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:

«Вычисление арифметических выражений (стеки)»

**Выполнил(а):** студент(ка)

Группы: 3822Б1ФИ2

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Фролова Е.А./

Подпись

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

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

Подпись

Нижний Новгород  
2023

**Содержание**

[Введение 3](#_Toc154140025)

[1 Постановка задачи 4](#_Toc154140026)

[2 Руководство пользователя 5](#_Toc154140027)

[2.1 Приложение для демонстрации работы стека 5](#_Toc154140028)

[2.2 Приложение для демонстрации работы класса арифметическое выражение 7](#_Toc154140029)

[3 Руководство программиста 9](#_Toc154140030)

[3.1 Описание алгоритмов 9](#_Toc154140031)

[3.1.1 Стек 9](#_Toc154140032)

[3.1.2 Постфиксная форма 9](#_Toc154140033)

[3.2 Описание программной реализации 12](#_Toc154140034)

[3.2.1 Описание класса STack 12](#_Toc154140035)

[3.2.2 Описание класса TArithmeticExpression 14](#_Toc154140036)

[Заключение 17](#_Toc154140037)

[Литература 18](#_Toc154140038)

[Приложения 19](#_Toc154140039)

[Приложение А. Реализация класса TStack 19](#_Toc154140040)

[Приложение Б. Реализация класса TArithmeticExpression 20](#_Toc154140041)

# Введение

Лабораторная работа направлена на практическое освоение динамической структуры данных Стек. С этой целью в лабораторной работе изучаются различные варианты структуры хранения стеков и разрабатываются методы и программы решения ряда задач с использованием стеков. В качестве области приложений выбрана тема вычисления арифметических выражений, возникающей при трансляции программ на языке программирования высокого уровня в исполняемые программы.

Актуальность постфиксной формы заключается в следующем:

1. Простота чтения и записи: в постфиксной форме выражение представляется линейно, без скобок или приоритетов операций. Это делает запись и чтение выражений проще и более понятными для людей.
2. Отсутствие неоднозначности: в постфиксной форме каждый оператор имеет строго определенную позицию относительно своих операндов. Это исключает возможность неоднозначных интерпретаций выражений.
3. Простота вычисления: выражения в постфиксной форме могут быть вычислены с использованием стека. Каждый операнд добавляется в стек, а операторы вызывают вычисление соответствующей операции над операндами в вершине стека. Это делает вычисление выражений быстрым и эффективным.
4. Универсальность: постфиксная форма является одной из форм, которую понимают различные типы вычислительных систем. Она может быть использована для представления выражений в компьютерных программах или в калькуляторах.

В целом, постфиксная форма арифметического выражения является актуальным и удобным способом представления и вычисления выражений, который находит широкое применение в различных областях.

# Постановка задачи

Ставится задача реализации программ, обеспечивающих поддержку стеков, и разработки программных средств, производящих обработку арифметических выражений, включая проверку правильности записи выражения, перевод в постфиксную форму и вычисление результата.

Задача:

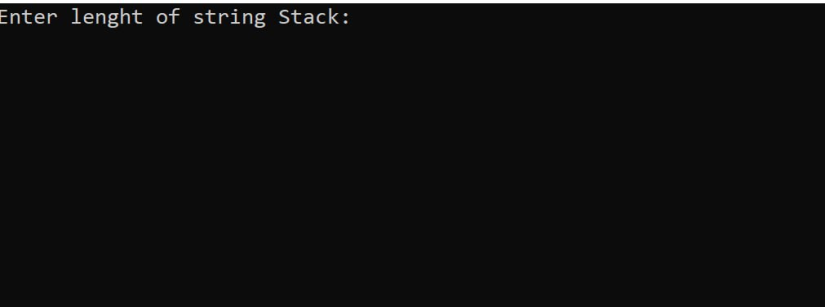
Разработать класс для работы с арифметическими выражениями. Класс арифметическое выражение должен предоставлять, как минимум, следующие операции:

* Получение, хранение и возврат исходной (инфиксной) формы.
* Преобразование в постфиксную форму.
* Возврат постфиксной формы.
* Вычисление арифметического выражения при заданных значениях операндов.

