

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

Отчет по учебной практике
ВЫЧИСЛЕНИЕ АРИФМЕТИЧЕСКИХ
ВЫРАЖЕНИЙ (СТЕКИ)

Выполнил: Власов Максим Сергеевич, студент
группы 381806-1

Проверил: к. т. н., доцент кафедры МОСТ Кустикова
В. Д.

Нижний Новгород
2019

Содержание

ВВЕДЕНИЕ.....	4
1. ПОСТАНОВКА ЗАДАЧИ	5
2. РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ	6
3. РУКОВОДСТВО ПРОГРАММИСТА.....	8
3.1. СТРУКТУРА ПРОГРАММЫ	8
3.2. ОПИСАНИЕ АЛГОРИТМА.....	8
3.2.1. Структура данных Стек	8
3.2.2. Перевод арифметического выражения в постфиксную форму	8
3.2.3. Вычисление значения выражения, представленного в постфиксной форме	10
3.3. ОПИСАНИЕ СТРУКТУР ДАННЫХ И ФУНКЦИЙ	12
3.3.1. Класс <i>TStack</i>	12
3.3.2. Класс <i>PostfixFormProcessor</i>	14
3.3.3. Основная программа	18
ЗАКЛЮЧЕНИЕ	19
СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ.....	20
ПРИЛОЖЕНИЕ.....	21

Введение

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

Таким образом, для этого и было создано консольное приложение, работающее с математическими выражениями, изначально представленными в классической (человеко-понятной, инфиксной) форме записи. При этом используется динамическая структура данных Стек, на практическое освоение которой и направлена данная лабораторная работа.

1. Постановка задачи

Задача: разработать и реализовать приложение, вычисляющее арифметические выражения, представленные в инфиксной форме.

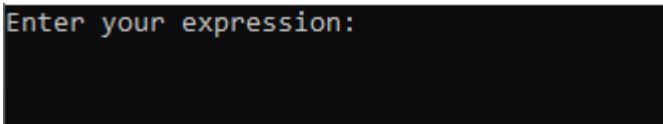
Входные данные: арифметическое выражение в инфиксной форме, значения переменных.

Выходные данные: постфиксная форма выражения, результат вычисления значения выражения.

2. Руководство пользователя

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

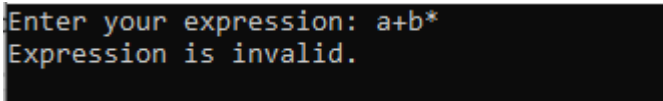
1. Запустите файл **02_Stack.exe** из папки с программой. Перед вами отобразится приветственный экран с предложением ввести арифметическое выражение, представленное в классической (инфиксной) форме.



```
Enter your expression:
```

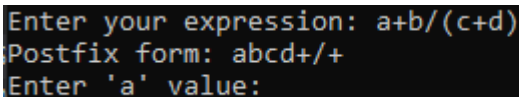
Рис 1. Программа после запуска.

2. С помощью клавиатуры введите выражение и нажмите Enter. Если вы указали неверное выражение (например, присутствуют недопустимые символы, то есть, любые, кроме a-z, A-Z, +, -, *, /, (,), или допущена логическая ошибка, как-то: более одного операнда рядом, не соблюдена вложенность скобок, бинарные операторы имеют менее двух операндов, или названия переменных содержат более одного символа), то программа сообщит вам об этом соответствующим сообщением об ошибке (см. Рис. 2). Повторите ввод, перезапустив приложение. Если арифметическое выражение корректное, программа преобразует его в постфиксную форму и выведет (см. Рис. 3).



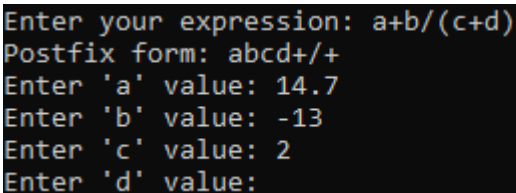
```
Enter your expression: a+b*  
Expression is invalid.
```

Рис 2. Сообщение об ошибке при вводе неверного выражения.



```
Enter your expression: a+b/(c+d)  
Postfix form: abcd+/  
Enter 'a' value:
```

Рис 3. Программа после ввода корректного выражения.



```
Enter 'a' value: 14.7  
Enter 'b' value: -13  
Enter 'c' value: 2  
Enter 'd' value:
```

Рис 4. Программа во время ввода значений переменных.

