Министерство науки и высшего образования РФ
Федеральное государственное автономное образовательное учреждение
высшего образования «Национальный исследовательский Нижегородский
государственный университет им. Н.И. Лобачевского»
Институт информационных технологий, математики и механики

Отчет по учебной практике
АНАЛИТИЧЕСКИЕ ПРЕОБРАЗОВАНИЯ ПОЛИНОМОВ ОТ НЕСКОЛЬКИХ ПЕРЕМЕННЫХ (СПИСКИ)

Выполнил: Власов Максим Сергеевич, студент группы 381806-1
Проверил: к. т. н., доцент кафедры МОСТ Кустикова В. Д.

Нижний Новгород
2020

Содержание

Введение

В повседневной жизни человеку постоянно необходимо выполнять вычисления различных арифметических выражений разного уровня сложности как в прикладных, так и в научных целях. Для того чтобы избежать ошибок, можно поручить это компьютеру с помощью специальных приложений.

Таким образом, для этого и было создано консольное приложение, работающее с полиномами от нескольких переменных (с некоторыми ограничениями). При этом используется динамическая структура данных Список, на практическое освоение которой и направлена данная лабораторная работа.

1. Постановка задачи

Задача: разработать и реализовать приложение, выполняющее алгебраические операции (сложение, вычитание, умножение) над полиномами от трех переменных (степени от 0 до 9).

Входные данные: полиномы, операции (в виде строки).

Выходные данные: полиномы (в виде строки).

2. Руководство пользователя

В данном руководстве содержатся пошаговые инструкции по работе с программой, для того чтобы вы могли как можно быстрее приступить к использованию приложения.

1. Запустите файл **04_Polynomial.exe** из папки с программой. Перед вами отобразится приветственный экран с предложением выбрать режим работы. Введите 0, если хотите запустить автоматическое тестирование, или 1, чтобы перейти в режим ручного ввода, и нажмите Enter.

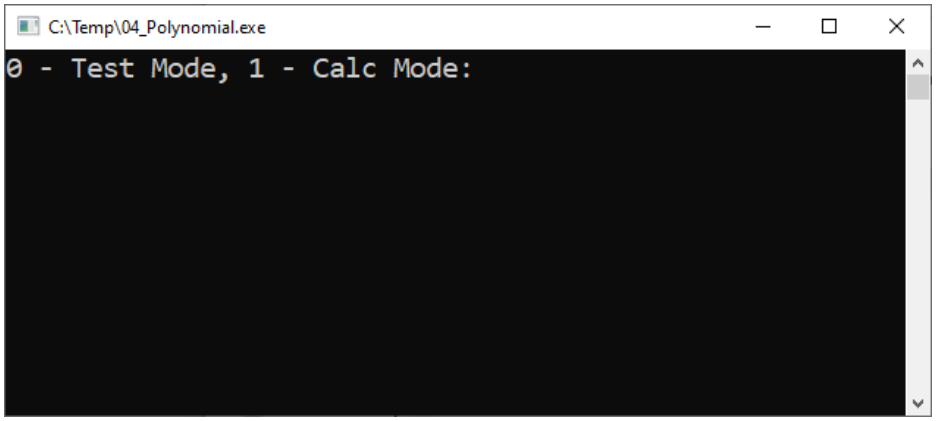

Рис 1. Программа после запуска.

2. Если вы выбрали режим автоматического тестирования, то программа сгенерирует несколько полиномов, моном и число и выполнит различные действия над ними, а результат будет выведен на экран (см. Рис 2). Вы также можете ввести строковое представление полинома с клавиатуры и нажать Enter, чтобы проверить работу алгоритма преобразования строки. Правила ввода следующие: знак * (между коэффициентом и переменными) не опускать, степени вводить не меньше 1 и не больше 9 (если при какой-либо переменной нужно ввести степень 0, то пропустите эту переменную). Чтобы сменить режим, перезапустите программу.
3. После выбора режима ручного ввода (см. Рис 3) программа предложит вам ввести полиномы. В случае неверного ввода полиномов (см. Рис 4) вы увидите сообщение об ошибке с предложением повторить ввод.
4. Если все введено правильно, программа выведет результаты выполнения операций (см. Рис 5) (включая, возможно, сообщение об ошибке, если при вычислении степени при переменных превысили 9, см. Рис 6). Нажмите Enter для выхода из программы.

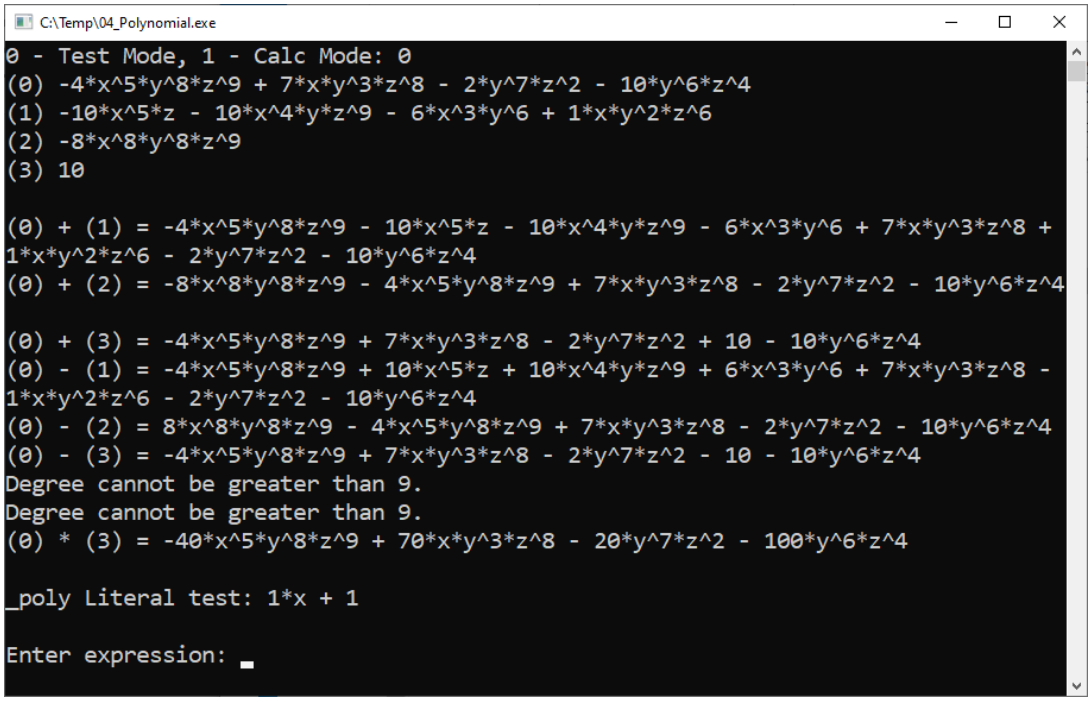

Рис 2. Программа в режиме автоматического тестирования.

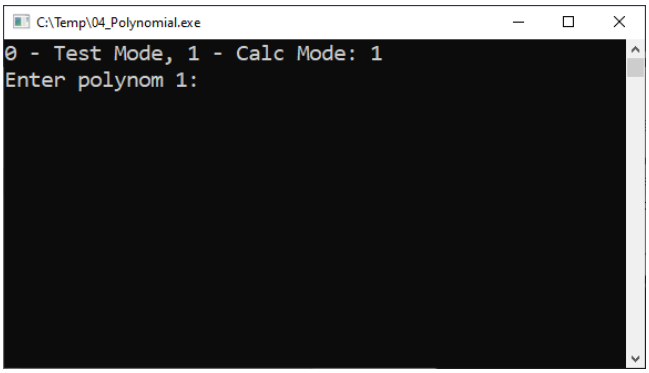

Рис 3. Программа в режиме ручного ввода.

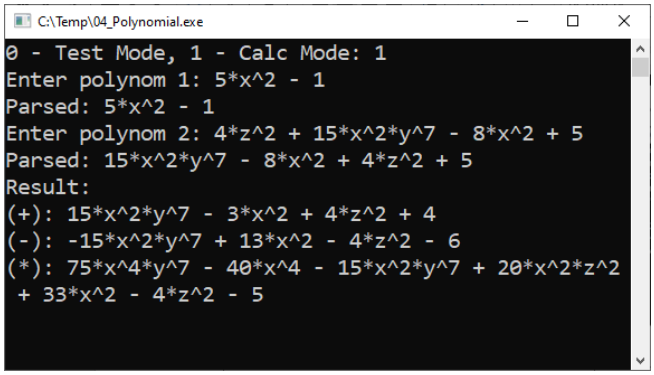

Рис 5. Отображение результатов вычислений.

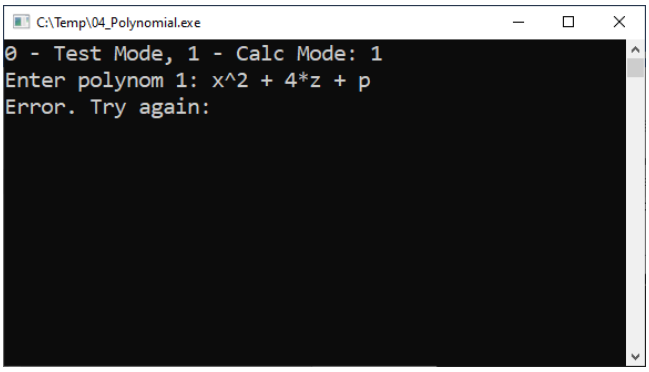

Рис 4. Сообщение об ошибке при вводе некорректного выражения (полинома).

