

Министерство образования и науки 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

Отчёт по лабораторной работе

Аналитические преобразования полиномов от нескольких переменных

Выполнил:
студент гр. 381806-1

Сидорова А.К.

Проверил:
доцент каф. МОСТ, ИИТММ

Кустикова В.Д.

Нижний Новгород
2019 г.

Содержание

Введение	3
Постановка задачи	4
Руководство пользователя	5
Руководство программиста.....	6
Описание структуры программы	6
Описание структур данных	6
Описание алгоритмов.....	9
Заключение.....	12
Литература	13
Приложения	14
Приложение 1. Исходный код основной функции.....	14
Приложение 2. Класс Exceptions.....	17
Приложение 3. Класс TNode	17
Приложение 4. Класс TNode <int, float>	18
Приложение 5. Класс TList.....	19
Приложение 6. Класс TList <int, float>.....	28
Приложение 7. Класс TPolinom.....	39

Введение

Лабораторная работа направлена на разработку системы для арифметических действий над многочленами от трех переменных с помощью списков. Для этого введем несколько необходимых понятий.

Моном (также одночлен) – простое математическое выражение, прежде всего рассматриваемое и используемое в элементарной алгебре, а именно, произведение, состоящее из числового множителя и одной или нескольких переменных, взятых каждая в неотрицательной целой степени.

Мономом также считается каждое отдельное число (без буквенных множителей), причём степень такого одночлена равняется нулю.

Полином (или многочлен) от n переменных – это сумма одночленов или, строго, – конечная формальная сумма вида $\sum_I C_I * x_1^{i_1} * x_2^{i_2} * x_3^{i_3} * \dots * x_n^{i_n}$, где $I = (i_1, i_2, \dots, i_n)$ – набор всевозможных целых неотрицательных чисел (мультииндекс), C_I – число, (именуемое коэффициентом многочлена) зависящее только от мультииндекса I .

В данной лабораторной работе будет реализована программа для работы с полиномами от трех переменных: $\sum_I C_I * x^i * y^j * z^k$, где $0 \leq i, j, k \leq 9$

Постановка задачи

В рамках данной лабораторной работы необходимо было реализовать систему для выполнения арифметических действий над многочленами трех переменных с помощью списков.

Таким образом, в программе требуется реализовать:

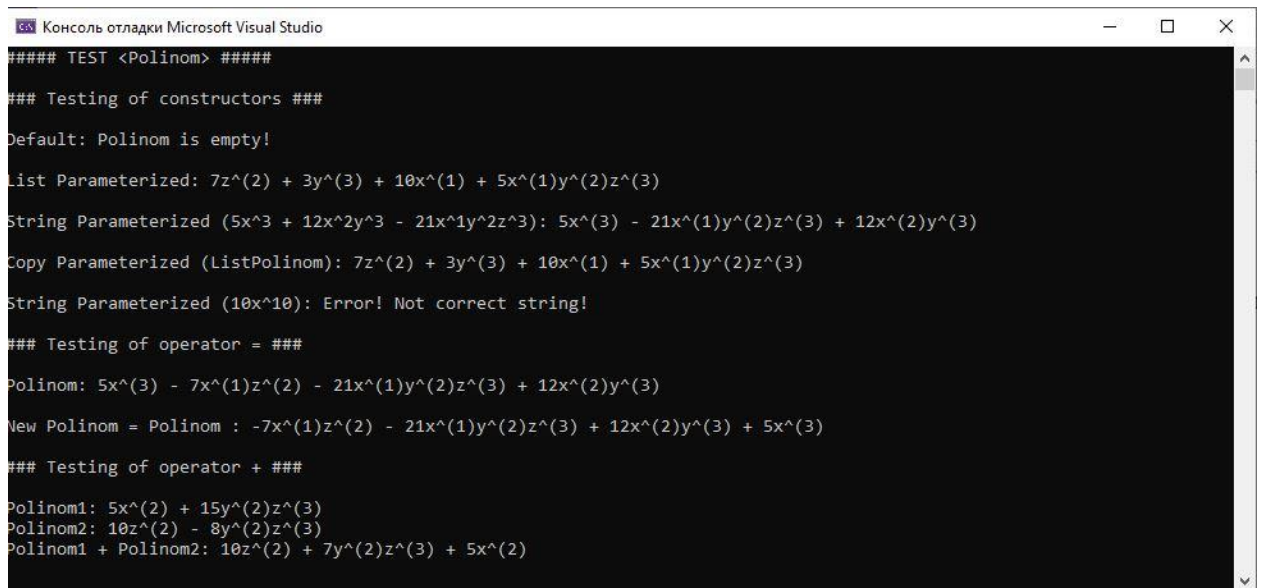
- Организация хранения полинома;
- Преобразование введенной с консоли строки в полином;
- Преобразование списка в полином;
- Удаление введенного ранее полинома;
- Копирование полинома;
- Сумма двух полиномов;
- Разность двух полиномов;
- Произведение двух полиномов;
- Вывод.

При выполнении лабораторной работы используются следующие основные предположения:

- Разработка структуры хранения должна быть ориентирована на представление полиномов от трех неизвестных.
- Степени переменных полиномов не могут превышать значения 9, т.е. $0 \leq i, j, k \leq 9$.
- Число мономов в полиномах существенно меньше максимально возможного количества (тем самым, в структуре хранения должны находиться только мономы с ненулевыми коэффициентами).

Руководство пользователя

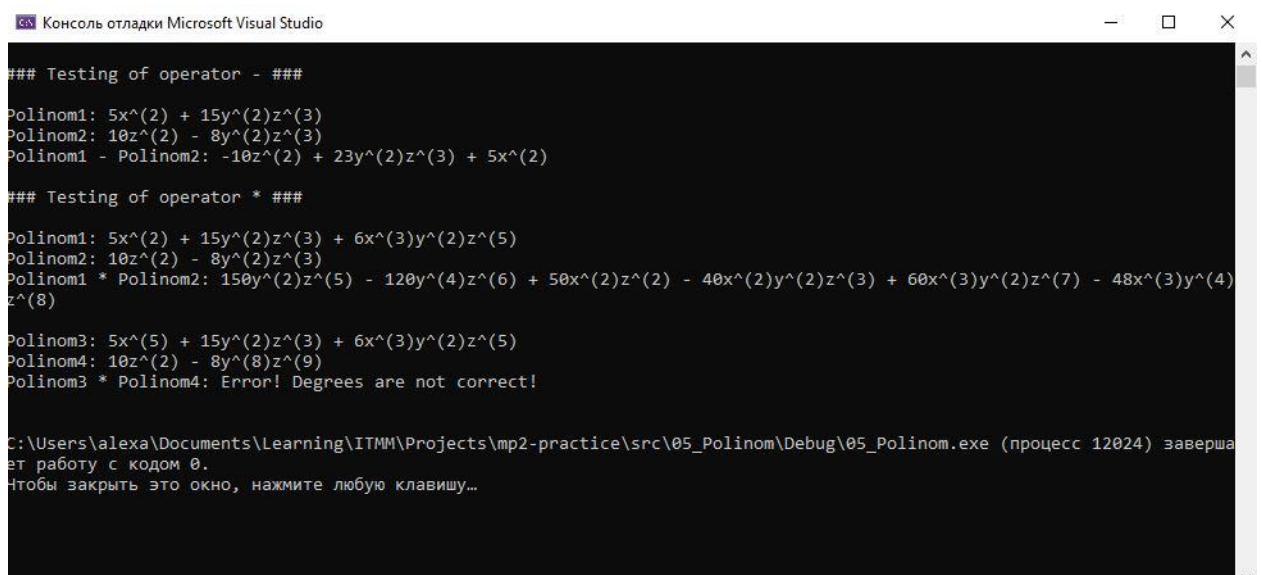
В лабораторной работе реализовано тестирование, которое запускается при включении программы.



```
Консоль отладки Microsoft Visual Studio
#### TEST <Polinom> ####
### Testing of constructors ###
Default: Polinom is empty!
List Parameterized: 7z^(2) + 3y^(3) + 10x^(1) + 5x^(1)y^(2)z^(3)
String Parameterized (5x^3 + 12x^2y^3 - 21x^1y^2z^3): 5x^(3) - 21x^(1)y^(2)z^(3) + 12x^(2)y^(3)
Copy Parameterized (ListPolinom): 7z^(2) + 3y^(3) + 10x^(1) + 5x^(1)y^(2)z^(3)
String Parameterized (10x^10): Error! Not correct string!
### Testing of operator = ###
Polinom: 5x^(3) - 7x^(1)z^(2) - 21x^(1)y^(2)z^(3) + 12x^(2)y^(3)
New Polinom = Polinom : -7x^(1)z^(2) - 21x^(1)y^(2)z^(3) + 12x^(2)y^(3) + 5x^(3)
### Testing of operator + ###
Polinom1: 5x^(2) + 15y^(2)z^(3)
Polinom2: 10z^(2) - 8y^(2)z^(3)
Polinom1 + Polinom2: 10z^(2) + 7y^(2)z^(3) + 5x^(2)
```

