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

Институт информационных технологий, математики и механики

ЛАБОРАТОРНАЯ РАБОТА

на тему:

«ВЫЧИСЛЕНИЕ АРИФМЕТИЧЕСКИХ ВЫРАЖЕНИЙ (СТЕКИ)»

Выполнил: студент группы
3822Б1ФИ1
_____ / Созонов И.С. /
Подпись

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

Нижний Новгород
2023

Содержание

Введение

Как известно, структура данных – алгебраическая система $<A,O,R>$

- непустое множество (операндов) A;
- с заданным на нём набором операций O: $A^{n_i} \rightarrow A$, $i \in I$ (n_i – арность операций);
- и отношенийR: $r_j \subseteq A^{m_j}$, $j \in J$ (m_j – арность отношений).

Структуры данных не только строятся на множестве операндов и наборе операций, но и сами являются операндами. Результаты операций над структурами данных также являются структурами данных, как того же вида, так и иного. Во всех таких операциях структуры данных остаются статическими (статичными) – изначальное множество (операндов) A не меняется. Однако существуют такие структуры данных, обладающие состоянием, которое может изменяться операциями, например, вставкой/добавлением элемента в структуру данных, удалением/исключением элемента из структуры данных. Такие структуры данных называются динамическими.

Динамическая структура данных – это структура данных, обладающая состоянием и переменным размером, которые могут меняться с течением времени.

Свойства динамических структур данных:

- Число элементов (размер множества операндов A) может меняться;
- Может иметь пустое состояние – без элементов;
- Эффективная реализация требует стратегии управления памятью.

Одной из типовых динамических структур данных является стек.

Стек – динамическая структура данных, построенная по принципу «последним вошел – первым вышел» (last in –first out, LIFO). Принцип работы стека часто сравнивают со стопкой тарелок –взять и поставить тарелку можно только сверху стопки.

Стек широко используется при работе синтаксических и иных парсеров; для алгоритмов, построенных по принципу перебора с возвратом; для обхода различных структур данных. Сегмент стека используется в WAP (Wireless Application Protocol) процессе. Некоторые языки программирования используют стековую модель вычислений.

Стек также используется при вычислении арифметических выражений.

Арифметическое выражение – выражение, составленное из операндов, соединенных арифметическими операциями (+, -, *, /).

Если в список операций добавить возведение в степень и извлечение корня (с целыми показателями), арифметическое выражение станет **алгебраическим**.

Расширив список операций обозначениями произвольных действий и функций, получим **аналитическое выражение** или **формулу**.

Условимся далее все такие действия называть операциями, а все такие выражения – арифметическими.

Традиционная запись арифметического выражения, например, $(a + b * c) * (c / d - e)$ подразумевает, что

- операнды отделяются друг от друга операциями;
- порядок действий определяется расстановкой скобок и приоритетом операций.

Такой способ записи называется **инфиксной формой** арифметического выражения. Однако данный способ записи не очень удобна для вычисления значения арифметического выражения, т.к. необходимо учитывать приоритет операций.

Поэтому В 1920 годах польский логик Ян Лукасевич разработал форму записи арифметических и логических выражений, в которой операция располагается в выражении слева от ее операндов. Так арифметическому выражению $(a + b * c) * (c / d - e)$ будет соответствовать запись: $* + a * b c - / c d e$.

Такой способ записи называется **префиксной (польской или прямой польской) формой** арифметического выражения.

Далее В 1950 годах уже австралийский ученый Чарльз Хэмблин на основе польской нотации разработал форму записи арифметических и логических выражений, в которой операнды располагаются в выражении перед операциями. Так арифметическому выражению $(a + b * c) * (c / d - e)$ будет соответствовать запись $a b c * + c d / e - *$.

Такой способ записи называется **постфиксной (обратной польской) формой** арифметического выражения.

Постфиксная форма стала очень популярной. Она используется во многих областях, математики, техники и программирования:

- Стек-ориентированные языки программирования (Forth, Factor, PostScript, BibTeX);
- Настольные калькуляторы (Hewlett-Packard9100A, HP-35, советские инженерные и программируемые калькуляторы, такие как Б3-19М, российские программируемые калькуляторы «Электроника МК-152» и «Электроника МК-161»);
- Программные калькуляторы (Mac OS X Calculator, Unix system calculator «dc»).

