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:

«Стек. Постфиксная форма.»

Выполнил(а): студент группы 3822Б1ФИ2
_____ / Холин К.И/
Подпись

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

Нижний Новгород
2023

Содержание

Введение....3
1 Постановка задачи....4
2 Руководство пользователя....5
2.1 Приложение для демонстрации работы стека....5
2.2 Приложение для демонстрации работы постфиксной формы арифметического выражения....7
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

Часто в любой программе программисты сталкиются с ситуациями, когда нужно провести некоторые простые или сложные вычисления. для того чтобы решить конкретную задачу. Эти вычисления содержат различные операции над операндами. Соответственно, чтобы проводить вычисления, необходимо подумать о том, с помощью какой структуры данных можно это сделать, где мы можем хранить операнды и их значения и операции, каким образом подсчитать результат? В этом случае хорошо подходит стек и постфиксная форма арифметического выражения.

1 Постановка задачи

Цель – Реализовать шаблонный класс TStack и на его основе реализовать класс ArithmeticExpression.

Задачи

По классу TStack:

Реализовать операции для работы со стеком: добавление элемента на вершину стека, извлечение элемента с вершины стека, проверка на пустоту, проверка на заполненность, получение значения верщины стека.

По классу TArithmeticExpression:

Воспользоваться готовыми решениями для хранения операндов и операций арифметического выражения и при помощи их реализовать следующие операции для работы с арифметическим выражением: разбор выражения на отдельные лексемы, перевод в постфиксную форму, вычисление по постфиксной форме. Класс должен содержать методы проверки на корректность введённого 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).

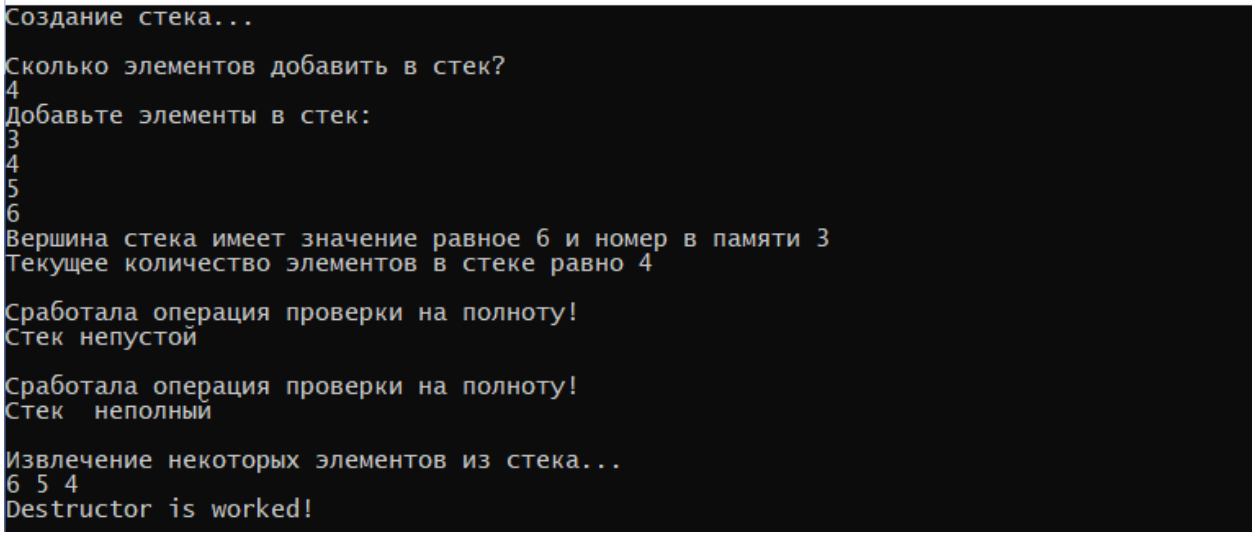

Рис. 1. Основное окно программы

2. На первом шаге создаётся пустой стек. (**Error! Reference source not found.**).

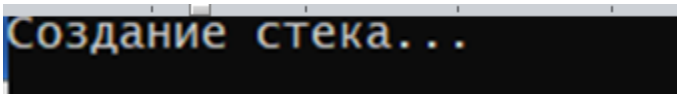

Рис. 2. Создание пустого стека

3. На втором этапе идёт добавление введённого количества элементов пользователем в стек (**Error! Reference source not found.**).