Рис. 1 Демонстрация работы программы.

Таким образом, пользователь может увидеть работоспособность программы, ее возможности и проверить на корректность. Например, на Рис. 1 представлены тесты всех конструкторов: по умолчанию, по параметрам (по списку и по строке), копирования; а также тесты перегрузки таких операций, как равенство и сложение. Более того, на Рис. 2 есть продолжение тестирующей программы: тесты перегрузок таких операций, как вычитания и умножения.



```
Консоль отладки Microsoft Visual Studio
### Testing of operator - ###
Polinom1: 5x^(2) + 15y^(2)z^(3)
Polinom2: 10z^(2) - 8y^(2)z^(3)
Polinom1 - Polinom2: -10z^(2) + 23y^(2)z^(3) + 5x^(2)
### Testing of operator * ###
Polinom1: 5x^(2) + 15y^(2)z^(3) + 6x^(3)y^(2)z^(5)
Polinom2: 10z^(2) - 8y^(2)z^(3)
Polinom1 * Polinom2: 150y^(2)z^(5) - 120y^(4)z^(6) + 50x^(2)z^(2) - 40x^(2)y^(2)z^(3) + 60x^(3)y^(2)z^(7) - 48x^(3)y^(4)z^(8)
Polinom3: 5x^(5) + 15y^(2)z^(3) + 6x^(3)y^(2)z^(5)
Polinom4: 10z^(2) - 8y^(8)z^(9)
Polinom3 * Polinom4: Error! Degrees are not correct!
C:\Users\alexa\Documents\Learning\ITMM\Projects\mp2-practice\src\05_Polinom\Debug\05_Polinom.exe (процесс 12024) завершил работу с кодом 0.
Чтобы закрыть это окно, нажмите любую клавишу...
```

Рис. 2 Демонстрация работы программы.

Стоит отметить, что случаи, когда возможны исключения, в программе реализованы.

Руководство программиста

Описание структуры программы

Программа состоит из следующих файлов:

- Includes\exceptions.h – заголовочный файл, который содержит в себе описание и реализацию собственного класса исключений.
- Includes\TNode.h – заголовочный файл, в котором описан класс звена списка, как структуры данных.
- Includes\TNode_TMonom.h – заголовочный файл, в котором описана специализация класса звена списка для возможной работы с полиномами.
- Includes\TList.h – заголовочный файл, в котором описан класс списка, как структуры данных.
- Includes\TList_TPolinom.h – заголовочный файл, в котором описана специализация класса списка для возможной работы с полиномами.
- Includes\TPolinom.h – заголовочный файл, в котором реализован класс полинома.
- main.cpp – исполняемый файл, в котором содержится главная функция программы с тестами.

Описание структур данных

1. Класс Exceptions – класс исключений.

Класс содержит в себе:

- `string msg` – переменная, которая хранит в себе строку с информацией об ошибке.
- `Exception(string _msg) : msg(_msg) {}` – конструктор с параметром.
- `const char* what() const noexcept` – константный метод, который возвращает строку, содержащую в себе информацию об ошибке.

2. Класс TNode – шаблонный класс звена списка.

Класс содержит в себе:

- `TKey key` – переменная – ключ.
- `TData* pData` – указатель на область памяти, в которой хранятся данные.
- `TNode<TKey, TData>* pNext` – указатель на следующее звено.
- `TNode()` – конструктор по умолчанию.
- `TNode(TKey _key, TData* _data, TNode* _node = NULL)` – конструктор с параметрами.
- `TNode(const TNode&)` – конструктор копирования.
- `~TNode()` – деструктор.

3. Класс TNode <int, float> – специализация шаблонного класса TNode для работы с полиномами.

Класс содержит в себе:

- `int key` – переменная, которая хранит в себе степень монома.
- `float pData` – переменная, которая хранит в себе коэффициент монома.

- `TNode<int, float>* pNext` – указатель на следующее звено (моном).
- `TNode()` – конструктор по умолчанию.
- `TNode(int _key, float _data, TNode* _node = NULL)` – конструктор с параметрами.
- `TNode(const TNode&)` – конструктор копирования.
- `~TNode()` – деструктор.

4. Класс **TList** – шаблонный класс списка.

Класс содержит в себе:

- `TNode<TKey, TData>* pFirst` – указатель на первое звено списка.
- `TNode<TKey, TData>* pCurrent` – указатель на текущее звено списка.
- `TNode<TKey, TData>* pNext` – указатель на следующее звено списка.
- `TNode<TKey, TData>* pPrev` – указатель на предыдущее звено списка.
- `TList()` – конструктор по умолчанию.
- `TList(const TList&)` – конструктор копирования.
- `TList(const TNode<TKey, TData>*)` – конструктор с параметром.
- `~TList()` – деструктор.
- `void Reset()` – метод, который возвращает указатель на текущее звено в начало.
- `void Next()` – метод, который двигает указатель на текущее звено к следующему за ним.
- `bool IsEnded() const` – константный метод, который проверяет, достигнут ли конец списка, то есть является ли указатель на текущее звено нулевым.
- `TNode<TKey, TData>* GetpFirst() const` – константный метод, который возвращает указатель на начало списка.
- `TNode<TKey, TData>* Search(TKey)` – метод, который находит звено списка по заданному ключу и его возвращает.
- `void PushBegin(TKey, TData*)` – метод, который вставляет в начало списка звено с заданным ключом и указателем на данные.
- `void PushEnd(TKey, TData*)` – метод, который вставляет в конец списка звено с заданным ключом и указателем на данные.
- `void PushBefore(TKey, TKey, TData*)` – метод, который вставляет в звено с заданным ключом и указателем на данные перед звеном с указанным ключом.
- `void PushAfter(TKey, TKey, TData*)` – метод, который вставляет в звено с заданным ключом и указателем на данные после звена с указанным ключом.
- `void Delete(TKey)` – метод, который удаляет звено из списка с заданным ключом.
- `template<typename TKey, class TData> friend ostream& operator<<(ostream&, TList<TKey, TData>&)` – перегрузка вывода.

5. Класс `TList<int, float>` – специализация шаблонного класса списка для работы с полиномами.

