МИНИСТЕРСТВО НАУКИ И ВЫСШЕГООБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
**«Национальный исследовательский
Нижегородский государственный университет им. Н.И. Лобачевского»
(ННГУ)**

Институт информационных технологий, математики и механики

ЛАБОРАТОРНАЯ РАБОТА

на тему:

«Постфиксная форма записи арифметических выражений»

Выполнил: студент группы 3822Б1ФИ2
_____/Коробейников А.П./
Подпись

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

Нижний Новгород
2023

Содержание

Введение

Задача представления арифметических выражений в компьютер встречается повсеместно и на любом уровне программирования. Но делают это не привычным нам способом - инфиксной записью, а другими. В данной работе мы будем рассматривать один из способов представлять арифметические выражение в компьютере. А именно - постфиксную (обратную польскую) запись арифметических выражений. Стандартная библиотека языка C++ не предоставляет такую возможность, как постфиксная запись арифметических выражений, поэтому напишем такую библиотеку сами.

1 Постановка задачи

Цель:

Разработать библиотеку, которая содержит в себе структуру хранения для стека и арифметических операций, которые предоставляются в компьютер постфиксной записью и вычисляются с помощью стека.

Задачи:

1. Изучить необходимую теорию по работе со стеком.
2. Написать класс стек.
3. Написать тесты для класса стек, чтобы убедится в корректности работы.
4. Написать программу, демонстрирующую работу класса стек.
5. Изучить алгоритм преобразования инфиксной записи арифметического выражения в постфиксную.
6. Написать класс, для работы с арифметическими выражениями.
7. Реализовать в этом классе функции преобразования в постфиксную запись и вычисления, используя стек.
8. Написать тесты для этого класса, чтобы убедится в корректности его работы.
9. Написать приложение, демонстрирующее работу наших алгоритмов.

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).

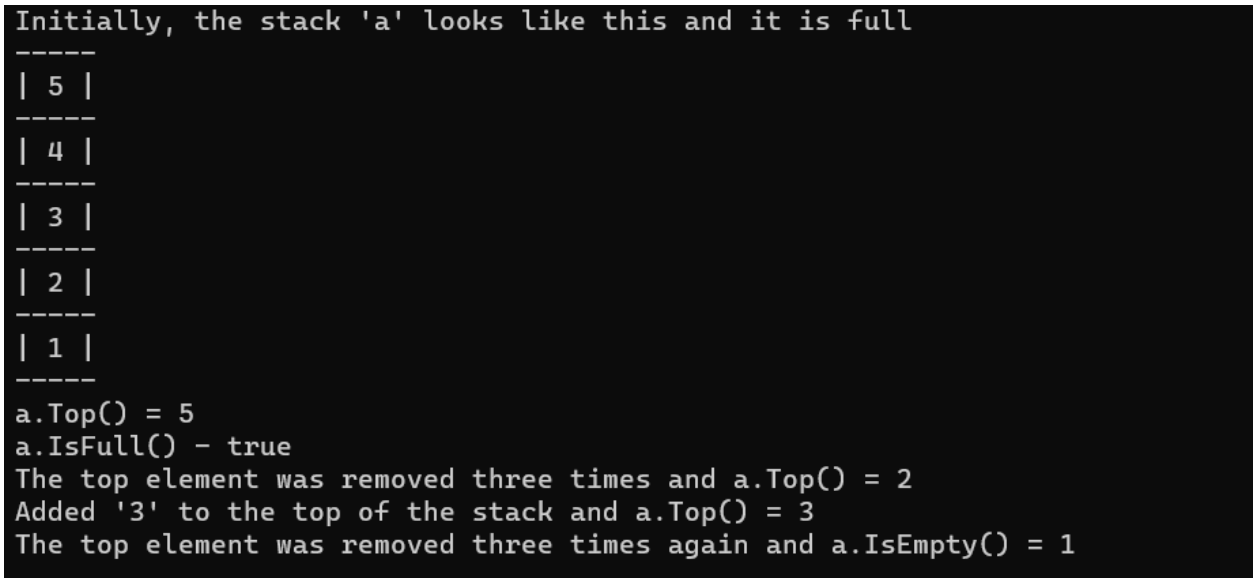

Рис. 1. Основное окно программы sample_tstack.exe

2.2 Приложение для демонстрации работы перевода арифметического выражения в постфиксную запись

1.Запустите приложение с названием sample_tarithmeticexpression.exe. В результате появится окно, показанное ниже, вам будет предложено ввести арифметическое выражение. После ввода арифметического выражения вам выведется его запись в инфиксной форме и постфиксной форме, а также попросят ввести операнды, использующиеся в вашем выражении.