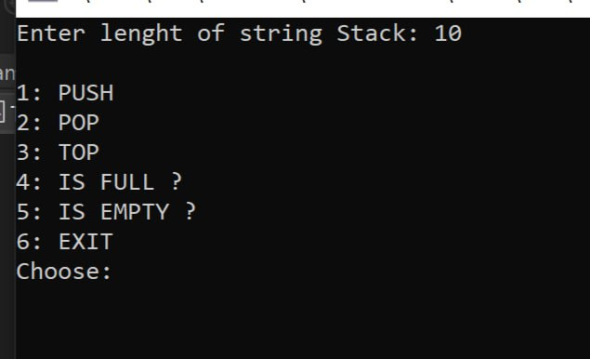
# Руководство пользователя

## Приложение для демонстрации работы стека

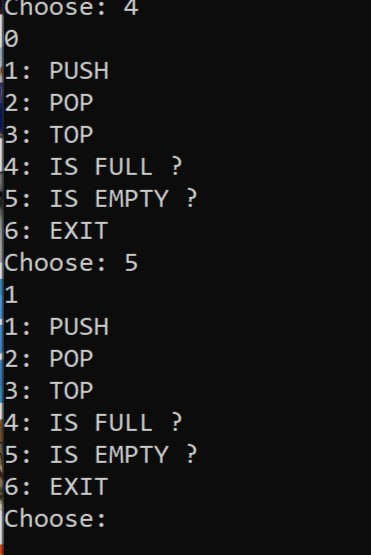
1. Запустите приложение с названием \*.exe. В результате появится окно, показанное ниже (рис. 1).



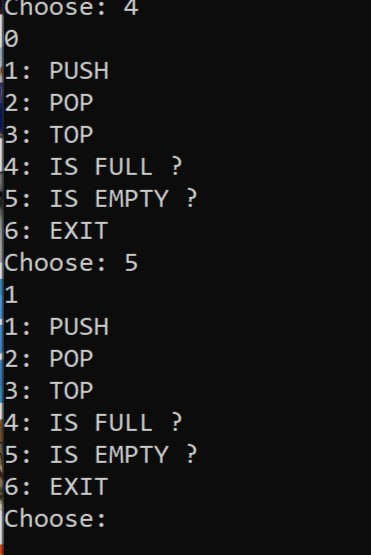
1. Основное окно программы
2. Введем размер стека. Далее мы увидим, что можем сделать со стеком (рис. 2).



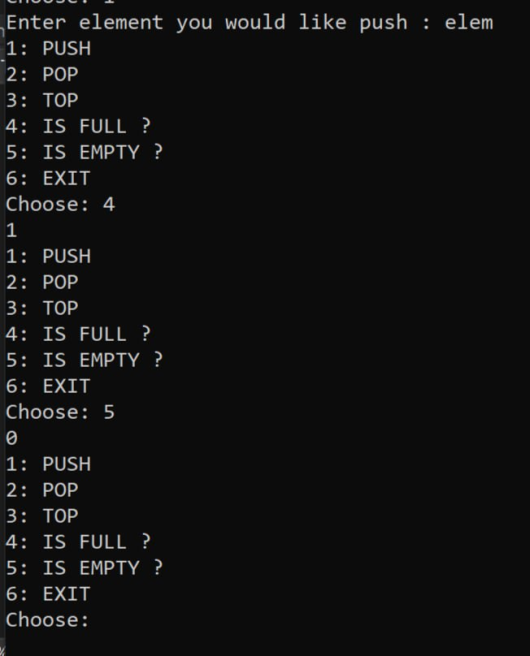
1. Возможные операции со стеком.
2. Проверим стек на полноту (рис. 3).



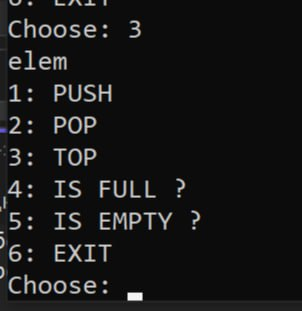
1. Стек не полон.
2. Проверим стек на пустоту (рис. 4).