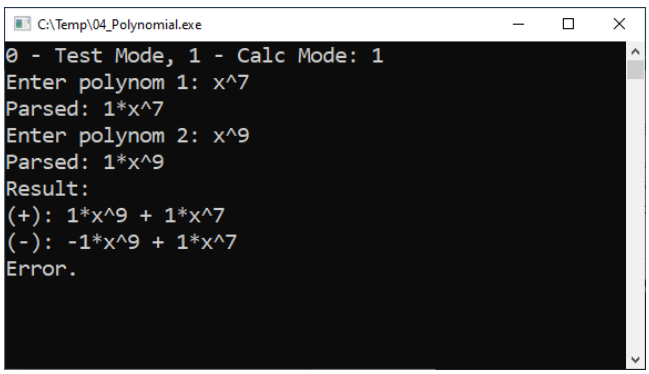

Рис 6. Сообщение об ошибке при переполнениии степени.

3. Руководство программиста

3.1. Структура программы

Исходный код программы содержится в следующих файлах:

1. **04_Polynomial.cpp** – основной модуль (консольное приложение, использующее модули ниже).
2. **TList.h** – модуль Список (содержит объявление и реализацию класса для работы со структурой данных Список, включая классы узлов, итераторов и исключений).
3. **TMonomial.h** – модуль Моном (содержит объявление и реализацию специализации узла списка, используемого в модуле выше, включая специальные классы исключений).
4. **TPolynomial.h** – модуль Полином (содержит объявление класса для работы с полиномами от нескольких переменных с некоторыми ограничениями на степени, включая классы исключений).
5. **TPolynomial.cpp** – модуль Полином (содержит реализацию методов класса для работы с полиномами).

3.2. Описание алгоритма

3.2.1. Структура данных Список

Линейный односвязный список – базовая динамическая структура данных, состоящая из узлов, каждый из которых содержит как собственно данные, так и указатель на следующий узел списка. Принципиальным преимуществом перед массивом является структурная гибкость: порядок элементов связного списка может не совпадать с порядком расположения элементов данных в памяти компьютера, а порядок обхода списка всегда явно задаётся его внутренними связями.

Для работы со списками предлагается реализовать следующие операции:

- методы проверки на пустоту и проверки на полноту списка;
- методы навигации по списку (итератор);
- методы вставки в начало, в конец, после звена с заданным ключи и перед звеном с заданным ключом;
- методы удаления звена с заданным ключом и поиск звена с заданным ключом.

3.2.1.1. Вставка звена в начало

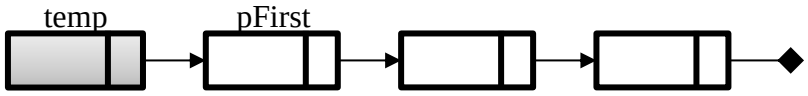

Рис 7. Вставка звена в начало.

1. Создать новое звено temp.
2. Установить указатель на следующее звено у temp на pFirst.
3. Поменять указатель на начало списка pFisrt на temp.

3.2.1.2. Вставка звена в конец

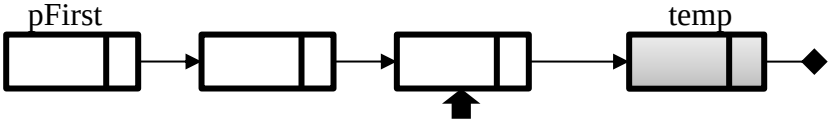

Рис 8. Вставка звена temp в конец.

1. Создать новое звено temp.
2. По списку дойти до последнего звена.
3. Установить указатель на следующее звено у последнего на temp.
4. Установить указатель на следующее звено у temp на nullptr.

3.2.1.3. Вставка звена после заданного звена

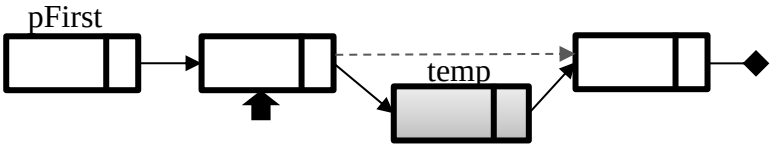

Рис 9. Вставка звена temp после заданного звена.

1. По списку дойти до заданного звена.
2. Создать новое звено temp.
3. Установить указатель на следующее звено у temp на звено, следующее за заданным.
4. Установить указатель на следующее звено у заданного звена на temp.

3.2.1.4. Вставка звена перед заданным звеном

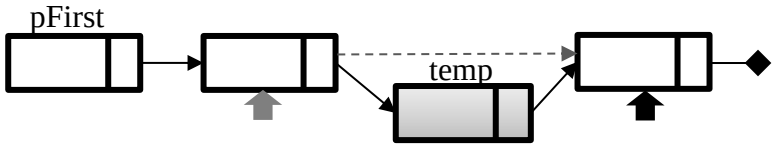

Рис 10. Вставка звена temp перед заданным звеном.

1. По списку дойти до звена, предшествующего заданному.
2. Создать новое звено temp.
3. Установить указатель на следующее звено у temp на заданное звено.
4. Установить указатель на следующее звено у звена, предшествующего заданному, на temp.

3.2.1.5. Удаление заданного звена

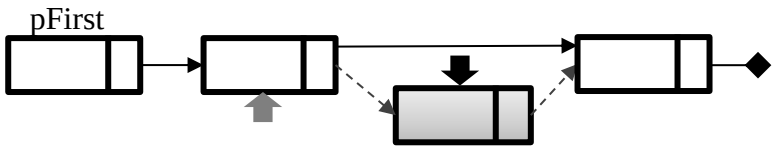

Рис 11. Удаление заданного звена.

1. По списку дойти до звена, предшествующего заданному.
2. Установить указатель на следующее звено у звена, предшествующего заданному, на следующее за заданным звено, а заданное звено удалить.

3.2.2. Полином

Полином можно определить также как выражение из нескольких термов, соединенных знаками сложения или вычитания. Терм включает коэффициент и моном, содержащий одну или несколько переменных, каждая из которых может иметь степень. При выполнении данной лабораторной работы предполагается, что степени переменных в мономах могут принимать значения 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Для организации быстрого доступа может быть использовано упорядоченное хранение мономов. Для задания порядка следования можно принять лексикографическое упорядочивание по степеням переменных, при котором мономы упорядочиваются по степеням первой переменной, потом по второй переменной, и только затем по третьей переменной. В общем виде это правило можно записать как соотношение: моном $X^{A1}Y^{B1}Z^{C1}$ предшествует моному $X^{A2}Y^{B2}Z^{C2}$ тогда и только тогда, если

(A1 > A2) ∨ (A1 = A2) & (B1 > B2) ∨ (A1 = A2) & (B1 = B2) & (C1 > C2).

Проверка лексикографического порядка занимает сравнительно много времени. Ее можно существенно упростить при помощи свернутой степени (индекса) монома, образуемой с использованием позиционной системы счисления: для монома со степенями (A, B, C) ставится в соответствие величина

ABC = A * 100 + B * 10 + C.

Данное соответствие является взаимно-однозначным. Обратное соответствие определяется при помощи выражений

A= [ABC / 100], B= [ABC % 100 / 10], C=ABC % 10.

Кроме того, введенное соответствие порождает порядок, полностью совпадающий с лексикографическим порядком

$X^{A1}Y^{B1}Z^{C1} > X^{A2}Y^{B2}Z^{C2}$ ↔ ABC1 > ABC2.

Выполненное обсуждение позволяет определить, что наиболее эффективным способом организации структуры хранения полиномов являются линейный (односвязный) список. Тем самым, в рамках лабораторной работы появляется подзадача – разработка структуры хранения в виде линейных списков. Данная разработка должна быть выполнена в некоторой общей постановке с тем, чтобы разработанные программы работы со списками могли быть далее использованы и в других ситуациях, в которых необходимы списковые структуры хранения.

Для работы с полиномами предлагается реализовать следующие операции:

- конструкторы инициализации и копирования;

- метод присваивания;
- методы сложения, вычитания, умножения полиномов.

Ниже более подробно рассматриваются алгоритмы для выполнения указанных методов.

3.2.2.1. Сложение двух полиномов (на линейных списках)

Пусть необходимо выполнить операцию P + Q (P, Q – полиномы от трех переменных, хранящиеся в виде линейного списка). В результирующий полином R копируются мономы из P. Далее для каждого монома в полиноме Q выполняется поиск монома такой же степени в полиноме R. Если такой моном найден, то их коэффициенты складываются, иначе выполняется упорядоченная вставка этого монома в полином R.

3.2.2.2. Вычитание двух полиномов (на линейных списках)

Пусть необходимо выполнить операцию P – Q (P, Q – полиномы от трех переменных, хранящиеся в виде линейного списка). В результирующий полином R копируются мономы из P. Далее для каждого монома в полиноме Q выполняется поиск монома такой же степени в полиноме R. Если такой моном найден, то их коэффициенты вычитаются, иначе выполняется упорядоченная вставка этого монома (с коэффициентом, умноженным на -1) в полином R.