(рис. 2).

Рис. 2. Основное окно программы sample_tarithmeticexpression.exe

2.После ввода всех операндов выражения вам выведется его результат (рис. 3).

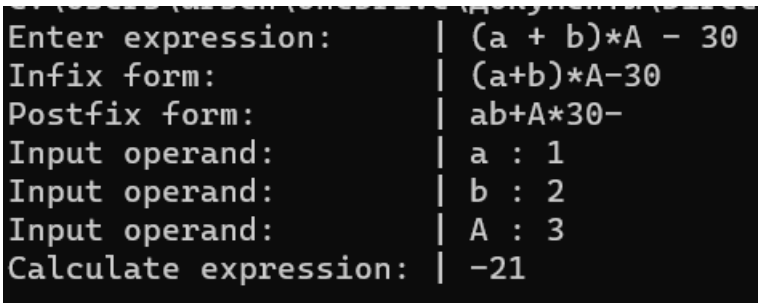

Рис. 3. Результат вычислений программы sample_tarithmeticexpression.exe

3 Руководство программиста

3.1 Описание алгоритмов

3.1.1 Стек

Стек (от англ. stack — стопка) — структура данных, представляющая из себя упорядоченный набор элементов, в которой добавление новых элементов и удаление существующих производится с одного конца, называемого вершиной стека. Притом первым из стека удаляется элемент, который был помещен туда последним, то есть в стеке реализуется стратегия «последним вошел — первым вышел» (last-in, first-out — LIFO). Примером стека в реальной жизни может являться стопка тарелок: когда мы хотим вытащить тарелку, мы должны снять все тарелки выше.

Операция добавления элемента в вершину стека

Добавляем элемент на «верхушку стека», сдвигая все остальные «глубже в стек».

Пример:

33	18	26			

Операция добавления элемента ‘45’ в вершину:

33	18	26	45		

Операция удаления элемента из вершины стека

Мы забираем элемент с верхушки стека. Всё остальное остаётся также.

Пример:

33	27	126	492		

Удаление элемента с вершины стека:

33	27	126			

Операция взятия элемента с вершины.

Мы узнаём значение элемента на вершине стека, сам стек при этом не меняется.

Пример:

67	34	815	7		

Операция взятия элемента с вершины стека:

Результат: 7

Операция проверки на полноту.

Операция проверки на полноту проверяет, полон ли стек.

Пример 1:

67	34	815	7		

Операция проверки на полноту:

Результат: false

Пример 2:

67	34	815	7	52	43

Операция проверки на полноту:

Результат: true

Операция проверки на пустоту.

Операция проверки на полноту проверяет, есть ли хотя бы один элемент в стеке. Пример 1:

67	34	815	7		

Операция проверки на полноту:

Результат: false

Пример 2:

Операция проверки на полноту:

Результат: true

3.1.2 Арифметическое выражение

Арифметическое выражение – это последовательность чисел, констант, переменных, и других арифметических выражений, заключённых в круглые скобки, которые соединены между собой знаками арифметических операций. Переменные, входящие в состав арифметического выражения должны иметь числовой тип. Мы привыкли видеть арифметические выражения в инфиксной форме, например:

A + B*2 - C

Это инфиксная форма записи арифметического выражения.

Однако компьютеру так видеть не удобно, и придумали постфиксную (обратную польскую) запись арифметических выражений. Её суть в том, что операция следует только после своих операндов. И как раз для перевода в постфиксную запись используется стек, а также он используется при вычислении этого выражения из постфиксной формы.

