

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
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:
«Стек. Постфиксная форма.»

Выполнил(а): студент группы
3822Б1ФИ2

_____ / Савченко М.П./
Подпись

Проверил: к.т.н, доцент каф. ВВиСП
_____ / Кустикова В.Д./
Подпись

Нижний Новгород
2023

Содержание

Введение.....	3
1 Постановка задачи.....	5
2 Руководство пользователя.....	6
2.1 Приложение для демонстрации работы стека.....	6
2.2 Приложение для демонстрации работы постфиксной формы арифметического выражения	8
3 Руководство программиста	9
3.1 Описание алгоритмов	9
3.1.1 Стек.....	9
3.1.2 Постфиксная форма	9
3.2 Описание программной реализации	12
3.2.1 Описание класса TStack.....	12
3.2.2 Описание класса TArithmeticExpression	14
Заключение	17
Литература	18
Приложения	19
Приложение А. Реализация класса TStack	19
Приложение Б. Реализация класса TArithmeticExpression.....	21

Введение

Стек и постфиксная (или обратная польская) форма арифметического выражения - это концепции, которые могут быть актуальными и полезными в различных областях программирования и вычислительной математики.

1. Стек (Stack):

Стек - это структура данных, работающая по принципу "последний вошел, первый вышел" (Last In, First Out - LIFO). Это означает, что элементы добавляются и удаляются только с одного конца стека (вершины). Стек может быть реализован как массив или связанный список.

Применение стека:

2. **Управление вызовами функций:** Стек используется для хранения информации о вызовах функций, чтобы знать, куда возвращаться после завершения каждой функции.
3. **Обратная трассировка (debugging):** Стек помогает отслеживать порядок вызовов функций и точки, в которых произошла ошибка.
4. **Вычисления с использованием рекурсии:** Рекурсивные алгоритмы часто используют стек для хранения промежуточных результатов.

5. Постфиксная форма:

Постфиксная форма (или обратная польская запись) - это способ записи арифметических выражений, при котором операторы расположены после своих операндов. Это исключает необходимость в скобках и упрощает вычисление выражений.

Применение постфиксной формы:

1. **Калькуляторы:** Некоторые карманные калькуляторы используют постфиксную форму для упрощения вычислений.
2. **Оптимизация вычислений:** Постфиксная форма позволяет избежать проблем с приоритетом операторов и порядком операций, делая выражения более однозначными.
3. **Автоматическая генерация кода:** Некоторые компиляторы используют постфиксную форму внутри своих промежуточных представлений.

Вместе стек и постфиксная форма могут использоваться, например, для вычисления постфиксных выражений без использования рекурсии. Выражение обрабатывается слева направо, операнды помещаются в стек, и когда встречается оператор, извлекаются нужное количество операндов из стека, выполняется операция, и результат помещается обратно в стек.

Хотя стек и постфиксная форма могут казаться несколько абстрактными, они оказываются полезными инструментами в различных областях программирования и алгоритмов, особенно в тех случаях, когда нужно эффективно управлять порядком операций и сохранять контекст выполнения.

1 Постановка задачи

Цель – Реализовать шаблонный класс для представления стека TStack, и на его основе реализовать класс для представления арифметического выражения с его постфиксной формой TArithmeticExpression.

Задачи при реализации класса TStack:

1. Описать и реализовать конструктор, конструктор копирования, деструктор.
2. Описать и реализовать методы проверки заполнен ли стек и пуст ли стек.
3. Описать и реализовать методы добавления элемента на верхушку стека, удаление элемента с верхушки стека, получить элемент с верхушки стека.

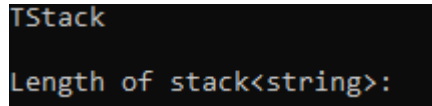
Задачи при реализации класса TArithmeticExpression:

1. Описать и реализовать конструктор.
2. Описать и реализовать метод парсинга арифметического выражения для поиска операторов, операндов и констант.
3. Реализовать алгоритм перевода арифметического выражения в его постфиксную форму.
4. Реализовать алгоритм вычисления значения арифметического выражения по его постфиксной форме.
5. Описать и реализовать разные способы ввода значений арифметического выражения.
6. Описать и реализовать вспомогательные методы: проверка на оператор, проверка на константу, поиск первого встречного оператора от какой-нибудь позиции.
7. Описать и реализовать методы получения инфиксной и постфиксной форм арифметического выражения.
8. Описать и реализовать метод конвертации строки арифметического выражения в форму без сокращений (унарный минус, умножение скобок без символа «*» и т.д.).
9. Описать и реализовать метод на проверку корректности арифметического выражения.