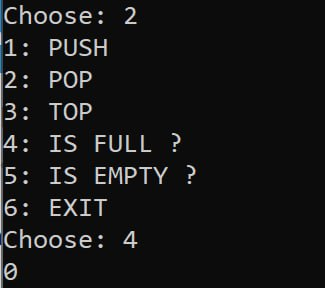
1. Стек пуст.
2. Заполним стек до максимального значения. И опять проверим стек на полноту/пустоту (рис. 5).



1. Стек полон.
2. Получим элемент, лежащий на вершине стека и уберём его (рис. 6). Проверим стек на полноту (рис. 7).



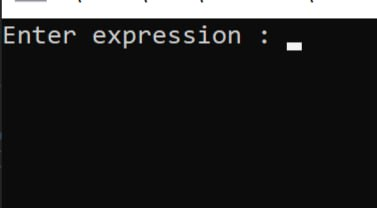
1. Elem-элемент, лежащий на вершине стека.



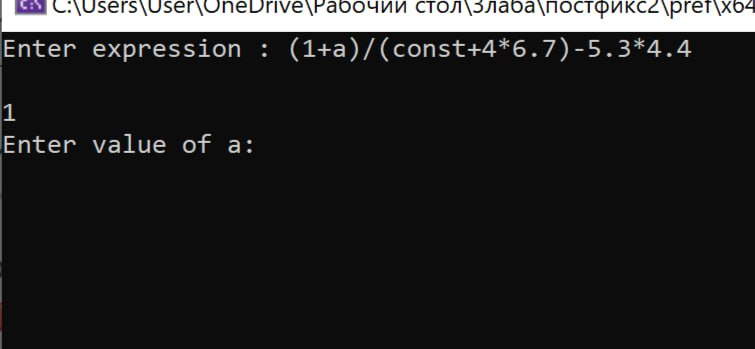
1. Стек не полон.

## Приложение для демонстрации работы класса арифметическое выражение

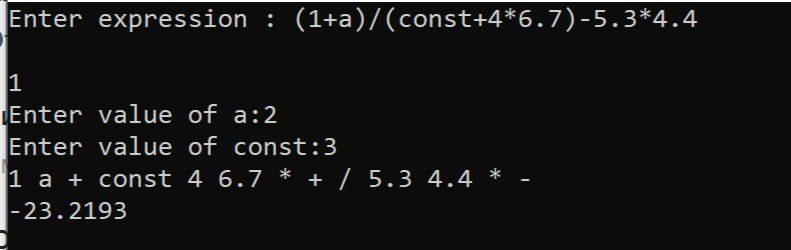
1. Запустите приложение с названием \*.exe. В результате появится окно, показанное ниже (рис. 8).



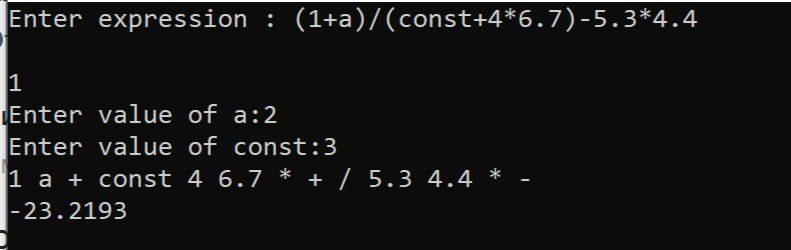
1. Запрос на ввод выражения.
2. Введём арифметическое выражение (рис. 9).



1. Основное окно программы.
2. Программа запрашивает значения операндов (рис. 10).



1. Значения элементов.
2. После ввода элементов программа выводит выражение в постфиксной форме и его результат (рис. 11).



1. Результат арифметического выражения и его постфиксная форма.

# Руководство программиста

## Описание алгоритмов

### Стек

Для работы со стеком предлагается реализовать следующие операции:

* Метод Push – добавить элемент;

При добавлении элемента в стек необходимо переместить указатель вершины стека, записать элемент в соответствующую позицию динамического массива и увеличить количество элементов.

* Метод Pop – удалить элемент;

При удалении элемента из стека необходимо возвратить значение из динамического массива по индексу вершины стека, переместить указатель вершины стека и уменьшить количество элементов.

* Метод IsEmpty – проверить стек на пустоту;

Стек пуст, если в нем нет ни одного элемента, т.е. когда количество элементов равно нулю.

* Метод IsFull – проверить стек на полноту.