Класс содержит в себе:

- `TNode<int, float>* pFirst` – указатель на первое звено списка.
- `TNode<int, float>* pCurrent` – указатель на текущее звено списка.
- `TNode<int, float>* pNext` – указатель на следующее звено списка.
- `TNode<int, float>* pPrev` – указатель на предыдущее звено списка.
- `TList()` – конструктор по умолчанию.
- `TList(const TList&)` – конструктор копирования.
- `TList(const TNode<int, float>*)` – конструктор с параметром.
- `~TList()` – деструктор.
- `void Reset()` – метод, который возвращает указатель на текущее звено в начало.
- `void Next()` – метод, который двигает указатель на текущее звено к следующему за ним.
- `bool IsEnded() const` – константный метод, который проверяет, достигнут ли конец списка, то есть является ли указатель на текущее звено нулевым.
- `TNode<int, float>* GetpFirst() const` – константный метод, который возвращает указатель на начало списка.
- `TNode<int, float>* GetpCurrent() const` – константный метод, который возвращает указатель на текущее звено списка.
- `void SortKey()` – метод, который сортирует по возрастанию список по ключам.
- `TNode<int, float>* Search(int)` – метод, который находит звено списка по заданному ключу и его возвращает.
- `void PushBegin(int, float)` – метод, который вставляет в начало списка звено с заданным ключом и указателем на данные.
- `void PushEnd(int, float)` – метод, который вставляет в конец списка звено с заданным ключом и указателем на данные.
- `void PushBefore(int, int, float)` – метод, который вставляет в звено с заданным ключом и указателем на данные перед звеном с указанным ключом.
- `void PushAfter(int, int, float)` – метод, который вставляет в звено с заданным ключом и указателем на данные после звена с указанным ключом.
- `void Delete(TKey)` – метод, который удаляет звено из списка с заданным ключом.
- `friend ostream& operator<<(ostream&, TList<int, float>&)` – перегрузка вывода.

6. Класс `TPolinom` – класс полинома.

Класс содержит в себе:

- `TList<int, float>* monoms` – указатель на список мономов.
- `void CastToDefault()` – метод, который приводит полином к стандартному виду.
- `bool IsOperation(const char) const` – константный метод, который проверяет, данный символ является операцией в полиноме или нет.
- `TPolinom()` – конструктор по умолчанию.
- `TPolinom(const string)` – конструктор с параметром.
- `TPolinom(TList<int, float>*)` – конструктор с параметром.
- `TPolinom(const TPolinom&)` – конструктор копирования.
- `~TPolinom()` – деструктор.
- `const TPolinom& operator=(const TPolinom&)` – перегрузка операции '='.
- `TPolinom operator+(const TPolinom&)` – перегрузка операции '+'.
`TPolinom operator-(const TPolinom&)` – перегрузка операции '-'.
`TPolinom operator*(const TPolinom&)` – перегрузка операции '*'.
`TPolinom operator-()` – перегрузка унарного минуса.
- `friend ostream& operator<<(ostream&, TPolinom&)` – перегрузка вывода.

Описание алгоритмов

Описание структуры список.

В данной лабораторной работе реализована такая структура данных, как список. Список представляет собой сцепление звеньев памяти. В свою очередь, звено – это некоторый блок, который хранит в себе ключ, указатель на область памяти, в которой хранятся некоторые данные, и указатель на следующее звено (Рис. 3).

Таким образом, список – набор сцепленных между собой последовательно звеньев (Рис. 4).

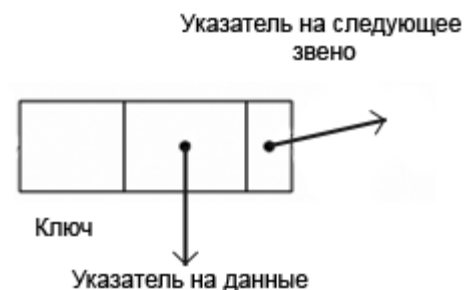


Рис. 3 Звено списка.

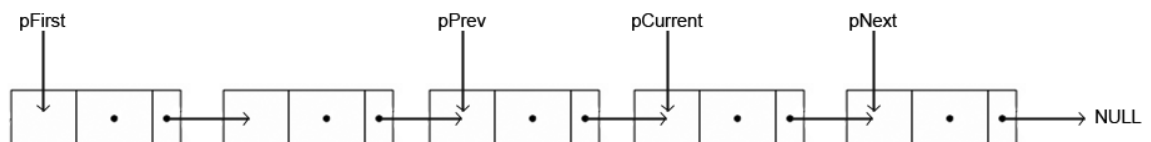


Рис. 4 Список

На данном рисунке представлен список, который задается указателем на первый элемент списка – `pFirst`. Для более удобной работы с такой структурой данной существуют

еще три указателя: `pCurrent` (указатель на текущий элемент списка), `pPrev` (указатель на предыдущий элемент списка, то есть на тот, что является предыдущим для `pCurrent`) и `pNext` (указатель на следующий элемент списка, то есть на тот, что следует после `pCurrent`).

Для работы со списком реализованы такие методы, как:

- **Поиск элемента списку по ключу:**
По списку выполняется проход до тех пор, пока не встретим звено с ключом, которое заданное. Его и возвращаем. Если дошли до конца списка и такое звено не обнаружено, то возвращаем нулевой указатель.
- **Удаление элемента списка по ключу:**
Сначала проверяем первый элемент списка на соответствие ключа. Если это он, то элемент удаляется, а указатель на первый элемент списка переходит к следующему звену. В противном случае по списку выполняется проход до тех пор, пока не встретится звено с указанным ключом. Указатель на следующее звено предыдущего элемента от найденного ссылаем на следующее звено для найденного, а его самого удаляем. Если проход совершен, а элемент не найден, то бросается исключение.
- **Добавить элемент в начало списка:**
Указатель следующего звена в новом элементе мы ссылаем на первый элемент списка (или на нулевой, если список пуст), а указатель на начало списка переадресуем на новое звено.
- **Добавить элемент в конец списка:**
Сначала проверяем, если список пуст, то указатель на начало списка ссылаем на новый элемент. В противном случае доходим до конца списка, последнему звену указатель на следующее звено ссылаем на новое звено, а у нового звена тот же указатель приравниваем к нулю.
- **Добавить элемент перед заданным элементом списка:**
Проходим по списку в поиске нужного элемента. Его предыдущему звену меняем указатель на следующий элемент на новое звено, а новому звену ставим тот же указатель на текущий, найденный элемент списка. Если проход совершен, а элемент не найден, то бросается исключение.
- **Добавить элемент после заданного элемента списка:**
Проходим по списку в поиске нужного элемента. Его указатель на следующий элемент меняем на новое звено, а новому звену ставим тот же указатель на следующий элемент списка от текущего. Если проход совершен, а элемент не найден, то бросается исключение.
- **Сортировка списка:**
Проходим по списку в поиску элемента с наименьшим ключом. Его кладем в конец нового, временного списка. Запоминаем указатель на этот элемент, как временный указатель на первый элемент списка. А сам этот элемент из списка удаляем путем перезаписывания указателя предыдущего от него элемента на следующее звено на тот элемент, который шел после минимального звена. Затем аналогично проходим по всему списку, добавляя новые минимумы в конец временного списка. Когда список кончится, его указатель на первый элемент ссылаем на указатель временного первого элемента отсортированного списка.

Описание структуры полином.

Структура данных полином представляет собой список, элементами которых являются мономы. У мономов в свою очередь ключ является степенью, которая целочисленная и не превышает значения 999, но и не ниже значения 0. Степень представляет собой число `хуз`, где каждый символ – это степень при данной переменной.

Кроме того, данными для мономов являются их коэффициенты, которые могут быть вещественными.

Для полиномов реализованы методы:

- Создание полинома из строки;
- Сумма двух полиномов;
- Разность двух полиномов;
- Произведение двух полиномов;
- Перегружена операция «унарный минус» для полиномов;
- Приведение к стандартному виду.