2 Руководство пользователя

2.1 Приложение для демонстрации работы стека

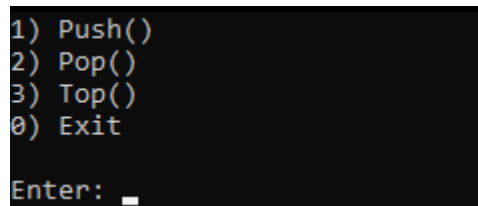
1. Запустите приложение с названием `sample_tstack.exe`. В результате появится окно, показанное ниже (рис. 1).



```
TStack
Length of stack<string>:
```

Рис. 1. Основное окно программы

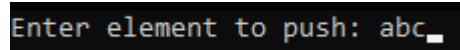
2. Вам будет предложено ввести размер стека строк. Введите значение (например 4), и будет выведено основное меню программы (рис. 2).



```
1) Push()
2) Pop()
3) Top()
0) Exit
Enter: _
```

Рис. 2. Основное меню программы

3. Выберите какую-нибудь предложенную опцию, введя значение, которое указано в меню у этой опции.
4. При выборе пункта `Push()`, вам будет предложено ввести строку, для добавления ее в стек. Введите любую строку, например, `abc` (рис. 3).

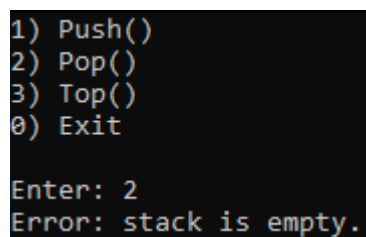


```
Enter element to push: abc_
```

Рис. 3. Пример меню функции `Push()`

5. При выборе пункта `Pop()` из стека будет удален верхний элемент. Далее будет опять выведено основное меню программы (рис. 2), и вы сможете продолжить работу с ней.

Если при выборе этого пункта стек будет пуст, то программа выдаст ошибку, что стек пуст, и программа завершит свою работу (рис. 4).

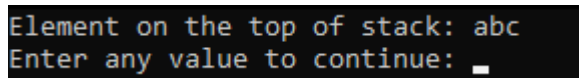


```
1) Push()
2) Pop()
3) Top()
0) Exit
Enter: 2
Error: stack is empty.
```

Рис. 4. Ошибка, что стек пуст

6. При выборе пункта Top() будет выведен верхний элемент стека (рис. 5). Введите любое значение, чтобы вернуться в основное меню программы (рис. 2) и продолжить свою работу с программой.

Если при выборе этого пункта стек будет пуст, то программа выдаст ошибку, что стек пуст, и программа завершит свою работу (рис. 4).



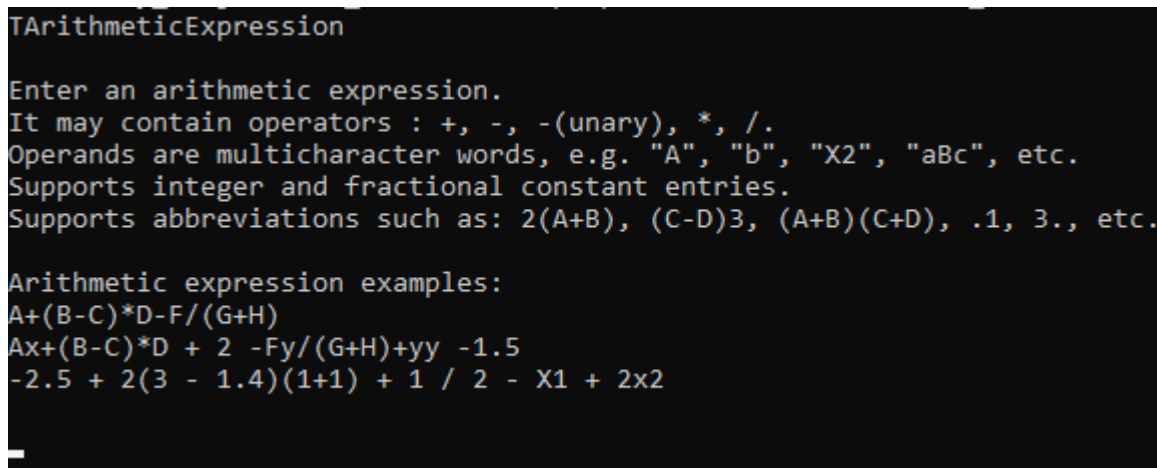
```
Element on the top of stack: abc
Enter any value to continue: _
```

Рис. 5. Пример меню функции Top()

7. При выборе пункта Exit будет завершена работа с программой.