Стек полон при исчерпании всей отведенной под хранение элементов памяти.

### Постфиксная форма

Постфиксная запись арифметического выражения (или обратная польская нотация) - это способ записи математических выражений, в котором операторы идут после своих операндов. В постфиксной записи каждое арифметическое выражение записывается без использования скобок и с явным указанием порядка операций.

Например, вместо записи "2 + 3 \* 4" в инфиксной (стандартной) записи, в постфиксной записи это выражение будет записано как "2 3 4 \* +" (где \* обозначает умножение, а + - сложение). При вычислении постфиксного выражения, сначала выполняются операции над операндами, и результаты операций используются в последующих операциях.

Перевод в постфиксную форму.

Алгоритм:

* 1. Создайте пустой стек операторов и операндов.
  2. Инициализируйте пустую строку (в которую будет записано преобразованное выражение).
  3. Просмотрите каждый символ в исходном инфиксном выражении слева направо.
  4. Если символ - операнд (число), добавьте его в строку.
  5. Если символ - открывающая скобка, положите его в стек.
  6. Если символ - оператор, выполните следующие действия:
  + Проверьте стек. Если он не пустой и верхний оператор стека имеет больший или равный приоритет, чем текущий оператор, переведите верхний оператор из стека в строку. Повторяйте эту операцию, пока это условие выполняется.
  + Положите текущий оператор в стек.
  1. Если символ - закрывающая скобка, переведите операторы из стека в строку, пока не встретите открывающую скобку. Удалите открывающую скобку из стека, но не добавляйте ее в строку.
  2. Когда весь входной символ просмотрен, переведите оставшиеся операторы из стека в строку.
  3. Возвращайте преобразованную строку, она будет содержать постфиксную форму исходного выражения.

Пример:

(a + b \* c) \* (c/d – e)

|  |  |  |
| --- | --- | --- |
| Лексема | Постфиксная форма | Стек |
| ( |  | ( |
| a | a | ( |
| + | a | (+ |
| b | ab | (+ |
| \* | ab | (+\* |
| c | abc | (+\* |
| ) | abc\*+ |  |
| \* | abc\*+ | \* |
| ( | abc\*+ | \*( |
| c | abc\*+c | \*( |
| / | abc\*+c | \*(/ |
| d | abc\*+cd | \*(/ |
| - | abc\*+cd/ | \*(- |
| e | abc\*+cd/e | \*(- |
| ) | abc\*+cd/e-\* |  |

Вычисление по постфиксной форме:

Алгоритм:

1. Создать пустой стек операндов.
2. Перебирать каждый символ в постфиксном выражении, начиная с первого.
3. Если символ является операндом (числом), положить его в стек операндов.
4. Если символ является оператором, то:
   * - Извлечь два операнда из стека операндов.
   * - Выполнить операцию над этими операндами с использованием оператора.
   * - Результат положить в стек операндов.
5. Повторять шаги 3-4 до тех пор, пока не будет обработан последний символ в постфиксном выражении.
6. По окончании обработки последнего символа, в стеке операндов останется ровно один элемент - результат операции.

Пример:

Давайте рассмотрим пример " 4 2 1 / 5 - +":

1. Создаем пустой стек.
2. Разбираем выражение:

* 4: является операндом, помещаем его в стек. Стек [4].
* 2: является операндом, помещаем его в стек. Стек [4, 2].
* 1: является операндом, помещаем его в стек. Стек [4, 2, 1].
* /: является оператором, извлекаем из стека два операнда (1 и 2) и выполняем операцию деления. Результат (0.5) помещаем обратно в стек. Стек [4, 0.5].
* 5: является операндом, помещаем его в стек. Стек [4, 0.5, 5].
* -: является оператором, извлекаем из стека два операнда (5 и 0.5) и выполняем операцию вычитания. Результат (-4.5) помещаем обратно в стек. Стек [4, -4.5].
* +: является оператором, извлекаем из стека два операнда (-4.5 и 4) и выполняем операцию сложения. Результат (0.5) помещаем обратно в стек. Стек [0.5].

1. Разбор выражения завершен. В стеке осталось единственное значение - результат (0.5).