3.2.2.3. Умножение двух полиномов (на линейных списках)

Пусть необходимо выполнить операцию P * Q (P, Q – полиномы от трех переменных, хранящиеся в виде линейного списка). Создается пустой результирующий полином R. Далее для каждого монома в полиноме P выполняется умножение с каждым мономом из полинома Q, после чего результаты складываются и упорядоченно вставляются в полином R.

3.3. Описание структур данных и функций

3.3.1. Класс TList

Объявление класса для работы со списком выглядит следующим образом:

```
template <typename TKey, typename TData>
class TList
{
public:
    class TNode
    {
        template <typename, typename> friend class TList;
        template <typename> friend class TListIterator;
        TNode* pNext;
    public:
        TKey key;
        TData data;
        explicit TNode(TKey key = 0, TData data = 0, TNode* pNext = nullptr);
        TNode(const TNode& other);
    };
private:
    TList::TNode* pFirst;
public:
    TList();
    TList(const TList& other);
    TList(const TList<TKey, TData>::TNode* firstNode);
    ~TList();
    typedef TListIterator<TList::TNode> iterator;
    typedef TListIterator<TList::TNode> const_iterator;
    typename TNode* find(TKey needle);
    typename TNode* getFirst();
    void insertToStart(TKey key, TData data = 0);
    void insertToEnd(TKey key, TData data = 0);
    void insertBefore(TKey needle, TKey key, TData data = 0);
    void insertBefore(iterator i, TKey key, TData data = 0);
    void insertAfter(TKey needle, TKey key, TData data = 0);
    void insertAfter(iterator i, TKey key, TData data = 0);
    void remove(TKey needle);
    void remove(iterator& i);
    void removeAll();
    size_t size() const;
```

```
    bool empty() const;
    typename iterator begin();
    typename iterator end();
    typename const_iterator begin() const;
    typename const_iterator end() const;
};
namespace TListException
{
    class NodeNotFound : ::std::exception
    {
        const ::std::string whatStr = "Node with given key not found.";
    public:
        virtual const char* what() { return whatStr.c_str(); }
    };
};
```

3.3.1.1. Пользовательские типы данных

`TNode` – класс узла (элемента) линейного односвязного списка.

3.3.1.2. Классы исключений

`NodeNotFound` – узел с заданным ключом не найден (при попытке найти, вставить, удалить элемент).

3.3.1.3. Поля класса

`TNode* pFirst;`

Назначение: хранит указатель на первый узел линейного односвязного списка.

3.3.1.4. Методы класса

`TList();`

Назначение: конструктор по умолчанию.

Входные параметры: отсутствуют.

Выходные данные: отсутствуют.

`TList(const TList& other);`

Назначение: конструктор копирования.

Входные параметры: other – копируемый список.

Выходные данные: отсутствуют.

```
TList(const TNode* firstNode);
```

Назначение: конструктор преобразования цепи связанных узлов в односвязный список.

Входные параметры: firstNode – указатель на первый узел.

Выходные данные: отсутствуют.

```
~ TList();
```

Назначение: деструктор.

Входные параметры: отсутствуют.

Выходные данные: отсутствуют.

```
typename TNode* find(TKey needle);
```

Назначение: поиск элемента в списке по ключу.

Входные параметры: needle – ключ искомого элемента.

Выходные данные: отсутствуют.

```
typename TNode* getFirst();
```

Назначение: получение первого элемента списка.

Входные параметры: отсутствуют.

Выходные данные: отсутствуют.

```
void insertToStart(TKey key, TData data);
```

Назначение: вставка элемента в начало списка.

Входные параметры: key – ключ вставляемого элемента, data – данные вставляемого элемента.

Выходные данные: отсутствуют.

```
void insertToEnd(TKey key, TData data);
```

Назначение: вставка элемента в конец списка.

Входные параметры: key – ключ вставляемого элемента, data – данные вставляемого элемента.

Выходные данные: отсутствуют.

```
void insertBefore(TKey needle, TKey key, TData data);
```

Назначение: вставка элемента перед заданным элементом.

Входные параметры: needle – ключ элемента, перед которым нужно вставить новый элемент, key – ключ вставляемого элемента, data – данные вставляемого элемента.

Выходные данные: отсутствуют.

```
void insertBefore(iterator i, TKey key, TData data);
```

Назначение: вставка элемента перед заданным элементом.

Входные параметры: i – итератор, установленный на элемент, перед которым нужно вставить новый элемент, key – ключ вставляемого элемента, data – данные вставляемого элемента.

Выходные данные: отсутствуют.

```
void insertAfter(TKey needle, TKey key, TData data);
```

Назначение: вставка элемента после заданного элемента.

Входные параметры: needle – ключ элемента, после которого нужно вставить новый элемент, key – ключ вставляемого элемента, data – данные вставляемого элемента.

Выходные данные: отсутствуют.

```
void insertAfter(iterator i, TKey key, TData data);
```

Назначение: вставка элемента после заданного элемента.

Входные параметры: i – итератор, установленный на элемент, после которого нужно вставить новый элемент, key – ключ вставляемого элемента, data – данные вставляемого элемента.

Выходные данные: отсутствуют.

```
void remove(TKey needle);
```

Назначение: удаление заданного элемента из списка.

Входные параметры: needle – ключ удаляемого элемента.

Выходные данные: отсутствуют.

```
void remove(iterator i);
```

Назначение: удаление заданного элемента из списка.

Входные параметры: i – итератор, установленный на удаляемый элемент.

Выходные данные: отсутствуют.

```
void removeAll();
```

Назначение: удаление всех элементов из списка.

Входные параметры: отсутствуют.

Выходные данные: отсутствуют.

```
size_t size() const;
```

Назначение: подсчет количества элементов в списке.

Входные параметры: отсутствуют.

Выходные данные: количество элементов в списке.

```
bool empty() const;
```

Назначение: проверяет, является ли список пустым.

Входные параметры: отсутствуют.

Выходные данные: истина или ложь.

```
typename iterator begin();
```

Назначение: получение начального итератора для списка.

Входные параметры: отсутствуют.

Выходные данные: итератор.

```
typename iterator end();
```

Назначение: получение конечного итератора для списка.

Входные параметры: отсутствуют.

Выходные данные: итератор.

```
typename const_iterator begin() const;
```

Назначение: получение начального итератора для константного списка.

Входные параметры: отсутствуют.

Выходные данные: итератор.

```
typename const_iterator end() const;
```

Назначение: получение конечного итератора для константного списка.

Входные параметры: отсутствуют.

Выходные данные: итератор.

3.3.2. Класс TMonomial

Для хранения мономов используется специализированный шаблонный класс звена линейного списка.

```
template<>
class TList<unsigned, double>::TNode
{
public:
	class DegreeOverflow : std::exception
	{
		const std::string whatStr = "Degree cannot be greater than 9.";
	public:
		virtual const char* what() { return whatStr.c_str(); }
	};

	class DegreeUnequality : std::exception
	{
		const std::string whatStr = "Degrees are not equal.";
	public:
		virtual const char* what() { return whatStr.c_str(); }
	};
private:
    template <typename, typename> friend class TList;
    template <typename> friend class TListIterator;
    friend class TPolynomial;
    friend std::ostream& operator<<(std::ostream& stream, const TPolynomial& polynomial);
    unsigned key;
    double data;
    TNode* pNext;
    bool checkDegrees(unsigned degrees) const;
    bool checkDegrees(unsigned x, unsigned y, unsigned z) const;
public:
    explicit TNode(unsigned degree = 0U, double coefficient = 0., TNode* pNext_ = nullptr);
    TNode(const TNode&) = default;
    TNode(double coefficient, unsigned degree);
    TNode operator+(const TNode& other) const;
    TNode operator-(const TNode& other) const;
    TNode operator*(const TNode& other) const;
    TNode operator*(double number) const;
    TNode operator+() const;
    TNode operator-() const;
};
```

3.3.2.1. Классы исключений

1. `DegreeOverflow` – одна или несколько степеней переменных монома превышает 9.
2. `DegreeUnequality` – степени мономов различаются (при сложении, вычитании).

3.3.2.2. Методы класса

```
bool checkDegrees(unsigned degrees) const;
```

Назначение: проверка свертки степени монома.

Входные параметры: degrees – свертка.

Выходные данные: истина или ложь.

```
bool checkDegrees(unsigned x, unsigned y, unsigned z) const;
```

Назначение: проверка степеней монома.

Входные параметры: x – степень при переменной x, y – степень при переменной y, z – степень при переменной z.

Выходные данные: истина или ложь.

```
explicit TNode(unsigned degree, double coefficient, TNode* pNext);
```

Назначение: конструктор ().

Входные параметры: degree.

Выходные данные: отсутствуют.