1 Постановка задачи

Цель – реализовать классы для представления стеков TStack и обработки арифметических выражений TArithmeticExpression.

Задачи:

1. Разработать класс TStack для работы со стеками. Написать следующие операции для работы со стеками: добавление элемента в стек, получение значения элемента из стека, изъятие элемент из стека и проверка на пустоту.
2. Разработать класс TArithmeticExpression для обработки арифметических выражений. Написать следующие операции для обработки арифметических выражений: проверка корректности записи выражения, перевод в постфиксную форму и вычисление результата.

2 Руководство пользователя

2.1 Приложение для демонстрации работы стеков

1. Запустить sample_tstack.exe. В результате появится окно для ввода количество элементов, которое необходимо поместить в стек (рис. 1).

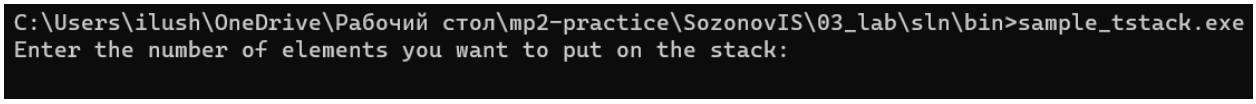


Рис. 1. Основное окно приложения

2. Ввести количество элементов. В результате появится окно для ввода элементов, которые необходимо поместить в стек (Рис. 2).

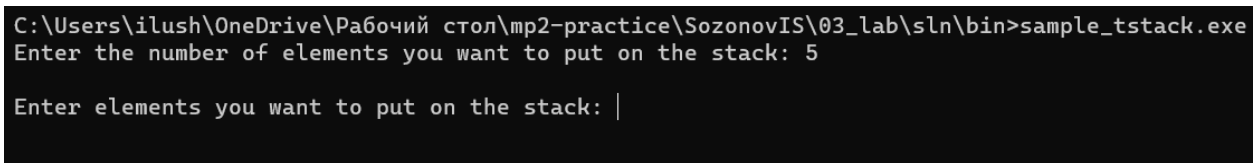


Рис. 2. Ввод количества элементов

3. Ввести элементы. В результате будет выполнена проверка стека на пустоту и появится окно для ввода количество элементов, которое необходимо изъять из стека (Рис. 3).

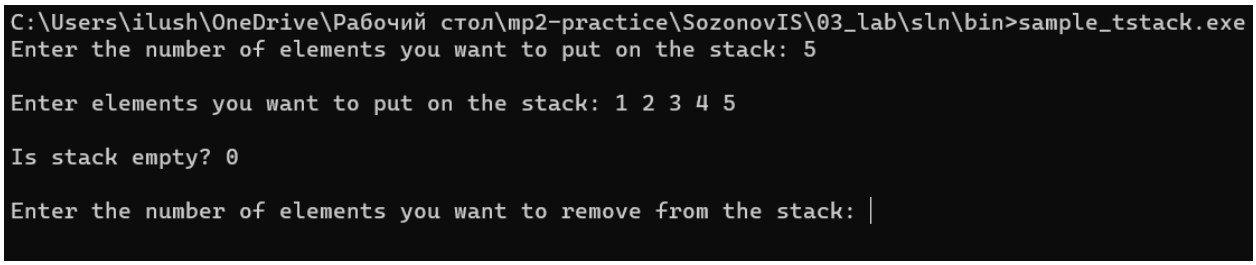


Рис. 3. Ввод элементов

4. Ввести количество элементов. В результате будут изъяты элементы, находящиеся в стеке и выведены их значения элементов. Затем будет повторно выполнена проверка на пустоту стека (Рис. 4).

```
C:\Users\ilush\OneDrive\Рабочий стол\mp2-practice\SozonovIS\03_lab\sln\bin>sample_tstack.exe
Enter the number of elements you want to put on the stack: 5

Enter elements you want to put on the stack: 1 2 3 4 5

Is stack empty? 0

Enter the number of elements you want to remove from the stack: 5

Stack item: 5
Stack item: 4
Stack item: 3
Stack item: 2
Stack item: 1

Is stack empty? 1

C:\Users\ilush\OneDrive\Рабочий стол\mp2-practice\SozonovIS\03_lab\sln\bin>
```