3. Если арифметическое выражение введено верно (программа не обнаружила ошибок), то далее вам будет предложено ввести значения переменных. Последовательно введите значения указанных переменных, нажимая Enter после ввода значения (см. Рис. 4).

```
Enter your expression: a+b/(c+d)
Postfix form: abcd+/*
Enter 'a' value: 14.7
Enter 'b' value: -13
Enter 'c' value: 2
Enter 'd' value: -2
Cannot divide by zero.
```

Рис 5. Сообщение об ошибке при делении на ноль.

4. После ввода значений переменных программа выполнит попытку вычисления значения выражения. В случае если на каком-то этапе вычислений будет осуществлено деление на ноль, программа покажет сообщение об ошибке (см. Рис. 5). Перезапустите приложение и введите выражение и допустимые значения переменных заново. Если вычисление завершится успешно, программа выведет его (см. Рис. 6). После этого вы можете закрыть программу, для этого нажмите «крестик» в правом верхнем углу окна.

```
Enter your expression: a+b*(c+d)
Postfix form: abcd+*+
Enter 'a' value: 14.7
Enter 'b' value: -13
Enter 'c' value: 2
Enter 'd' value: -2.1

Result of expression calculation: 16
```

Рис 6. Успешное завершение программы.

3. Руководство программиста

3.1. Структура программы

Исходный код программы содержится в следующих файлах:

1. **02_Stack.cpp** – основной модуль (консольное приложение, использующее модули ниже).
2. **TStack.h** – модуль Стек (содержит объявление и реализацию класса для работы со структурой данных Стек, включая классы исключений)
3. **PostfixFormProcessor.h** – модуль Обработчик для постфиксной формы (содержит объявление класса для работы с постфиксной формой, включая классы исключений и структуру для хранения значений переменных).
4. **PostfixFormProcessor.cpp** – модуль Обработчик для постфиксной формы (содержит реализацию методов класса для работы с постфиксной формой).

3.2. Описание алгоритма

3.2.1. Структура данных Стек

Стек – схема запоминания информации, при которой каждый вновь поступающий ее элемент как бы «проталкивает» вглубь отведенного участка памяти находящиеся там элементы и занимает крайнее положение (так называемую вершину стека). При выдаче информации из стека выдается элемент, расположенный в вершине стека, а оставшиеся элементы продвигаются к вершине; следовательно, элемент, поступивший последним, выдается первым.

3.2.2. Перевод арифметического выражения в постфиксную форму

Приложение использует алгоритм перевода арифметического выражения, представленного в классической (инфиксной) форме в постфиксную. Этот алгоритм основан на использовании стека. Ниже подробно рассмотрен принцип работы алгоритма.

1. На вход алгоритму подается строка символов, содержащая арифметическое выражение в инфиксной форме (считаем, что выражение верное – каждый символ является либо оператором, либо операндом).
2. Входная строка просматривается слева направо от начала до конца. Символы попадают в один из стеков: стек №1 (для операций) или стек №2 (для операндов и операций). На

i-м шаге от 1 до N, где N – длина строки (под текущим символом будем понимать i-й символ строки):

3. если текущий символ – операнд, то кладем его в стек №2;
4. если текущий символ – открывающая скобка, то кладем ее в стек №1;
5. если текущий символ – закрывающая скобка, то последовательно перекладываем операции из стека №1 в стек №2 до тех пор, пока не встретим открывающую скобку, после чего удаляем ее из стека №1;
6. если текущий символ – оператор (+, −, *, /), то в случае если приоритет этой операции больше либо равен приоритету операции, находящейся на вершине стека №1, кладем ее в стек №1, иначе перекладываем более приоритетные либо равные по приоритету операции из стека №1 в стек №2 (считаем приоритет операций *, / выше приоритета операций +, −, которые, в свою очередь, более приоритетные по сравнению с ().
7. После прохождения N шагов в пункте 2 необходимо переложить все символы из стека №1 в стек №2.

В результате в стеке №2 будет находиться арифметическое выражение в постфиксной форме, причем в обратном порядке – при извлечении символов необходимо записывать их в строку, начиная с ее конца. В качестве примера рассмотрим выражение $A+B/(C+D)*(E-F)+G$:

Таблица 1. Пример преобразования выражения в постфиксную форму.