2.2 Приложение для демонстрации работы постфиксной формы арифметического выражения

1. Запустите приложение с названием sample_prefix.exe. В результате появится окно, показанное ниже (рис. 6).



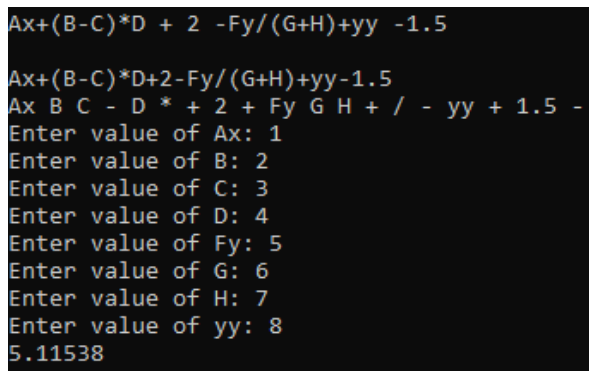
```
TArithmeticExpression

Enter an arithmetic expression.
It may contain operators : +, -, -(unary), *, /.
Operands are multicharacter words, e.g. "A", "b", "X2", "aBc", etc.
Supports integer and fractional constant entries.
Supports abbreviations such as: 2(A+B), (C-D)3, (A+B)(C+D), .1, 3., etc.

Arithmetic expression examples:
A+(B-C)*D-F/(G+H)
Ax+(B-C)*D + 2 -Fy/(G+H)+yy -1.5
-2.5 + 2(3 - 1.4)(1+1) + 1 / 2 - X1 + 2x2
```

Рис. 6. Основное окно программы

2. В главном окне программы пользователю предоставляется возможность ввода арифметического выражения. Также представлены примеры арифметических выражений и дополнительные пояснения относительно содержания арифметического выражения. В случае возникновения ошибки при вводе, программа выведет соответствующее сообщение об ошибке и завершит свою работу. Пожалуйста, введите ваше арифметическое выражение
3. После ввода арифметического выражения программа выведет его инфиксную и постфиксную формы. В случае наличия операндов в арифметическом выражении пользователю предложено последовательно ввести значения каждого операнда. После ввода значений будет вычислено и отображено значение арифметического выражения (рис. 7).



```
Ax+(B-C)*D + 2 -Fy/(G+H)+yy -1.5
Ax+(B-C)*D+2-Fy/(G+H)+yy-1.5
Ax B C - D * + 2 + Fy G H + / - yy + 1.5 -
Enter value of Ax: 1
Enter value of B: 2
Enter value of C: 3
Enter value of D: 4
Enter value of Fy: 5
Enter value of G: 6
Enter value of H: 7
Enter value of yy: 8
5.11538
```

Рис. 7. Окно программы после ввода арифметического выражения

3 Руководство программиста

3.1 Описание алгоритмов

3.1.1 Стек

Стек (или стопка) представляет собой абстрактную структуру данных, организованную по принципу "последний вошел - первый вышел" (Last In, First Out, LIFO). Это означает, что элементы добавляются и удаляются только с одного конца стека, который называется вершиной.

Операции, которые можно выполнять со стеком, включают добавление элемента (push) на вершину стека и удаление элемента (pop) с вершины стека. При этом доступ к остальным элементам стека осуществлять нельзя, кроме верхнего элемента.

3.1.2 Постфиксная форма

Постфиксная форма арифметического выражения, также известная как обратная польская запись (ОПЗ) или постфиксная нотация, представляет собой способ записи математических выражений, при котором операторы расположены после своих операндов. В отличие от традиционной инфиксной формы, где операторы находятся между операндами, в постфиксной форме порядок операндов и операторов определяется последовательностью их появления.