Рис. 4. Ввод количества элементов

2.2 Приложение для демонстрации работы арифметических выражений

1. Запустить sample_tarithmeticexpression.exe. В результате появится окно для ввода инфиксной формы арифметического выражения (Рис. 5).

```
C:\Users\ilush\OneDrive\Рабочий стол\mp2-practice\SozonovIS\03_lab\sln\bin>sample_tarithmeticexpression.exe
Enter expression:
```

Рис. 5. Основное окно приложения

2. Ввести инфиксную форму арифметического выражения. В результате будет выведена постфиксная форма введенного арифметического выражения. Появится окно для ввода значения переменной a (Рис. 6).

```
C:\Users\ilush\OneDrive\Рабочий стол\mp2-practice\SozonovIS\03_lab\sln\bin>sample_tarithmeticexpression.exe
Enter expression: -3*num1+a*(3.14-2/b)
Postfix: 0 3 num1 * - a 3.14 2 b / - * +

Enter value of a:
```

Рис. 6. Ввод инфиксной формы арифметического выражения

3. Ввести значение переменной a. В результате появится окно для ввода значения переменной b (Рис. 7).

```
C:\Users\ilush\OneDrive\Рабочий стол\mp2-practice\SozonovIS\03_lab\sln\bin>sample_tarithmeticexpression.exe
Enter expression: -3*num1+a*(3.14-2/b)
Postfix: 0 3 num1 * - a 3.14 2 b / - * +

Enter value of a: 1.01
Enter value of b:
```

Рис. 7. Ввод значения переменной a

4. Ввести значение переменной b. В результате появится окно для ввода значения переменной num1 (рРис. 8).

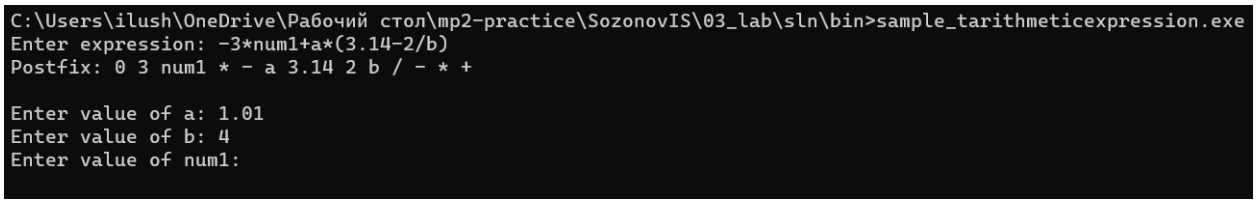


Рис. 8. Ввод значения переменной b

5. Ввести значение переменной num1. В результате будет выведен результат вычисления арифметического выражения (Рис. 9).

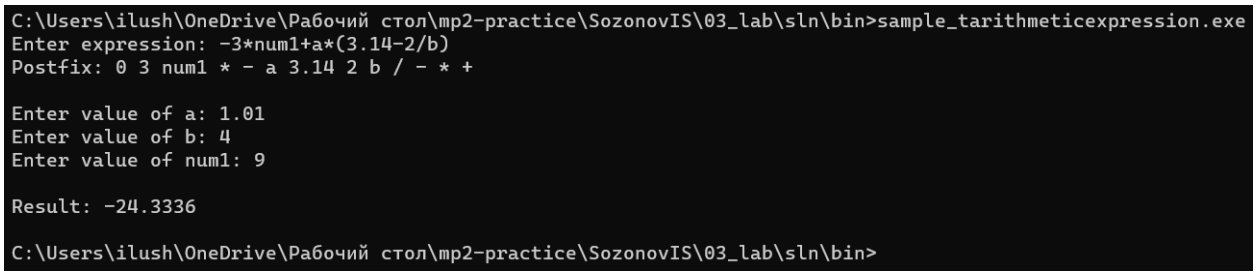


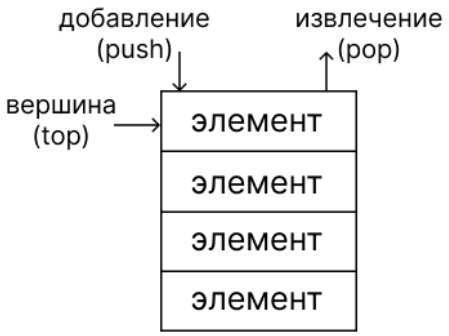
Рис. 9. Ввод значения переменной num1

3 Руководство программиста

3.1 Использованные алгоритмы

3.1.1 Стеки

Стек – динамическая структура данных, построенная по принципу «последним вошел –первым вышел» (last in –first out, LIFO). Элементы стека можно разместить в памяти, используя статический массив, динамический массив фиксированного размера, динамический массив с возможностью перевыделения памяти или контейнер std::vector.



