МИНИСТЕРСТВО НАУКИ И ВЫСШЕГООБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

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

**(ННГУ)**

**Институт информационных технологий, математики и механики**

**ЛАБОРАТОРНАЯ РАБОТА**

на тему:

**«Постфиксная форма записи арифметических выражений»**

**Выполнил:** студент группы 3822Б1ФИ2

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ /Миронов А. И./

Подпись

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

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

Подпись

Нижний Новгород  
2023

**Содержание**

[Введение 3](#_Toc153624766)

[1 Постановка задачи 4](#_Toc153624767)

[2 Руководство пользователя 5](#_Toc153624768)

[2.1 Приложение для демонстрации работы стека 5](#_Toc153624769)

[2.2 Приложение для демонстрации работы перевода арифметического выражения в постфиксную запись 6](#_Toc153624770)

[3 Руководство программиста 8](#_Toc153624771)

[3.1 Описание алгоритмов 8](#_Toc153624772)

[3.1.1 Стек 8](#_Toc153624773)

[3.1.2 Арифметическое выражение 9](#_Toc153624774)

[3.2 Описание программной реализации 12](#_Toc153624775)

[3.2.1 Описание класса TStack 12](#_Toc153624776)

[3.2.2 Описание класса Expression 13](#_Toc153624777)

[Заключение 17](#_Toc153624778)

[Литература 18](#_Toc153624779)

[Приложения 19](#_Toc153624780)

[Приложение А. Реализация класса TStack 19](#_Toc153624781)

[Приложение Б. Реализация класса Expression 20](#_Toc153624782)

# Введение

Лабораторная работа направлена на изучение алгоритма преобразования математических выражений из инфиксной записи в постфиксную (обратную польскую) запись. Инфиксная запись — это традиционный способ записи математических выражений, где операторы расположены между операндами. Постфиксная запись, наоборот, предполагает расположение операторов после соответствующих операндов.

В данной лабораторной работе студенты будут изучать основные принципы работы алгоритма преобразования инфиксной записи в постфиксную и реализовывать его на практике. Это позволит им лучше понять принципы работы стека и освоить навыки работы с алгоритмами обработки строк и вычисления математических выражений.

# Постановка задачи

**Цель:**

Реализовать шаблонный класс TStack. Используя класс TStack реализовать класс перевода арифметического выражения в постфиксную форму Expression. Научиться использовать стек для преобразования инфиксного (обычного) арифметического выражения в постфиксную (обратную польскую) форму.

**Задачи:**

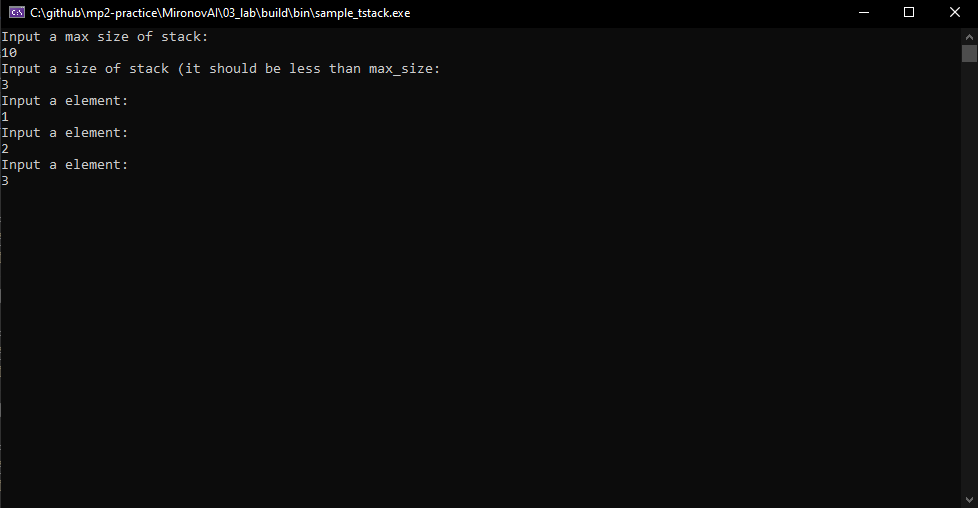
1. Изучение основных принципов работы со стеком.
2. Изучение правил преобразования инфиксного выражения в постфиксное.
3. Написание программы на С++, использующей стек для преобразования арифметического выражения.
4. Анализ времени выполнения программы и оценка эффективности использования стека для данной задачи.
5. Тестирование программы на различных входных данных, включая выражения с разными операциями и скобками.