Например, в инфиксной форме выражение записывается с оператором между операндами: $3 + 4 * 5$.

В постфиксной форме это же выражение будет выглядеть следующим образом: $3\ 4\ 5\ *\ +$.

Алгоритм преобразования инфиксной формы арифметического выражения в постфиксную с использованием стека:

1. Создайте пустой стек для операторов и операндов.
2. Инициализируйте пустой список для хранения выходного постфиксного выражения.
3. Пройдите по каждому символу в инфиксной форме слева направо.
4. Если символ - операнд (число), добавьте его в выходной список.
5. Если символ - открывающая скобка, поместите ее в стек.
6. Если символ - оператор, то:

- Пока стек не пуст и верхний элемент стека не является открывающей скобкой, и приоритет оператора на вершине стека больше или равен приоритету текущего оператора, извлеките оператор из стека и добавьте его в выходной список.
 - Поместите текущий оператор в стек.
7. Если символ - закрывающая скобка, извлекайте операторы из стека и добавляйте их в выходной список до тех пор, пока не встретите открывающую скобку. Извлекните открывающую скобку, но не добавляйте ее в выходной список.
 8. После обхода всего выражения извлеките оставшиеся операторы из стека и добавьте их в выходной список.

Алгоритм вычисления значения арифметического выражения в постфиксной форме с использованием стека:

1. Создайте пустой стек.
2. Пройдите по каждому символу в постфиксной форме слева направо.
3. Если символ - операнд (число), поместите его в стек.
4. Если символ - оператор, извлеките из стека необходимое количество операндов (в соответствии с арностью оператора), выполните операцию, и поместите результат обратно в стек.
5. Продолжайте этот процесс до тех пор, пока не обработаете все символы в постфиксной форме.
6. После обработки всех символов в постфиксной форме, результат вычисления будет находиться на вершине стека.

Пример преобразования инфиксной формы в постфиксную:

$$A + (B - C) * D - F / (G - H)$$

																	-
																	/
																+	+
																H	H
																G	G
																F	F
																+	+
																*	*
																D	D
																-	-
																C	C
																B	B
ст1	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
ст2																	
№	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	

Полученная постфиксная форма: $A B C - D * + F G H + / -$

Пример вычисления значения по постфиксной форме:

$A=0, B=1, C=2, D=-1, F=2, G=0.5, H=0.5$

																	0.5
																	2
																	-1
																	0.5
																	0.5
																	1
																	2
Ст.	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	-1	
Преф.	A	B	C	-	D	*	+	F	G	H	+	/	-				

Полученное значение: -1

3.2 Описание программной реализации

3.2.1 Описание класса TStack

```
template <typename T>
class TStack {
private:
    int maxSize;
    int top;
    T* elems;

    void Realloc(int extraSize = 10);
public:
    TStack(int _maxSize = 50);
    TStack(const TStack<T>& s, int extraSize = 0);
    virtual ~TStack();

    bool IsEmpty(void) const noexcept;
    bool IsFull(void) const noexcept;

    T Top();

    void Push(const T& e);
    void Pop();
};
```

Назначение: представление стека.

Поля:

maxSize – размер стека, количество доступной памяти.

top – индекс верхнего элемента в стеке.

elems – память для представления стека.

Конструкторы:

```
TStack(int _maxSize = 50);
```

Назначение: конструктор по умолчанию и с параметром.

Входные параметры: **_maxSize** – количество выделяемой памяти.

```
TStack(const TStack<T>& s, int extraSize = 0);
```

Назначение: выделение памяти и копирование данных из другого объекта стека.

Входные параметры: **s** – экземпляр класса, на основе которого создаем новый объект, **extraSize** – количество дополнительной выделяемой памяти.

```
virtual ~TStack();
```

Назначение: деструктор.

Методы:

void Realloc(int extraSize = 10);

Назначение: выделение дополнительной памяти для объекта стека.

Входные параметры: **extraSize** - количество дополнительной выделяемой памяти.

bool IsEmpty(void) const noexcept;

Назначение: проверка на пустоту стека.

Выходные параметры: **true** или **false**.

bool IsFull(void) const noexcept;

Назначение: проверка на полноту стека.

Выходные параметры: **true** или **false**.

T Top();

Назначение: получение данных элемента с верхушки стека.