Для работы со стеком предлагается реализовать следующие операции:

- Метод **Push** – добавить элемент в стек;

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

- Метод **Top** – получить значение элемента из стека;

При получении значения элемента из стека необходимо возвратить значение из динамического массива по индексу вершины стека.

- Метод **Pop** – изъять элемент из стека;

При изъятии элемента из стека необходимо переместить указатель вершины стека и уменьшить количество элементов.

- Метод **IsEmpty** – проверить стек на пустоту;

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

- Метод **IsFull** – проверить стек на полноту.

Стек полон при исчерпании всей отведенной под хранение элементов памяти, т.е. когда значение DataCount совпадает со значением MemSize.

3.1.2 Арифметические выражения

Арифметическое выражение – выражение, в котором операндами являются объекты, над которыми выполняются арифметические операции: $(a + b * c) * (c / d - e)$.

При такой форме записи (называемой **инфиксной**, где знаки операций стоят между операндами) порядок действий определяется расстановкой скобок и приоритетом операций.

Постфиксная (или обратная польская) форма записи не содержит скобок, а знаки операций следуют после соответствующих операндов. Тогда для приведённого примера постфиксная форма будет иметь вид: $a\ b\ c * +\ c\ d / e - *$..

Известный ученый Эдсгер Дейкстра предложил **алгоритм для перевода арифметических выражений из инфиксной в постфиксную форму**. Данный алгоритм основан на использовании стека:

1) Для каждой лексемы в инфиксной форме:
   1.1) Если лексема – операнд, поместить ее в постфиксную форму;
   1.2) Если лексема – открывающая скобка, поместить ее в стек;
   1.3) Если лексема – закрывающая скобка:
      1.3.1) Пока на вершине стека не открывающая скобка:
         1.3.1.1) Извлечь из стека элемент;
         1.3.1.1) Поместить элемент в постфиксную форму;
      1.3.2) Извлечь из стека открывающую скобку;
   1.4) Если лексема – операция:
      1.4.1) Пока приоритет лексемы меньше или равен приоритета верхнего элемента стека:
         1.4.1.1) Извлечь из стека элемент;
         1.4.1.2) Поместить элемент в постфиксную форму;
      1.4.2) Поместить лексему в стек;
2) По исчерпании лексем в инфиксной форме перенести все элементы из стека в постфиксную форму.

$(a + b * c) * (c / d - e)$

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

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

Далее по полученной постфиксной необходимо **вычислить значение арифметического выражения**, используя следующий алгоритм:

1) Для каждой лексемы в постфиксной форме:
   1.1) Если лексема – операнд, поместить ее значение в стек;
   1.2) Если лексема – операция:
   1.2.1) Извлечь из стека значения двух операндов;
   1.2.2) Выполнить операцию (верхний элемент из стека является правым операндом, следующий за ним – левым;
   1.2.3) Положить результат операции в стек;
2) По исчерпании лексем из постфиксной формы на вершине стека будет результат вычисления выражения

3.2 Описание классов

3.2.1 Класс TStack

Объявление класса:

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

Поля:

maxSize – максимальное количество элементов, которые можно поместить в стек.

top – индекс последнего элемента в стеке.

elems – указатель типа **ValueType** на первый элемент вектора.

Конструкторы:

```
TStack(size_t maxSize = 10);
```

Назначение: инициализация полей класса **TStack** и выделение памяти под хранение элементов вектора.

Входные данные: **maxSize** – максимальное количество элементов, которые можно поместить в стек.

Выходные данные: отсутствуют.

```
TStack(const TStack<ValueType>& s);
```

Назначение: создание копии вектора.

Входные данные: **TStack<ValueType>& s** – константная ссылка на стек.

Выходные данные: отсутствуют.

Деструктор:

```
~TStack();
```

Назначение: освобождение памяти, занимаемой динамическими полями класса **TStack**.

Входные данные: отсутствуют.

Выходные данные: отсутствуют.

Методы:

```
void Push(const ValueType& e);
```

Назначение: добавление элемента в стек.

Входные данные: **const ValueType& e** – константная ссылка на элемент.