3.3.3. Класс TPolynomial

Этот класс используется для работы с полиномами от трех переменных.

```
class TPolynomial
{
    TList<unsigned, double>* monomials;
    TMonomialList::iterator findPrevOrderedDegree(unsigned degree) const;
    TMonomialList::iterator getNextIterator(TMonomialList::iterator iterator) const;
    void add(double coefficient, unsigned degree);
    void addNonzero(TMonomial* primary, const TMonomial* secondary);
    void reduce();
    void sort();
    void nullify();
    void parse(const char* const expression);
    const std::string monomToStr(const TMonomial& monomial) const;
    const std::string getFirstFloatNumber(const char* const expression, size_t& offset) const;
    const std::string getFirstUIntNumber(const char* const expression, size_t& offset) const;
    unsigned getDegreeMask(const char* const expression, unsigned& factor) const;
public:
    TPolynomial();
    TPolynomial(const char* const expression);
    TPolynomial(const TMonomial& monomial);
    TPolynomial(double number);
    TPolynomial(const TPolynomial& other);
    TPolynomial(const TMonomialList& list);
    ~TPolynomial();
    TPolynomial& operator=(const TPolynomial& other);
    TPolynomial operator+(const TPolynomial& other);
    TPolynomial operator+(const TMonomial& monomial);
    TPolynomial operator+(double number);
    TPolynomial operator-(const TPolynomial& other);
    TPolynomial operator-(const TMonomial& monomial);
    TPolynomial operator-(double number);
    TPolynomial operator*(const TPolynomial& other);
    TPolynomial operator*(const TMonomial& monomial);
    TPolynomial operator*(double number);
    TPolynomial& operator+=(const TPolynomial& other);
    TPolynomial& operator+=(const TMonomial& monomial);
    TPolynomial& operator+=(double number);
    TPolynomial& operator-=(const TPolynomial& other);
    TPolynomial& operator-=(const TMonomial& monomial);
    TPolynomial& operator-=(double number);
    TPolynomial& operator*=(const TPolynomial& other);
    TPolynomial& operator*=(const TMonomial& monomial);
    TPolynomial& operator*=(double number);
    friend std::ostream& operator<<(std::ostream& stream, const TPolynomial& polynomial);
    friend std::istream& operator>>(std::istream& stream, TPolynomial& polynomial);
    class SyntaxError : std::exception
    {
```

```
        const std::string whatStr = "Serialization cannot be parsed.";
    public:
        virtual const char* what() { return whatStr.c_str(); }
    };
};
```

3.3.3.1. Классы исключений

`SyntaxError` – ошибка при обработке строки с полиномом.

3.3.3.2. Поля класса

`TList<unsigned, double>* monomials;`

Назначение: хранит указатель на список мономов.

3.3.3.3. Методы класса

`TPolynomial();`

Назначение: конструктор по умолчанию.

Входные параметры: отсутствуют.

Выходные данные: отсутствуют.

`TPolynomial(const char* const expression);`

Назначение: конструктор преобразования из строки.

Входные параметры: expression – выражение (строковое представление полинома).

Выходные данные: отсутствуют.

`TPolynomial(const TMonomial& monomial);`

Назначение: конструктор преобразования из монома.

Входные параметры: monomial – моном.

Выходные данные: отсутствуют.

`TPolynomial(double number);`

Назначение: конструктор преобразования из числа.

Входные параметры: number – число.

Выходные данные: отсутствуют.

`TPolynomial(const TPolynomial& other);`

Назначение: конструктор копирования.

Входные параметры: other – полином.

Выходные данные: отсутствуют.

`TPolynomial(const TMonomialList& list);`

Назначение: конструктор преобразования из списка мономов.

Входные параметры: list – список мономов.

Выходные данные: отсутствуют.

`~TPolynomial();`

Назначение: деструктор.

Входные параметры: отсутствуют.

Выходные данные: отсутствуют.

`TMonomialList::iterator findPrevOrderedDegree(unsigned degree) const;`

Назначение: ищет моном, который должен предшествовать моному с указанной степенью.

Входные параметры: degree – степень монома.

Выходные данные: итератор, установленный на нужный моном.

`TMonomialList::iterator getNextIterator(TMonomialList::iterator iterator) const;`

Назначение: переводит итератор на один шаг вперед.

Входные параметры: iterator – итератор.

Выходные данные: итератор.

`void add(double coefficient, unsigned degree);`

Назначение: выполняет упорядоченную вставку монома в список.

Входные параметры: coefficient – коэффициент монома, degree – степень монома.

Выходные данные: отсутствуют.

`void addNonzero(TMonomial* primary, const TMonomial* secondary);`

Назначение: выполняет «умную» вставку суммы мономов.

Входные параметры: primary, secondary – указатели на мономы.

Выходные данные: отсутствуют.

`void reduce();`

Назначение: приводит подобные мономы.

Входные параметры: отсутствуют.

Выходные данные: отсутствуют.

```
void sort();
```

Назначение: выполняет сортировку мономов по убыванию степеней.

Входные параметры: отсутствуют.

Выходные данные: отсутствуют.

```
void nullify();
```

Назначение: обнуляет полином.

Входные параметры: отсутствуют.

Выходные данные: отсутствуют.

```
void parse(const char* const expression);
```

Назначение: выполняет преобразование строки в полином.

Входные параметры: expression – выражение (строковое представление полинома).

Выходные данные: отсутствуют.

```
const std::string monomToStr(const TMonomial& monomial) const;
```

Назначение: возвращает строковое представление монома.

Входные параметры: monomial – моном.

Выходные данные: строка – представление монома.

```
const std::string getFirstFloatNumber(const char* const expression, size_t& offset) const;
```

Назначение: находит и возвращает первое число с плавающей точкой в строке.

Входные параметры: expression – строка, offset – индекс символа начала числа.

Выходные данные: строковое представление числа.

```
const std::string getFirstUIntNumber(const char* const expression, size_t& offset) const;
```

Назначение: находит и возвращает первое целое число без знака в строке.

Входные параметры: expression – строка, offset – индекс первой цифры числа.

Выходные данные: строковое представление числа.

```
unsigned getDegreeMask(const char* const expression, unsigned& factor) const;
```

Назначение: возвращает свертку степени для одной переменной монома.

Входные параметры: expression – строка, в которой производится поиск, factor – маска свертки степени (1 – найдена переменная z, 10 – найдена переменная y, 100 – найдена переменная x, 1000 – переменные не найдены).

Выходные данные: свертка степени для одной переменной монома.

3.3.3.4. Перегруженные операторы класса

```
TPolynomial& operator=(const TPolynomial& other);
```

Назначение: выполняет операцию присваивания полинома.

Входные параметры: other – полином.

Выходные данные: ссылка на результирующий полином.

```
TPolynomial operator+(const TPolynomial& other);
```

Назначение: выполняет операцию сложения с полиномом.

Входные параметры: other – полином.

Выходные данные: результирующий полином.

```
TPolynomial operator+(const TMonomial& monomial);
```

Назначение: выполняет операцию сложения с мономом.

Входные параметры: monomial – моном.

Выходные данные: результирующий полином.

```
TPolynomial operator+(double number);
```

Назначение: выполняет операцию сложения с числом.

Входные параметры: number – число.

Выходные данные: результирующий полином.

```
TPolynomial operator-(const TPolynomial& other);
```

Назначение: выполняет операцию вычитания с полиномом.

Входные параметры: other – полином.

Выходные данные: результирующий полином.

```
TPolynomial operator-(const TMonomial& monomial);
```

Назначение: выполняет операцию вычитания с мономом.

Входные параметры: monomial – моном.

Выходные данные: результирующий полином.

```
TPolynomial operator-(double number);
```

Назначение: выполняет операцию вычитания с числом.

Входные параметры: number – число.

Выходные данные: результирующий полином.

```
TPolynomial operator*(const TPolynomial& other);
```

Назначение: выполняет операцию умножения с полиномом.

Входные параметры: other – полином.

Выходные данные: результирующий полином.

```
TPolynomial operator*(const TMonomial& monomial);
```

Назначение: выполняет операцию умножения с мономом.

Входные параметры: monomial – моном.

Выходные данные: результирующий полином.

```
TPolynomial operator*(double number);
```

Назначение: выполняет операцию умножения с числом.

Входные параметры: number – число.

Выходные данные: результирующий полином.

```
TPolynomial operator+=(const TPolynomial& other);
```

Назначение: выполняет операцию сложения с полиномом и присваивание.

Входные параметры: other – полином.

Выходные данные: ссылка на результирующий полином.

```
TPolynomial operator+=(const TMonomial& monomial);
```

Назначение: выполняет операцию сложения с мономом и присваивание.

Входные параметры: monomial – моном.

Выходные данные: ссылка на результирующий полином.

```
TPolynomial operator+=(double number);
```

Назначение: выполняет операцию сложения с числом и присваивание.

Входные параметры: number – число.

Выходные данные: ссылка на результирующий полином.

```
TPolynomial operator-=(const TPolynomial& other);
```

Назначение: выполняет операцию вычитания с полиномом и присваивание.

Входные параметры: other – полином.

Выходные данные: ссылка на результирующий полином.

```
TPolynomial operator-=(const TMonomial& monomial);
```

Назначение: выполняет операцию вычитания с мономом и присваивание.

Входные параметры: monomial – моном.

Выходные данные: ссылка на результирующий полином.

```
TPolynomial operator-=(double number);
```

Назначение: выполняет операцию вычитания с числом и присваивание.

Входные параметры: number – число.

Выходные данные: ссылка на результирующий полином.

`TPolynomial operator*=(const TPolynomial& other);`

Назначение: выполняет операцию умножения с полиномом и присваивание.

Входные параметры: other – полином.

Выходные данные: ссылка на результирующий полином.

`TPolynomial operator*=(const TMonomial& monomial);`

Назначение: выполняет операцию умножения с мономом и присваивание.

Входные параметры: monomial – моном.

Выходные данные: ссылка на результирующий полином.

`TPolynomial operator*=(double number);`

Назначение: выполняет операцию умножения с числом и присваивание.

Входные параметры: number – число.

