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:** студент группы 3822Б1ФИ2

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ /Ясакова Т.Е./

Подпись

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

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

Подпись

Нижний Новгород  
2023

**Содержание**

[Введение 3](#_Toc154691052)

[1 Постановка задачи 4](#_Toc154691053)

[2 Руководство пользователя 5](#_Toc154691054)

[2.1 Приложение для демонстрации работы стека 5](#_Toc154691055)

[2.2 Приложение для демонстрации работы перевода арифметического выражения в постфиксную запись 6](#_Toc154691056)

[3 Руководство программиста 7](#_Toc154691057)

[3.1 Описание алгоритмов 7](#_Toc154691058)

[3.1.1 Стек 7](#_Toc154691059)

[3.1.2 Арифметическое выражение 8](#_Toc154691060)

[3.2 Описание программной реализации 11](#_Toc154691061)

[3.2.1 Описание класса Stack 11](#_Toc154691062)

[3.2.2 Описание класса Talgorithm 12](#_Toc154691063)

[Заключение 15](#_Toc154691064)

[Литература 16](#_Toc154691065)

[Приложения 17](#_Toc154691066)

[Приложение А. Реализация класса Stack 17](#_Toc154691067)

[Приложение Б. Реализация класса Talgorithm 18](#_Toc154691068)

# 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 форму.

**Задачи:**

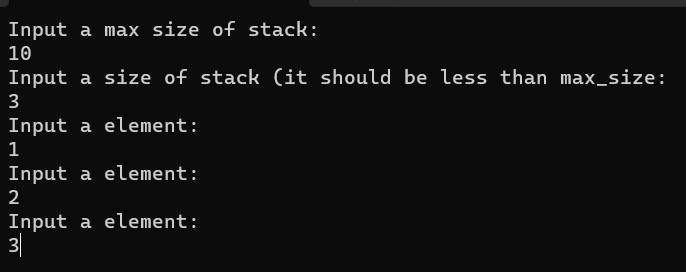
1. Изучение основных принципов работы со стеком.
2. Изучение правил преобразования инфиксного выражения в постфиксное.
3. Написание программы на С++, использующей стек для преобразования арифметического выражения.
4. Анализ времени выполнения программы и оценка эффективности использования стека для данной 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.

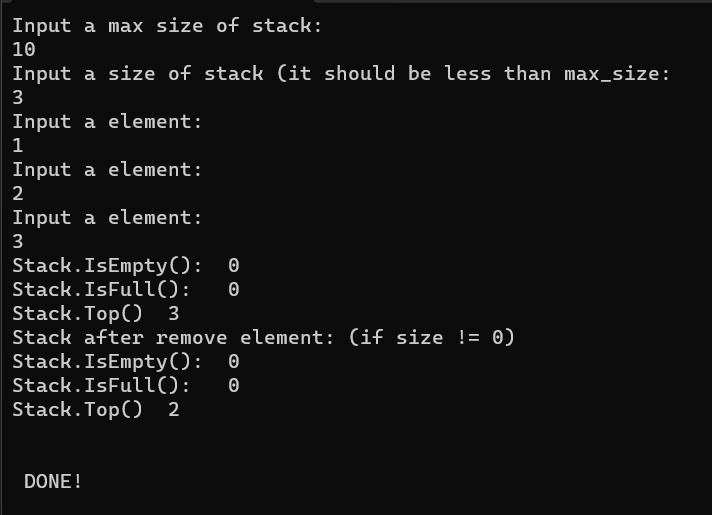
# Руководство пользователя

## Приложение для демонстрации работы стека

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



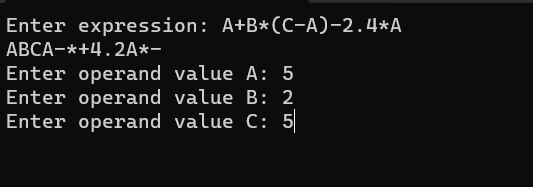
1. Основное окно программы
2. После ввода будет выведены результаты соответствующих операций и функций стека. Если введенный размер равен нулю, то метод Top() не вызывается.(рис. 2).