Шаг 1	$A+B/(C+D)*(E-F)+G$														
	Стек №1														
	Стек №2	A													
Шаг 2	$A+B/(C+D)*(E-F)+G$														
	Стек №1	+													
	Стек №2	A													
Шаг 3	$A+B/(C+D)*(E-F)+G$														
	Стек №1	+													
	Стек №2	A	B												
Шаг 4	$A+B/(C+D)*(E-F)+G$														
	Стек №1	+	/												
	Стек №2	A	B												
Шаг 5	$A+B/(C+D)*(E-F)+G$														
	Стек №1	+	/	(
	Стек №2	A	B												
Шаг 6	$A+B/(C+D)*(E-F)+G$														
	Стек №1	+	/	(
	Стек №2	A	B	C											

Шаг 7	$A+B/(C+D)*(E-F)+G$														
	Стек №1	+	/	(+										
	Стек №2	A	B	C											
Шаг 8	$A+B/(C+D)*(E-F)+G$														
	Стек №1	+	/	(+										
	Стек №2	A	B	C	D										
Шаг 9	$A+B/(C+D)*(E-F)+G$														
	Стек №1	+	/												
	Стек №2	A	B	C	D	+									
Шаг 10	$A+B/(C+D)*(E-F)+G$														
	Стек №1	+	/	*											
	Стек №2	A	B	C	D	+									
Шаг 11	$A+B/(C+D)*(E-F)+G$														
	Стек №1	+	/	*	(
	Стек №2	A	B	C	D	+									
Шаг 12	$A+B/(C+D)*(E-F)+G$														
	Стек №1	+	/	*	(
	Стек №2	A	B	C	D	+	E								
Шаг 13	$A+B/(C+D)*(E-F)+G$														
	Стек №1	+	/	*	(-									
	Стек №2	A	B	C	D	+	E	F							
Шаг 14	$A+B/(C+D)*(E-F)+G$														
	Стек №1	+	/	*	(-									
	Стек №2	A	B	C	D	+	E	F							
Шаг 15	$A+B/(C+D)*(E-F)+G$														
	Стек №1	+	/	*											
	Стек №2	A	B	C	D	+	E	F	-						
Шаг 16	$A+B/(C+D)*(E-F)+G$														
	Стек №1	+													
	Стек №2	A	B	C	D	+	E	F	-	*	/	+			
Шаг 17	$A+B/(C+D)*(E-F)+G$														
	Стек №1	+													
	Стек №2	A	B	C	D	+	E	F	-	*	/	+	G		
Шаг 18	$A+B/(C+D)*(E-F)+G$														
	Стек №1														
	Стек №2	A	B	C	D	+	E	F	-	*	/	+	G	+	

3.2.3. Вычисление значения выражения, представленного в постфиксной форме

Вычисление значения выражения, представленного в постфиксной форме, происходит следующим образом: арифметическое выражение просматривается посимвольно слева направо. На i -м шаге при $i = 1, 2, \dots, N$, где N – длина постфиксного выражения:

1. Если i -й символ – операнд, то получаем его значение и помещаем в стек.

2. Если i -й символ – оператор, то из стека последовательно извлекаются значения x , y и выполняется оператор, причем таким образом, что x – правостороннее значение, а y – левостороннее, результат помещается в стек.

По окончании просмотра строки в стеке должно находиться единственное значение – это и будет результат вычисления выражения, представленного в постфиксной форме. В качестве примера рассмотрим выражение $ABCD+EF-*/+G+$:

Таблица 2. Пример вычисления выражения, представленного в постфиксной форме.

Шаг 1	$ABCD+EF-*/+G+$					
	Стек	A				
Шаг 2	$ABCD+EF-*/+G+$					
	Стек	A	B			
Шаг 3	$ABCD+EF-*/+G+$					
	Стек	A	B	C		
Шаг 4	$ABCD+EF-*/+G+$					
	Стек	A	B	C	D	
Шаг 5	$ABCD+EF-*/+G+$					
	Стек	A	B	C+D		
Шаг 6	$ABCD+EF-*/+G+$					
	Стек	A	B	C+D	E	
Шаг 7	$ABCD+EF-*/+G+$					
	Стек	A	B	C+D	E	F
Шаг 8	$ABCD+EF-*/+G+$					
	Стек	A	B	C+D	E-F	
Шаг 9	$ABCD+EF-*/+G+$					
	Стек	A	B	$(C+D)*(E-F)$		
Шаг 10	$ABCD+EF-*/+G+$					
	Стек	A	$B/(C+D)*(E-F)$			
Шаг 11	$ABCD+EF-*/+G+$					
	Стек	$A+B/(C+D)*(E-F)$				
Шаг 12	$ABCD+EF-*/+G+$					
	Стек	$A+B/(C+D)*(E-F)$	G			
Шаг 13	$ABCD+EF-*/+G+$					
	Стек	$A+B/(C+D)*(E-F)+G$				

