p) const {

    TPolynomial tmpThis(\*this);

    tmpThis.monomials.reset();

    TPolynomial tmpP(p);

    tmpP.monomials.reset();

    while (!tmpP.monomials.IsEnded() || !tmpThis.monomials.IsEnded()) {

        if ((!tmpP.monomials.IsEnded() && tmpThis.monomials.IsEnded()) || (tmpP.monomials.IsEnded() && !tmpThis.monomials.IsEnded()))

            return false;

        if (tmpP.monomials.GetCurrent()->data != tmpThis.monomials.GetCurrent()->data)

            return false;

        tmpP.monomials.next();

        tmpThis.monomials.next();

    }

    return true;

}

string TPolynomial::ToString() const {

    string S;

    TPolynomial p(\*this);

    p.monomials.reset();

    if (p.monomials.IsEmpty())

        return "0";

    if ((p.monomials.GetCurrent()->data.coef == 0) && (p.monomials.GetCurrent()->data.degr == 0))

        S.push\_back('0');

    else if ((p.monomials.GetCurrent()->data.coef == 0) && (p.monomials.GetCurrent()->data.degr != 0)) {}

    else if ((p.monomials.GetCurrent()->data.coef < 0) && (p.monomials.GetCurrent()->data.degr == 0)) {

        S.append("-");

        string tmp;

        tmp = p.monomials.GetCurrent()->data.ToString();

        tmp.erase(0, 1);

        S.append(tmp);

    }else if ((p.monomials.GetCurrent()->data.coef < 0) && (p.monomials.GetCurrent()->data.degr != 0)) {

        S.append("-");

        S.append(p.monomials.GetCurrent()->data.ToString());

    }

    else

        S.append(p.monomials.GetCurrent()->data.ToString());

    try { p.monomials.next(); }

    catch (...) { return S; }

    int k = -1;

    while (!p.monomials.IsEnded()) {

        k++;

        if ((p.monomials.GetCurrent()->data.coef < 0)&&(p.monomials.GetCurrent()->data.degr == 0)) {

            S.append(" - ");

            string tmp;

            tmp = p.monomials.GetCurrent()->data.ToString();

            tmp.erase(0, 1);

            S.append(tmp);

            p.monomials.next();

            continue;

        }

        else if ((p.monomials.GetCurrent()->data.coef < 0) && (p.monomials.GetCurrent()->data.degr != 0)) {

            S.append(" - ");

            S.append(p.monomials.GetCurrent()->data.ToString());

            p.monomials.next();

            continue;

        }

        else if (p.monomials.GetCurrent()->data.coef > 0)

            S.append(" + ");

        else if ((p.monomials.GetCurrent()->data.coef == 0) && (k == 0))

            return S;

        else if ((p.monomials.GetCurrent()->data.coef == 0) && (k != 0)) {

            p.monomials.next();

            continue;

        }

        S.append(p.monomials.GetCurrent()->data.ToString());

        p.monomials.next();

    }

    return S;

}

void TPolynomial::PutNewMonomialInPlace(TMonomial& m) {

    monomials.reset();

    if (monomials.IsEmpty()) {

        monomials.InsertFirst(m);

        return;

    }

    if ((monomials.GetCurrent()->data.coef == 0) &&(monomials.GetCurrent()->data.degr == 0)){

        monomials.clear();

        monomials.InsertFirst(m);

        return;

    }

    if (monomials.GetCurrent()->data < m) {

        monomials.InsertFirst(m);

        return;

    }

    while ((monomials.GetCurrent()->data > m) && (!monomials.IsEnded()))

        monomials.next();

    if (monomials.GetCurrent()->data.degr == m.degr) {

        monomials.GetCurrent()->data.coef += m.coef;

        if (monomials.GetCurrent()->data.coef == 0) {

            monomials.DeleteCurrentNode();

        }

        return;

    }

    if (!monomials.IsEnded())

        monomials.InsertBefore(monomials.GetCurrent()->data, m);

    else

        monomials.InsertLast(m);