Заключение

В результате предложенной лабораторной работы была разработана программа с заголовочными файлами, реализующие классы полиномов и мономов, как специализации классов списка и элемента списка. Они позволяют создать объект класса `TPolynomial` и выполнить с ним некоторые операции, задача реализации которых была поставлена в начале данной лабораторной работы.

Кроме того, были разработаны и доведены до успешного выполнения тесты.

Программное решение было продемонстрировано с помощью простейшего набора операций над полиномами. Описание примера работы с полиномами было представлено в разделе «Руководство пользователя».

Литература

1. Электронный ресурс.
 - 1.1. Алгоритмы, методы, исходники [Электронный ресурс] // URL:
<http://algolist.manual.ru/>
2. Учебники, рабочие материалы.
 - 2.1. Гергель В. П. Рабочие материалы к учебному курсу «Методы программирования», часть 1 – Нижний Новгород.

Приложения

Приложение 1. Исходный код основной функции

```
#include "Includes/TList_TPolinom.h"
#include "Includes/TPolinom.h"
#include "Includes/exceptions.h"

#include <conio.h>
#include <iostream>

void main()
{
    cout << "##### TEST <Polinom> #####" << endl << endl;

    try
    {
        cout << "### Testing of constructors ###" << endl << endl;

        cout << "Default: ";
        TPolinom defPolinom;
        cout << defPolinom << endl;

        cout << "List Parameterized: ";
        TNode<int, float>* node3 = new TNode<int, float>(123, 5);
        TNode<int, float>* node2 = new TNode<int, float>(100, 10, node3);
        TNode<int, float>* node1 = new TNode<int, float>(30, 3, node2);
        TNode<int, float>* node0 = new TNode<int, float>(2, 7, node1);
        TList<int, float>* listParametr = new TList<int, float>(node0);
        TPolinom listPolinom(listParametr);
        cout << listPolinom << endl;

        cout << "String Parameterized (5x^3 + 12x^2y^3 - 21x^1y^2z^3 + 10): ";
        string str1 = "5x^3 + 12x^2y^3 - 21x^1y^2z^3 + 10";
        TPolinom stringPolinom1(str1);
        cout << stringPolinom1 << endl;

        cout << "Copy Parameterized (ListPolinom): ";
        TPolinom copyPolinom(listPolinom);
        cout << copyPolinom << endl;

        cout << "String Parameterized (10x^10): ";
        string str2 = "10x^10";
        TPolinom stringPolinom2(str2);
        cout << stringPolinom2 << endl;
```

```

    }
    catch (Exception ex)
    {
        cout << ex.what() << endl << endl;
    }

    try
    {
        cout << "### Testing of operator = ###" << endl << endl;

        cout << "Polinom: ";
        TPolinom stringPolinom("5x^3 + 12x^2y^3 - 21x^1y^2z^3 - 7z^2x^1");
        cout << stringPolinom << endl;

        cout << "New Polinom = Polinom : ";
        TPolinom newPolinom = stringPolinom;
        cout << newPolinom << endl;
    }
    catch (Exception ex)
    {
        cout << ex.what() << endl << endl;
    }

    try
    {
        cout << "### Testing of operator + ###" << endl << endl;

        cout << "Polinom1: ";
        TPolinom polinom1("5x^2 + 15y^2z^3");
        cout << polinom1;
        cout << "Polinom2: ";
        TPolinom polinom2("10z^2 - 8y^2z^3");
        cout << polinom2;
        cout << "Polinom1 + Polinom2: ";
        TPolinom sum = polinom1 + polinom2;
        cout << sum << endl;
    }
    catch (Exception ex)
    {
        cout << ex.what() << endl << endl;
    }

    try
    {
        cout << "### Testing of operator - ###" << endl << endl;

```

```

        cout << "Polinom1: ";
        TPolinom polinom1("5x^2 + 15y^2z^3");
        cout << polinom1;
        cout << "Polinom2: ";
        TPolinom polinom2("10z^2 - 8y^2z^3");
        cout << polinom2;
        cout << "Polinom1 - Polinom2: ";
        TPolinom sub = polinom1 - polinom2;
        cout << sub << endl;
    }
    catch (Exception ex)
    {
        cout << ex.what() << endl << endl;
    }

    try
    {
        cout << "### Testing of operator * ###" << endl << endl;

        cout << "Polinom1: ";
        TPolinom polinom1("5x^2 + 15y^2z^3 + 6x^3y^2z^5");
        cout << polinom1;
        cout << "Polinom2: ";
        TPolinom polinom2("10z^2 - 8y^2z^3");
        cout << polinom2;
        cout << "Polinom1 * Polinom2: ";
        TPolinom sub1 = polinom1 * polinom2;
        cout << sub1 << endl;

        cout << "Polinom3: ";
        TPolinom polinom3("5x^5 + 15y^2z^3 + 6x^3y^2z^5");
        cout << polinom3;
        cout << "Polinom4: ";
        TPolinom polinom4("10z^2 - 8y^8z^9");
        cout << polinom4;
        cout << "Polinom3 * Polinom4: ";
        TPolinom sub2 = polinom3 * polinom4;
        cout << sub2 << endl;
    }
    catch (Exception ex)
    {
        cout << ex.what() << endl << endl;
    }
};

```


Приложение 2. Класс Exceptions

```
#ifndef _EXCEPTIONS_H_
#define _EXCEPTIONS_H_

#include <iostream>
#include <string>
#include <exception>

using namespace std;

class Exception : public exception
{
private:
    string msg;
public:
    Exception(string _msg) : msg(_msg) {};

    const char* what() const noexcept
    {
        return msg.c_str();
    }
};

#endif
```

Приложение 3. Класс TNode

```
#ifndef _TNODE_H_
#define _TNODE_H_

template<typename TKey, class TData>
class TNode
{
public:
    TKey key;
    TData* pData;
    TNode<TKey, TData>* pNext;

    TNode();
    TNode(TKey _key, TData* _data, TNode* _node = NULL);
    TNode(const TNode&);
    ~TNode();
};

//-----

template<typename TKey, class TData>
TNode<TKey, TData>::TNode()
{
    key = 0;
    pData = NULL;
}
```

```

        pNext = NULL;
};

template<typename TKey, class TData>
TNode<TKey, TData>::TNode(TKey _key, TData* _data, TNode<TKey, TData>* _node)
{
    key = _key;
    pData = _data;
    pNext = _node;
};

template<typename TKey, class TData>
TNode<TKey, TData>::TNode(const TNode<TKey, TData>& _copy)
{
    key = _copy.key;
    pData = _copy.pData;
    pNext = _copy.pNext;
};

template<typename TKey, class TData>
TNode<TKey, TData>::~~TNode()
{
    key = 0;
    pData = NULL;
    pNext = NULL;
};

#endif

```

Приложение 4. Класс TNode <int, float>

```

#ifndef _TNODE_TMONOM_H_
#define _TNODE_TMONOM_H_

#include "exceptions.h"
#include "TNode.h"

template<>
class TNode <int, float>
{
public:
    int key;    // degree, 3 symbols (xyz)
    float pData; // coeff
    TNode<int, float>* pNext;

    TNode() : key(0), pData(0), pNext(NULL) {}

```

```

        TNode(int _key, float _data, TNode* _node = NULL) : key(_key), pData(_data),
pNext(_node) {}

        TNode(const TNode& _copy) : key(_copy.key), pData(_copy.pData),
pNext(_copy.pNext) {}

        ~TNode()
        {
            key = 0;
            pData = 0;
            pNext = NULL;
        }
};

#endif