Выходные данные: ссылка на результирующий полином.

`friend std::ostream& operator<<(std::ostream& stream, const TPolynomial& polynomial);`

Назначение: выполняет вывод строкового представления полинома в поток.

Входные параметры: stream – поток вывода, polynomial – полином.

Выходные данные: ссылка на поток вывода.

`friend std::istream& operator>>(std::istream& stream, TPolynomial& polynomial);`

Назначение: выполняет ввод строкового представления полинома из потока.

Входные параметры: stream – поток ввода, polynomial – полином.

Выходные данные: ссылка на поток ввода.

3.3.4. Основная программа

Во всех функциях основной программы входные параметры и выходные данные отсутствуют.

`void testMode()`

Назначение: запускает режим автоматического тестирования.

`void calcMode()`

Назначение: запускает режим ручного ввода.

`int main()`

Назначение: основная функция (точка входа).

Заключение

В ходе выполнения практической работы «Аналитические преобразования полиномов от нескольких переменных (списки)» было разработано и реализовано консольное приложение для работы с полиномами от трех переменных, включая выполнение алгебраических операций над ними, использующее динамическую структуру данных Список.

Список используемых источников

1. Лабораторный практикум. Барышева И. В., Мееров И. Б., Сысоев А. В., Шестакова Н. В. Учебно-методическое пособие. – Нижний Новгород (ННГУ), 2017. – 105 с.
2. Рабочие материалы к учебному курсу «Методы программирования» (часть 1). Гергель В. П. Нижний Новгород, 2015. – 100 с.

Приложение

Приложение 1. Исходный код основной программы.

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "TPolynomial.h"
#define RND(MIN, MAX) (rand() % ((MAX) - (MIN) + 1) + (MIN))

void testMode();
void calcMode();

int main()
{
    std::cout << "0 - Test Mode, 1 - Calc Mode: ";
    int menu;
    std::cin >> menu;
    std::cin.ignore(1);
    if (menu == 0)
        testMode();
    else
        calcMode();
    std::cin.ignore(1);
    return 0;
}

void testMode()
{
    srand((unsigned)time(nullptr));
    TPolynomial poly[2];
    TMonomial monomial = TMonomial(static_cast<double>(RND(-10, 10)), RND(0,
999));
    double number = RND(-10, 10);
    int sizes[2] = { RND(3, 6), RND(3, 6) };
    int degree = RND(0, 999);
    for (int i = 0; i < 2; i++)
    {
        for (int j = 0; j < sizes[i]; j++)
            poly[i] += TMonomial(static_cast<double>(RND(-10, 10)), RND(0,
999));
        std::cout << '(' << i << ") " << poly[i] << '\n';
    }
    std::cout << "(2) " << TPolynomial(monomial) << '\n';
    std::cout << "(3) " << number << '\n';
    std::cout << '\n';
    std::cout << "(0) + (1) = " << (poly[0] + poly[1]) << '\n';
    std::cout << "(0) + (2) = " << (poly[0] + monomial) << '\n';
    std::cout << "(0) + (3) = " << (poly[0] + number) << '\n';
    std::cout << "(0) - (1) = " << (poly[0] - poly[1]) << '\n';
    std::cout << "(0) - (2) = " << (poly[0] - monomial) << '\n';
    std::cout << "(0) - (3) = " << (poly[0] - number) << '\n';
    try
    {
        std::cout << "(0) * (1) = " << (poly[0] * poly[1]) << '\n';
    }
    catch (TMonomial::DegreeOverflow & e)
```

```
    {
        std::cerr << e.what() << '\n';
    }
    try
    {
        std::cout << "(0) * (2) = " << (poly[0] * monomial) << '\n';
    }
    catch (TMonomial::DegreeOverflow & e)
    {
        std::cerr << e.what() << '\n';
    }
    std::cout << "(0) * (3) = " << (poly[0] * number) << '\n';
    std::cout << "\n_poly Literal test: ";
    try
    {
        std::cout << "x + 1"_poly << '\n';

    }
    catch (TPolynomial::SyntaxError & e)
    {
        std::cerr << e.what();
    }
    std::cout << "\nEnter expression: ";
    try
    {
        TPolynomial ipoly;
        std::cin >> ipoly;
        std::cout << "Parsed: " << ipoly << '\n';

    }
    catch (std::exception& e)
    {
        std::cerr << e.what();
    }
    TList<unsigned, double> l;
    l.insertToEnd(0U, 1);
    l.insertToEnd(150U, 3);
    l.insertToEnd(350U, 5);
    TPolynomial sp = l;
    std::cout << sp;
}

void calcMode()
{
    TPolynomial lhs, rhs;
    bool success;
    std::cout << "Enter polynom 1: ";
    success = false;
    do
    {
        try
        {
            std::cin >> lhs;
            std::cout << "Parsed: " << lhs << '\n';
            success = true;
        }
        catch (...)
        {
            std::cerr << "Syntax Error.";
```

```
            std::cout << " Try again: ";
        }
    } while (!success);
    std::cout << "Enter polynom 2: ";
    success = false;
    do
    {
        try
        {
            std::cin >> rhs;
            std::cout << "Parsed: " << rhs << '\n';
            success = true;
        }
        catch (...)
        {
            std::cerr << "Syntax Error.";
            std::cout << " Try again: ";
        }
    } while (!success);
    std::cout << "Result: \n";
    try
    {
        std::cout << "(+): " << (lhs + rhs) << '\n';
        std::cout << "(-): " << (lhs - rhs) << '\n';
        std::cout << "(*): " << (lhs * rhs) << '\n';
    }
    catch (...)
    {
        std::cerr << "Degree overflow.";
    }
    std::cout << '\n';
}
```

Приложение 2. Исходный код заголовочного файла TList.

```
#ifndef _TLIST_H_
#define _TLIST_H_
#include <iostream>
#include <iterator>
#include <exception>
#include <string>

template <typename TNodeTypename>
class TListIterator : public std::iterator<std::input_iterator_tag,
TNodeTypename>
{
    template <typename, typename> friend class TList;
    TNodeTypename* pNode;
    TListIterator(TNodeTypename* pNode);
public:
    TListIterator(const TListIterator& other) = default;
    ~TListIterator() = default;
    bool operator!=(TListIterator const& other) const;
    bool operator==(TListIterator const& other) const;
    typename TNodeTypename* operator*() const;
    TListIterator operator++();
    TListIterator operator++(int);
};

template <typename TKey, typename TData>
class TList
{
public:
    class TNode
    {
        template <typename, typename> friend class TList;
        template <typename> friend class TListIterator;
        TNode* pNext;
    public:
        TKey key;
        TData data;
        explicit TNode(TKey key = 0, TData data = 0, TNode* pNext = nullptr);
        TNode(const TNode& other);
    };
private:
    TList::TNode* pFirst;
public:
    TList();
    TList(const TList& other);
    TList(const TList<TKey, TData>::TNode* firstNode);
    ~TList();

    typedef TListIterator<TList::TNode> iterator;
    typedef TListIterator<TList::TNode> const_iterator;

    typename TNode* find(TKey needle);
    typename TNode* getFirst();
    void insertToStart(TKey key, TData data = 0);
    void insertToEnd(TKey key, TData data = 0);
    void insertBefore(TKey needle, TKey key, TData data = 0);
    void insertBefore(iterator i, TKey key, TData data = 0);
    void insertAfter(TKey needle, TKey key, TData data = 0);
    void insertAfter(iterator i, TKey key, TData data = 0);
```

```
    void remove(TKey needle);
    void remove(iterator& i);
    void removeAll();

    size_t size() const;
    bool empty() const;

    typename iterator begin();
    typename iterator end();
    typename const_iterator begin() const;
    typename const_iterator end() const;

    void output(std::ostream& stream, const char* separator = " ", const char*
ending = "\n") const;
    void outputRaw(std::ostream& stream) const;
};

namespace TListException
{
    class NodeNotFound : ::std::exception
    {
        const ::std::string whatStr = "Node with given key not found.";
    public:
        virtual const char* what() { return whatStr.c_str(); }
    };
}

// ---------------------------------------------------------------------
// ------------------------------ TNODE --------------------------------
// ---------------------------------------------------------------------

template<typename TKey, typename TData>
TList<TKey, TData>::TNode::TNode(TKey key, TData data, TList<TKey,
TData>::TNode* pNext)
    : key(key), data(data), pNext(pNext)
{
}

template<typename TKey, typename TData>
TList<TKey, TData>::TNode::TNode(const TNode& other)
    : key(other.key), data(other.data), pNext(pNext)
{
}

// ---------------------------------------------------------------------
// ------------------------------ TLIST --------------------------------
// ---------------------------------------------------------------------

template<typename TKey, typename TData>
TList<TKey, TData>::TList()
{
    pFirst = nullptr;
}

template<typename TKey, typename TData>
TList<TKey, TData>::TList(const TList& other) : TList()
{
    if (!other.pFirst)
        return;
```