}

TPolynomial TPolynomial::operator+(const TPolynomial& p) {

    TPolynomial result(\*this);

    TPolynomial tmp(p);

    tmp.monomials.reset();

    while (!tmp.monomials.IsEnded() && (tmp.monomials.GetCurrent() != nullptr))

    {

        result.PutNewMonomialInPlace(tmp.monomials.GetCurrent()->data);

        tmp.monomials.next();

    }

    return result;

}

TPolynomial TPolynomial::operator\*(const double& b) {

    TPolynomial tmp(\*this);

    tmp.monomials.reset();

    while (!tmp.monomials.IsEnded())

    {

        try {

            tmp.monomials.GetCurrent()->data.coef \*= b;

            tmp.monomials.next();

        }

        catch (...) {

            continue;

        }

    }

    return tmp;

}

TPolynomial TPolynomial::operator-(const TPolynomial& p) {

    TPolynomial tmpNeg(p);

    return this->operator+(tmpNeg \* (-1));

}

TPolynomial TPolynomial::operator\*(const TPolynomial& p) {

    TPolynomial result("0");

    TPolynomial tmpThis(\*this);

    TPolynomial tmpP(p);

    tmpThis.monomials.reset();

    while (!tmpThis.monomials.IsEnded()) {

        tmpP.monomials.reset();

        while (!tmpP.monomials.IsEnded()) {

            TMonomial tmpMonomial = tmpThis.monomials.GetCurrent()->data;

            TMonomial tmpMonomial1 = tmpP.monomials.GetCurrent()->data;

            double Coef = tmpMonomial.coef \* tmpMonomial1.coef;

            short int Degr = tmpMonomial.degr + tmpMonomial1.degr;

            if (Degr > 999)

                throw;

            result.PutNewMonomialInPlace(TMonomial(Coef, Degr));

            tmpP.monomials.next();

        }

        tmpThis.monomials.next();

    }

    return result;

}

const TPolynomial& TPolynomial::operator=(const TPolynomial& p) {

    if (this == &p)

    {

        return \*this;

    }

    name = p.name;

    monomials = p.monomials;

    return (\*this);

}

double TPolynomial::operator()(const double x, const double y, const double z) const {

    TPolynomial tmp(\*this);

    double result = 0;

    tmp.monomials.reset();

    while (!tmp.monomials.IsEnded()) {

        double i = tmp.monomials.GetCurrent()->data.coef;

        i \*= pow(x, tmp.monomials.GetCurrent()->data.degr / 100);

        i \*= pow(y, tmp.monomials.GetCurrent()->data.degr / 10 % 10);

        i \*= pow(z, tmp.monomials.GetCurrent()->data.degr % 10);

        result += i;

        tmp.monomials.next();

    }

    return result;

}

TPolynomial TPolynomial::DerivativeX() const {

    TPolynomial tmp(\*this);

    tmp.monomials.reset();

    TPolynomial tmpdx = TPolynomial("0");

    while (!tmp.monomials.IsEnded()) {

        if (tmp.monomials.GetCurrent()->data.degr / 100 != 0) {

            double Coef = tmp.monomials.GetCurrent()->data.coef \* (tmp.monomials.GetCurrent()->data.degr / 100);

            int16\_t Degr = tmp.monomials.GetCurrent()->data.degr - 100;

            tmpdx.PutNewMonomialInPlace(TMonomial(Coef, Degr));

        }

        tmp.monomials.next();

    }

    tmpdx.name = tmpdx.ToString();

    return tmpdx;

}

TPolynomial TPolynomial::DerivativeY() const {

    TPolynomial tmp(\*this);

    tmp.monomials.reset();

    TPolynomial tmpdx = TPolynomial("0");

    while (!tmp.monomials.IsEnded()) {

        if (tmp.monomials.GetCurrent()->data.degr / 10 % 10 != 0) {

            double Coef = tmp.monomials.GetCurrent()->data.coef \* (tmp.monomials.GetCurrent()->data.degr / 10 % 10);

            int16\_t Degr = tmp.monomials.GetCurrent()->data.degr - 10;

            tmpdx.PutNewMonomialInPlace(TMonomial(Coef, Degr));