Выходные параметры: объект класса **T**.

void Push(const T& e);

Назначение: добавление нового элемента на верхушку стека.

Входные параметры: **e** - объект класса **T**.

void Pop();

Назначение: удаление элемента с верхушки стека.

3.2.2 Описание класса TArithmeticExpression

```
class TArithmeticExpression {
private:
    string infix;
    vector<string> postfix;
    vector<string> lexems;
    static map<string, int> priority;
    map<string, double> operands;

    void Parse();
    void ToPostfix();

public:
    TArithmeticExpression(const string& _infix);

    string GetInfix() const { return infix; }
    vector<string> GetPostfix() const { return postfix; }
    string GetStringPostfix() const;

    vector<string> GetOperands() const;
    void SetValues();
    void SetValues(map<string, double>& values);
    void ClearValues();

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

private:
    bool IsOperator(const string& isopr) const;
    bool IsConst(const string& isopd) const;

    int FindOperator(int pos = 0) const;

    void ConvertInfix();
    void CorrectnessCheck();
};
```

Назначение: представление арифметического выражения.

Поля:

infix – инфиксная форма арифметического выражения.

postfix – постфиксная форма арифметического выражения.

lexems – инфиксная форма арифметического выражения с выделенными каждым лексемами.

priority – список доступных операций с их приоритетами.

operands – список всех операндов арифметического выражения с их значениями.

Конструкторы:

TArithmeticExpression(const string& _infix);

Назначение: создание постфиксной формы арифметического выражения.

Входные параметры: **_infix** – инфиксная форма арифметического выражения.

Методы:

void Parse() ;

Назначение: анализ строки инфиксной формы арифметического выражения.

void ToPostfix() ;

Назначение: преобразование инфиксной формы арифметического выражения в постфиксную форму.

string GetInfix() const;

Назначение: возвращение строки инфиксной формы арифметического выражения.

Выходные параметры: инфиксная форма арифметического выражения.

vector<string> GetPostfix() const;

Назначение: возвращение постфиксной формы арифметического выражения.

Выходные параметры: постфиксная форма арифметического выражения.

string GetStringPostfix() const;

Назначение: возвращение строки постфиксной формы арифметического выражения.

Выходные параметры: постфиксная форма арифметического выражения.

vector<string> GetOperands() const;

Назначение: получение всех операндов арифметического выражения.

Выходные параметры: вектор операндов арифметического выражения.

void SetValues() ;

Назначение: присваивание значений каждому операнду.

void SetValues(map<string, double>& values);

Назначение: присваивание значений каждому операнду.

Входные параметры: **values** - список операндов и их значений.

void ClearValues() ;

Назначение: зануление всех операндов.

double Calculate() ;

Назначение: вычисление значения арифметического выражения на основе внесенных значений операндов.

Выходные параметры: значение арифметического выражения.

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

Назначение: вычисление значения арифметического выражения

Входные параметры: **values** - список операндов и их значений.

Выходные параметры: значение арифметического выражения.

bool IsOperator(const string& isopr) const;

Назначение: проверка строки, является ли она доступным оператором.

Входные параметры: **isopr** – проверяемая строка.

Выходные параметры: **true** или **false**.

bool IsConst(const string& isopd) const;

Назначение: проверка строки, является ли она константой, но не операндом.

Входные параметры: **isopd** – проверяемая строка.

Выходные параметры: **true** или **false**.

int FindOperator(int pos = 0) const;

Назначение: поиск первого встречного оператора.

Входные параметры: **pos** – начальная позиция поиска.

Выходные параметры: индекс первого встречного оператора.

void ConvertInfix() ;

Назначение: корректирование строки инфиксной формы арифметического выражения (избавление от пробелов; добавление отсутствующих операторов умножения на скобки; избавление от сокращенной записи действительных констант).

void CorrectnessCheck() ;

Назначение: проверка инфиксной формы арифметического выражения на наличие критических ошибок в ее записи.

Заключение

В ходе выполнения лабораторной работы был успешно разработан шаблонный класс **TStack** для представления стека, который включает в себя реализацию конструктора, конструктора копирования и деструктора. Были также описаны и реализованы методы проверки заполненности и пустоты стека, а также методы добавления, удаления и получения элемента с верхушки стека.