```
    pFirst = new TNode(other.pFirst->key, other.pFirst->data);
    TNode* temp = other.pFirst->pNext;
    TNode* prev = pFirst;
    while (temp)
    {
        TNode* pNode = new TNode(temp->key, temp->data);
        prev->pNext = pNode;
        prev = pNode;
        temp = temp->pNext;
    }
}

template<typename TKey, typename TData>
TList<TKey, TData>::TList(const TList<TKey, TData>::TNode* firstNode) : TList()
{
    if (!firstNode)
        return;
    pFirst = new TNode(firstNode->key, firstNode->data);
    TNode* temp = firstNode->pNext;
    TNode* prev = pFirst;
    while (temp)
    {
        TNode* pNode = new TNode(temp.key, temp.data);
        prev->pNext = pNode;
        prev = pNode;
        temp = temp->pNext;
    }
}

template<typename TKey, typename TData>
TList<TKey, TData>::~TList()
{
    removeAll();
}

template<typename TKey, typename TData>
typename TList<TKey, TData>::TNode* TList<TKey, TData>::find(TKey needle)
{
    if (!pFirst)
        return nullptr;
    TNode* temp = pFirst;
    while (temp)
    {
        if (temp->key == needle)
            return temp;
        temp = temp->pNext;
    }
    return nullptr;
}

template<typename TKey, typename TData>
typename TList<TKey, TData>::TNode* TList<TKey, TData>::getFirst()
{
    return pFirst;
}

template<typename TKey, typename TData>
void TList<TKey, TData>::insertToStart(TKey key, TData data)
{
```

```
    TNode* pNode = new TNode(key, data, pFirst);
    pFirst = pNode;
}

template<typename TKey, typename TData>
void TList<TKey, TData>::insertToEnd(TKey key, TData data)
{
    if (!pFirst)
    {
        pFirst = new TNode(key, data);
        return;
    }
    TNode* temp = pFirst;
    while (temp->pNext)
        temp = temp->pNext;
    temp->pNext = new TNode(key, data, nullptr);
}

template<typename TKey, typename TData>
void TList<TKey, TData>::insertBefore(TKey needle, TKey key, TData data)
{
    if (!pFirst)
        throw TListException::NodeNotFound();
    if (pFirst->key == needle)
    {
        TNode* pNode = new TNode(key, data, pFirst);
        pFirst = pNode;
        return;
    }
    TNode* temp = pFirst;
    while (temp->pNext && (temp->pNext->key != needle))
        temp = temp->pNext;
    if (!temp->pNext)
        throw TListException::NodeNotFound();
    TNode* pNode = new TNode(key, data, temp->pNext);
    temp->pNext = pNode;
}

template<typename TKey, typename TData>
void TList<TKey, TData>::insertBefore(iterator i, TKey key, TData data)
{
    if (!pFirst)
        throw TListException::NodeNotFound();
    if (!i.pNode)
    {
        insertToEnd((*i).key, (*i).data);
        return;
    }
    TNode* needle = i.pNode;
    if (pFirst == needle)
    {
        TNode* pNode = new TNode(key, data, pFirst);
        pFirst = pNode;
        return;
    }
    TNode* temp = pFirst;
    while (temp->pNext && (temp->pNext != needle))
        temp = temp->pNext;
    if (!temp->pNext)
```

```
        throw TListException::NodeNotFound();
    TNode* pNode = new TNode(key, data, temp->pNext);
    temp->pNext = pNode;
}

template<typename TKey, typename TData>
void TList<TKey, TData>::insertAfter(iterator i, TKey key, TData data)
{
    if (!i.pNode)
        throw TListException::NodeNotFound();
    TNode* pNode = new TNode(key, data, i.pNode->pNext);
    i.pNode->pNext = pNode;
}

template<typename TKey, typename TData>
void TList<TKey, TData>::insertAfter(TKey needle, TKey key, TData data)
{
    TNode* prev = findNode(needle);
    if (!prev)
        throw TListException::NodeNotFound();
    TNode* pNode = new TNode(key, data, prev->pNext);
    prev->pNext = pNode;
}

template<typename TKey, typename TData>
void TList<TKey, TData>::remove(TKey needle)
{
    if (!pFirst)
        throw TListException::NodeNotFound();
    if (pFirst->key == needle)
    {
        TNode* pNode = pFirst;
        pFirst = pFirst->pNext;
        delete pNode;
        return;
    }
    TNode* temp = pFirst;
    while (temp->pNext && (temp->pNext->key != needle))
        temp = temp->pNext;
    if (!temp->pNext)
        throw TListException::NodeNotFound();
    TNode* pNode = temp->pNext;
    temp->pNext = pNode->pNext;
    delete pNode;
}

template<typename TKey, typename TData>
void TList<TKey, TData>::remove(iterator& i)
{
    if (!pFirst || !i.pNode)
        throw TListException::NodeNotFound();
    TNode* needle = i.pNode;
    if (pFirst == needle)
    {
        TNode* pNode = pFirst;
        pFirst = pFirst->pNext;
        delete pNode;
        i.pNode = nullptr;
        return;
```

```
    }
    TNode* temp = pFirst;
    while (temp->pNext && (temp->pNext != needle))
        temp = temp->pNext;
    if (!temp->pNext)
        throw TListException::NodeNotFound();
    temp->pNext = needle->pNext;
    delete needle;
    i.pNode = nullptr;

}

template<typename TKey, typename TData>
void TList<TKey, TData>::removeAll()
{
    if (!pFirst)
        return;
    TNode* temp = pFirst;
    while (temp)
    {
        TNode* pNode = temp;
        temp = temp->pNext;
        delete pNode;
    }
    pFirst = nullptr;
}

template<typename TKey, typename TData>
size_t TList<TKey, TData>::size() const
{
    if (!pFirst)
        return 0;
    TNode* temp = pFirst;
    size_t i = 0;
    while (temp)
    {
        i++;
        temp = temp->pNext;
    }
    return i;
}

template<typename TKey, typename TData>
bool TList<TKey, TData>::empty() const
{
    return pFirst == nullptr;
}

template<typename TKey, typename TData>
void TList<TKey, TData>::output(std::ostream& stream, const char* separator,
const char* ending) const
{
    TNode* temp = pFirst;
    while (temp)
    {
        if (temp->data)
            stream << *(temp->data);
        else
            stream << "nullptr";
```

```
        stream << separator;
        temp = temp->pNext;
    }
    stream << ending;
}

template<typename TKey, typename TData>
void TList<TKey, TData>::outputRaw(std::ostream& stream) const
{
    TNode* temp = pFirst;
    stream << "[\n";
    while (temp)
    {
        stream << "    {\n        " << temp->key << "\n        ";
        if (temp->data)
            stream << *(temp->data);
        else
            stream << "nullptr";
        stream << "\n    }\n";
        temp = temp->pNext;
    }
    stream << "]\n";
}

// ------------------------------------------------------------------
// ---------------------- ITERATION IN TLIST ------------------------
// ------------------------------------------------------------------

template<typename TKey, typename TData>
typename TList<TKey, TData>::iterator TList<TKey, TData>::begin()
{
    return iterator(pFirst);
}

template<typename TKey, typename TData>
typename TList<TKey, TData>::iterator TList<TKey, TData>::end()
{
    return iterator(nullptr);
}

template<typename TKey, typename TData>
typename TList<TKey, TData>::const_iterator TList<TKey, TData>::begin() const
{
    return iterator(pFirst);
}

template<typename TKey, typename TData>
typename TList<TKey, TData>::const_iterator TList<TKey, TData>::end() const
{
    return iterator(nullptr);
}

// ------------------------------------------------------------------
// ------------------------- TLISTITERATOR --------------------------
// ------------------------------------------------------------------

template <typename TNodeTypename>
TListIterator<TNodeTypename>::TListIterator(TNodeTypename* pNode) : pNode(pNode)
{}
```

```
template <typename TNodeTypename>
bool TListIterator<TNodeTypename>::operator!=(TListIterator const& other) const
{
    return pNode != other.pNode;
}

template <typename TNodeTypename>
bool TListIterator<TNodeTypename>::operator==(TListIterator const& other) const
{
    return pNode == other.pNode;
}

template <typename TNodeTypename>
typename TNodeTypename* TListIterator<TNodeTypename>::operator*() const
{
    return pNode;
}

template <typename TNodeTypename>
TListIterator<TNodeTypename> TListIterator<TNodeTypename>::operator++()
{
    if (pNode)
        pNode = pNode->pNext;
    return *this;
}

template <typename TNodeTypename>
TListIterator<TNodeTypename> TListIterator<TNodeTypename>::operator++(int)
{
    TNodeTypename* temp = pNode;
    if (pNode)
        pNode = pNode->pNext;
    return TListIterator<TNodeTypename>(temp);
}

#endif // !_TLIST_H_
```

Приложение 3. Исходный код заголовочного файла TMonomial.