        }

        tmp.monomials.next();

    }

    return tmpdx;

}

TPolynomial TPolynomial::DerivativeZ() const {

    TPolynomial tmp(\*this);

    tmp.monomials.reset();

    TPolynomial tmpdx = TPolynomial("0");

    while (!tmp.monomials.IsEnded()) {

        if (tmp.monomials.GetCurrent()->data.degr % 10 != 0) {

            double Coef = tmp.monomials.GetCurrent()->data.coef \* (tmp.monomials.GetCurrent()->data.degr % 10);

            int16\_t Degr = tmp.monomials.GetCurrent()->data.degr - 1;

            tmpdx.PutNewMonomialInPlace(TMonomial(Coef, Degr));

        }

        tmp.monomials.next();

    }

    return tmpdx;

}

void TPolynomial::DividePolynomialName(const string& str1) {

    if (str1 == "") {

        this->PutNewMonomialInPlace(TMonomial(0, 0));

        return;

    }

    string str;

    for (int i = 0; i < str1.length(); i++) {

        if (str1[i] != ' ')

            str.push\_back(str1[i]);

    }

    if (str.length() == 1)

    {

        this->PutNewMonomialInPlace(TMonomial(str[0]-48, 0));

        return;

    }

    string tmpMonoms[30];

    if ((str[0] != '-') && (str[0] != '+'))

        str = "+" + str;

    for (int i = 0, k = -1; i < str.length(); i++) {

        if ((str[i] != '-') && (str[i] != '+')) {

            tmpMonoms[k] += str[i];

        }

        else

            k++;

    }

    string signs;

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

        if (str[i] == '+')

            signs += "+";

        else if (str[i] == '-')

            signs += "-";

    }

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

        if (signs[i] == '-')

            tmpMonoms[i] = "-" + tmpMonoms[i];

    }

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

        short degr = 0;

        double coef = 1.0;

        int t = 0;

        if (tmpMonoms[i][t] == '-') {

            coef \*= -1;

            t++;

        }

        bool wasChanged = false;

        while (((int)(tmpMonoms[i][t]) - 48 >= 0) && ((int)(tmpMonoms[i][t]) - 48 <= 9)) {

            if (((coef == 1) || (coef == -1)) && (!wasChanged)) {

                coef \*= (int)(tmpMonoms[i][t]) - 48;

                wasChanged = true;

            }

            else if (coef >= 0) {

                coef = coef \* 10 + ((int)(tmpMonoms[i][t]) - 48);

                wasChanged = true;

            }

            else {

                coef = coef \* 10 - ((int)(tmpMonoms[i][t]) - 48);

                wasChanged = true;

            }

            t++;

        }

        for (int n = 0; n < tmpMonoms[i].size(); n++) {

            if ((tmpMonoms[i][n] == 'x') && (tmpMonoms[i][n + 1] == '^'))

                degr += (int(tmpMonoms[i][n + 2]) - 48) \* 100;

            if ((tmpMonoms[i][n] == 'x') && (tmpMonoms[i][n + 1] != '^'))

                degr += 100;

            if ((tmpMonoms[i][n] == 'y') && (tmpMonoms[i][n + 1] == '^'))

                degr += (int(tmpMonoms[i][n + 2]) - 48) \* 10;

            if ((tmpMonoms[i][n] == 'y') && (tmpMonoms[i][n + 1] != '^'))

                degr += 10;

            if ((tmpMonoms[i][n] == 'z') && (tmpMonoms[i][n + 1] == '^'))

                degr += (int(tmpMonoms[i][n + 2]) - 48);

            if ((tmpMonoms[i][n] == 'z') && (tmpMonoms[i][n + 1] != '^'))

                degr += 1;

        }

        this->PutNewMonomialInPlace(TMonomial(coef, degr));

    }

}