```

Приложение 5. Класс TList

```

#ifndef _TLIST_H_
#define _TLIST_H_

#include <iostream>

#include "TNode.h"
#include "exceptions.h"

template<typename TKey, class TData>
class TList
{
private:
    TNode<TKey, TData>* pFirst;
    TNode<TKey, TData>* pCurrent;
    TNode<TKey, TData>* pNext;
    TNode<TKey, TData>* pPrev;

public:
    TList();
    TList(const TList&);
    TList(const TNode<TKey, TData>*);
    ~TList();

    void Reset();
    void Next();
    bool IsEnded() const;
    TNode<TKey, TData>* GetpFirst() const;

    TNode<TKey, TData>* Search(TKey);
    void PopBegin(TKey, TData*);
    void PopEnd(TKey, TData*);

```

```

    void PopBefore(TKey, TKey, TData*);
    void PopAfter(TKey, TKey, TData*);
    void Delete(TKey);

    template<typename TKey, class TData> friend ostream& operator<<(ostream&,
TList<TKey,TData>&);
};
//-----

template<typename TKey, class TData>
TList<TKey, TData>::TList()
{
    pFirst = pNext = pPrev = pCurrent = NULL;
};

template<typename TKey, class TData>
TList<TKey, TData>::TList(const TList& _copy)
{
    pNext = pPrev = pCurrent = NULL;

    if (!_copy.pFirst)
        pFirst = NULL;
    else
    {
        pFirst = new TNode<TKey, TData>(*_copy.pFirst);
        pFirst->pNext = NULL;
        pCurrent = pFirst;

        TNode<TKey, TData>* iter = new TNode<TKey, TData>;
        iter = _copy.pFirst;

        while (iter->pNext)
        {
            iter = iter->pNext;
            pCurrent->pNext = new TNode<TKey, TData>(*iter);

            pPrev = pCurrent;
            pCurrent = pCurrent->pNext;
            pNext = pCurrent->pNext = NULL;
        }

        pPrev = NULL;
        pCurrent = pFirst;
        pNext = pFirst->pNext;
    }
}

```

```

};

template<typename TKey, class TData>
TList<TKey, TData>::TList(const TNode<TKey, TData>* _node)
{
    pNext = pPrev = pCurrent = NULL;

    if (!_node)
        pFirst = NULL;
    else
    {
        TNode<TKey, TData>* node = new TNode<TKey, TData>(*_node);
        pFirst = node;

        TNode<TKey, TData>* iter = _node->pNext;
        TNode<TKey, TData>* prev = pFirst;

        while (iter)
        {
            TNode<TKey, TData>* tmp = new TNode<TKey, TData>(*iter);
            prev->pNext = tmp;
            prev = tmp;
            iter = iter->pNext;
        }

        pCurrent = pFirst;
        pNext = pCurrent->pNext;
    }
};

template<typename TKey, class TData>
TList<TKey, TData>::~~TList()
{
    this->Reset();
    while (!this->IsEnded())
    {
        this->Next();
        delete pPrev;
    }

    delete pCurrent;

    pFirst = pNext = pPrev = pCurrent = NULL;
};
//-----

```

```

template<typename TKey, class TData>
void TList<TKey, TData>::Reset()
{
    pPrev = NULL;
    pCurrent = pFirst;
    if (pFirst)
        pNext = pCurrent->pNext;
    else
        pNext = NULL;
};

template<typename TKey, class TData>
void TList<TKey, TData>::Next()
{
    pPrev = pCurrent;
    pCurrent = pNext;

    if (pCurrent)
        pNext = pCurrent->pNext;
    else
        pNext = NULL;
};

template<typename TKey, class TData>
bool TList<TKey, TData>::IsEnded() const
{
    return (pCurrent == NULL);
};

template<typename TKey, class TData>
TNode<TKey, TData>* TList<TKey, TData>::GetpFirst() const
{
    return pFirst;
};
//-----

template<typename TKey, class TData>
TNode<TKey, TData>* TList<TKey, TData>::Search(TKey _key)
{
    TNode<TKey, TData>* tmppCurrent = pCurrent;
    TNode<TKey, TData>* tmppNext = pNext;
    TNode<TKey, TData>* tmppPrev = pPrev;

    this->Reset();
}

```

```

while (!this->IsEnded())
{
    if (_key == pCurrent->key)
    {
        TNode<TKey, TData>* findNode = pCurrent;
        pCurrent = tmppCurrent;
        pNext = tmppNext;
        pPrev = tmppPrev;

        return findNode;
    }

    this->Next();
}

pCurrent = tmppCurrent;
pNext = tmppNext;
pPrev = tmppPrev;

return NULL;
};

template<typename TKey, class TData>
void TList<TKey, TData>::PopBegin(TKey _key, TData* _data)
{
    TNode<TKey, TData>* newNode = new TNode<TKey, TData>(_key, _data, pFirst);

    if (pCurrent == pFirst)
        pPrev = newNode;

    pFirst = newNode;
};

template<typename TKey, class TData>
void TList<TKey, TData>::PopEnd(TKey _key, TData* _data)
{
    TNode<TKey, TData>* tmppCurrent = pCurrent;
    TNode<TKey, TData>* tmppNext = pNext;
    TNode<TKey, TData>* tmppPrev = pPrev;

    this->Reset();

    while (pNext)
        this->Next();
}

```

```

TNode<TKey, TData>* newNode = new TNode<TKey, TData>(_key, _data);

if (!pFirst)
    pFirst = newNode;
else
    pCurrent->pNext = newNode;

if (tmppCurrent == pCurrent)
    pNext = newNode;
else
    pNext = tmppNext;

pCurrent = tmppCurrent;
pPrev = tmppPrev;
};

template<typename TKey, class TData>
void TList<TKey, TData>::PopBefore(TKey _superKey, TKey _key, TData* _data)
{
    TNode<TKey, TData>* tmppCurrent = pCurrent;
    TNode<TKey, TData>* tmppNext = pNext;
    TNode<TKey, TData>* tmppPrev = pPrev;

    this->Reset();

    if ((this->IsEnded()) || (pFirst->key == _superKey))
    {
        this->PopBegin(_key, _data);
        pCurrent = pFirst;
        return;
    }

    TNode<TKey, TData>* nodeFind = Search(_superKey);

    if (!nodeFind)
    {
        throw Exception("Error: key didn't find!");
        return;
    }

    while (pCurrent != nodeFind)
        this->Next();

    TNode<TKey, TData>* newNode = new TNode<TKey, TData>(_key, _data, pCurrent);

```



```

    pPrev->pNext = newNode;

    if (tmppCurrent == pPrev)
        pNext = newNode;
    else
        pNext = tmppNext;

    if (tmppCurrent == pCurrent)
        pPrev = newNode;
    else
        pPrev = tmppPrev;

    pCurrent = tmppCurrent;
};

template<typename TKey, class TData>
void TList<TKey, TData>::PopAfter(TKey _superKey, TKey _key, TData* _data)
{
    TNode<TKey, TData>* tmppCurrent = pCurrent;
    TNode<TKey, TData>* tmppNext = pNext;
    TNode<TKey, TData>* tmppPrev = pPrev;

    this->Reset();

    TNode<TKey, TData>* nodeFind = Search(_superKey);

    if (!nodeFind)
    {
        throw Exception("Error: key didn't find!");
        return;
    }

    while (pCurrent != nodeFind)
        this->Next();

    TNode<TKey, TData>* newNode = new TNode<TKey, TData>(_key, _data, pNext);
    pCurrent->pNext = newNode;

    if (tmppCurrent == pCurrent)
        pNext = newNode;
    else
        pNext = tmppNext;

    if (tmppCurrent == pNext)
        pPrev = newNode;