Алгоритм получения постфиксной записи выражения:

Входные данные:

Арифметическое выражение в инфиксной форме

Выходные данные:

Арифметическое выражение в постфиксной форме

Алгоритм:

1. Создаём пустой стек операторов.

2. Создаём 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: (a + b * c)*(c/d - e)

Символ	Постфиксная форма	Стек операторов
(		(
a	a	(
+	a	(+
b	ab	(+
*	ab	(+*
c	abc	(+*
)	abc*+	
*	abc*+	*
(	abc*+	*(
c	abc*+c	*(

/	abc*+c	*(/
d	abc*+cd	*(/
-	abc*+cd/	*(-
e	abc*+cd/e	*(-
)	abc*+cd/e-*	

Вычисление результата.

Входные данные:

Постфиксная запись выражения.

Выходные данные:

Результат арифметического выражения.

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

1. Создаём 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. Причём если порядок важен, то первым будет считаться элемент, который мы достали вторым.

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 a + b - c

Постфиксная запись: ab+c-

Значение операндов: A = 33, B = 18, C = 21

Символ	Стек операндов	Вычисления
a	a	
b	ab	
+	51	a+b = 51
c	51c	
-	30	51 - c = 51 - 21 = 30

3.2 Описание программной реализации

3.2.1 Описание класса TStack

```
template <typename T>
class TStack
{
private:
      int maxsize;
      int top;
      T* elems;
public:
      TStack(int maxsize = 10);
      TStack(const TStack& s);
      ~TStack();
      bool  IsEmpty(void) const;
      bool IsFull(void) const;
      T Top() const;
      void Push(const T& e);
      void Pop();
};
```

Назначение: представление стека .

Поля:

maxSize – максимальный размер стека.

***elems** – память для представления элементов стека.

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

Методы:

TStack(int maxSize = 10);

Назначение: конструктор по умолчанию и конструктор с параметрами.

Входные параметры:

maxSize – максимальный размер стека (по умолчанию 10).

Выходные параметры: отсутствуют.

TStack(const TStack<T>& s);

Назначение: конструктор копирования.

Входные параметры:

s – стек, **на основе которого создаем новый стек.**

Выходные параметры: отсутствуют.

~TStack();

Назначение: деструктор.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

T Top() const;

Назначение: получение элемента, находящийся в вершине стека.

Входные параметры отсутствуют.

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

bool IsEmpty() const;

Назначение: проверка на пустоту.

Входные параметры: отсутствуют.

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

bool IsFull() const;

Назначение: проверка на полноту.

Входные параметры: отсутствуют.

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

void push(const T& elem);

Назначение: добавление элемента в стек.

Входные параметры:

elem – элемент, который добавляем.

Выходные параметры отсутствуют.

void pop();

Назначение: удаление элемента из вершины стека.

Входные параметры отсутствуют.

3.2.2 Описание класса TArithmeticExpression

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

      //вспомогательные функции
      bool IsOperator(const string& op) const;
      bool IsConst(const string& op) const;
      bool IsBinaryOperator(const char& op) const;
      int FindFirstOperator(int pos) const;

      void SetOperand(const string& operand);
```

```
	void GetFinishedLine();
	void CheckCorrect() const;


	void Parse();
	void ToPostfix();
public:
	TArithmeticExpression(string infx, const map<string, double> operands_ =
map<string, double>());

	string GetInfix() const {
		return infix;
	}
	string GetPostfix()  const {
		string postfix_str = postfix[0];
		for  (int  i  =  1;  i  <  postfix.size();  i++)  postfix_str    +=
postfix[i];
		return postfix_str;
	}