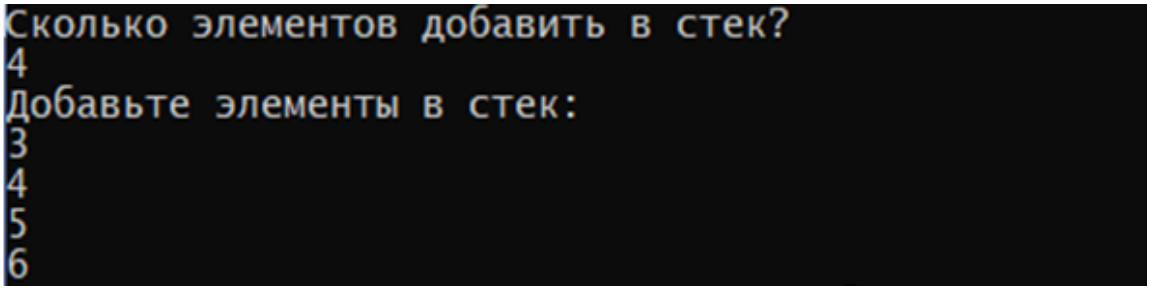

Рис. 3. Добавление элементов в стек с помощью Push()

4. Информацию о вершине стека и его количестве элементов можно увидеть на рисунке ниже
(**Error! Reference source not found.**).

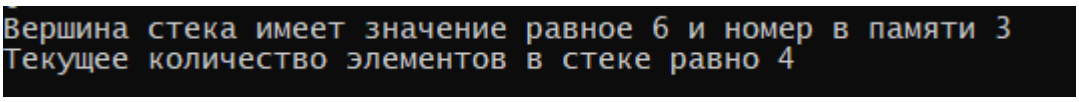

Рис. 4. Вершина стека и текущее количество элементов

5. Далее проверяем операции стека для проверки на полноту и пустоту (рис. 5).

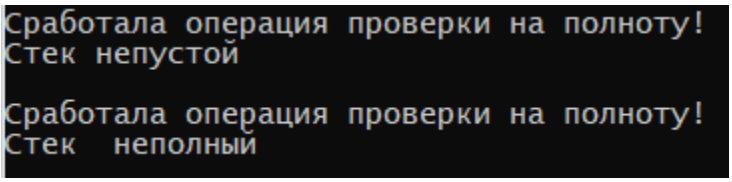

Рис. 5. Проверка на полноту и пустоту

6. В завершение программы все элементы стека извлекаются и стек удаляется (рис. 6).

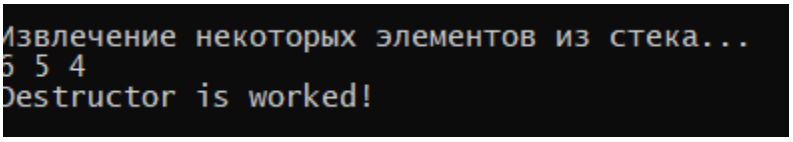

Рис. 6. Извлечение элементов из стека и удаление стека

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. 7).

Рис. 7. Основное окно программы

2. В главном окне программы пользователю предоставляется возможность ввода арифметического выражения. В случае возникновения ошибки при вводе, программа выведет соответствующее сообщение об ошибке и завершит свою работу (рис. 8).

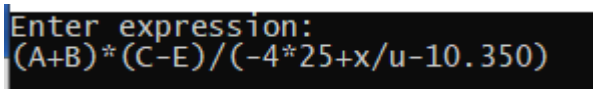

Рис. 8. Ввод арифметического выражения

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. 9).

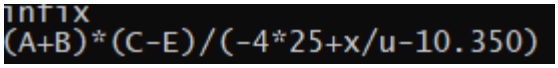

Рис. 9. Вывод инфиксной формы арифметического выражения

4. После на экран выводится постфиксная форма на основании инфиксной с помощью соответствующих методов преобразования в постфиксную форму. (рис. 10).

Рис. 10. Постфиксная форма арифметичекого выражения

5. В завершение пользователю предлагается ввести значения операндов. На основе введённых данных по постфиксной форме вычисляется результат арифметического выражения (рис. 11).