3.3. Описание структур данных и функций

3.3.1. Класс TStack

Объявление класса для работы со стеком выглядит следующим образом:

```
template <typename ValueType>
class TStack
{
    size_t size;
    size_t nextEmpty;
    ValueType* elements;
public:
    explicit TStack(size_t size = 0);
    TStack(const TStack& other);
    ~TStack();

    void push(ValueType value);
    ValueType top() const;
    void pop();

    size_t height() const;
    size_t capacity() const;
    bool empty() const;
    bool full() const;

    class FullError : std::exception
    {
        const std::string whatStr = "Stack is full.";
    public:
        virtual const char* what() { return whatStr.c_str(); }
    };

    class EmptyError : std::exception
    {
        const std::string whatStr = "Stack is empty.";
    public:
        virtual const char* what() { return whatStr.c_str(); }
    };
};
```

3.3.1.1. Классы исключений

1. `FullError` – стек полон (при попытке добавить элемент).
2. `EmptyError` – стек пуст (при попытке извлечь элемент).

3.3.1.2. Поля класса

```
size_t size;
```

Назначение: хранит максимальное количество элементов, которое можно хранить в данном стеке.

```
size_t nextEmpty;
```

Назначение: хранит индекс следующего доступного для перезаписи элемента массива.

```
ValueType* elements;
```

Назначение: хранит указатель на начало массива, выделенного для хранения элементов (`typename ValueType` – шаблонный параметр класса; тип элементов, хранимых в стеке).

3.3.1.3. Методы класса

```
explicit TStack(size_t size = 0);
```

Назначение: конструктор (выделение размера и инициализация членов-данных).

Входные параметры: `size` – максимальное количество элементов стека.

Выходные данные: отсутствуют.

```
TStack(const TStack& other);
```

Назначение: конструктор копирования.

Входные параметры: `other` – копируемый стек.

Выходные данные: отсутствуют.

```
~TStack();
```

Назначение: деструктор.

Входные параметры: отсутствуют.

Выходные данные: отсутствуют.

```
void push(ValueType value);
```

Назначение: добавление элемента на вершину стека.

Входные параметры: `value` – значение добавляемого элемента.

Выходные данные: отсутствуют.

```
ValueType top() const;
```

Назначение: получение элемента с вершины стека (без его изъятия).

Входные параметры: отсутствуют.

Выходные данные: копия элемента с вершины стека.

```
ValueType pop();
```

Назначение: извлечение элемента с вершины стека.

Входные параметры: отсутствуют.

Выходные данные: элемент с вершины стека.

```
size_t height() const;
```

Назначение: получение количества элементов, содержащихся в стеке.

Входные параметры: отсутствуют.

Выходные данные: количество элементов, содержащихся в стеке.

```
size_t capacity() const;
```

Назначение: получение количества элементов, которые можно хранить в стеке (вместимость).

Входные параметры: отсутствуют.

Выходные данные: количество элементов, которые можно хранить в стеке (вместимость).

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

Назначение: получение информации о том, является ли стек пустым.

Входные параметры: отсутствуют.

Выходные данные: истина или ложь — является ли стек пустым.

```
bool full() const;
```

Назначение: получение информации о том, является ли стек полным.

Входные параметры: отсутствуют.

Выходные данные: истина или ложь — является ли стек полным.

3.3.2. Класс PostfixFormProcessor

Для работы с арифметическими выражениями и их постфиксными формами был спроектирован класс `PostfixFormProcessor`, содержащий набор статических методов, а также вспомогательные пользовательские типы данных и перечисления. Объявление представлено ниже.

```
class PostfixFormProcessor
{
public:
```

```

struct Variable
{
    char    name  = '\0';
    double value = 0.0;
};

struct Variables
{
    Variable* variables = nullptr;
    size_t    count     = 0U;
};

private:
    enum class TokenType
    {
        unknown,
        operand,
        operation,
        closingBrace,
        space
    };

    enum class Priority
    {
        notLower,
        notHigher,
        unknown
    };

    static TokenType checkToken(const char token);
    static Priority  checkPriority(const char first, const char second);
    static Variable findVariableByName(const Variables variables, const char
name);
    static bool checkExpression(const std::string& expression);
    static bool test(const std::string& postfixForm);
public:
    static std::string parse(const std::string& expression);
    static std::string findVariables(const std::string& expression);
    static double calculate(const std::string& postfixForm, const Variables
variables);

    class InvalidExpressionError : std::exception
    {
        const std::string whatStr = "Expression is invalid.";
    }