В результате лабораторной работы студент должен освоить принципы работы со стеком, понять преимущества использования постфиксной формы для вычисления арифметических выражений и научиться применять их на практике.

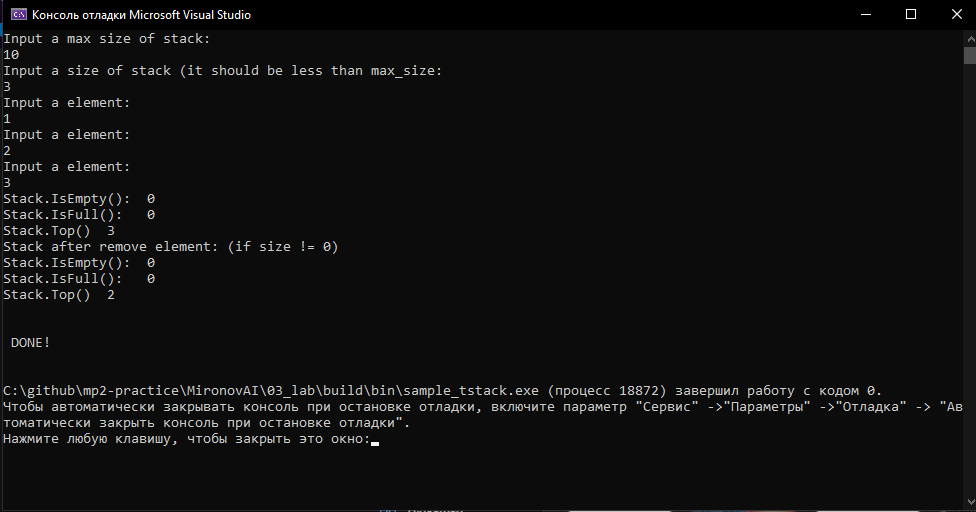
# Руководство пользователя

## Приложение для демонстрации работы стека

1. Запустите приложение с названием sample\_tstack.exe. В результате появится окно, показанное ниже и вам будет предложено ввести размер максимальный стека, число элементов в стеке и сами целочисленные элементы. ().



1. Основное окно программы
2. После ввода будет выведены результаты соответствующих операций и функций стека. Если введенный размер равен нулю, то метод Top() не вызывается.().

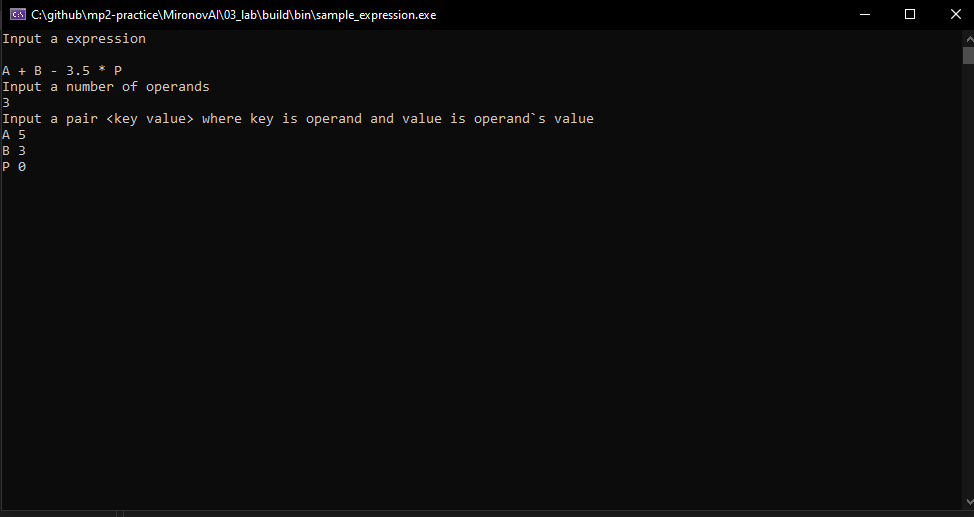


1. Результат тестирования функций класса TStack

## Приложение для демонстрации работы перевода арифметического выражения в постфиксную запись

1. Запустите приложение с названием sample\_expression.exe. В результате появится окно, показанное ниже, вам будет предложено ввести арифметическое выражение. После ввода арифметического выражения необходимо ввести количество используемых в нем переменных, далее ввести каждый из них

(рис. 3).