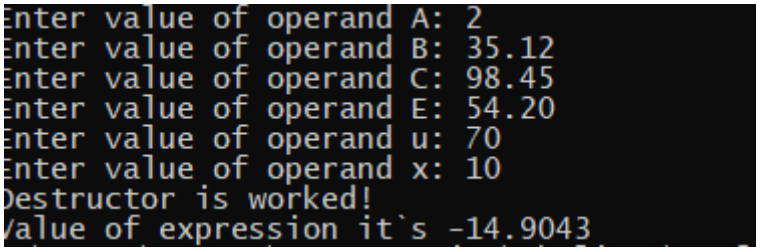

Рис. 11. Вввод значений операндов и подсчёт результата

3 Руководство программиста

3.1 Описание алгоритмов

3.1.1 Стек

Стек (или стопка) представляет собой структуру данных, организованную по принципу "последний вошел - первый вышел" (Last In, First Out, LIFO). Это значит, что первые элементы, добавленные в стек, будут извлечены позже всех остальных.

Операции, которые можно выполнять со стеком, включают добавление элемента (push) на вершину стека и удаление элемента (pop) с вершины стека. Доступ к любому элементу получить нельзя, кроме как последовательного извлечения элементов, начиная с вершины стека.

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.

Алгоритм преобразования в постфиксную форму:

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 лексема – операнд поместить в постфиксную форму.
5. Если лексема – открывающая скобка, поместить её в стек.
6. Если лексема – операция:
    - Пока приоритет лексемы меньше или равен приоритета верхнего элемента стека, извлечь элемент стека и поместить в постфиксную форму.
    - Извлечь из стека открывающую скобка.
7. Если лексема - закрывающая скобка:

- Пока на вершине не открывающая скобка, извлечь элемент стека и поместить в постфиксную форму.
- Поместить лексему в стек.

8. По исчерпании лексем инфиксной формы извлечь все элементы стека и поместить в постфиксную форму.

ПРИМЕР:

Выполнить преобразование в постфиксную форму следующего выражения:

$$(A + B) * \left(\frac{C}{E} - T * 25\right) - 10$$

На рисунке ниже показано, как изменяется состояние стека на каждой итерации разбора арифметического выражения на отдельные лексемы (рис. 12).

I = 1 I = 2 I = 3 I = 4 I = 5 I = 6 I = 7 I = 8 I = 9 I= 10 I = 11 I = 12

								*			
						/	-	-			
		+			(	(	(	(			
(	(	(		*	*	*	*	*	*	-	

Рис. 12. Состояния стека операций на каждой итерации

Операнды попадают в стек операндов(постфиксную форму). Результат вы увидите на (рис. 13).

A B + C E / T 25 * - * 10 -

Рис. 13. Постфиксная форма

Алгоритм вычисления значения арифметического выражения по постфиксной форме

1. Создайте пустой стек.
2. Пройдите по каждому символу в постфиксной форме слева направо.
3. Если лексема - операнд (число), поместить в стек.
4. Если лексема – операция:
   - Извлечь из стека значения двух операндов.
   - Выполнить операцию(верхний элемент стека – правый операнд, следующим за ним – левый операнд).
   - Положить результат операции в стек.

По исчерпании лексем постфиксной формы на вершине стека будет результат выражения.

3.2 Описание программной реализации

3.2.1 Описание класса TStack

```
template <typename T> class Stack {
private:
	int top;
	int mem_size;
	T* pMem;
public:
	Stack(int size = 100);
	Stack(const Stack<T>& obj);
	~Stack();

	T Top();

	bool IsEmpty()const { return top == -1; }
	bool IsFull()const { return top == mem_size - 1; }

	void Push(const T val);
	T Pop();

	friend istream& operator>>(istream& istr, Stack<T>& st) {
		T elem;
		istr >> elem;
		st.Push(elem);
		return istr;
	}
};
```

Поля:

mem_size – размер стека.

top – индекс верхнего элемента в стеке.

pMem – память для представления стека.

Конструкторы:

TStack(int size = 100);

Назначение: конструктор по умолчанию и с параметром.

Входные параметры: **size** – количество выделенной памяти.

Выходные параметры: отсутствуют.

TStack(const TStack<T>& s);

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

Входные параметры: **s** – объект класса **TStack**, из которого копируем данных.

Выходные параметры: отсутствуют.

```
~TStack();
```

Назначение: освобождение памяти.

Входные данные: отсутствуют.

Выходные данные: отсутствуют.

Методы:

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

Назначение: проверка на пустоту стека.

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

```
bool IsFull()const;
```

Назначение: проверка на полноту стека.

Входные параметры: отсутствуют.

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

```
T Top();
```

Назначение: получение данных элемента с верхушки стека.

Входные параметры: отсутствуют.

Выходные параметры: элемент на вершине стека типа данных **`T`**.

```
void Push(const T val);
```

Назначение: добавление нового элемента на вершину стека.