	double  Calculate();
};
```

Назначение: работа с инфиксной формой записи арифметических выражений

Поля:

infix – выражение в инфиксной записи.

postfix – выражение в постфиксной записи.

lexems – набор лексем инфиксной записи

priority – приоритет арифметических операндов

operands – операнды и их значения

Методы:

TArithmeticExpression(const string& expression, const map<string, double> operands_);

Назначение: конструктор с параметрами.

Входные параметры:

infx – выражение в инфиксной форме.

operands_ – значение операндов

Выходные параметры: отсутствуют.

string GetInfix();

Назначение: получение инфиксной формы.

Входные параметры отсутствуют.

Выходные параметры: инфиксная форма записи.

string GetPostfix();

Назначение: получение постфиксной формы.

Входные параметры отсутствуют.

Выходные параметры: постфиксная форма записи.

`double Calculate();`

Назначение: вычисление выражения.

Входные параметры отсутствуют.

Выходные параметры: результат вычислений

`void CheckCorrect();`

Назначение: проверка на корректность введенных данных.

Входные параметры отсутствуют.

Выходные параметры: отсутствуют

`void GetFinishedLine();`

Назначение: подготовка инфиксной формы записи.

Входные параметры отсутствуют.

Выходные параметры: отсутствуют

`void ToPostfix();`

Назначение: конвертирование инфиксной формы в постфиксную.

Входные параметры отсутствуют.

Выходные параметры: отсутствуют

`void Parse();`

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

Входные параметры отсутствуют.

Выходные параметры: отсутствуют

`void SetOperand (const string& operand);`

Назначение: ввод значения операнда

Входные параметры: операнд, значение которого нам нужно.

Выходные параметры: отсутствуют

`int FindFirstOperator(int pos = 0);`

Назначение: поиск первого оператора , начиная с позиции **`pos.`**

Входные параметры:

`pos –` позиция, начиная с которой ищется оператор.

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

bool IsOperator(const string &op) const;

Назначение: проверяет строку оператор ли это.

Входные параметры:

op – оператор или операнд.

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

bool IsConst(const string &op) const;

Назначение: проверяет строку константа ли это.

Входные параметры:

op – оператор или операнд.

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

bool IsBinaryOperator(const char* &op) const;

Назначение: проверяет строку бинарный оператор ли это.

Входные параметры:

op – оператор или операнд.

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

Заключение

В ходе выполнения лабораторной работы была изучена структура хранения Стек и все её особенности. Кроме того, мы изучили способы представления арифметического выражения в компьютере, что является очень важным знанием для программиста. Мы изучили алгоритм преобразования выражения в постфиксную форму, а также алгоритм вычисления результата выражения по его постфиксной форме. Знание этих алгоритмов и структуры хранения стек, очень важно для программиста, потому что это требуется и в образовательной, и в профессиональной деятельности.

Литература

1. Польская запись [https://ru.wikipedia.org/wiki/Польская_запись].
2. Лекции на тему арифметических выражений [https://cloud.unn.ru/s/4Pyf24EBmowGsQ2]
3. Стек [https://neerc.ifmo.ru/wiki/index.php?title=Стек#:~:text=Стек%20(от%20англ.%20stack%20—,вышел»%20(last-in%2C%20first-out%20—%20LIFO)]

Приложения

Приложение А. Реализация класса TStack

```
template <typename T>
TStack<T>::TStack(int maxsize = 10) {
      if (maxsize <= 0) {
            throw "maxsize < 0";
      }
      top = -1;
      this->maxsize = maxsize;
      elems = new T[maxsize];
}

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

template <typename T>
TStack<T>:: ~TStack() {
      delete[] elems;
      maxsize = 0;
      top = -1;
}

template <typename T>
bool TStack<T>::IsEmpty(void) const {
      if (top == -1) {
            return true;
      }
      return false;
}


template <typename T>
bool TStack<T>::IsFull(void) const {
      if (top == maxsize - 1) {
            return true;
      }
      return false;
}

template <typename T>
T TStack<T>::Top() const {
      if (top != -1) {
            return elems[top];
      }
      else throw "stack is empty";
}

template <typename T>
void TStack<T>::Push(const T& e) {
      if (IsFull()) {
            T* tmp_elem = new T[maxsize + 10];
            for (int i = 0; i < maxsize; ++i) {
```

```
                tmp_elem[i] = elems[i];
            }
            delete[] elems;
            elems = tmp_elem;
            maxsize = maxsize + 10;
      }
      elems[++top] = e;
}

template <typename T>
void TStack<T>::Pop() {
      if (IsEmpty()) {
            throw "Stack is empty";
      }
      top -= 1;
}