1. Основное окно программы
2. После ввода арифметического выражения будет выведены результаты соответствующих операций и функций (рис. 4).



1. Результат тестирования функций класса Expression

# Руководство программиста

## Описание алгоритмов

### Стек

Стек – это структура хранения, основанная на принципе «Last in, first out». Операции, доступные с данной структурой хранения, следующие: добавление элемента в вершину стека, удаление элемента из вершины стека, взять элемент с вершины стека, проверка на полноту, проверка на пустоту.

**Операция добавления элемента в вершину стека**

Операция добавления элемента реализуется при помощи флага, указывающий на последний занятый элемент ( на вершину стека ). Если структура хранения ещё не полна, то мы можем добавить элемент на top+1 место.

Пример:

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 2 |  |  |

Операция добавления элемента (1) в вершину:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 4 | 2 |  |

**Операция удаления элемента из вершины стека**

Операция удаления элемента реализуется при помощи флага, указывающий на последний занятый элемент ( на вершину стека ) . Если структура хранения ещё не пуста, то мы можем удалить элемент с индексом top .

Пример:

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 2 |  |  |

Операция добавления элемента (1) в вершину:

|  |  |  |  |
| --- | --- | --- | --- |
| 2 |  |  |  |

**Операция взятия элемента с вершины.**

Операция взятия элемента с вершины также реализуется при помощи флага, указывающий на последний занятый элемент ( на вершину стека ). Если структура хранения не пуста, мы можем взять элемент с вершины.

Пример:

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 2 |  |  |

Операция взятия элемента с вершины стека:

Результат: 4

**Операция проверки на полноту.**

Операция проверки на полноту проверяет, полон ли стек. Также реализуется при помощи флага, указывающий на вершину стека.

Пример 1:

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 2 |  |  |

Операция проверки на полноту:

Результат: false

Пример 2:

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 2 | 2 | 2 |

Операция проверки на полноту:

Результат: true

**Операция проверки на пустоту.**

Операция проверки на полноту проверяет, есть ли хотя бы один элемент в стеке. Также реализуется при помощи флага, указывающий на вершину стека.

Пример 1:

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 2 |  |  |

Операция проверки на полноту:

Результат: false

Пример 2:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

Операция проверки на полноту:

Результат: true

### Арифметическое выражение

Программа предоставляет возможности для работы с арифметическими выражениями: получение инфиксной записи, получение постфиксной записи, получение результата.

Алгоритм на входе требует строку, которая представляет некоторое арифметическое выражение, и хэш-таблицу, элементы которой представляют операнды в арифметическом выражении. Алгоритм также вводит приоритет арифметических операций согласно математическим правилам: скобки, умножение/деление, сложение/вычитание.

**Получение инфиксной записи.**

Функция просто выведет исходную строку в инфиксной записи.

**Получение постфиксной записи.**

Изначально алгоритм подготавливает выражение: убирает лишние пробелы, проверяет на корректность введенных данных, разделяет строку на операции и операнды. Таким образом, до начала перевода в постфиксную форму в программе уже есть разделенный набор операций и операндов.

Алгоритм:

1. Создаем пустой стек операторов.

2. Создаем пустой массив для хранения постфиксной записи.

3. Проходим по каждому символу в инфиксной записи слева на право:

- Если символ является операндом, добавляем его в массив постфиксной записи.

- Если символ является открывающей скобкой, помещаем его в стек операторов.

- Если символ является закрывающей скобкой, извлекаем операторы из стека и добавляем их в массив постфиксной записи до тех пор, пока не встретится открывающая скобка. Удаляем открывающую скобку из стека.