Входные параметры: **`val`** – добавляемый элемент типа данных **`T`**.

```
T Pop();
```

Назначение: извлечение элемента с вершины стека.

Входные параметры: отсутствуют.

Выходные параметры: элемент на вершине стека типа данных **`T`**.

3.2.2 Описание класса TArithmeticExpression

```
class ArithmeticExpression {
private:
	string infix;
	vector<string> postfix;
	vector<string> lexemes;
	map<char, int> priority;
	map<string, double> operands;

	void Parse();
	void ToPostfix();

public:
	ArithmeticExpression(string infix_);

	string GetInfix()const { return infix; }
	vector<string> GetPostfix()const { return postfix; }
	vector<string> GetOperands()const;
	map<string, double> SetOperands(const vector<string> operands);

	double Calculate(const map<string, double>& values);
private:
	void Check()const;
	bool Is_Operator(char c)const;
	bool Is_Operand_String(char c)const;
	bool Is_Operand_const(char c)const;
	double Transform(string str)const;
};
```

Поля:

infix – инфиксная форма арифметического выражения.

postfix – постфиксная форма арифметического выражения.

lexemes –контейнер **vector**, содержащий упорядоченную последовательность лексем инфиксной формы выражения.

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.

Выходные данные: отсутствуют.

Методы:

```
void Parse();
```

Назначение: разбор арифметического выражения на лексемы.

Входные данные: отсутствуют.

Выходные данные: отсутствуют.

```
void ToPostfix();
```

Назначение: преобразование инфиксной формы арифметического выражения в постфиксную форму.

```
string GetInfix() const;
```

Назначение: получение инфиксной формы выражения.

Входные параметры: отсутствуют.

Выходные параметры: инфиксная форма арифметического выражения.

```
vector<string> GetPostfix() const;
```

Назначение: получение постфиксной формы выражения.

Входные параметры отсутствуют.

Выходные параметры: постфиксная форма арифметического выражения.

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

Назначение: получение операндов постфиксной формы выражения.

Входные параметры: отсутствуют.

Выходные параметры: вектор операндов арифметического выражения.

```
map<string, double> SetOperands(const vector<string> operands);
```

Назначение: инициализация операндов постфиксной формы арифметического выражения.

Входные параметры: отсутствуют.

Выходные параметры: контейнер инициализированных 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.

`void Check()const;`
Назначение: комплексная проверка арифметического выражения на наличие ошибок.

Входные параметры: отсутствуют.

Выходные параметры: отсутствуют.

`bool Is_Operator(char c)const;`
Назначение: проверка входного символа на оператор.

Входные параметры: **`c –`** входной символ для проверки.

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

`bool Is_Operand_String(char c)const;`
Назначение: проверка на многосимвольный операнд.

Входные параметры: **`c –`** входной символ для проверки.

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

`bool Is_Operand_const(char c)const;`
Назначение: проверка на операнд-константу.

Входные параметры: **`c –`** входной символ для проверки.

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

`double Transform(string str)const;`
Назначение: разбиение дробного числа на целую и дробную части.

Входные параметры: **`str`** – строка, содержащая вещественное дробное число.

Выходные параметры: преобразованный тип данных string в double.

.

Заключение

В ходе выполнения лабораторной работы был успешно разработан шаблонный класс **`TStack`** для представления стека и класс **ArithmeticExpression,** был реализован с помощью дополнительных способов хранения данных, включая основной – стек.

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

```
template <typename T> class Stack {
private:
	int top;
	int mem_size;
	T* pMem;
public:
	Stack(int size = 100);
	Stack(const Stack<T>& obj);
	~Stack();

	T Top();
	size_t Size()const noexcept { return top + 1; }
	int GetTop() { return top; }

	bool IsEmpty()const { return top == -1; }
	bool IsFull()const { return top == mem_size - 1; }

	void Push(const T val);
	T Pop();

	friend istream& operator>>(istream& istr, Stack<T>& st) {
		T elem;
		istr >> elem;
		st.Push(elem);
		return istr;
	}
};

template <typename T>
Stack<T>::Stack(int size) {
	if (size <= 0) {
		throw "Negative or zero size!";
	}
	top = -1;
	mem_size = size;
	pMem = new T[mem_size];
}



template <typename T>
Stack<T>::Stack(const Stack <T>& obj) {
	top = obj.top;
	mem_size = obj.mem_size;
	pMem = new T[mem_size];
	for(int i = 0; i <= top;i++){
		pMem[i] = obj.pMem[i];
	}
	cout << "Constructor copy is worked!" << endl;
}


template <typename T>
```

```
T Stack<T>::Top() {
	if (IsEmpty()) {
		throw "Stack is empty!";
	}
	else {
		return pMem[top];
	}
}


template <typename T>
void Stack<T>::Push(const T val) {
	if (IsFull()) {
		throw "Stack is full!";
	}
	else {
		pMem[++top] = val;
	}
}


template <typename T>
T Stack<T>::Pop() {
	if (IsEmpty()) {
		throw "Stack is empty!";
	}
	else {

		return pMem[top--];
	}
}

template <typename T>
Stack<T>::~Stack() {
	delete[] pMem;
	mem_size = 0;
	top = -1;
	cout << "Destructor is worked!" << endl;
}


}
```