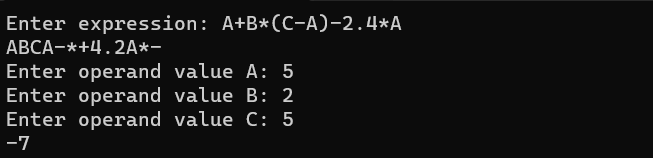
1. Результат тестирования функций класса TStack

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

1. Запустите приложение с названием sample\_talgorithm.exe. В результате появится окно, показанное ниже, вам будет предложено ввести арифметическое выражение. Затем будет выведена постфиксная форма выражения. Затем необходимо ввести используемые операнды. Операнды имеют тип числа с плавающей запятой (рис. 3).



1. Основное окно программы
2. После ввода операндов будет выведен результат соответствующей операции (рис. 4).



1. Результат тестирования функций класса Talgotithm

# 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 Stack

template <typename T> class Stack {

private:

T\* data;

int maxSize;

int top\_;

void resize(int step = 10);

public:

Stack(int size = 10);

Stack(const Stack<T>& stack);

~Stack();

T pop();

T top() const;

void push(const T& element);

bool isEmpty() const;

bool isFull() const;

Stack<T>& operator=(const Stack<T>& stack);

};

Назначение: представление стека.

Поля:

maxSize – максимальный размер стека.

\*data – память для представления элементов стека.

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

Методы:

**Stack(int size = 10);**

Назначение: конструктор по умолчанию и конструктор с параметрами.

Входные параметры:

size – максимальный размер стека (по умолчанию 10).

Выходные параметры: отсутствуют.

**Stack(const Stack<T>& stack);**

Назначение: конструктор копирования.

Входные параметры:

stack – стек, на основе которого создаем новый стек.

Выходные параметры: отсутствуют.

**~Stack();**

Назначение: деструктор.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

**T pop()**;

Назначение: удаление элемента из вершины стека.

Входные параметры: отсутствуют.

**T top() const;**

Назначение: получение элемента, находящийся в вершине стека.

Входные параметры отсутствуют.

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

void push(const T& element);

Назначение: добавление элемента в стек.

Входные параметры:

element – элемент, который добавляем.

Выходные параметры отсутствуют.

**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.

### Описание класса Talgorithm

class Talgorithm {

private:

static bool isOperator(const string& str);

static int getPrecedence(const string& op);

public:

static void isValidTalgorithm(const vector<string>& infixTalgorithm);

static vector<string> splitTalgorithm(const string& expression);

static Stack<string> infixToPostfix(const vector<string>& infixTalgorithm);

static map<string, double> getOperandValues(const vector<string>& strs);

static double evaluatePostfixTalgorithm(const map<string, double>& operandValues, Stack<string>& postfixTalgorithm);

};

Назначение: работа с инфиксной формой записи арифметических выражений

Методы:

static bool isOperator(const string& str);

Назначение: метод распознавания арифметических операторов.

Входные параметры: str – строка, представляющая арифметический оператор.

Выходные параметры: 1, если арифметический оператор, 0 иначе.

**static int getPrecedence(const string& op);**

Назначение: метод для определения приоритета оператора.

Входные параметры:

op – оператор, для которого нужно определить приоритет.

Выходные параметры:1, если оператор “+” или “-”, 2, если оператор “\*” или “/”, 0 в остальных случаях.

**static void isValidTalgorithm(const vector<string>& infixTalgorithm);**

Назначение: метод для проверки, является ли постфиксное выражение допустимым.

Входные параметры:

infixTalgorithm – вектор строк, хранящий постфиксную запись.

Выходные параметры: отсутствуют.

**static vector<string> splitTalgorithm(const string& expression);**

Назначение: метод разделения арифметического выражения.

Входные параметры:

expression – арифметическое выражение

Выходные параметры: вектор строк.

**static Stack<string> infixToPostfix(const vector<string>& infixTalgorithm);**

Назначение: метод для преобразования инфиксного выражения в постфиксное.

Входные параметры:

infixTalgorithm – вектор строк, хранящий постфиксную запись.