- Если символ является оператором, извлекаем операторы из стека и добавляем их в массив постфиксной записи до тех пор, пока не будет найден оператор с меньшим или равным приоритетом. Затем помещаем текущий оператор в стек.

4. Извлекаем оставшиеся операторы из стека и добавляем их в массив постфиксной записи.

После завершения алгоритма массив постфиксной записи будет содержать инфиксное выражение в постфиксной форме.

Пример:

Выражение: А + (В – C) \* D

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  | + |
|  |  |  |  |  |  |  |  | \* |
|  |  |  |  |  |  |  |  | D |
|  |  |  |  |  |  | - | - | - |
|  |  |  |  |  | C | C | C | C |
|  |  |  | B | B | B | B | B | B |
| A | A | A | A | A | A | A | A | A |

Стек:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  | - | - |  |  |  |
|  |  | ( | ( | ( | ( |  | \* |  |
|  | + | + | + | + | + | + | + |  |

**Вычисление результата.**

Алгоритм вычисления значения выражения в постфиксной записи (обратной польской записи) выглядит следующим образом:

1. Создаем пустой стек для хранения операндов.

2. Проходим по каждому символу в постфиксной записи:

- Если символ является операндом, помещаем его в стек операндов.

- Если символ является оператором, извлекаем два операнда из стека, применяем оператор к этим операндам и помещаем результат обратно в стек.

3. После завершения прохода по всем символам, результат вычисления будет находиться на вершине стека операндов.

Полученное значение на вершине стека будет являться результатом вычисления постфиксной записи.

Пример:

Выражение: А + (В – C) \* D

Постфиксная запись: ABC-D\*+

Значение операндов: A = B = 3, C = D = 2

Стек:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  | 3 | 3 | 1 | 1 | 2 |  |
| 3 | 3 | 3 | 3 | 3 | 3 | 2 | 4 |

## Описание программной реализации

### Описание класса TStack

template <class Type>

class TStack {

private:

int maxSize;

int top;

Type\* elems;

public:

TStack(int maxSize1 = 100);

TStack(const TStack<Type>& s);

~TStack();

Type Top() const;

bool IsEmpty() const;

bool IsFull() const;

void Push(const Type& elem);

void Pop();

};

Назначение: представление стека .

Поля:

maxSize – максимальный размер стека.

\*elems – память для представления элементов стека.

top – индекс вершины стека (-1, если стек пустой).

Методы:

TStack(int maxSize1 = 100);

Назначение: конструктор по умолчанию и конструктор с параметрами.

Входные параметры:

maxSize1 – максимальный размер стека (по умолчанию 100).

Выходные параметры: отсутствуют.

TStack(const TStack<Type>& s);

Назначение: конструктор копирования.

Входные параметры:

s – стек, на основе которого создаем новый стек.

Выходные параметры: отсутствуют.

~TStack();

Назначение: деструктор.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

**Type Top() const;**

Назначение: получение элемента, находящийся в вершине стека.

Входные параметры отсутствуют.

Выходные параметры: элемент с вершины стека, последний добавленный элемент.

**bool IsEmpty() const;**

Назначение: проверка на пустоту.

Входные параметры: отсутствуют.

Выходные параметры: 1, если стек пуст, 0 иначе.

**bool IsFull() const;**

Назначение: проверка на полноту.

Входные параметры: отсутствуют.

Выходные параметры: 1, если стек полон, 0 иначе.

void push(const Type& elem);

Назначение: добавление элемента в стек.

Входные параметры:

elem – элемент, который добавляем.

Выходные параметры отсутствуют.

void pop();

Назначение: удаление элемента из вершины стека.

Входные параметры отсутствуют.

### Описание класса Expression

class Expression {

private:

string infix;

vector<string> postfix;

vector<string> lexems;

map<string, short> priority;

map<string, double> operands;

double res;

void Parse();

void Convert();

void Preparation();

bool IsOperator(const string& op) const;

bool IsConst(const string& op) const;

bool IsArOperator(const char& op) const;

void IsCorrect() const;

int FindFirstOperator(int pos = 0) const;

void Calculate();

public:

Expression(const string& expression, const map<string, double> operands\_);

string GetInfix() const { return infix; }

string GetPostfix() const;

double Get\_res() const { return res; }

};

Назначение: работа с инфиксной формой записи арифметических выражений

Поля:

infix – выражение в инфиксной записи.

postfix – выражение в посфиксной записи.

lexems – набор лексем инфиксной записи

priority – приоритет арифметических операндов

operands – операнды и их значения

res – результат вычисления выражения

Методы:

Expression(const string& expression, const map<string, double> operands\_);

Назначение: конструктор с параметрами.