```
#ifndef _TMONOMIAL_H_
#define _TMONOMIAL_H_

#define DEGX(A) ((A) / 100U)
#define DEGY(A) ((A) / 10U % 10U)
#define DEGZ(A) ((A) % 10U)
#define MAKEDEG(X, Y, Z) ((X) * 100U + (Y) * 10U + (Z))

#include "TList.h"

typedef TList<unsigned, double> TMonomialList;
typedef TMonomialList::TNode TMonomial;

template<>
class TList<unsigned, double>::TNode
{
public:
      class DegreeOverflow : std::exception
      {
            const std::string whatStr = "Degree cannot be greater than 9.";
      public:
            virtual const char* what() { return whatStr.c_str(); }
      };

      class DegreeUnequality : std::exception
      {
            const std::string whatStr = "Degrees are not equal.";
      public:
            virtual const char* what() { return whatStr.c_str(); }
      };

private:
    template <typename, typename> friend class TList;
    template <typename> friend class TListIterator;
    friend class TPolynomial;
    friend std::ostream& operator<<(std::ostream& stream, const TPolynomial&
polynomial);
    unsigned key;
    double data;
    TNode* pNext;

    bool checkDegrees(unsigned degrees) const
    {
        return degrees <= 999U;
    }

    bool checkDegrees(unsigned x, unsigned y, unsigned z) const
    {
        return (x <= 9) && (y <= 9) && (z <= 9);
    }

public:
    explicit TNode(unsigned degree = 0U, double coefficient = 0., TNode* pNext_
= nullptr)
    {
        if (degree > 999U)
            throw DegreeOverflow();
            key = degree;
```

```
        data = coefficient;
        pNext = pNext_;
    }
      TNode(const TNode&) = default;
      TNode(double coefficient, unsigned degree) : TNode(degree, coefficient)
{};

    TNode operator+(const TNode& other) const
    {
        if (key != other.key)
            throw DegreeUnequality();
        double coefficient = data + other.data;
        return TMonomial(key, coefficient, pNext);
    }

    TNode operator-(const TNode& other) const
    {
        if (key != other.key)
            throw DegreeUnequality();
        double coefficient = data - other.data;
        return TMonomial(key, coefficient, pNext);
    }

    TNode operator*(const TNode& other) const
    {
        if (!checkDegrees(key + other.key))
            throw DegreeOverflow();
        unsigned x = DEGX(key) + DEGX(other.key);
        unsigned y = DEGY(key) + DEGY(other.key);
        unsigned z = DEGZ(key) + DEGZ(other.key);
        if (!checkDegrees(x, y, z))
            throw DegreeOverflow();
        unsigned degree = MAKEDEG(x, y, z);
            double coefficient = data * other.data;
        return TMonomial(degree, coefficient, pNext);
    }

    TNode operator*(double number) const
    {
        return TMonomial(key, data * number, pNext);
    }

      TNode operator+() const
      {
            return *this;
      }

      TNode operator-() const
      {
            return TNode(key, -data, pNext);
      }
};

#endif //!_TMONOMIAL_H_
```

Приложение 4. Исходный код заголовочного класса TPolynomial.

```
#ifndef _TPOLYNOMIAL_H_
#define _TPOLYNOMIAL_H_

#include <iostream>
#include "TMonomial.h"

class TPolynomial
{
    TList<unsigned, double>* monomials;
    TMonomialList::iterator findPrevOrderedDegree(unsigned degree) const;
    TMonomialList::iterator getNextIterator(TMonomialList::iterator iterator)
const;
    void add(double coefficient, unsigned degree);
    void addNonzero(TMonomial* primary, const TMonomial* secondary);
    void reduce();
    void sort();
    void nullify();
    void parse(const char* const expression);
    const std::string monomToStr(const TMonomial& monomial) const;
    const std::string getFirstFloatNumber(const char* const expression, size_t&
offset) const;
    const std::string getFirstUIntNumber(const char* const expression, size_t&
offset) const;
    unsigned getDegreeMask(const char* const expression, unsigned& factor)
const;
public:
    TPolynomial();
    TPolynomial(const char* const expression);
    TPolynomial(const TMonomial& monomial);
    TPolynomial(double number);
    TPolynomial(const TPolynomial& other);
    TPolynomial(const TMonomialList& list);
    ~TPolynomial();
    TPolynomial& operator=(const TPolynomial& other);
    TPolynomial operator+(const TPolynomial& other);
    TPolynomial operator+(const TMonomial& monomial);
    TPolynomial operator+(double number);
    TPolynomial operator-(const TPolynomial& other);
    TPolynomial operator-(const TMonomial& monomial);
    TPolynomial operator-(double number);
    TPolynomial operator*(const TPolynomial& other);
    TPolynomial operator*(const TMonomial& monomial);
    TPolynomial operator*(double number);
    TPolynomial& operator+=(const TPolynomial& other);
    TPolynomial& operator+=(const TMonomial& monomial);
    TPolynomial& operator+=(double number);
    TPolynomial& operator-=(const TPolynomial& other);
    TPolynomial& operator-=(const TMonomial& monomial);
    TPolynomial& operator-=(double number);
    TPolynomial& operator*=(const TPolynomial& other);
    TPolynomial& operator*=(const TMonomial& monomial);
    TPolynomial& operator*=(double number);
    friend std::ostream& operator<<(std::ostream& stream, const TPolynomial&
polynomial);
    friend std::istream& operator>>(std::istream& stream, TPolynomial&
polynomial);
    class SyntaxError : std::exception
    {
```

```
        const std::string whatStr = "Serialization cannot be parsed.";
    public:
        virtual const char* what() { return whatStr.c_str(); }
    };
};

TPolynomial operator ""_poly(const char* literal, size_t);

#endif //!_TPOLYNOMIAL_H_
```

Приложение 5. Исходный код реализации класса TPolynomial.

```
#include "TPolynomial.h"

TMonomialList::iterator TPolynomial::findPrevOrderedDegree(unsigned degree)
const
{
    if (!monomials->empty())
    {
        TMonomial* first = *(monomials->begin());
        if (degree >= first->key)
            return monomials->end();
    }
    TMonomialList::iterator i = monomials->begin();
    TMonomialList::iterator temp = i;
    for (; getNextIterator(i) != monomials->end(); i++)
    {
        temp = i;
        TMonomialList::iterator next = getNextIterator(temp);
        if ((*next)->key <= degree)
            return temp;
    }
    if (getNextIterator(temp) == monomials->end())
        return temp;
    else
        return i;
}

TMonomialList::iterator TPolynomial::getNextIterator(TMonomialList::iterator
iterator) const
{
    return ++iterator;
}

void TPolynomial::add(double coefficient, unsigned degree)
{
    TMonomialList::iterator prevOrdered = findPrevOrderedDegree(degree);
    if (prevOrdered != monomials->end())
        monomials->insertAfter(prevOrdered, degree, coefficient);
    else
        monomials->insertToStart(degree, coefficient);
}

void TPolynomial::addNonzero(TMonomial* primary, const TMonomial* secondary)
{
      if (primary)
      {
        if(secondary)
                *primary = *primary + *secondary;
            if (primary->data == 0)
                  monomials->remove(primary->key);
            return;
      }
      if (secondary && (secondary->data != 0))
            add(secondary->data, secondary->key);
}

void TPolynomial::reduce()
{
```

```
    for (TMonomialList::iterator i = monomials->begin(); i != monomials->end();
i++)
    {
        TMonomial* control = *i;
        if (control->data == 0)
        {
            monomials->remove(i);
            continue;
        }
        for (TMonomialList::iterator j = getNextIterator(i); j != monomials-
>end(); j++)
        {
            TMonomial* monomial = *j;
            if (monomial->key == control->key)
            {
                control->data += monomial->data;
                monomials->remove(j);
            }
        }
    }
}

void TPolynomial::sort()
{
    if (monomials->empty() || monomials->begin()++ == monomials->end())
        return;
    for (TMonomialList::iterator i = monomials->begin(); i != monomials->end();
i++)
    {
        TMonomial* prev = *i;
        for (TMonomialList::iterator j = monomials->begin()++; j != monomials-
>end(); j++)
        {
            TMonomial* current = *j;
            if (prev->key > current->key)
            {
                std::swap(prev->key, current->key);
                std::swap(prev->data, current->data);
            }
        }
    }
}

void TPolynomial::nullify()
{
    monomials->removeAll();
}

void TPolynomial::parse(const char* const expression)
{
    std::string rawStr = expression;
    rawStr = rawStr.substr(rawStr.find_first_not_of(' '));
    if (rawStr.find_first_not_of("0123456789+-*xyz^. \t\n") < rawStr.size())
        throw SyntaxError();
    std::string str;
    for (char symbol : rawStr)
        if (symbol != ' ')
            str += symbol;
    rawStr.clear();
```

```
    do
    {
        size_t delta = 0U;
        size_t offset = str.find_first_of("0123456789.+-");
        size_t offset2 = str.find_first_of("+-", offset + 1);
        size_t offset3 = str.find_first_of("xyz");
        if ((offset >= str.size()) && (offset3 >= str.size()))
            throw SyntaxError();
        bool hasCoef = true;
        if (offset3 < offset)
        {
            offset = offset3;
            offset2 = str.find_first_of("+-", offset);
            hasCoef = false;
        }
        std::string strMonom;
        if (offset2 >= str.size())
            strMonom = str;
        else
            strMonom = str.substr(0, offset2);
        std::string strCoef;
        size_t strCoefSize = 0ULL;
        if (hasCoef)
        {
            strCoef = getFirstFloatNumber(strMonom.c_str(), offset);
            strCoefSize = strCoef.size();
            if (strCoef == "-")
            {
                strCoefSize = 1ULL;
                strCoef = "-1";
            }
            else if (strCoef == "+")
            {
                strCoefSize = 1ULL;
                strCoef = "1";
            }
        }
        else
        {
            strCoef = "1";
        }
        std::string remain = hasCoef ? strMonom.substr(strCoefSize + offset) :
strMonom;
        size_t remainFirstNums = remain.find_first_of("0123456789");
        if (remainFirstNums < remain.find_first_of("xyz"))
            throw SyntaxError(); // like -123 45*x^2*...
        size_t remainVars;
        unsigned degree = 0U;
        while ((remainVars = remain.find_first_of("xyz")) < remain.size())
        {
            size_t remainVarsNext = remain.find_first_of("xyz", remainVars + 1U)
- 1U;
            std::string strVarArea = remain.substr(remainVars, remainVarsNext >
0ULL ? remainVarsNext : 1ULL);
            unsigned factor;
            unsigned addingDegree = getDegreeMask(strVarArea.c_str(), factor);
            unsigned currentDegree = factor == 1U ? DEGZ(degree) : factor == 10U
? DEGY(degree) : DEGX(degree);
            if (currentDegree + addingDegree > 9U)
```