```

```

        else
            pPrev = tmppPrev;

        pCurrent = tmppCurrent;
    };

template<typename TKey, class TData>
void TList<TKey, TData>::Delete(TKey _key)
{
    if (!pFirst)
        throw Exception("Error: list is empty!");

    if (pFirst->key == _key)
    {
        if (pCurrent == pFirst)
        {
            pCurrent = pNext;
            if (pNext)
                pNext = pNext->pNext;
            else
                pNext = NULL;

            delete pFirst;
            pFirst = pCurrent;

            return;
        }

        if (pCurrent == pFirst->pNext)
        {
            pPrev = NULL;

            delete pFirst;
            pFirst = pCurrent;

            return;
        }

        delete pFirst;

        return;
    }

    TNode<TKey, TData>* tmppCurrent = pCurrent;
    TNode<TKey, TData>* tmppPrev = pPrev;

```

```

TNode<TKey, TData>* tmppNext = pNext;

this->Reset();

TNode<TKey, TData>* nodeFind = Search(_key);

if (!nodeFind)
{
    throw Exception("Error: key didn't find!");
    return;
}

while (pCurrent != nodeFind)
    this->Next();

pPrev->pNext = pNext;

if (tmppCurrent == pCurrent)
{
    pCurrent = tmppNext;
    pNext = pCurrent->pNext;
    delete nodeFind;

    return;
}

if (tmppCurrent == pPrev)
{
    pCurrent = pPrev;
    pPrev = tmppPrev;
    pNext = pCurrent->pNext;
    delete nodeFind;

    return;
}

if (tmppCurrent == pNext)
{
    pCurrent = pNext;
    pNext = pCurrent->pNext;
    delete nodeFind;

    return;
}

```

```

        pNext = tmppCurrent->pNext;
        pCurrent = tmppCurrent;
        delete nodeFind;

        return;
};

template<typename TKey, class TData>
ostream& operator<<(ostream& _out, TList<TKey, TData>& _list)
{
    if (!_list.pFirst)
    {
        _out << "List is empty!" << endl;
        return _out;;
    }

    TNode<TKey, TData>* tmppCurrent = _list.pCurrent;
    TNode<TKey, TData>* tmppNext = _list.pNext;
    TNode<TKey, TData>* tmppPrev = _list.pPrev;

    _list.Reset();

    while (!_list.IsEnded())
    {
        _out << _list.pCurrent->key << " ";
        _list.Next();
    }

    _out << endl;

    _list.pCurrent = tmppCurrent;
    _list.pNext = tmppNext;
    _list.pPrev = tmppPrev;

    return _out;
};

#endif

```

Приложение 6. Класс TList <int, float>

```

#ifndef _TLIST_TPOLINOM_H_
#define _TLIST_TPOLINOM_H_

#include <iostream>

```

```

#include "TNode_TMonom.h"
#include "TList.h"
#include "exceptions.h"

template<>
class TList <int, float>
{
private:
    TNode<int, float>* pFirst;
    TNode<int, float>* pCurrent;
    TNode<int, float>* pNext;
    TNode<int, float>* pPrev;

public:
    TList();
    TList(const TList&);
    TList(const TNode<int, float>*);
    ~TList();

    void Reset();
    void Next();
    bool IsEnded() const;
    TNode<int, float>* GetpFirst() const;
    TNode<int, float>* GetpCurrent() const;

    void SortKey();
    TNode<int, float>* Search(int);
    void PushBegin(int, float);
    void PushEnd(int, float);
    void PushBefore(int, int, float);
    void PushAfter(int, int, float);
    void Delete(int);

    friend ostream& operator<<(ostream& _out, TList<int, float>& _list)
    {
        if (!_list.pFirst)
        {
            _out << "List is empty!" << endl;
            return _out;;
        }

        TNode<int, float>* tmppCurrent = _list.pCurrent;
        TNode<int, float>* tmppNext = _list.pNext;
        TNode<int, float>* tmppPrev = _list.pPrev;
    }
};

```

```

        _list.Reset();

        while (!_list.IsEnded())
        {
            _out << _list.pCurrent->pData << " ";
            _list.Next();
        }

        _out << endl;

        _list.pCurrent = tmppCurrent;
        _list.pNext = tmppNext;
        _list.pPrev = tmppPrev;

        return _out;
    };
};

//-----

TList<int, float>::TList()
{
    pFirst = NULL;
    pNext = NULL;
    pPrev = NULL;
    pCurrent = NULL;
};

TList<int, float>::TList(const TList& _copy)
{
    pNext = pPrev = pCurrent = NULL;

    if (!_copy.pFirst)
        pFirst = NULL;
    else
    {
        pFirst = new TNode<int, float>(*_copy.pFirst);
        pFirst->pNext = NULL;
        pCurrent = pFirst;

        TNode<int, float>* iter = new TNode<int, float>;
        iter = _copy.pFirst;

        while (iter->pNext)
        {
            iter = iter->pNext;

```

```

        pCurrent->pNext = new TNode<int, float>(*iter);

        pPrev = pCurrent;
        pCurrent = pCurrent->pNext;
        pNext = pCurrent->pNext = NULL;
    }

    pPrev = NULL;
    pCurrent = pFirst;
    pNext = pFirst->pNext;
}

};

TList<int, float>::TList(const TNode<int, float>* _node)
{
    pNext = pPrev = pCurrent = NULL;

    if (!_node)
        pFirst = NULL;
    else
    {
        TNode<int, float>* node = new TNode<int, float>(*_node);
        pFirst = node;

        TNode<int, float>* iter = _node->pNext;
        TNode<int, float>* prev = pFirst;

        while (iter)
        {
            TNode<int, float>* tmp = new TNode<int, float>(*iter);
            prev->pNext = tmp;
            prev = tmp;
            iter = iter->pNext;
        }

        pCurrent = pFirst;
        pNext = pCurrent->pNext;
    }
};

TList<int, float>::~~TList()
{
    this->Reset();
    while (!this->IsEnded())
    {

```

```

        this->Next();
        delete pPrev;
    }

    delete pCurrent;

    pFirst = pNext = pPrev = pCurrent = NULL;
};
//-----

void TList<int, float>::Reset()
{
    pPrev = NULL;
    pCurrent = pFirst;
    if (pFirst)
        pNext = pCurrent->pNext;
    else
        pNext = NULL;
};

void TList<int, float>::Next()
{
    pPrev = pCurrent;
    pCurrent = pNext;

    if (pCurrent)
        pNext = pCurrent->pNext;
    else
        pNext = NULL;
};

bool TList<int, float>::IsEnded() const
{
    return (pCurrent == NULL);
};

TNode<int, float>* TList<int, float>::GetpFirst() const
{
    return pFirst;
};

TNode<int, float>* TList<int, float>::GetpCurrent() const
{
    return pCurrent;
};

```



```

//-----

void TList<int, float>::SortKey()
{
    TNode<int, float>* last = new TNode<int, float>;
    TNode<int, float>* tmpfirst = new TNode<int, float>;
    bool first = true;

    while (this->pFirst)
    {
        this->Reset();
        TNode<int, float>* min = new TNode<int, float>;
        min = pFirst;

        while (!this->IsEnded())
        {
            if (pCurrent->key < min->key)
                min = pCurrent;
            this->Next();
        }

        this->Reset();
        while ((this->pCurrent != min) && (!this->IsEnded()))
            this->Next();

        if (pCurrent == pFirst)
            pFirst = pCurrent = pNext;
        else if (pNext == NULL)
            pPrev->pNext = NULL;
        else
            pPrev->pNext = pNext;
        this->Reset();

        if (first)
        {
            tmpfirst = min;
            first = false;
        }
        else last->pNext = min;
        last = min;
    }

    last->pNext = NULL;
    pFirst = tmpfirst;
    pPrev = NULL;
}