Входные параметры:

expression – выражение в инфиксной форме.

operands\_ – значение операндов

Выходные параметры: отсутствуют.

string GetInfix();

Назначение: получение инфиксной формы.

Входные параметры отсутствуют.

Выходные параметры: инфиксная форма записи.

string GetPostfix();

Назначение: получение постфиксной формы.

Входные параметры отсутствуют.

Выходные параметры: постфиксная форма записи.

double Get\_res();

Назначение: получение результата вычисления выражения.

Входные параметры отсутствуют.

Выходные параметры: результат

void Calculate();

Назначение: вычисление выражения.

Входные параметры отсутствуют.

Выходные параметры: отсутствуют

void IsCorrect();

Назначение: проверка на корректность введенных данных.

Входные параметры отсутствуют.

Выходные параметры: отсутствуют

void Preparation();

Назначение: подготовка инфиксной формы записи.

Входные параметры отсутствуют.

Выходные параметры: отсутствуют

void Convert();

Назначение: конвертирование инфиксной формы в постфиксную.

Входные параметры отсутствуют.

Выходные параметры: отсутствуют

void Parse();

Назначение: подготовка к конвертированию инфиксной формы в постфиксную. Разделение инфиксной формы на лексемы.

Входные параметры отсутствуют.

Выходные параметры: отсутствуют

int FindFirstOperator(int pos = 0);

Назначение: поиск первого оператора , начиная с позиции pos.

Входные параметры:

pos – позиция, начиная с которой ищется оператор.

Выходные параметры: индекс оператора (-1, если он не нашёлся).

bool IsOperator(const string &op) const;

Назначение: проверяет строку оператор ли это.

Входные параметры:

op – оператор или операнд.

Выходные параметры: 1, если строка – это оператор, 0 иначе.

bool IsConst(const string &op) const;

Назначение: проверяет строку константа ли это.

Входные параметры:

op – оператор или операнд.

Выходные параметры: 1, если строка – это константа, 0 иначе.

bool IsArOperator(const char\* &op) const;

Назначение: проверяет строку арифметический оператор ли это.

Входные параметры:

op – оператор или операнд.

Выходные параметры: 1, если строка – это арифметический оператор, 0 иначе.

# Заключение

В ходе выполнения лабораторной работы было изучены основные принципы работы алгоритма преобразования математических выражений из инфиксной записи в постфиксную. Также были получено практические навыки реализации этого алгоритма и работы с постфиксной записью.

Изучение данного алгоритма позволило студентам лучше понять принципы работы стека, освоить навыки работы с алгоритмами обработки строк и вычисления математических выражений. Также они узнали о преимуществах постфиксной записи перед инфиксной и научились применять её в практических задачах.

Таким образом, выполнение лабораторной работы позволило студентам расширить свои знания в области алгоритмов обработки математических выражений и приобрести навыки работы с постфиксной записью. Эти знания и навыки будут полезны им в дальнейшем образовании и профессиональной деятельности.

# Литература

1. Польская запись [https://ru.wikipedia.org/wiki/Польская\_запись].

# Приложения

## Приложение А. Реализация класса TStack

#ifndef \_TSTACK\_H

#define \_TSTACK\_H

using namespace std;

template <class Type>

class TStack {

private:

int maxSize;

int top;

Type\* elems;

public:

TStack(int maxSize1 = 100);

TStack(const TStack<Type>& s);

~TStack();

Type Top() const;

bool IsEmpty() const;

bool IsFull() const;

void Push(const Type& elem);

void Pop();

};

template <class Type>

TStack<Type>::TStack(int maxSize1)

{

if (maxSize1 < 1)

throw exception("Size should be > 0");

maxSize = maxSize1;

elems = new Type[maxSize];

top = -1;