Выходные данные: отсутствуют.

```
void Pop();
```

Назначение: изъятие элемента из стека.

Входные данные: отсутствуют.

Выходные данные: отсутствуют.

```
ValueType Top() const;
```

Назначение: получение значения элемента из стека.

Входные данные: отсутствуют.

Выходные данные: значение элемента.

```
bool IsEmpty() const;
```

Назначение: проверка на пустоту.

Входные данные: отсутствуют.

Выходные данные: результат проверки (1 – стек пустой, 0 – стек не пустой).

3.2.2 Класс TArithmeticExpression

Объявление класса:

```
class TArithmeticExpression {
private:
       std::string infix;
       std::vector<std::string> postfix;
       std::vector<std::string> lexems;
       std::map<std::string, int> priority;
       std::map<std::string, double> operands;
       bool IsOperator(char c)const;
       bool IsConst(const std::string& str)const;
       void Check();
       void Parse();
       void ToPostfix();
public:
       TArithmeticExpression(std::string infx);
       std::string GetInfix() const;
       std::string GetPostfix() const;
       std::vector<std::string> GetOperands() const;
       std::map<std::string, double> SetValues();
       double Calculate(const std::map<std::string, double>& values);
};
```

Поля:

infix – инфиксная форма арифметического выражения.

postfix – постфиксная форма арифметического выражения.

lexems – лексемы арифметического выражения.

priority – приоритет операций в арифметическом выражении.

operands – операнды арифметического выражения.

Конструкторы:

```
TArithmeticExpression(std::string infx);
```

Назначение: инициализация полей класса **TArithmeticExpression**.

Входные данные: **std::string infx** – инфиксная форма арифметического выражения.

Выходные данные: отсутствуют.

Методы:

bool IsOperator(char c)const;

Назначение: проверка, является ли символ оператором.

Входные данные: **c** – проверяемый символ.

Выходные данные: результат проверки (1 – оператор, 0 – не оператор).

bool IsConst(const std::string& str)const;

Назначение: проверка, является ли строка константой.

Входные данные: **const std::string& str** - константная ссылка на строку.

Выходные данные: результат проверки (1 – константа, 0 – не константа).

void Check();

Назначение: проверка корректности записи арифметического выражения.

Входные данные: отсутствуют.

Выходные данные: отсутствуют.

void Parse();

Назначение: разбор арифметического выражения на лексемы.

Входные данные: отсутствуют.

Выходные данные: отсутствуют.

void ToPostfix();

Назначение: перевод арифметического выражения из инфиксной формы в постфиксную.

Входные данные: отсутствуют.

Выходные данные: отсутствуют.

std::string GetInfix() const;

Назначение: получение инфиксной формы арифметического выражения.

Входные данные: отсутствуют.

Выходные данные: полученная строка.

```
std::string GetPostfix() const;
```

Назначение: получение постфиксной формы арифметического выражения.

Входные данные: отсутствуют.

Выходные данные: полученная строка.

```
std::vector<std::string> GetOperands() const;
```

Назначение: получение операнд арифметического выражения.

Входные данные: отсутствуют.

Выходные данные: вектор строк.

```
std::map<std::string, double> SetValues();
```

Назначение: получение значений и их присваивание операндам.

Входные данные: отсутствуют.

Выходные данные: контейнер значений.

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

Назначение: вычисление результата арифметического выражения.

Входные данные: `const std::map<std::string, double>& values` – константная ссылка на контейнер значений.

Выходные данные: результат вычисления.

Заключение

В ходе лабораторной работы были изучены основные термины и понятия, связанные со стеками, а также наиболее эффективные способы их представления (хранения). Были изучены понятия о арифметическом выражении и алгоритмы перевода из инфиксной формы в постфиксную и вычисления результата.

На основе подготовленной теоретической базы, были реализованы классы для представления стеков TStack и обработки арифметических выражений TArithmeticExpression со всеми необходимыми операциями. Для проверки работоспособности и эффективности реализации перечисленных выше классов были написаны приложения sample_tstack и sample_tarithmeticexpression, а также модульные тесты.

Литература

1. Барышева И.В. Учебно-методическое пособие. – Нижний Новгород: Нижегородский госуниверситет, 2017 – 105 с.

2. Динамические структуры данных. Односвязный список, стек и очередь. Алгоритмы их обработки. [https://op-al.gitbook.io/s-30-voprosy-i-dop.-voprosy/22.-dinamicheskie-struktury-dannykh.-odnosvyaznyi-spisok-stek-i-ochered.-algoritmy-ikh-obrabotki].