На основе класса **TStack** был создан класс **TArithmeticExpression**, решающий задачи по обработке арифметических выражений. Был разработан конструктор класса, а также методы парсинга арифметического выражения для поиска операторов, операндов и констант. Реализованы алгоритмы перевода арифметического выражения в его постфиксную форму и вычисления значения по постфиксной форме.

Кроме того, были реализованы различные способы ввода значений арифметического выражения, вспомогательные методы (проверка на оператор, проверка на константу, поиск первого встречного оператора от заданной позиции) и методы получения инфиксной и постфиксной форм арифметического выражения.

Дополнительно, был реализован метод конвертации строки арифметического выражения в форму без сокращений, таких как унарный минус или умножение скобок без символа '*'. Также был создан метод для проверки корректности арифметического выражения.

Литература

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

Приложения

Приложение А. Реализация класса TStack

```
#include <iostream>
using namespace std;

template <typename T>
class TStack {
private:
    int maxSize;
    int top;
    T* elems;

    void Realloc(int extraSize = 10);
public:
    TStack(int _maxSize = 50);
    TStack(const TStack<T>& s, int extraSize = 0);
    virtual ~TStack();

    bool IsEmpty(void) const noexcept;
    bool IsFull(void) const noexcept;

    T Top();

    void Push(const T& e);
    void Pop();
};

template <typename T>
void TStack<T>::Realloc(int extraSize) {
    if (extraSize <= 0) {
        string exp = "Error: extraSize must be bigger than 0.";
        throw exp;
    }
    T* tmp = new T[maxSize + extraSize];
    for (int i = 0; i < maxSize; i++)
        tmp[i] = elems[i];

    delete[] elems;
    maxSize = maxSize + extraSize;
    elems = tmp;
}

template <typename T>
TStack<T>::TStack(int _maxSize) {
    if (_maxSize <= 0) {
        string exp = "Error: maxSize must be bigger than 0.";
        throw exp;
    }
    maxSize = _maxSize;
    top = -1;
    elems = new T[maxSize];
}

template <typename T>
TStack<T>::TStack(const TStack<T>& s, int extraSize) {
    if (extraSize < 0) {
        string exp = "Error: Stack extra size less than 0.";
        throw exp;
    }
    maxSize = s.maxSize + extraSize;
    top = s.top;
```

```

        elems = new T[maxSize];

        for (int i = 0; i <= top; i++)
            elems[i] = s.elems[i];
    }
    template <typename T>
    TStack<T>::~~TStack() {
        delete[] elems;
    }

    template <typename T>
    bool TStack<T>::IsEmpty(void) const noexcept {
        return (top == -1);
    }
    template <typename T>
    bool TStack<T>::IsFull(void) const noexcept {
        return (top == maxSize - 1);
    }

    template <typename T>
    T TStack<T>::Top() {
        if (IsEmpty()) {
            string exp = "Error: stack is empty.";
            throw exp;
        }
        return elems[top];
    }

    template <typename T>
    void TStack<T>::Push(const T& e) {
        //if (IsFull()) throw "Error: stack is full.";
        if (IsFull()) Realloc(maxSize / 2);
        elems[++top] = e;
    }
    template <typename T>
    void TStack<T>::Pop() {
        if (IsEmpty()) {
            string exp = "Error: stack is empty.";
            throw exp;
        }
        top--;
    }
}