```

```

public:
    virtual const char* what() { return whatStr.c_str(); }
};
class InvalidPostfixFormError : InvalidExpressionError
{
    const std::string whatStr = "Postfix form is invalid.";
public:
    virtual const char* what() { return whatStr.c_str(); }
};
class UndefinedVariable : std::exception
{
    const std::string whatStr = "Variable not defined in variables array.";
public:
    virtual const char* what() { return whatStr.c_str(); }
};
class DivisionByZero : std::exception
{
    const std::string whatStr = "Cannot divide by zero.";
public:
    virtual const char* what() { return whatStr.c_str(); }
};
};

```

3.3.2.1. Пользовательские типы данных

Класс содержит следующие типы данных:

1. `struct Variable` – хранение информации об имени и значении переменной, используемой в арифметическом выражении.
2. `struct Variables` – агрегатор динамического массива, содержащего данные о переменных.
3. `enum class TokenType` – перечисление, содержащее возможные типы (классификацию) символов (токенов) арифметического выражения.
4. `enum class Priority` – перечисление, содержащее виды отношений порядка приоритетов арифметических операторов.

3.3.2.2. Классы исключений

1. `InvalidExpressionError` – неверное арифметическое выражение (в инфиксной форме).
2. `InvalidPostfixFormError` – неверное арифметическое выражение (в постфиксной форме).
3. `UndefinedVariable` – неизвестная переменная (при обращении к ней в выражении).
4. `DivisionByZero` – попытка деления на ноль (при вычислении значения выражения).

3.3.2.3. Методы класса

```
static TokenType checkToken(const char token);
```

Назначение: определение типа (классификация) символа (токена) в арифметическом выражении.

Входные параметры: `token` – проверяемый символ.

Выходные данные: тип символа.

```
static Priority checkPriority(const char first, const char second);
```

Назначение: определение отношения порядка приоритета арифметических операций.

Входные параметры: `first`, `second` – символы арифметических операций.

Выходные данные: отношение порядка приоритета между операциями.

```
static Variable findVariableByName(const Variables variables, const char name);
```

Назначение: поиск переменной в массиве переменных по ее имени.

Входные параметры: `variables` – массив переменных, `name` – имя переменной.

Выходные данные: имя и значение переменной.

```
static std::string findVariables(const std::string& expression);
```

Назначение: поиск лексем-переменных в арифметическом выражении.

Входные параметры: `expression` – арифметическое выражение.

Выходные данные: строка из неповторяющихся символов – имен найденных переменных.

```
static bool checkExpression(const std::string& expression);
```

Назначение: проверка корректности выражения, заданного в инфиксной форме.

Входные параметры: `expression` – арифметическое выражение.

Выходные данные: истина или ложь.

```
static std::string parse(const std::string& expression);
```

Назначение: преобразование инфиксной формы арифметического выражения в постфиксную.

Входные параметры: expression – выражение, заданное в инфиксной форме, testFinally – значение логического типа, определяющее, нужно ли выполнить тестирование выражения после преобразования.

Выходные данные: арифметическое выражение в постфиксной форме.

```
static bool test(const std::string& postfixForm);
```

Назначение: тестирование арифметического выражения в постфиксной форме.

Входные параметры: postfixForm - выражение.

Выходные данные: истина или ложь – успешно ли выполнилось тестирование.

```
static double calculate(const std::string& postfixForm, const  
Variables variables);
```

Назначение: вычисление значения выражения, представленного в постфиксной форме.

Входные параметры: postfixForm – выражение в постфиксной форме, variables – массив переменных и их значений.

Выходные данные: значение выражения.

3.3.3. Основная программа

```
int main()
```

Назначение: основная функция (точка входа).

Входные параметры: отсутствуют.

Заключение

В ходе выполнения практической работы «Вычисление арифметических выражений (стеки)» было разработано и реализовано консольное приложение для работы с арифметическими выражениями, представленными в инфиксной и постфиксной форме, включая вычисление значения, использующее динамическую структуру данных Стек.