Выходные параметры: стек строк, каждая отдельный токен из постфиксного выражения.

**static map<string, double> getOperandValues(const vector<string>& strs);**

Назначение: метод для получения значений операндов.

Входные параметры:

strs – вектор строк, в котором хранятся значения операндов.

Выходные параметры: ключ - это операнд, а значение - это соответствующее ему числовое значение.

**static double evaluatePostfixTalgorithm(const map<string, double>& operandValues, Stack<string>& postfixTalgorithm);**

Назначение: метод вычисления постфиксного выражения.

Входные параметры:

operandValues – значения операндов.

postfixTalgoritm – стек, в котором находится постфиксное выражение.

Выходные параметры: числовое значение, которое является результатом вычисления постфиксного выражения.

# 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. Лекция «Динамическая структура данных Стек» Сысоев А.В. [https://cloud.unn.ru/s/jXmxFzAQoTDGfNe].
2. Лекция «Разбор и вычисление арифметических выражений с помощью постфиксной формы» Сысоев А.В. [<https://cloud.unn.ru/s/4Pyf24EBmowGsQ2>].

# Приложения

## Приложение А. Реализация класса Stack

#ifndef \_STACK\_H

#define \_STACK\_H

#include <iostream>

template <typename T> class Stack {

private:

T\* data;

int maxSize;

int top\_;

void resize(int step = 10);

public:

Stack(int size = 10);

Stack(const Stack<T>& stack);

~Stack();

T pop();

T top() const;

void push(const T& element);

bool isEmpty() const;

bool isFull() const;

Stack<T>& operator=(const Stack<T>& stack);

};

template <typename T> Stack<T>::Stack(int size) {

if (size <= 0) throw "maxSize must be bigger than 0";

maxSize = size;

top\_ = -1;

data = new T[maxSize];

}

template <typename T> Stack<T>::Stack(const Stack<T>& stack) {

maxSize = stack.maxSize;

top\_ = stack.top\_;

data = new T[maxSize];

for (int i = 0; i <= top\_; i++) {

data[i] = stack.data[i];

}

}

template <typename T> Stack<T>::~Stack() {

if (data != NULL) {

delete[] data;

top\_ = -1;

maxSize = 0;

}

}

template <typename T> void Stack<T>::resize(int step) {

if (step <= 0) { throw "Step<0 or step=0"; }

int newSize = maxSize + step;

T\* newData = new T[newSize];

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

newData[i] = data[i];

}

delete[] data;

data = newData;

maxSize = newSize;

}

template <typename T> bool Stack<T>::isFull() const {

return (top\_ == maxSize - 1);

}

template <typename T> bool Stack<T>::isEmpty() const {

return (top\_ == -1);

}

template <typename T> T Stack<T>::top() const {

if (isEmpty()) throw "Stack is empty";

else { return data[top\_]; }

}

template <typename T> void Stack<T>::push(const T& element) {

if (isFull()) { resize(5); }

data[++top\_] = element;

}

template <typename T> T Stack<T>::pop() {

if (isEmpty()) throw "Stack is empty!";

return data[top\_--];

}

template <typename T> Stack<T>& Stack<T>::operator=(const Stack<T>& stack) {

if (this != &stack) {

delete[] data;

maxSize = stack.maxSize;

top\_ = stack.top\_;

data = new T[maxSize];

for (int i = 0; i <= top\_; ++i) {

data[i] = stack.data[i];

}

}

return \*this;

}

#endif

## Приложение Б. Реализация класса Talgorithm

#include <iostream>

#include "talgorithm.h"

#include <set>

#include <string>

vector<string> Talgorithm::splitTalgorithm(const string& expression) {

vector<string> result;

string currentToken;

for (char ch : expression) {

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

if (!currentToken.empty()) {

result.push\_back(currentToken);

currentToken.clear();

}

result.push\_back(string(1, ch));

}

else if (isspace(ch)) {

continue;

}

else {

currentToken += string(1,ch);

}

}

if (!currentToken.empty()) {

result.push\_back(currentToken);

}

return result;