```
                throw SyntaxError();
            degree += addingDegree * factor;
            remain = remain.substr(remainVars + strVarArea.size());
        }
        double coefficient = 0.;
        try
        {
            coefficient = std::stod(strCoef);
        }
        catch (...)
        {
            throw SyntaxError();
        }
        add(coefficient, degree);
        str = str.substr(strMonom.size() + delta);
        if(str.size() > 0)
            str = str.substr(str.find_first_not_of(' '));
    } while (str.size() > 0);
    reduce();
}

const std::string TPolynomial::monomToStr(const TMonomial& monomial) const
{
    constexpr double EPSILON = 1e-4;
    std::string str;
    if (monomial.data == 0)
        return std::string("0");
    if (fabs(monomial.data - (int)monomial.data) < EPSILON)
        str += std::to_string((int)monomial.data);
    else
        str += std::to_string(monomial.data);
    if (monomial.key > 0U)
    {
        if (DEGX(monomial.key) > 0)
            str += "*x";
        if (DEGX(monomial.key) > 1)
        {
            str += '^';
            str += std::to_string(DEGX(monomial.key));
        }
        if (DEGY(monomial.key) > 0)
            str += "*y";
        if (DEGY(monomial.key) > 1)
        {
            str += '^';
            str += std::to_string(DEGY(monomial.key));
        }
        if (DEGZ(monomial.key) > 0)
            str += "*z";
        if (DEGZ(monomial.key) > 1)
        {
            str += '^';
            str += std::to_string(DEGZ(monomial.key));
        }
    }
    return str;
}
```

```
const std::string TPolynomial::getFirstFloatNumber(const char* const expression,
size_t& offset) const
{
    std::string str = expression;
    offset = str.find_first_of("0123456789.+-");
    return str.substr(offset, str.find_first_not_of("0123456789.+-", offset));
}

const std::string TPolynomial::getFirstUIntNumber(const char* const expression,
size_t& offset) const
{
    std::string str = expression;
    offset = str.find_first_of("0123456789+");
    size_t count = str.find_first_not_of("0123456789+", offset);
    if(count < str.size())
        return str.substr(offset, count);
    return str.substr(offset);
}

unsigned TPolynomial::getDegreeMask(const char* const expression, unsigned&
factor) const
{
    std::string str = expression;
    if (str.find_first_of("+-.") < str.size())
        return 1000U;
    size_t varsFound = str.find_first_of("xyz");
    size_t numsFound = str.find_first_of("0123456789");
    if ((numsFound < varsFound) || (varsFound >= str.size()))
        return 1000U;
    factor = 1U;
    if (str[varsFound] == 'x')
        factor = 100U;
    else if (str[varsFound] == 'y')
        factor = 10U;
    if (numsFound >= str.size())
        return 1U;
    size_t offset;
    unsigned degree =
static_cast<unsigned>(std::stoul(getFirstUIntNumber(expression, offset)));
    return degree;
}

TPolynomial::TPolynomial()
{
    monomials = new TMonomialList;
}

TPolynomial::TPolynomial(const char* const expression) : TPolynomial()
{
    parse(expression);
}

TPolynomial::TPolynomial(const TMonomial& monomial) : TPolynomial()
{
    monomials->insertToStart(monomial.key, monomial.data);
}

TPolynomial::TPolynomial(double number)
    : TPolynomial(TMonomial(number, 0U))
```

```
{
}

TPolynomial::TPolynomial(const TPolynomial& other)
{
    monomials = new TMonomialList(*other.monomials);
}

TPolynomial::TPolynomial(const TMonomialList& list)
{
    monomials = new TMonomialList(list);
    sort();
    reduce();
}

TPolynomial::~TPolynomial()
{
    delete monomials;
}

TPolynomial& TPolynomial::operator=(const TPolynomial& other)
{
    if (this == &other)
        return *this;
    delete monomials;
    monomials = new TMonomialList(*other.monomials);
    return *this;
}

TPolynomial TPolynomial::operator+(const TPolynomial& other)
{
    TPolynomial result(*this);
    for (TMonomialList::iterator i = other.monomials->begin(); i !=
other.monomials->end(); i++)
    {
        TMonomial* current = *i;
        TMonomial* monomial = result.monomials->find(current->key);
            result.addNonzero(monomial, current);
    }
    return result;
}

TPolynomial TPolynomial::operator+(const TMonomial& monomial)
{
    TPolynomial result(*this);
    TMonomial* needle = result.monomials->find(monomial.key);
      result.addNonzero(needle, &monomial);
    return result;
}

TPolynomial TPolynomial::operator+(double number)
{
    if (number == 0.)
        return *this;
    return *this + TMonomial(number, 0U);
}

TPolynomial TPolynomial::operator-(const TPolynomial& other)
{
```

```
    TPolynomial result(*this);
    for (TMonomialList::iterator i = other.monomials->begin(); i !=
other.monomials->end(); i++)
    {
        TMonomial* current = *i;
        TMonomial* monomial = result.monomials->find(current->key);
            result.addNonzero(monomial, &-*current);
    }
    return result;
}

TPolynomial TPolynomial::operator-(const TMonomial& monomial)
{
    TPolynomial result(*this);
    TMonomial* needle = result.monomials->find(monomial.key);
      result.addNonzero(needle, &-monomial);
    return result;
}

TPolynomial TPolynomial::operator-(double number)
{
    if (number == 0.)
        return *this;
    return *this - TMonomial(number, 0U);
}

TPolynomial TPolynomial::operator*(const TPolynomial& other)
{
    TPolynomial result;
    for (TMonomialList::iterator i = monomials->begin(); i != monomials->end();
i++)
    {
        for (TMonomialList::iterator j = other.monomials->begin(); j !=
other.monomials->end(); j++)
        {
            TMonomial* lhs = *i;
            TMonomial* rhs = *j;
            TMonomial monomial = *lhs * (*rhs);
            if(monomial.data)
                result.add(monomial.data, monomial.key);
        }
    }
    result.reduce();
    return result;
}

TPolynomial TPolynomial::operator*(const TMonomial& monomial)
{
    if (monomial.data == 0.)
        return TPolynomial();
    TPolynomial result(*this);
    for (TMonomialList::iterator i = monomials->begin(); i != monomials->end();
i++)
    {
        TMonomial* current = *i;
        TMonomial res = *current * monomial;
        if (res.data)
            result.add(res.data, res.key);
    }
```

```
    result.reduce();
    return result;
}

TPolynomial TPolynomial::operator*(double number)
{
    if (number == 0.)
        return TPolynomial();
    TPolynomial result(*this);
    for (TMonomialList::iterator i = result.monomials->begin(); i !=
result.monomials->end(); i++)
    {
        TMonomial* current = *i;
        current->data *= number;
    }
    return result;
}

TPolynomial& TPolynomial::operator+=(const TPolynomial& other)
{
    *this = *this + other;
    return *this;
}

TPolynomial& TPolynomial::operator+=(const TMonomial& monomial)
{
    *this = *this + monomial;
    return *this;
}

TPolynomial& TPolynomial::operator+=(double number)
{
    *this = *this + number;
    return *this;
}

TPolynomial& TPolynomial::operator-=(const TPolynomial& other)
{
    *this = *this - other;
    return *this;
}

TPolynomial& TPolynomial::operator-=(const TMonomial& monomial)
{
    *this = *this - monomial;
    return *this;
}

TPolynomial& TPolynomial::operator-=(double number)
{
    *this = *this - number;
    return *this;
}

TPolynomial& TPolynomial::operator*=(const TPolynomial& other)
{
    *this = *this * other;
    return *this;
}
```

```
TPolynomial& TPolynomial::operator*=(const TMonomial& monomial)
{
    *this = *this * monomial;
    return *this;
}

TPolynomial& TPolynomial::operator*=(double number)
{
    *this = *this * number;
    return *this;
}

std::ostream& operator<<(std::ostream& stream, const TPolynomial& polynomial)
{
    if (polynomial.monomials->empty())
        return stream << "0";
    TMonomialList::iterator i = polynomial.monomials->begin();
    stream << polynomial.monomToStr(TMonomial((*i)->data, (*i)->key));
    for (i++; i != polynomial.monomials->end(); i++)
    {
        TMonomial* monomial = *i;
        if (monomial->data >= 0)
            stream << " + ";
        else
            stream << " - ";
        stream << polynomial.monomToStr(TMonomial(fabs(monomial->data),
monomial->key));
    }
    return stream;
}

std::istream& operator>>(std::istream& stream, TPolynomial& polynomial)
{
    std::string str;
    std::getline(stream, str, '\n');
    polynomial.parse(str.c_str());
    return stream;
}

TPolynomial operator""_poly(const char* literal, size_t)
{
    return TPolynomial(literal);
}
```