Список используемых источников

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

Приложение

Приложение 1. Исходный код основной программы

```
#include <iostream>
#include "PostfixFormProcessor.h"

int main()
{
    std::string expression;
    std::cout << "Enter your expression: ";
    std::getline(std::cin, expression, '\n');
    std::string postfixForm, variablesNames;
    PostfixFormProcessor::Variables variables;
    try
    {
        postfixForm = PostfixFormProcessor::parse(expression);
        variablesNames = PostfixFormProcessor::findVariables(expression);
        std::cout << "Postfix form: " << postfixForm << '\n';
    }
    catch (PostfixFormProcessor::InvalidExpressionError& e)
    {
        std::cerr << e.what() << '\n';
        return 1;
    }
    variables.variables = new PostfixFormProcessor::Variable[variables.count =
variablesNames.size()];
    size_t i = 0;
    for (std::string::iterator variableName = variablesNames.begin();
variableName != variablesNames.end(); variableName++)
    {
        variables.variables[i].name = *variableName;
        std::cout << "Enter '\" << *variableName << "\" value: ";
        std::cin >> variables.variables[i].value;
        i++;
    }
    try
    {
```

```

        double    result    =    PostfixFormProcessor::calculate(postfixForm,
variables);
        std::cout << "\nResult of expression calculation: " << result;
    }
    catch (PostfixFormProcessor::InvalidPostfixFormError& e)
    {
        std::cerr << e.what() << '\n';
        return 2;
    }
    catch (PostfixFormProcessor::DivisionByZero& e)
    {
        std::cerr << e.what() << '\n';
        return 3;
    }
    delete[] variables.variables;
    variables.variables = nullptr;
    variables.count = 0ull;
}

```

```
#ifndef _TSTACK_H_
#define _TSTACK_H_

#include <string>
#include <cstring>
#include <exception>

template <typename ValueType>
class TStack
{
    size_t size;
    size_t nextEmpty;
    ValueType* elements;
public:
    explicit TStack(size_t size = 0);
    TStack(const TStack& other);
    ~TStack();

    TStack<ValueType>& push(ValueType value);
    ValueType top() const;
    void pop();

    size_t height() const;
    size_t capacity() const;
    bool empty() const;
    bool full() const;

    class FullError : std::exception
    {
        const std::string whatStr = "Stack is full.";
    public:
        virtual const char* what() { return whatStr.c_str(); }
    };

    class EmptyError : std::exception
    {
        const std::string whatStr = "Stack is empty.";
    public:
        virtual const char* what() { return whatStr.c_str(); }
    };
};
```

```

};

template<typename ValueType>
TStack<ValueType>::TStack(size_t size) : size(size)
{
    nextEmpty = 0;
    elements = size ? new ValueType[size] : nullptr;
}

template<typename ValueType>
TStack<ValueType>::TStack(const TStack& other) : size(other.size),
nextEmpty(other.nextEmpty)
{
    elements = size ? new ValueType[size] : nullptr;
    memcpy(elements, other.elements, size * sizeof(ValueType));
}

template<typename ValueType>
TStack<ValueType>::~~TStack()
{
    if (elements)
        delete[] elements;
}

template<typename ValueType>
TStack<ValueType>& TStack<ValueType>::push(ValueType value)
{
    if (full())
        throw FullError();
    elements[nextEmpty++] = value;
    return *this;
}

template<typename ValueType>
ValueType TStack<ValueType>::top() const
{
    if (empty())
        throw EmptyError();
    return elements[nextEmpty - 1];
}

```



```

}

template<typename ValueType>
void TStack<ValueType>::pop()
{
    if (empty())
        throw EmptyError();
    nextEmpty--;
}

template<typename ValueType>
size_t TStack<ValueType>::height() const
{
    return nextEmpty;
}

template<typename ValueType>
size_t TStack<ValueType>::capacity() const
{
    return size;
}

template<typename ValueType>
bool TStack<ValueType>::empty() const
{
    return nextEmpty == 0;
}

template<typename ValueType>
bool TStack<ValueType>::full() const
{
    return nextEmpty == size;
}

#endif // !_TSTACK_H_

```

Приложение 3. Исходный код PostfixFormProcessor.h

```
#ifndef _POSTFIXFORMPROCESSOR_H_
#define _POSTFIXFORMPROCESSOR_H_

#include "TStack.h"
#include <string>

class PostfixFormProcessor
{
public:
    struct Variable
    {
        char    name  = '\0';
        double value = 0.0;
    };
    struct Variables
    {
        Variable* variables = nullptr;
        size_t    count     = 0U;
    };
private:
    enum class TokenType
    {
        unknown,
        operand,
        operation,
        closingBrace,
        space
    };
    enum class Priority
    {
        notLower,
        notHigher,
        unknown
    };
    static TokenType checkToken(const char token);
    static Priority checkPriority(const char first, const char second);
    static Variable findVariableByName(const Variables variables, const char
name);
    static bool checkExpression(const std::string& expression);
```

```

        static bool test(const std::string& postfixForm);
public:
    static std::string parse(const std::string& expression);
    static std::string findVariables(const std::string& expression);
    static double calculate(const std::string& postfixForm, const Variables
variables);

class InvalidExpressionError : std::exception
{
    const std::string whatStr = "Expression is invalid.";
public:
    virtual const char* what() { return whatStr.c_str(); }
};
class InvalidPostfixFormError : InvalidExpressionError
{
    const std::string whatStr = "Postfix form is invalid.";
public:
    virtual const char* what() { return whatStr.c_str(); }
};
class UndefinedVariable : std::exception
{
    const std::string whatStr = "Variable not defined in variables array.";
public:
    virtual const char* what() { return whatStr.c_str(); }
};
class DivisionByZero : std::exception
{
    const std::string whatStr = "Cannot divide by zero.";
public:
    virtual const char* what() { return whatStr.c_str(); }
};
};

#endif // !_POSTFIXFORMPROCESSOR_H_