```

```

        pCurrent = pFirst;
        pNext = pCurrent->pNext;
    };

TNode<int, float>* TList<int, float>::Search(int _key)
{
    TNode<int, float>* tmppCurrent = pCurrent;
    TNode<int, float>* tmppNext = pNext;
    TNode<int, float>* tmppPrev = pPrev;

    this->Reset();

    while (!this->IsEnded())
    {
        if (_key == pCurrent->key)
        {
            TNode<int, float>* findNode = pCurrent;
            pCurrent = tmppCurrent;
            pNext = tmppNext;
            pPrev = tmppPrev;

            return findNode;
        }

        this->Next();
    }

    pCurrent = tmppCurrent;
    pNext = tmppNext;
    pPrev = tmppPrev;

    return NULL;
};

void TList<int, float>::PushBegin(int _key, float _data)
{
    TNode<int, float>* newNode = new TNode<int, float>(_key, _data, pFirst);

    if (pCurrent == pFirst)
        pPrev = newNode;

    pFirst = newNode;
};

void TList<int, float>::PushEnd(int _key, float _data)

```

```

{
    TNode<int, float>* tmppCurrent = pCurrent;
    TNode<int, float>* tmppNext = pNext;
    TNode<int, float>* tmppPrev = pPrev;

    this->Reset();

    while (pNext)
        this->Next();

    TNode<int, float>* newNode = new TNode<int, float>(_key, _data);

    if (!pFirst)
        pFirst = newNode;
    else
        pCurrent->pNext = newNode;

    if (tmppCurrent == pCurrent)
        pNext = newNode;
    else
        pNext = tmppNext;

    pCurrent = tmppCurrent;
    pPrev = tmppPrev;
};

void TList<int, float>::PushBefore(int _superKey, int _key, float _data)
{
    TNode<int, float>* tmppCurrent = pCurrent;
    TNode<int, float>* tmppNext = pNext;
    TNode<int, float>* tmppPrev = pPrev;

    this->Reset();

    if ((this->IsEnded()) || (pFirst->key == _superKey))
    {
        this->PushBegin(_key, _data);
        pCurrent = pFirst;
        return;
    }

    TNode<int, float>* nodeFind = Search(_superKey);

    if (!nodeFind)
    {

```

```

        throw Exception("Error: key didn't find!");
        return;
    }

    while (pCurrent != nodeFind)
        this->Next();

    TNode<int, float>* newNode = new TNode<int, float>(_key, _data, pCurrent);
    pPrev->pNext = newNode;

    if (tmppCurrent == pPrev)
        pNext = newNode;
    else
        pNext = tmppNext;

    if (tmppCurrent == pCurrent)
        pPrev = newNode;
    else
        pPrev = tmppPrev;

    pCurrent = tmppCurrent;
};

void TList<int, float>::PushAfter(int _superKey, int _key, float _data)
{
    TNode<int, float>* tmppCurrent = pCurrent;
    TNode<int, float>* tmppNext = pNext;
    TNode<int, float>* tmppPrev = pPrev;

    this->Reset();

    TNode<int, float>* nodeFind = Search(_superKey);

    if (!nodeFind)
    {
        throw Exception("Error: key didn't find!");
        return;
    }

    while (pCurrent != nodeFind)
        this->Next();

    TNode<int, float>* newNode = new TNode<int, float>(_key, _data, pNext);
    pCurrent->pNext = newNode;

```

```

        if (tmppCurrent == pCurrent)
            pNext = newNode;
        else
            pNext = tmppNext;

        if (tmppCurrent == pNext)
            pPrev = newNode;
        else
            pPrev = tmppPrev;

        pCurrent = tmppCurrent;
    };

void TList<int, float>::Delete(int _key)
{
    if (!pFirst)
        throw Exception("Error: list is empty!");

    if (pFirst->key == _key)
    {
        if (pCurrent == pFirst)
        {
            pCurrent = pNext;
            if (pNext)
                pNext = pNext->pNext;
            else
                pNext = NULL;

            delete pFirst;
            pFirst = pCurrent;

            return;
        }

        if (pCurrent == pFirst->pNext)
        {
            pPrev = NULL;

            delete pFirst;
            pFirst = pCurrent;

            return;
        }

        delete pFirst;
    }
}

```

```

        return;
    }

    TNode<int, float>* tmppCurrent = pCurrent;
    TNode<int, float>* tmppPrev = pPrev;
    TNode<int, float>* tmppNext = pNext;

    this->Reset();

    TNode<int, float>* nodeFind = Search(_key);

    if (!nodeFind)
    {
        throw Exception("Error: key didn't find!");
        return;
    }

    while (pCurrent != nodeFind)
        this->Next();

    pPrev->pNext = pNext;

    if (tmppCurrent == pCurrent)
    {
        pCurrent = tmppNext;
        pNext = pCurrent->pNext;
        delete nodeFind;

        return;
    }

    if (tmppCurrent == pPrev)
    {
        pCurrent = pPrev;
        pPrev = tmppPrev;
        pNext = pCurrent->pNext;
        delete nodeFind;

        return;
    }

    if (tmppCurrent == pNext)
    {
        pCurrent = pNext;

```

```

        pNext = pCurrent->pNext;
        delete nodeFind;

        return;
    }

    pNext = tmppCurrent->pNext;
    pCurrent = tmppCurrent;
    delete nodeFind;

    return;
};

#endif

```

Приложение 7. Класс TPolinom

```

#ifndef _TPOLINOM_H_
#define _TPOLINOM_H_

#include <cstring>
#include <iostream>

#include "TList_TPolinom.h"
#include "TNode_TMonom.h"
#include "exceptions.h"

using namespace std;

class TPolinom
{
private:
    TList<int, float>* monoms;

    void CastToDefault();
    bool IsOperation(const char) const;
public:
    TPolinom();
    TPolinom(const string);
    TPolinom(TList<int, float>*);
    TPolinom(const TPolinom&);
    ~TPolinom();

    const TPolinom& operator=(const TPolinom&);
    TPolinom operator+(const TPolinom&);
    TPolinom operator-(const TPolinom&);

```

```

    TPolinom operator*(const TPolinom&);
    TPolinom operator-();
    friend ostream& operator<<(ostream&, TPolinom&);
};
//-----

TPolinom::TPolinom()
{
    monoms = new TList<int, float>();
};

TPolinom::TPolinom(const string _str)
{
    if (!_str.length())
        throw Exception("Error! String is empty!");

    monoms = new TList<int, float>;

    int i = 0; // number of symbol
    bool ismin = false; // check for minus

    while (i < _str.length())
    {
        float coeff = 0;
        int degree = 0;
        bool point = false; // check for floating point
        bool isdegree = false; // check for degree
        bool iscoeff = true; // check for coeff
        bool isx = false; // check for x
        bool isy = false; // check for y
        bool isz = false; // check for z

        do
        {
            char symbol = static_cast<char>(_str[i]);

            if (isspace(symbol)) // space processing
            {
                i++;
                continue;
            }

            if (symbol == '.')
            {
                point = true;
            }

```



```

        i++;
        continue;
    }

    if (isdigit(symbol) && iscoeff && !point && !isdegree)
    {
        coeff = coeff * 10 + (int)symbol - 48; // ASCII
        i++;
        continue;
    }

    if (isdigit(symbol) && iscoeff && point && !isdegree)
    {
        int tmp = (int)coeff;
        float tmpcoeff = coeff;
        float ex = ((int)symbol - 48) / 10.; // ASCII

        while (tmp != tmpcoeff)
        {
            ex /= 10.;
            tmpcoeff *= 10.;
            tmp = (int)tmpcoeff;
        }

        coeff = coeff + ex;
        i++;
        continue;
    }

    if (symbol == 'x')
    {
        isx = true;
        isy = false;
        isz = false;
        isdegree = false;
        iscoeff = false;
        point = false;
        i++;
        continue;
    }

    if (symbol == 'y')
    {
        isx = false;
        isy = true;