```

Приложение Б. Реализация класса TArithmeticExpression

```
#include <iostream>
#include <string>
#include <map>
#include <vector>
#include "tstack.h"

using namespace std;

class TArithmeticExpression {
private:
    string infix;
    vector<string> postfix;
    vector<string> lexems;
    static map<string, int> priority;
    map<string, double> operands;

    void Parse();
    void ToPostfix();

public:
    TArithmeticExpression(const string& _infix);

    string GetInfix() const { return infix; }
    vector<string> GetPostfix() const { return postfix; }
    string GetStringPostfix() const;

    vector<string> GetOperands() const;
    void SetValues();
    void SetValues(map<string, double>& values);
    void ClearValues();

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

private:
    bool IsOperator(const string& isopr) const;
    bool IsConst(const string& isopd) const;

    int FindOperator(int pos = 0) const;

    void ConvertInfix();
    void CorrectnessCheck();
};

map<string, int> TArithmeticExpression::priority = {
    {"*", 3},
    {"/", 3},
    {"+", 2},
    {"-", 2},
    {"(", 1},
    {")", 1}
};

bool TArithmeticExpression::IsOperator(const string& isopr) const {
    bool flag = false;
    for (const auto& opr : priority) {
        if (isopr == opr.first) {
            flag = true;
            break;
        }
    }
}
```

```

    }
    return flag;
}
bool TArithmeticExpression::IsConst(const string& isopd) const {
    bool flag = true;
    for (int i = 0; i < isopd.size(); i++)
        if (isopd[i] < '0' || isopd[i] > '9') {
            if (isopd[i] != '.')
                flag = false;
            break;
        }
    return flag;
}

// If found any operator, returns index. Else returns -1.
int TArithmeticExpression::FindOperator(int pos) const {
    if (pos < 0 || pos >= infix.size()) return -1;

    int ind = -1;
    for (int i = pos; i < infix.size(); i++) {
        string isopr;
        isopr += infix[i];

        if (IsOperator(isopr)) {
            ind = i;
            break;
        }
    }
    return ind;
}

void TArithmeticExpression::ConvertInfix() {
    string nospaces;
    for (int i = 0; i < infix.size(); i++) {
        if (infix[i] != ' ')
            nospaces += infix[i];
    }
    infix = nospaces;

    string tmp;
    if (infix[0] == '-') tmp += "0-";
    else if (infix[0] == '.') tmp += "0.";
    else if (infix[0] == '(') tmp += '(';
    else tmp += infix[0];

    for (int i = 1; i < infix.size() - 1; i++) {
        char elem = infix[i];
        if (elem == ' ') {
            continue;
        }
        else if (elem == '-') {
            if (infix[i - 1] == '(')
                tmp += '0';
            tmp += '-';
        }
        else if (elem == '(') {
            if (infix[i - 1] == ')') || infix[i - 1] == '.' || (infix[i - 1] >= '0' && infix[i - 1] <= '9'))
                tmp += '*';
            tmp += '(';
        }
        else if (elem == '.') {
            if (infix[i - 1] < '0' || infix[i - 1] > '9') {

```

```

        if (infix[i - 1] == ')')
            tmp += '*';
        tmp += '0';
    }
    tmp += '.';
    if (infix[i + 1] < '0' || infix[i + 1] > '9') {
        tmp += '0';
        if (infix[i + 1] == '(')
            tmp += '*';
    }
}
else if (elem >= '0' && elem <= '9') {
    if (infix[i - 1] == ')')
        tmp += '*';
    tmp += elem;
}
else {
    tmp += elem;
}
}
if (infix[infix.size() - 1] == '.') tmp += ".0";
else if (infix[infix.size() - 1] == ')') tmp += ')';
else tmp += infix[infix.size() - 1];
infix = tmp;
}

void TArithmeticExpression::CorrectnessCheck() {
    int op_bracket = 0;
    int cl_bracket = 0;
    int dot = 0;

    string exp = "Incorrect arithmetic expression";

    if (infix[0] == '+' || infix[0] == '*' || infix[0] == '/' || infix[0] ==
')') throw exp;
    if (infix[infix.size() - 1] == '+' || infix[infix.size() - 1] == '-' ||
infix[infix.size() - 1] == '*' || infix[infix.size() - 1] == '/' ||
infix[infix.size() - 1] == '(') throw exp;

    if (infix[0] == '(') op_bracket++;
    else if (infix[0] == '.') dot++;
    if (infix[infix.size() - 1] == ')') cl_bracket++;
    else if (infix[infix.size() - 1] == '.') dot++;

    for (int i = 1; i < infix.size()-1; i++) {
        char elem = infix[i];
        if (elem == '(')
            op_bracket++;
        else if (elem == ')') {
            if (infix[i - 1] == '(' || infix[i - 1] == '+' || infix[i -
1] == '-' || infix[i - 1] == '*' || infix[i - 1] == '/') throw exp;
            cl_bracket++;
        }
        else if (elem == '.')
            dot++;
        else if (elem == '+' || elem == '-' || elem == '*' || elem == '/')
{
            if (dot > 1) throw exp;
            dot = 0;
            if (infix[i - 1] == '+' || infix[i - 1] == '-' || infix[i -
1] == '*' || infix[i - 1] == '/' || infix[i - 1] == '(') throw exp;
        }
    }
}