```

Приложение 4. Исходный код PostfixFormProcessor.cpp

```
#include "PostfixFormProcessor.h"

PostfixFormProcessor::TokenType PostfixFormProcessor::checkToken(const char
token)
{
    if ((token == ' ') || (token == '\n') || (token == '\t') || (token ==
'\b'))
        return TokenType::space;
    if (token == ')')
        return TokenType::closingBrace;
    if (((token >= 'a') && (token <= 'z')) || ((token >= 'A') && (token <=
'Z'))))
        return TokenType::operand;
    if ((token == '+') || (token == '-') || (token == '*') || (token == '/')
|| (token == '('))
        return TokenType::operation;
    return TokenType::unknown;
}

PostfixFormProcessor::Priority PostfixFormProcessor::checkPriority(const char
first, const char second)
{
    if ((checkToken(first) != TokenType::operation) || (checkToken(second) !=
TokenType::operation))
        return Priority::unknown;
    int numericPriorityFirst = 0, numericPrioritySecond = 0;
    if ((first == '*') || (first == '/'))
        numericPriorityFirst = 3;
    else if ((first == '+') || (first == '-'))
        numericPriorityFirst = 2;
    else
        numericPriorityFirst = 1;
    if ((second == '*') || (second == '/'))
        numericPrioritySecond = 3;
    else if ((second == '+') || (second == '-'))
        numericPrioritySecond = 2;
    else
        numericPrioritySecond = 1;
    if (numericPriorityFirst >= numericPrioritySecond)
```

```

        return Priority::notLower;
    if (numericPriorityFirst <= numericPrioritySecond)
        return Priority::notHigher;
    return Priority::unknown;
}

PostfixFormProcessor::Variable PostfixFormProcessor::findVariableByName(const
Variables variables, const char name)
{
    for (size_t i = 0; i < variables.count; i++)
    {
        const Variable& variable = variables.variables[i];
        if (variable.name == name)
            return Variable(variable);
    }
    return Variable();
}

std::string PostfixFormProcessor::findVariables(const std::string& expression)
{
    if (!checkExpression(expression))
        throw InvalidExpressionError();
    std::string variablesNames;
    for (std::string::const_iterator token = expression.begin(); token !=
expression.end(); token++)
    {
        TokenType type = checkToken(*token);
        if (type == TokenType::operand)
        {
            bool isFirstTime = true;
            for (std::string::const_iterator prevToken =
expression.begin(); prevToken != token; prevToken++)
                if (*token == *prevToken)
                    isFirstTime = false;
            if (isFirstTime)
                variablesNames += *token;
        }
    }
    return variablesNames;
}

```

```

bool PostfixFormProcessor::checkExpression(const std::string& exp)
{
    std::string expression = exp.substr(exp.find_first_not_of(' '));
    unsigned openingBracesCount = 0, closingBracesCount = 0;
    size_t i = 0;
    TokenType prev = checkToken(expression[0]);
    if ((prev != TokenType::operand) && (expression[0] != '('))
        return false;
    if (expression[0] == '(')
        openingBracesCount++;
    for (std::string::const_iterator token = ++expression.begin(); token !=
expression.end(); token++, i++)
    {
        TokenType current = checkToken(*token);
        if (current == TokenType::space)
            continue;
        if (current == TokenType::closingBrace)
        {
            closingBracesCount++;
            if (prev != TokenType::operand)
                return false;
            if (openingBracesCount < closingBracesCount)
                return false;
        }
        else if (*token == '(')
        {
            openingBracesCount++;
            if (prev != TokenType::operation)
                return false;
        }
        else if (current == TokenType::operand)
        {
            if (prev != TokenType::operation)
                return false;
        }
        else if (current == TokenType::operation)
        {
            if ((prev != TokenType::operand) && (prev !=
TokenType::closingBrace))