```

```

        isz = false;
        isdegree = false;
        iscoeff = false;
        point = false;
        i++;
        continue;
    }

    if (symbol == 'z')
    {
        isx = false;
        isy = false;
        isz = true;
        isdegree = false;
        iscoeff = false;
        point = false;
        i++;
        continue;
    }

    if (symbol == '^' && (isx || isy || isz))
    {
        isdegree = true;
        i++;
        continue;
    }

    if (isdegree && (isx || isy || isz))
    {
        if (isx)
            degree = degree + ((int)symbol - 48) * 100;
        if (isy)
            degree = degree + ((int)symbol - 48) * 10;
        if (isz)
            degree = degree + ((int)symbol - 48);

        isx = isy = isz = false;
        isdegree = false;
        i++;
        continue;
    }

    throw Exception("Error! Not correct string!");

```

```

        } while (!IsOperation(static_cast<char>(_str[i])) && (i !=
_str.length()));

        if (ismin && (coeff != 0))
            monoms->PushEnd(degree, -coeff);
        if (!ismin && (coeff != 0))
            monoms->PushEnd(degree, coeff);

        if (static_cast<char>(_str[i]) == '-')
            ismin = true;
        else
            ismin = false;

        i++;
    }

    this->CastToDefault();
};

TPolinom::TPolinom(TList<int, float>* _monoms)
{
    while (!_monoms->IsEnded())
    {
        if (_monoms->GetpCurrent()->key > 999)
            throw Exception("Error! Degree of monom is not correct!");
        _monoms->Next();
    }

    monoms = new TList<int, float>(*_monoms);
    this->CastToDefault();
};

TPolinom::TPolinom(const TPolinom& _copy)
{
    monoms = new TList<int, float>(*_copy.monoms);
    this->CastToDefault();
};

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

//-----

void TPolinom::CastToDefault()

```

```

{
    if (!monoms)
        throw Exception("Error! Polinom is empty!");

    monoms->SortKey();
};

bool TPolinom::IsOperation(const char _s) const
{
    return (_s == '+' || _s == '-');
};
//-----

const TPolinom& TPolinom::operator=(const TPolinom& _copy)
{
    if (this == &_amp;_copy)
        return *this;

    if (monoms->GetpFirst())
        delete monoms;

    monoms = new TList<int, float>(*_copy.monoms);
    return *this;
};

TPolinom TPolinom::operator+(const TPolinom& _copy)
{
    TPolinom tmp(*this);
    _copy.monoms->Reset();

    while (!_copy.monoms->IsEnded())
    {
        tmp.monoms->Reset();

        while ((!tmp.monoms->IsEnded()) && (_copy.monoms->GetpCurrent()->key !=
tmp.monoms->GetpCurrent()->key))
            tmp.monoms->Next();

        if (tmp.monoms->IsEnded())
        {
            tmp.monoms->PushEnd(_copy.monoms->GetpCurrent()->key, _copy.monoms-
>GetpCurrent()->pData);
            _copy.monoms->Next();
            continue;
        }
    }
}

```

```

        tmp.monoms->GetpCurrent()->pData += _copy.monoms->GetpCurrent()->pData;
        _copy.monoms->Next();
    }

    tmp.monoms->Reset();
    while (!tmp.monoms->IsEnded())
    {
        if (tmp.monoms->GetpCurrent()->pData == 0)
        {
            tmp.monoms->Delete(tmp.monoms->GetpCurrent()->key);
            continue;
        }

        tmp.monoms->Next();
    }

    _copy.monoms->Reset();
    tmp.monoms->Reset();
    tmp.CastToDefault();
    return tmp;
};

TPolinom TPolinom::operator-(const TPolinom& _copy)
{
    TPolinom tmp(_copy);
    return *this + (-tmp);
};

TPolinom TPolinom::operator*(const TPolinom& _copy)
{
    TPolinom tmp;
    _copy.monoms->Reset();

    while (!_copy.monoms->IsEnded())
    {
        this->monoms->Reset();

        while (!this->monoms->IsEnded())
        {
            float coeff = this->monoms->GetpCurrent()->pData * _copy.monoms->GetpCurrent()->pData;

            int degreeX = this->monoms->GetpCurrent()->key / 100;
            int degreeY = (this->monoms->GetpCurrent()->key / 10) % 10;
            int degreeZ = this->monoms->GetpCurrent()->key % 10;

```

```

        int copyDegreeX = _copy.monoms->GetpCurrent()->key / 100;
        int copyDegreeY = (_copy.monoms->GetpCurrent()->key / 10) % 10;
        int copyDegreeZ = _copy.monoms->GetpCurrent()->key % 10;

        if (((degreeX + copyDegreeX) > 9) || ((degreeY + copyDegreeY) > 9)
            || ((degreeZ + copyDegreeZ) > 9))
            throw Exception("Error! Degrees are not correct!");

        int key = (degreeX + copyDegreeX) * 100 +
            (degreeY + copyDegreeY) * 10 + (degreeZ + copyDegreeZ);

        tmp.monoms->PushEnd(key, coeff);
        this->monoms->Next();
    }

    _copy.monoms->Next();
}

this->monoms->Reset();
_copy.monoms->Reset();
tmp.monoms->Reset();
tmp.CastToDefault();
return tmp;
};

TPolinom TPolinom::operator-()
{
    TPolinom tmp(*this);

    while (!tmp.monoms->IsEnded())
    {
        tmp.monoms->GetpCurrent()->pData *= -1;
        tmp.monoms->Next();
    }

    tmp.monoms->Reset();
    return tmp;
};

ostream& operator<<(ostream& _out, TPolinom& _p)
{
    _p.monoms->Reset();

    if (_p.monoms->IsEnded())

```

```

{
    _out << "Polinom is empty!" << endl;
    return _out;
}

_out << _p.monoms->GetpCurrent()->pData;

if (_p.monoms->GetpCurrent()->key / 100 != 0)
    _out << "x^(" << int(_p.monoms->GetpCurrent()->key / 100) << ")";
if (((int)_p.monoms->GetpCurrent()->key / 10) % 10 != 0)
    _out << "y^(" << int(((int)_p.monoms->GetpCurrent()->key / 10) % 10) <<
    ")";
if (((int)_p.monoms->GetpCurrent()->key % 10 != 0)
    _out << "z^(" << int(((int)_p.monoms->GetpCurrent()->key % 10) << ")";

_p.monoms->Next();

while (!_p.monoms->IsEnded())
{
    if (_p.monoms->GetpCurrent()->pData > 0)
        _out << " + " << _p.monoms->GetpCurrent()->pData;
    if (_p.monoms->GetpCurrent()->pData < 0)
        _out << " - " << abs(_p.monoms->GetpCurrent()->pData);

    if (_p.monoms->GetpCurrent()->key / 100 != 0)
        _out << "x^(" << int(_p.monoms->GetpCurrent()->key / 100) << ")";
    if (((int)_p.monoms->GetpCurrent()->key / 10) % 10 != 0)
        _out << "y^(" << int(((int)_p.monoms->GetpCurrent()->key / 10) % 10)
        << ")";
    if (((int)_p.monoms->GetpCurrent()->key % 10 != 0)
        _out << "z^(" << int(((int)_p.monoms->GetpCurrent()->key % 10) <<
        ")";

    _p.monoms->Next();
}

_out << endl;

return _out;
};

#endif

```