#endif
```

Приложение Б. Реализация класса Expression

```
#include "tarithmetic_expression.h"

bool TArithmeticExpression:: IsOperator(const string& op) const
{
            if ((op == "(") || (op == ")") || (op == "+") || (op == "-") ||
(op == "*") || (op == "/"))
            {
                  return true;
            }
      return false;
}
bool TArithmeticExpression::IsConst(const string& op) const
{
      for (int i = 0; i < op.size(); i++)

            if (op[i] < '0' || op[i] > '9')
            {
                  if (op[i] != '.')
                  {
                        return false;
                  }
                  return true;
            }
      return true;
}
bool TArithmeticExpression::IsBinaryOperator(const char& op) const
{
      return (op == '*' || op == '/' || op == '+' || op == '-');
}

int TArithmeticExpression::FindFirstOperator(int pos) const
{
      if (pos < 0 || pos >= infix.size()) return -1;

      for (int i = pos; i < infix.size(); i++)
      {
            string op;
            op += infix[i];

            if (IsOperator(op))
            {
```

```
                return i;
            }
        }

        return -1;
}

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

        if (expression_without_spaces[0] == '-')
        {
            finished_expression += "0-";
        }
        else finished_expression += expression_without_spaces[0];

        for (int i = 1; i < expression_without_spaces.size(); i++)
        {
            char t = expression_without_spaces[i];

            switch (t)
            {
            case '-':
                if (expression_without_spaces[i - 1] == '(')
                {
                    finished_expression += '0';
                }

                finished_expression += '-';
                break;
            case'.':
                if    (expression_without_spaces[i    -    1]    <    '0'    ||
expression_without_spaces[i - 1] > '9' ||
                    expression_without_spaces[i    +    1]    <    '0'    ||
expression_without_spaces[i + 1] > '9')
                {
                    throw "Incorrect expression (extra point)";
                }
                finished_expression += '.';
                break;
            case '(':
                if (expression_without_spaces[i - 1] == ')' ||
                   (expression_without_spaces[i - 1] >= '0' &&
                    expression_without_spaces[i - 1] <= '9'))
                {
                    finished_expression += '*';
                }

                finished_expression += '(';
                break;
            default:
                finished_expression += t;
                break;
            } //switch
        } //for
        infix = finished_expression;
}
```

```
void TArithmeticExpression::CheckCorrect() const {
	int open_count = 0;
	int closed_count = 0;
	int point_counter = 0;
	int len = infix.size() - 1;

	if (infix[0] == '*' || infix[0] == '/' || infix[0] == '+' || infix[0] == 
')' || infix[0] == '.')
		throw "Incorrest expression (first simbol isn't correct)";

	 if (infix[0] == '(')
		open_count++;

	if (IsBinaryOperator(infix[len]) || infix[len] == '('  || infix[len] == 
'.')
		throw "Incorrest expression (last simbol isn't correct)";

	if (infix[len] == ')')
		closed_count++;

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

		char t = infix[i];

		switch (t)
		{
		case '(':
			open_count++;
			break;
		case ')':
			if (IsBinaryOperator(infix[i - 1]) || infix[i - 1] == '(')
				throw "Incorrest expression (simbol beside brackets 
incorrect)";
			closed_count++;
			break;
		case '.':
			point_counter += 1;
			break;
		default:
			break;
		} // switch

		if (IsBinaryOperator(t))
		{
			if (point_counter > 1 || IsBinaryOperator(infix[i - 1]) || 
infix[i - 1] == '(')
			{
				throw "Incorrect expression (two point in one number 
OR two operators OR operator after bracket)";
			}
			point_counter = 0;
		}
		else
		{
			if (infix[i - 1] == ')')
			{
				throw "Incorrect expession";
			}
		}
	}
```