Приложение Б. Реализация класса TArithmeticExpression

```
class ArithmeticExpression {
private:
	string infix;
	vector<string> postfix;
	vector<string> lexemes;
	map<char, int> priority;
	map<string, double> operands;

	void Parse();
	void ToPostfix();
public:
	ArithmeticExpression(const string& infix_);

	string GetInfix()const { return infix; }
	vector<string> GetPostfix()const { return postfix; }
	vector<string> GetOperands()const;
	map<string, double> SetOperands(const vector<string> operands);

	double Calculate(const map<string, double>& values);
private:
	void Check()const;
	bool Is_Operator(char c)const;
	bool Is_Operand_String(char c)const;
	bool Is_Operand_const(char c)const;
	double Transform(string str)const;
};


ArithmeticExpression::ArithmeticExpression(const string& infix_) :infix(infix_) {
	priority =
	{
		{'(',1},{'+',2},{'-',2},{'*',3}, {'/',3}
	};
	ToPostfix();
}


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


bool ArithmeticExpression::Is_Operand_String(char c)const {
	return c >= 65 && c <= 90 || c >= 97 && c <= 122;
}


bool ArithmeticExpression::Is_Operand_const(char c)const {
	return c >= 48 && c <= 57;
}


double ArithmeticExpression::Transform(string str)const {
	int i = 0;
	string int_part;
	while (i < str.find(".")) {
		int_part += str[i];
		i++;
	}
	int index = str.find(".") + 1;
	string fractal_part;
```

```
        int count_signs = 0;
        while (index < str.size()) {
                count_signs++;
                fractal_part += str[index];
                index++;
        }
        double arg = 1;
       double fractal_part_transformed = stod(fractal_part) *(arg/pow(10,count_signs));
        double transformed = stod(int_part) + fractal_part_transformed;
        return transformed;
}


void ArithmeticExpression::Check()const {
        int i = 0;
        char c;
        while (i < infix.size()) {//проверяет на чужеродные символы
                c = infix[i];
                if (Is_Operand_String(c) || Is_Operand_const(c) ||
                        Is_Operator(c) || c == '.' || c == ' ') {
                        i++;
                        continue;
                }
                else {
                        throw "Mistake in arithmetic expression!";
                }
        }
        i = 0;

        int count_left_open_bracket = 0;
        int count_right_close_bracket = 0;
        char next_c;
        while (i < infix.size()) {
                c = infix[i];
                next_c = infix[i + 1];
                if (c == '(') {
                        if (next_c == '+' || next_c == '*' || next_c == '/' ||
                        next_c == ')' || next_c == ' ') {
                                throw "Mistake in arithmetic expression!";
                        }
                        count_left_open_bracket++;
                }
                if (c == ' ' && next_c == ')') {
                        throw "Mistake in arithmetic expression!";
                }
                if (c == ')') {
                        count_right_close_bracket++;
                }
                i++;
        }
        if (count_left_open_bracket != count_right_close_bracket) {
                throw "Mistake in arithmetic expression!";
        }
        i = 0;

        char cc = 0;
        while (i < infix.size()) {
                c = infix[i];
                switch (c) {
                case '+': case '-': case '*': case '/':
                {
                        cc = infix[i + 1];
                        if (cc == ')') {
                                throw "Mistake in arithmetic expression!";
```

```
                        }
                        auto tmp = priority.find(cc);
                        if (tmp != priority.end() && tmp->first != ')' && tmp->first
                        != '(') {
                              throw "Mistake in arithmetic expression!";
                        }
                  }
                  default:
                        cc = infix[i + 1];
                        if (c == '.' && cc == '.') {
                              throw "Mistake in arithmetic expression!";
                        }
                        if (c == ' ' && cc == ' ') {
                              throw "Mistake in arithmetic expression!";
                        }
                        break;
                  }
                  i++;
            }
      }

      void ArithmeticExpression::Parse() {
            char c;
            char cc;
            int count_points = 0;
            string str;
            int i = 0;
            char first_c;
            Check();
            for (i = 0; i < (infix.size()); i++) {
                  c = infix[i];
                  first_c = c;
                  switch (c) {
                  case '+': case '-': case '*': case '/': case '(': case ')':
                  {
                        if (i == 0 && c != '-' && c != '(') {
                              throw "Mistake in arithmetic expression!";
                        }
                        if (i == infix.size() - 1 && c != ')') {
                              throw "Mistake in arithmetic expression!";
                        }