## Описание программной реализации

### Описание класса STack

**class TStack {**

**private:**

**int maxSize;**

**int top;**

**T\* elem;**

**int step = 10;**

**public:**

**TStack(int maxSize = 50);**

**TStack(const TStack<T>& s);**

**virtual ~TStack();**

**void Realloc(int step);**

**bool IsEmpty(void) const;**

**bool IsFull(void) const;**

**void Push(const T& Val);**

**void Pop(void);**

**T Top();**

**};**

Назначение: представление стека.

Поля:

**maxSize** – размер стека.

**top** – индекс верхнего элемента стека.

**elem** – стек.

Методы:

**TStack(int maxSize = 50);**

Назначение: инициализация полей класса **TStack**.

Входные параметры: maxSize- размер стека.

Выходные параметры: нет.

**TStack(const TStack<T>& s)**

Назначение: получение копии существующего объекта.

Входные параметры: ссылка на копируемый объект шаблонного типа.

Выходные параметры: нет.

**virtual ~TStack()**

Назначение: удаление объекта.

Входные параметры: нет.

Выходные параметры: нет.

**void Realloc(int step)**

Назначение: перераспределение блока памяти.

Входные параметры: step-размер на который нужно увеличить стек.

Выходные параметры: нет.

**bool IsEmpty(void) const**

Назначение: проверка стека на пустоту.

Входные параметры: нет.

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

**bool IsFull(void) const**

Назначение: проверка стека на полноту.

Входные параметры: нет.

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

**void Push(const T& Val)**

Назначение: добавление элемента в стек.

Входные параметры: константная ссылка на элемент шаблонного типа.

Выходные параметры: нет.

**void Pop(void)**

Назначение: удаление верхнего элемента из стека.

Входные параметры: нет.

Выходные параметры: нет.

**T Top()**

Назначение: получение значения элемента на вершине стека.

Входные параметры: нет.

Выходные параметры: элемент шаблонного типа.

### Описание класса TArithmeticExpression

**class TArithmeticExpression**

**{**

**private:**

**string infix;**

**string str\_postfix;**

**vector<string> postfix;**

**vector<string> lexems;**

**map<string, int> priority;**

**map<string, double> operands;**

**void Parse();**

**void ToPostfix();**

**void InToPostfix();**

**vector<string> GetPostFix() const { return postfix; }**

**bool IsOperator(const string& lecsem) const;**

**bool IsConst(const string& lecsem) const;**

**bool isOperatorChar(char c) const;**

**bool isCorrectInfixExpression();**

**vector<string> GetOperands() const;**

**public:**

**TArithmeticExpression(const string& infix);**

**void SetValues();**

**string GetInfix() const { return infix; }**

**string GetPostFix\_str() const { return str\_postfix; }**

**double Calculate(const map<string, double>& values);**

**double Calculate();**

**void ShowPostfix();**

**};**

Назначение: представление арифметического выражения.

Поля:

**infix**- инфиксная форма арифметического выражения.

**postfix**- постфиксная форма арифметического выражения.

str\_postfix- строка постфиксного арифметического выражения.

**lexems**- инфиксная форма арифметического выражения с выделенными лексемами

**priority**- возможные операции с приоритетами.

**operands**- список всех операндов арифметического выражения.

Методы:

TArithmeticExpression(string infix)

Назначение: инициализация полей класса **TArithmeticExpression**.

Входные параметры: **infix** – арифметическое выражение в инфиксной форме.

Выходные параметры: нет.

void Parse()

Назначение: разбиение строки инфиксной формы арифметического выражения.

Входные параметры: нет.

Выходные параметры: нет.

void ToPostfix();

Назначение: перевод арифметического выражения в постфиксную форму.

Входные параметры: нет.

Выходные параметры: нет.

**void InToPostfix();**

Назначение: запись постфиксной форм арифметического выражения в стрку.

Входные параметры: нет.

Выходные параметры: нет.

bool IsOperator(const string& lecsem) const

Назначение: проверка является ли строка оператором.

Входные параметры: константная ссылка на строку.

Выходные параметры: 1- если сторка- оператор, 0- иначе.

**bool isOperatorChar(char c) const;**

Назначение: проверка является ли символ оператором.

Входные параметры: константная ссылка на строку.