Приложения

Приложение А. Реализация класса TStack

```
template <typename ValueType>
TStack<ValueType>::TStack(size_t _maxSize) {
	if (maxSize <= 0)
		throw std::exception("negative or zero max size");
	maxSize = _maxSize;
	top = -1;
	elems = new ValueType[maxSize];
}

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

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

template <typename ValueType>
void TStack<ValueType>::Push(const ValueType& e) {
	if (top == maxSize - 1) {
		ValueType* tmp = new ValueType[maxSize * 2];
		std::copy(elems, elems + maxSize, tmp);
		delete[]elems;
		elems = tmp;
		maxSize *= 2;
	}
	elems[++top] = e;
}

template <typename ValueType>
void TStack<ValueType>::Pop() {
	if (IsEmpty())
		throw std::exception("got empty stack");
	top--;
}

template <typename ValueType>
ValueType TStack<ValueType>::Top() const {
	if (IsEmpty())
		throw std::exception("got empty stack");
	return elems[top];
}

template <typename ValueType>
bool TStack<ValueType>::IsEmpty() const {
	return (top == -1);
}
```

Приложение Б. Реализация класса TArithmeticExpression

```
TArithmeticExpression::TArithmeticExpression(const std::string& infx) : infix(infx) {
	priority = { {"+", 1},{"-", 1},{"*", 2},{"/", 2} };
	ToPostfix();
}

std::string TArithmeticExpression::GetInfix() const {
	return infix;
```

```
}

bool TArithmeticExpression::IsOperator(char c) const {
	return c == '+' || c == '-' || c == '*' || c == '/' || c == '(' || c == ')';
}

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

void TArithmeticExpression::Check() {
	if (infix.empty()) {
		throw std::exception("got empty string");
	}
	if (infix[0] == '+' || infix[0] == '*' || infix[0] == '/' || infix[0] == '.' ||
infix[0] == ')') {
		throw std::exception("arithmetic expression start with operator");
	}
	int opening_brackets = 0, closing_brackets = 0, points = 0;
	if (infix[0] == '(') {
		if (infix[1] == ')' || infix[1] == '+' || infix[1] == '*' || infix[1] ==
'/' || infix[1] == '.') {
			throw std::exception("operator after opening bracket");
		}
		opening_brackets++;
	}
	for (int i = 1; i < infix.size() - 2; i++) {
		if (IsOperator(infix[i]) || infix[i] == '.' || infix[i] >= 65 && infix[i]
<= 90 || infix[i] >= 97 && infix[i] <= 122 || infix[i] >= 48 && infix[i] <= 57) {
			if (infix[i] == '(') {
				if (infix[i + 1] == ')' || infix[i + 1] == '+' || infix[i +
1] == '-' || infix[i + 1] == '*' || infix[i + 1] == '/' || infix[i + 1] == '.') {
					throw std::exception("operator after opening
bracket");
					break;
				}
				opening_brackets++;
			}
			if (infix[i] == ')') {
				if (infix[i - 1] == '(' || infix[i - 1] == '+' || infix[i -
1] == '-' || infix[i - 1] == '*' || infix[i - 1] == '/' || infix[i - 1] == '.') {
					throw std::exception("operator before closing
bracket");
					break;
				}
				closing_brackets++;
			}
			if (infix[i] == '/' && infix[i + 1] == '0') {
				throw std::exception("division by zero");
			}
			if (infix[i] == '+' || infix[i] == '-' || infix[i] == '*' ||
infix[i] == '/' || infix[i] == '.') {
				if (infix[i + 1] == '+' || infix[i + 1] == '-' || infix[i +
1] == '*' || infix[i + 1] == '/' || infix[i + 1] == '.') {
					throw std::exception("repeat operator");
					break;
				}
			}
			while (!IsOperator(infix[i])) {
				if (infix[i] >= 48 && infix[i] <= 57 || infix[i] == '.') {
					if (infix[i] == '.') {
						points++;
					}
```

```
                                if (points > 1) {
                                        std::cout << "constant contains more than one
point";
                                }
                        }
                        i++;
                        if (i == infix.size()) {
                        }
                }
        }
        else {
                throw std::exception("expression contains invalid characters");
                break;
        }
    }
    if (infix[infix.size() - 1] == ')') {
        if (infix[infix.size() - 2] == '(' || infix[infix.size() - 2] == '+' ||
infix[infix.size() - 2] == '-' || infix[infix.size() - 2] == '*' || infix[infix.size()
- 2] == '/' || infix[infix.size() - 2] == '.') {
                throw std::exception("operator before closing bracket");
        }
        closing_brackets++;
    }
    if (opening_brackets > closing_brackets) {