```
	if (open_count != closed_count || point_counter > 1)
	{
		throw "Incorrect expression (different number of brackets)";
	}
}


void TArithmeticExpression::Parse()
{
	GetFinishedLine();
	CheckCorrect();

	int id1 = FindFirstOperator(0), id2 = FindFirstOperator(id1 + 1);
	string substring;

	if (id1 == -1) {
		lexems.push_back(infix);
		return;
	}
	else
	{
		for (int i = 0; i < id1; i++)
		{
			substring += infix[i];
		}
		if (substring.size())
		{
			lexems.push_back(substring);
		}
	}

	while (id2 + 1)
	{
		string substring1;
		substring = infix[id1];
		lexems.push_back(substring);

		for (int i = id1 + 1; i < id2; i++)
		{
			substring1 += infix[i];
		}

		if (!substring1.empty())
			lexems.push_back(substring1);

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

	}

	substring = infix[id1];
	lexems.push_back(substring);
	substring.clear();


	if (id1 != infix.size() - 1)
	{
		substring = infix[id1 + 1];
		for (int i = id1 + 2; i < infix.size(); i++)
		{
			substring += infix[i];
		}
		lexems.push_back(substring);
```

```
	}

}


void TArithmeticExpression::ToPostfix() {
	Parse();

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

	for (string lexem : lexems)
	{
		if (lexem == "(")
		{
			stack.Push(lexem);
		}
		else if (lexem == ")")
		{
			op = stack.Top();
			stack.Pop();

			while (op != "(")
			{
				postfix.push_back(op);
				op = stack.Top();
				stack.Pop();
			}
		}
		else if ((lexem.size() == 1) && IsBinaryOperator(lexem[0]))
		{
			while (!stack.IsEmpty())
			{

				op = stack.Top();
				stack.Pop();
				if (priority[op] >= priority[lexem])
				{
					postfix.push_back(op);
				}
				else
				{
					stack.Push(op);
					break;
				}
			}

			stack.Push(lexem);

		}
		else
		{
			double value = 0.0;
			if (IsConst(lexem))
			{
				value = stod(lexem);
				operands[lexem] = value;
			}
			postfix.push_back(lexem);
		}
	}
	while (!stack.IsEmpty()) {
		op = stack.Top();
```

```
            stack.Pop();
            postfix.push_back(op);
      }
}



void TArithmeticExpression::SetOperand(const string& operand)
{
      string op;
      cout << "Input operand:        | " + operand + " : ";
      cin >> op;
      try
      {
            double number = stoi(op);
            operands[operand] = number;
      }
      catch (const exception& e)
      {
            throw "It's not a number: " + op + "\n";
      }
}


double TArithmeticExpression::Calculate() {
      TStack<double> st(infix.size() + 10);
      double op1, op2;

      for (string lexem : postfix) {

            //swutch для лексем нет, но если очень хочется switch, то можно
брать бубен и танцевать с map
            if (lexem == "+")
            {

                  op2 = st.Top();
                  st.Pop();

                  op1 = st.Top();
                  st.Pop();

                  st.Push(op1 + op2);
            }

            else if (lexem == "-")
            {

                  op2 = st.Top();
                  st.Pop();

                  op1 = st.Top();
                  st.Pop();

                  st.Push(op1 - op2);
            }

            else if (lexem == "/")
            {

                  op2 = st.Top();
                  st.Pop();

                  op1 = st.Top();
```

```
                st.Pop();

                if (op2 == 0) {
                      throw string("Division by 0 (prohibited)");
                }
                else st.Push(op1 / op2);
          }

          else if (lexem == "*")
          {

                op2 = st.Top();
                st.Pop();

                op1 = st.Top();
                st.Pop();

                st.Push(op1 * op2);

          }

          else
          {
                if (operands.find(lexem) == operands.end())
                {
                      SetOperand(lexem);
                }
                st.Push(operands[lexem]);
          }
    }
    return st.Top();
}



TArithmeticExpression::TArithmeticExpression(string   infx,const   map<string,
double> operands_): infix(infx)
{
    if (operands_ != map<string, double>() )
    {
          for (pair<string, double> elem : operands_)
          {
                operands[elem.first] = elem.second;
          }
    }

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

    ToPostfix();
}
```