Выходные параметры: 1- если сторка- оператор, 0- иначе.

bool IsConst(const string& lecsem) const

Назначение: проверка является ли строка числом.

Входные параметры: константная ссылка на строку.

Выходные параметры: 1- если сторка- число, 0- иначе.

string GetInfix() const

Назначение: получение инфиксной формы арифметического выражения.

Входные параметры: нет.

Выходные параметры: **infix** – инфиксная запись выражения.

vector<string> GetPostFix()

Назначение: получение постфиксной формы арифметического выражения.

Входные параметры: нет.

Выходные параметры: постфиксная запись выражения.

void ShowPostfix()

Назначение: вывод постфиксной формы арифметического выражения на экран.

Входные параметры: нет.

Выходные параметры: нет.

bool isCorrectInfixExpression()

Назначение: проверка инфиксного выражения на корректность.

Входные параметры: нет.

Выходные параметры: 1 – выражение корректно, иначе 0.

vector<string> GetOperands() const

Назначение: получение всех операндов арифметического выражения.

Входные параметры: нет.

Выходные параметры: вектор операндов арифметического выражения.

void SetValues()

Назначение: присваивание значений операндам.

Входные параметры: нет.

Выходные параметры: нет.

double Calculate(const map<string, double>& values)

Назначение: вычисление результата арифметического выражения.

Входные параметры: values – список операндов и их значений.

Выходные параметры: результат вычисления.

double Calculate()

Назначение: вычисление результата арифметического выражения.

Входные параметры: нет.

Выходные параметры: результат вычисления.

**void ShowPostfix();**

Назначение: вывод постфиксного арифметического выражения.

Входные параметры: нет.

Выходные параметры: нет.

# Заключение

Мы реализовали программу, обеспечивающую поддержку стеков, и разработали программные средства, производящие обработку арифметических выражений, включая проверку правильности записи выражения, перевод в постфиксную форму и вычисление результата.

# Литература

1. Барышева И.В., Мееров И.Б., Сысоев А.В., Шестакова Н.В. Под редакцией Гергеля В.П Учебно-методическое пособие. – Нижний Новгород: Нижегородский госуниверситет, 2017. – 105с.

# Приложения

## Приложение А. Реализация класса TStack

template <typename T>

TStack <T> ::TStack(int maxSize)

{

if (maxSize <= 0)

throw"Error";

this->maxSize = maxSize;

top = -1;

elem = new T[maxSize];

}

template <typename T>

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

{

maxSize = s.maxSize;

top = s.top;

elem = new T[maxSize];

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

elem[i] = s.elem[i];

}

template <class T>

TStack <T>::~TStack()

{

if (elem != NULL)

{

delete[] elem;

elem = NULL;

maxSize = 0;

}

}

template <class T>

void TStack<T>::Realloc(int step)

{

if (step <= 0)

throw "Error";

T\* tmp = new T[step + maxSize];

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

tmp[i] = elem[i];

delete[] elem;

elem = tmp;

maxSize = maxSize + step;

}

template <class T>

bool TStack<T>::IsEmpty(void) const

{

return(top == -1);

}

template <class T>

bool TStack<T>::IsFull(void) const

{

return(top == maxSize - 1);

}

template <class T>

void TStack<T>::Push(const T& Val)

{

if (IsFull())

{

Realloc(10);

}

top++;

elem[top] = Val;

}

template <class T>

void TStack<T>::Pop(void)

{

if (IsEmpty())

throw "Error";

top--;

}

template <class T>

T TStack<T>::Top()

{

if (IsEmpty())

throw "Error";

return elem[top];

}

## Приложение Б. Реализация класса TArithmeticExpression

TArithmeticExpression::TArithmeticExpression(const string& infx) : infix(infx)

{

priority = { {"(",1},{")",1},{"+",2},{"-",2}, {"\*",3},{"/",3}};

isCorrectInfixExpression();

ToPostfix();

InToPostfix();

}

void TArithmeticExpression::Parse()

{

string currentElement;

for (char c : infix) {

if (c == '+' || c == '-' || c == '\*' || c == '/' || c == '('||c==')') {

if (!currentElement.empty()) {

lexems.push\_back(currentElement);

currentElement = "";

}

lexems.push\_back(string(1, c));

}

else if (isdigit(c) || c == '.') {

currentElement += c;