```

```

        return false;
    }
    prev = current;
}
if (prev == TokenType::operation)
    return false;
if (closingBracesCount != openingBracesCount)
    return false;
return true;
}

std::string PostfixFormProcessor::parse(const std::string& expression)
{
    if(!checkExpression(expression))
        throw InvalidExpressionError();
    TStack<char> postfixForm(expression.size()), operations(expression.size());
    for (std::string::const_iterator token = expression.begin(); token !=
expression.end(); token++)
    {
        TokenType type = checkToken(*token);
        if (type == TokenType::space)
            continue;
        if (type == TokenType::operand)
            postfixForm.push(*token);
        else if (*token == '(')
        {
            operations.push(*token);
        }
        else if (type == TokenType::closingBrace)
        {
            while (operations.top() != '(')
            {
                postfixForm.push(operations.top());
                operations.pop();
            }
            if (!operations.empty())
                operations.pop(); // remove '('
        }
        else if (type == TokenType::operation)
        {

```

```

        if(operations.empty() || (checkPriority(operations.top(),
*token) == Priority::notHigher))
            operations.push(*token);
        else
        {
            while (!operations.empty() &&
(checkPriority(operations.top(), *token) == Priority::notLower))
            {
                postfixForm.push(operations.top());
                operations.pop();
            }
            operations.push(*token);
        }
    }
    else
        throw InvalidExpressionError(); // something unknown
}
while (!operations.empty())
{
    postfixForm.push(operations.top());
    operations.pop();
}
std::string reversedResult;
while (!postfixForm.empty())
{
    reversedResult += postfixForm.top();
    postfixForm.pop();
}
std::string result;
for (std::string::reverse_iterator i = reversedResult.rbegin(); i !=
reversedResult.rend(); i++)
{
    result += *i;
}
if (!test(result))
    throw InvalidExpressionError();
return result;
}

```

```

bool PostfixFormProcessor::test(const std::string& postfixForm)

```



```

{
    TStack<double> calculated(postfixForm.size());
    for (std::string::const_iterator i = postfixForm.begin(); i !=
postfixForm.end(); i++)
    {
        TokenType type = checkToken(*i);
        if (type == TokenType::operand)
            calculated.push(1.0);
        else if (type == TokenType::operation)
        {
            double first, second;
            try
            {
                second = calculated.top();
                calculated.pop();
                first = calculated.top();
                calculated.pop();

            }
            catch (TStack<double>::EmptyError&)
            {
                return false;
            }
            if (*i == '+')
                calculated.push(first + second);
            else if (*i == '-')
                calculated.push(first - second);
            else if (*i == '*')
                calculated.push(first * second);
            else if (*i == '/')
            {
                if (second == 0.0)
                    second = 1.0;
                calculated.push(first / second);
            }
            else
                return false;
        }
        else
            return false;
    }
}

```

```

        try
        {
            calculated.pop();
        }
        catch (TStack<double>::EmptyError&)
        {
            return false;
        }
        return true;
    }

double PostfixFormProcessor::calculate(const std::string& postfixForm, const
Variables variables)
{
    TStack<double> calculated(postfixForm.size());
    for (std::string::const_iterator i = postfixForm.begin(); i !=
postfixForm.end(); i++)
    {
        TokenType type = checkToken(*i);
        if (type == TokenType::operand)
        {
            Variable variable = findVariableByName(variables, *i);
            if (!variable.name)
                throw UndefinedVariable();
            calculated.push(variable.value);
        }
        else if (type == TokenType::operation)
        {
            double first, second;
            try
            {
                second = calculated.top();
                calculated.pop();
                first = calculated.top();
                calculated.pop();
            }
            catch (TStack<double>::EmptyError&)
            {
                throw InvalidPostfixFormError();
            }
        }
    }
}

```

```

        if (*i == '+')
            calculated.push(first + second);
        else if (*i == '-')
            calculated.push(first - second);
        else if (*i == '*')
            calculated.push(first * second);
        else if (*i == '/')
        {
            if (second == 0.0)
                throw DivisionByZero();
            calculated.push(first / second);
        }
        else
            throw InvalidPostfixFormError();
    }
    else
        throw InvalidPostfixFormError();
}

double result = 0.0;
try
{
    result = calculated.top();
}
catch(TStack<double>::EmptyError&)
{
    throw InvalidPostfixFormError();
}
return result;
}

```