}

int Talgorithm::getPrecedence(const string& op) {

if (op == "+" || op == "-") {

return 1;

}

else if (op == "\*" || op == "/") {

return 2;

}

return 0;

}

bool Talgorithm::isOperator(const string& token) {

return (token == "+" || token == "-" || token == "\*" || token == "/");

}

void Talgorithm::isValidTalgorithm(const vector<string>& infixTalgorithm) {

if (isOperator(infixTalgorithm[0])) throw invalid\_argument("Ошибка1: \n");

else if (isOperator(infixTalgorithm[infixTalgorithm.size() - 1])) throw invalid\_argument("Ошибка2: \n");

else {

for (int i = 0; i < infixTalgorithm.size() - 1; i++) {

if (isOperator(infixTalgorithm[i]) && isOperator(infixTalgorithm[i + 1])) throw invalid\_argument("Ошибка3: \n");

if (infixTalgorithm[i] == "/" && infixTalgorithm[i + 1] == "0") throw invalid\_argument("Ошибка4: \n");

}

}

}

map<string, double> Talgorithm::getOperandValues(const vector<string>& tokens) {

map<string, double> operandValues;

for (const auto& token : tokens) {

if (token != "+" && token != "-" && token != "\*" && token != "/" && token != "(" && token != ")") {

if (operandValues.find(token) == operandValues.end()) {

double value;

if (token.find\_first\_not\_of("0123456789.") == string::npos) {

operandValues[token] = stod(token);

continue;

}

cout << "Enter operand value " << token << ": ";

cin >> value;

operandValues[token] = value;

}

}

}

return operandValues;

}

Stack<string> Talgorithm::infixToPostfix(const vector<string>& infixTalgorithm) {

Stack<string>operands;

Stack<string> operators;

for (const string& token : infixTalgorithm) {

if (isalnum(token[0])) {

operands.push(token);

}

else if (isOperator(token)) {

while (!operators.isEmpty() && getPrecedence(operators.top()) >= getPrecedence(token)) {

operands.push(operators.top());

operators.pop();

}

operators.push(token);

}

else if (token == "(") {

operators.push(token);

}

else if (token == ")") {

while (!operators.isEmpty() && operators.top() != "(") {

operands.push(operators.top());

operators.pop();

}

operators.pop();

}

}

while (!operators.isEmpty()) {

operands.push(operators.top());

operators.pop();

}

Stack<string> tmp;

string result\_inverse = "";

tmp = operands;

while (!tmp.isEmpty())

{

result\_inverse += tmp.pop();

}

string result = "";

for (int i = result\_inverse.size() - 1; i >= 0; i--)

{

result.push\_back(result\_inverse[i]);

}

cout << result << endl;

return operands;

}

double Talgorithm::evaluatePostfixTalgorithm(const map<string, double>& operandValues, Stack<string>& postfixTalgorithm) {

Stack<double> resultStack;

Stack<string> tmp = postfixTalgorithm;

while (!postfixTalgorithm.isEmpty()) {

postfixTalgorithm.pop();

}

while (!tmp.isEmpty()) {

string str = tmp.top();

postfixTalgorithm.push(str);

tmp.pop();

}

while (!postfixTalgorithm.isEmpty()) {

const string token = postfixTalgorithm.top();

postfixTalgorithm.pop();

if (!(isOperator(token))) {

if (operandValues.find(token) != operandValues.end()) {

resultStack.push(operandValues.at(token));

}

else {

cerr << "Error: operands value '" << token << "' not exist.\n";

return 0.0;

}

}

else {

double operand2 = resultStack.top();

resultStack.pop();

double operand1 = resultStack.top();

resultStack.pop();

if (token == "+") {

resultStack.push(operand1 + operand2);

}

else if (token == "-") {

resultStack.push(operand1 - operand2);

}

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

resultStack.push(operand1 \* operand2);

}

else if (token == "/") {

if (operand2 == 0) {

cerr << "ERROR: divizion by zero.\n";

return 0.0;

}

resultStack.push(operand1 / operand2);

}

}

}

return resultStack.top();

}