}

else if (isalpha(c)) {

currentElement += c;

}

}

// Äîáàâëÿåì ïîñëåäíèé ýëåìåíò â âåêòîð

if (!currentElement.empty()) {

lexems.push\_back(currentElement);

}

}

vector<string> TArithmeticExpression::GetOperands() const

{

vector<string> op;

for (const auto& item : operands)

if (!IsConst(item.first))

{

op.push\_back(item.first);

}

return op;

}

bool TArithmeticExpression::IsOperator(const string& lecsem) const

{

bool flag = false;

for (const auto& c : priority)

{

if (lecsem == c.first)

{

flag = true;

}

}

return flag;

}

void TArithmeticExpression::SetValues()

{

double value;

for (auto& op : operands)

{

if (!IsConst(op.first))

{

cout << "Enter value of " << op.first << ":";

cin >> value;

operands[op.first] = value;

}

}

}

bool TArithmeticExpression::IsConst(const string& lecsem) const

{

bool hasDot = false;

bool flag = true;

for (char c : lecsem) {

if (!std::isdigit(c)) {

if (c == '.' && !hasDot) {

hasDot = true;

}

else {

flag = false;

break;

}

}

}

return flag;

}

void TArithmeticExpression::ToPostfix() {

Parse();

TStack<string> st;

string stackItem;

for (string item : lexems) {

if (item == "(") {

st.Push(item);

}

else if (item == ")") {

stackItem = st.Top();

st.Pop();

while (stackItem != "(") {

postfix.push\_back(stackItem);

stackItem = st.Top();

st.Pop();

}

}

else if (item == "+" || item == "-" || item == "\*" || item == "/") {

while (!st.IsEmpty()) {

stackItem = st.Top();

st.Pop();

if (priority[item] <= priority[stackItem])

postfix.push\_back(stackItem);

else {

st.Push(stackItem);

break;

}

}

st.Push(item);

}

else {

double value = 0.0;

if (IsConst(item)) {

value = stod(item);

}

operands.insert({ item, value });

postfix.push\_back(item);

}

}

while (!st.IsEmpty()) {

stackItem = st.Top();

st.Pop();

postfix.push\_back(stackItem);

}

}

double TArithmeticExpression::Calculate(const map<string, double>& values)

{

for (auto& val : values) {

try {

operands.at(val.first) = val.second;

}

catch (out\_of\_range& e) {}

}

TStack<double> st;

double leftOperand, rightOperand;

for (string lexem : postfix) {

if (lexem == "+") {

rightOperand = st.Top();

st.Pop();

leftOperand = st.Top();

st.Pop();

st.Push(leftOperand + rightOperand);

}

else if (lexem == "-") {

rightOperand = st.Top();

st.Pop();

leftOperand = st.Top();

st.Pop();

st.Push(leftOperand - rightOperand);

}

else if (lexem == "\*") {

rightOperand = st.Top();

st.Pop();

leftOperand = st.Top();

st.Pop();

st.Push(leftOperand \* rightOperand);

}

else if (lexem == "/") {

rightOperand = st.Top();

st.Pop();

leftOperand = st.Top();

st.Pop();

if (rightOperand == 0) {

throw"Error";

}

st.Push(leftOperand / rightOperand);

}

else {

st.Push(operands[lexem]);

}

}

return st.Top();

}

double TArithmeticExpression::Calculate()

{

return Calculate(operands);

}

bool TArithmeticExpression::isCorrectInfixExpression()

{

int count = 0;

for (char c : infix)

{

if (c == '(')

count++;

else if (c == ')')

count--;

}

if (count != 0)

throw"Error";

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

{

for (int j = i + 1; j < i + 2; j++)

{

if (isOperatorChar(infix[i]) && isOperatorChar(infix[j]))

throw"Error";

}

}

}

void TArithmeticExpression::ShowPostfix()

{

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

{

cout << postfix[i] << " ";

}

cout << endl;

}

void TArithmeticExpression::InToPostfix()

{

for (const auto& elem : postfix) {

str\_postfix += elem; // êîíêàòåíèðóåì êàæäûé ýëåìåíò â ñòðîêó result

}

}

bool TArithmeticExpression::isOperatorChar(char c) const {

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

}