}

template <class Type>

TStack<Type>::TStack(const TStack<Type>& s)

{

maxSize = s.maxSize;

top = s.top;

elems = new Type[maxSize];

for (int i = 0; i <= top; i++)

{

elems[i] = s.elems[i];

}

}

template <class Type>

TStack<Type>::~TStack()

{

if (elems != nullptr)

delete[] elems;

}

template <class Type>

bool TStack<Type>::IsFull() const

{

return top + 1 == maxSize;

}

template <class Type>

bool TStack<Type>::IsEmpty() const

{

return top == -1;

}

template <class Type>

Type TStack<Type>::Top() const

{

if (top == -1)

throw exception("Stack is empty");

return elems[top];

}

template <class Type>

void TStack<Type>::Push(const Type& elem)

{

if (top + 1 == maxSize)

throw exception("Stack is full");

elems[++top] = elem;

}

template <class Type>

void TStack<Type>::Pop()

{

if (top-- == -1)

throw exception("Stack is empty");

}

#endif // !\_TSTACK\_H

## Приложение Б. Реализация класса Expression

#include "expression.h"

bool Expression::IsOperator(const string& op) const

{

for (const pair<string, short>& operator\_ : priority) {

if (operator\_.first == op)

{

return true;

}

}

return false;

}

bool Expression::IsConst(const string& op) const

{

for (int i = 0; i < op.size(); i++)

if (op[i] < '0' || op[i] > '9')

{

if (op[i] != '.')

{

return false;

}

return true;

}

return true;

}

bool Expression::IsArOperator(const char& op) const

{

return (op == '\*' || op == '/' || op == '+' || op == '-');

}

int Expression::FindFirstOperator(int pos) const

{

if (pos < 0 || pos >= infix.size()) return -1;

for (int i = pos; i < infix.size(); i++)

{

string op;

op += infix[i];

if (IsOperator(op))

{

return i;

}

}

return -1;

}

void Expression::Preparation()

{

string massage = "Incorrect expression";

string expression\_without\_spaces;

for (int i = 0; i < infix.size(); i++) {

if (infix[i] - ' ')

expression\_without\_spaces += infix[i];

}

string expression;

if (expression\_without\_spaces[0] == '-')

{

expression += "0-";

}

else if (expression\_without\_spaces[0] == '.')

{

throw massage;

}

else expression += expression\_without\_spaces[0];

if (expression\_without\_spaces[expression\_without\_spaces.size() - 1] == '.')

{

throw massage;

}

for (int i = 1; i < expression\_without\_spaces.size(); i++)

{

char t = expression\_without\_spaces[i];

if (t == '-')

{

if (expression\_without\_spaces[i - 1] == '(')

{

expression += '0';

}

expression += '-';

}

else if (t == '.')

{

if (expression\_without\_spaces[i - 1] < '0' || expression\_without\_spaces[i - 1] > '9' ||

expression\_without\_spaces[i + 1] < '0' || expression\_without\_spaces[i + 1] > '9')

{

throw massage;

}

expression += '.';

}

else if (t == '(')

{

if (expression\_without\_spaces[i - 1] == ')' || (expression\_without\_spaces[i - 1] >= '0' && expression\_without\_spaces[i - 1] <= '9'))

{

expression += '\*';

}

expression += '(';

}

else

{

expression += t;

}

}

infix = expression;

}

void Expression::IsCorrect() const

{

int open\_count = 0;

int closed\_count = 0;

int dots\_count = 0;

int len = infix.size() - 1;

string massage = "Incorrect expression";

if (infix[0] == '\*' || infix[0] == '/' || infix[0] == '+' || infix[0] == ')')

{

throw massage;

}

else if (infix[0] == '(')

{

open\_count++;

}

if (IsArOperator(infix[len]) || infix[len] == '(')

{

throw massage;

}

else if (infix[len] == ')')

{

closed\_count++;

}

for (int i = 1; i < len; i++)

{

char t = infix[i];

if (t == '(')

{

open\_count++;

}

else if (t == ')')

{

if (IsArOperator(infix[i-1]) || infix[i - 1] == '(')

{

throw massage;

}

closed\_count++;

}

else if (t == '.')