        throw std::exception("missing closing bracket");
    }
    else if (opening_brackets < closing_brackets) {
        throw std::exception("missing opening bracket");
    }
    if (infix[infix.size()] == '+' || infix[infix.size()] == '-' ||
infix[infix.size()] == '*' || infix[infix.size()] == '/' || infix[infix.size()] == '.'
|| infix[infix.size()] == '(') {
        throw std::exception("arithmetic expression end with operator");
    }
}

void TArithmeticExpression::Parse() {
    Check();
    std::string str;
    for (int i = 0; i < infix.size(); i++) {
        if (IsOperator(infix[i])) {
            if (infix[i] == '-' && i == 0) {
                lexems.push_back("0");
                lexems.push_back("-");
                str.clear();
                continue;
            }
            else {
                str = infix[i];
                lexems.push_back(str);
                str.clear();
            }
            continue;
        }
        else {
            while (!IsOperator(infix[i])) {
                str += infix[i];
                i++;
                if (i == infix.size()) {
                    break;
                }
            }
            lexems.push_back(str);
            str.clear();
        }
        if (i != infix.size()) {
            str = infix[i];
            lexems.push_back(str);
            str.clear();
        }
```

```
	}
}

void TArithmeticExpression::ToPostfix() {
	Parse();
	TStack<std::string> st;
	std::string item;
	std::string stackItem;
	for (int i = 0; i <= lexems.size() - 1; i++) {
		item = lexems[i];
		if (item == "(") {
			st.Push(item);
		}
		else if (item == ")") {
			stackItem = st.Top();
			st.Pop();
			while (stackItem != "(") {
				postfix.push_back(stackItem);
				stackItem = st.Top();
				st.Pop();
			}
		}
		else if (item == "+" || item == "-" || item == "*" || item == "/") {
			while (!st.IsEmpty()) {
				stackItem = st.Top();
				st.Pop();
				if (priority[item] <= priority[stackItem])
					postfix.push_back(stackItem);
				else {
					st.Push(stackItem);
					break;
				}
			}
			st.Push(item);
		}
		else {
			operands.insert({ item, 0.0 });
			postfix.push_back(item);
		}
	}
	while (!st.IsEmpty()) {
		stackItem = st.Top();
		st.Pop();
		postfix.push_back(stackItem);
	}
}

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

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

std::map<std::string, double> TArithmeticExpression::SetValues() {
	double val;
	for (auto& op : operands) {
		if (IsConst(op.first)) {
			operands[op.first] = std::stof(op.first);
		}
```

```
            else {
                  std::cout << "Enter value of " << op.first << ": ";
                  std::cin >> val;
                  operands[op.first] = val;
            }
      }
      return operands;
}

double TArithmeticExpression::Calculate(const std::map<std::string, double>& values) {
      for (auto& val : values) {
            try {
                  operands.at(val.first) = val.second;
            }
            catch (std::out_of_range& e) {}
      }
      TStack<double> st;
      double leftOperand, rightOperand;
      for (std::string lexem : postfix) {
            if (lexem == "+") {
                  rightOperand = st.Top();
                  st.Pop();
                  leftOperand = st.Top();
                  st.Pop();
                  st.Push(leftOperand + rightOperand);
            }
            else if (lexem == "-") {
                  rightOperand = st.Top();
                  st.Pop();
                  leftOperand = st.Top();
                  st.Pop();
                  st.Push(leftOperand - rightOperand);
            }
            else if (lexem == "*") {
                  rightOperand = st.Top();
                  st.Pop();
                  leftOperand = st.Top();
                  st.Pop();
                  st.Push(leftOperand * rightOperand);
            }
            else if (lexem == "/") {
                  rightOperand = st.Top();
                  st.Pop();
                  leftOperand = st.Top();
                  st.Pop();
                  st.Push(leftOperand / rightOperand);
            }
            else {
                  st.Push(operands[lexem]);
            }
      }
      return st.Top();
}
```