```

```

        if (op_bracket != cl_bracket) throw exp;
    }

void TArithmeticExpression::Parse() {
    ConvertInfix();
    CorrectnessCheck();

    int firstind, secondind;;
    string token;

    firstind = FindOperator();
    secondind = FindOperator(firstind + 1);
    if (firstind == -1) {
        lexems.push_back(infix);
        return;
    }
    if (firstind > 0) {
        lexems.push_back(infix.substr(0, firstind));
    }
    while (secondind != -1) {
        string opr = infix.substr(firstind, 1);
        string opd = infix.substr(firstind + 1, secondind - firstind - 1);

        lexems.push_back(opr);
        if (opd != "")
            lexems.push_back(opd);
        firstind = secondind;
        secondind = FindOperator(firstind + 1);
    }
    lexems.push_back(infix.substr(firstind, 1));
    if (firstind != infix.size() - 1)
        lexems.push_back(infix.substr(firstind + 1));
}

void TArithmeticExpression::ToPostfix() {
    Parse();
    TStack<string> st;
    string stackToken;
    for (string token : lexems) {
        if (token == "(") {
            st.Push(token);
        }
        else if (token == ")") {
            stackToken = st.Top();
            st.Pop();
            while (stackToken != "(") {
                postfix.push_back(stackToken);
                stackToken = st.Top();
                st.Pop();
            }
        }
        else if (token == "+" || token == "-" || token == "*" || token ==
"/") {
            while (!st.IsEmpty()) {
                stackToken = st.Top();
                st.Pop();
                if (priority[token] <= priority[stackToken])
                    postfix.push_back(stackToken);
                else {
                    st.Push(stackToken);
                    break;
                }
            }
        }
    }
}

```



```

        st.Push(token);
    }
    else {
        double value = 0.0;
        if (IsConst(token)) {
            value = stod(token);
        }
        operands.insert({ token, value });
        postfix.push_back(token);
    }
} // for
while (!st.IsEmpty()) {
    stackToken = st.Top();
    st.Pop();
    postfix.push_back(stackToken);
}
}

TArithmeticExpression::TArithmeticExpression(const string& _infix) {
    infix = _infix;
    ToPostfix();
}

string TArithmeticExpression::GetStringPostfix() const {
    string pf;
    for (const string& item : postfix)
        pf += item + " ";
    if (!pf.empty())
        pf.pop_back();
    return pf;
}

vector<string> TArithmeticExpression::GetOperands() const {
    vector<string> op;
    for (const auto& item : operands)
        if (!IsConst(item.first))
            op.push_back(item.first);
    return op;
}

void TArithmeticExpression::SetValues() {
    double val;
    for (auto& op : operands) {
        if (!IsConst(op.first)) {
            cout << "Enter value of " << op.first << ": ";
            cin >> val;
            operands[op.first] = val;
        }
    }
}

void TArithmeticExpression::SetValues(map<string, double>& values) {
    for (auto& val : values) {
        try {
            operands.at(val.first) = val.second;
        }
        catch (out_of_range& e) {}
    }
}

void TArithmeticExpression::ClearValues() {
    for (auto& op : operands) {
        if (!IsConst(op.first))
            operands.at(op.first) = 0.0;
    }
}

```

```

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 leftOp, rightOp;
    for (string lexem : postfix) {
        if (lexem == "+") {
            rightOp = st.Top();
            st.Pop();
            leftOp = st.Top();
            st.Pop();
            st.Push(leftOp + rightOp);
        }
        else if (lexem == "-") {
            rightOp = st.Top();
            st.Pop();
            leftOp = st.Top();
            st.Pop();
            st.Push(leftOp - rightOp);
        }
        else if (lexem == "*") {
            rightOp = st.Top();
            st.Pop();
            leftOp = st.Top();
            st.Pop();
            st.Push(leftOp * rightOp);
        }
        else if (lexem == "/") {
            rightOp = st.Top();
            st.Pop();
            leftOp = st.Top();
            st.Pop();
            if (rightOp == 0) {
                string exp = "Error: division by 0";
                throw exp;
            }
            st.Push(leftOp / rightOp);
        }
        else {
            st.Push(operands[lexem]);
        }
    }
    return st.Top();
}

double TArithmeticExpression::Calculate() {
    return Calculate(operands);
}

```