{

dots\_count++;

}

else if (IsArOperator(t))

{

if (dots\_count > 1 || IsArOperator(infix[i-1]) || infix[i - 1] == '(')

{

throw massage;

}

dots\_count = 0;

}

else

{

if (infix[i - 1] == ')')

{

// ") A"

throw massage;

}

}

}

if (open\_count != closed\_count || dots\_count > 1)

{

throw massage;

}

}

void Expression::Parse()

{

Preparation();

IsCorrect();

int id1 = FindFirstOperator(), id2 = FindFirstOperator(id1 + 1);

string substring;

if (id1 == -1) {

lexems.push\_back(infix);

return;

}

else

{

for (int i = 0; i < id1; i++)

{

substring += infix[i];

}

if (substring.size())

{

lexems.push\_back(substring);

}

}

while (id2 + 1)

{

string substring1;

substring = infix[id1];

lexems.push\_back(substring);

for (int i = id1+1; i < id2 ; i++)

{

substring1 += infix[i];

}

if (!substring1.empty())

lexems.push\_back(substring1);

id1 = id2;

id2 = FindFirstOperator(id1 + 1);

}

substring = infix[id1];

lexems.push\_back(substring);

substring.clear();

for (int i = id1; i < infix.size(); i++)

{

substring += infix[i];

}

if (id1 != infix.size() - 1)

{

substring = infix[id1 + 1];

for (int i = id1 + 2; i < infix.size(); i++)

{

substring += infix[i];

}

lexems.push\_back(substring);

}

}

void Expression::Convert()

{

Parse();

string op;

TStack<string> stack(infix.size()+10);

for (string lexem : lexems)

{

if ((lexem.size() == 1) && IsArOperator(lexem[0]))

{

while (!stack.IsEmpty())

{

op = stack.Top();

stack.Pop();

if (priority[op] >= priority[lexem])

{

postfix.push\_back(op);

}

else

{

stack.Push(op);

break;

}

}

stack.Push(lexem);

}

else if (lexem == "(")

{

stack.Push(lexem);

}

else if (lexem == ")")

{

op = stack.Top();

stack.Pop();

while (op != "(")

{

postfix.push\_back(op);

op = stack.Top();

stack.Pop();

}

}

else

{

double value = 0.0;

if (IsConst(lexem))

{

value = stod(lexem);

operands[lexem] = value;

}

postfix.push\_back(lexem);

}

}

while (!stack.IsEmpty()) {

op = stack.Top();

stack.Pop();

postfix.push\_back(op);

}

}

string Expression::GetPostfix() const

{

string string\_postfix = postfix[0];

for (int i = 1; i < postfix.size(); i++) string\_postfix += postfix[i];

return string\_postfix;

}

void Expression::Calculate()

{

TStack<double> stack(infix.size()+10);

double op1, op2;

for (string lexem : postfix) {

if (lexem == "+")

{

op2 = stack.Top();

stack.Pop();

op1 = stack.Top();

stack.Pop();

stack.Push(op1 + op2);

}

else if (lexem == "-")

{

op2 = stack.Top();

stack.Pop();

op1 = stack.Top();

stack.Pop();

stack.Push(op1 - op2);

}

else if (lexem == "/")

{

op2 = stack.Top();

stack.Pop();

op1 = stack.Top();

stack.Pop();

if (op2 == 0) {

throw "Division by 0";

}

else stack.Push(op1 / op2);

}

else if (lexem == "\*")

{

op2 = stack.Top();

stack.Pop();

op1 = stack.Top();

stack.Pop();

stack.Push(op1 \* op2);

}

else

{

if (operands.find(lexem) == operands.end())

{

throw "Wrong list of operands";

}

stack.Push( operands[lexem] );

}

}

res = stack.Top();

}

Expression::Expression(const string& expression, const map<string, double> operands\_)

{

if (operands\_ != map<string, double>::map())

{

for (pair<string, double> elems : operands\_)

{

operands[elems.first] = elems.second;

}

}

if (expression.empty())

{

throw "Expression can`t be empty";

}

infix = expression;

priority = {

{"(", 1}, {")", 1},

{"+", 2}, {"-", 2},

{"\*", 3}, {"/", 3}

};

Convert();

Calculate();

}