                        if ((c == '-' && i == 0) || (c == '-' && infix[i - 1] ==
                        '(')) {
                              lexemes.push_back("0");
                              lexemes.push_back("-");
                              str = "";
                              continue;
                        }
                        str = c;
                        lexemes.push_back(str);
                        str = "";
                        continue;
                  }
                  default:
                        while (!Is_Operator(c)) {
                              if (c == '.' && str == "") {
                                    throw "Mistake in arithmetic expression!";
                              }
                              if  (Is_Operand_const(c)  &&  Is_Operand_const(first_c)
                              || c == '.') {
                                    if (c == '.') {
                                          count_points++;
                                          if (count_points > 1) {
```

```
                        throw "Mistake in arithmetic expression!";
                        }
                    }
                    str += c;
                    i++;
                    if (i == infix.size()) {
                        break;
                    }
                    c = infix[i];
                    cc = infix[i + 1];
                    if (Is_Operand_String(c)) {
                        throw "Mistake in arithmetic expression!";
                    }
                    if (c == '.' && Is_Operator(infix[i + 1]) && i +
                    1 != infix.size()) {
                        throw "Mistake in arithmetic expression!";
                    }
                    if (c == ' ' && Is_Operator(cc)) {
                        c = cc;
                        i++;
                    }
                    if (c == ' ') {
                        throw "Mistake in arithmetic expression!";
                    }
                }
                else {
                    if (c == ' ' && first_c != ' ') {
                        throw "Mistake in arithmetic expression!";
                    }
                    if (c == ' ' && first_c == ' ') {
                        break;
                    }
                    str += c;
                    i++;
                    if (i == infix.size()) {
                        break;
                    }

                    c = infix[i];
                    cc = infix[i + 1];
                    if (!(c != '.')) {
                        throw "Mistake in arithmetic expression!";
                    }
                    if (c == ' ' && Is_Operator(cc)) {
                        c = cc;
                        i++;
                    }
                }

            }
            if (c == ' ' && first_c == ' ') {
                break;
            }
            if (i == infix.size() - 1 && c != ')') {
                throw "Mistake in arithmetic expression!";
            }
            count_points = 0;
            lexemes.push_back(str);
            str = "";
            break;
        }
        if (c == ' ' && first_c == ' ') {
            continue;
```

```
                }
                if (c == '\0') {
                        continue;
                }
                if (i != infix.size()) {
                        str = c;
                        lexemes.push_back(str);
                        str = "";
                }
        }
}

void ArithmeticExpression::ToPostfix() {
        Parse();
        Stack<char> stack_ops;
        unsigned char c;
        string lexeme;
        char stack_op;
        for (int i = 0; i <= (lexemes.size() - 1); i++) {
                lexeme = lexemes[i];
                c = lexeme[0];
                switch (c) {
                case '(':
                {
                        stack_ops.Push(c);
                        break;
                }
                case '+': case '-': case '*': case '/': case '—':
                {
                        while (!stack_ops.IsEmpty()) {
                                stack_op = stack_ops.Pop();
                                if (priority[c] <= priority[stack_op]) {
                                        string tmp_str;
                                        tmp_str = stack_op;
                                        postfix.push_back(tmp_str);
                                }
                                else {
                                        stack_ops.Push(stack_op);
                                        break;
                                }
                        }
                        stack_ops.Push(c);
                        break;
                }
                case ')':
                {
                        stack_op = stack_ops.Pop();
                        string tmp_str;
                        while (stack_op != '(') {
                                tmp_str = stack_op;
                                postfix.push_back(tmp_str);
                                stack_op = stack_ops.Pop();
                        }
                        break;
                }
                default:
                        if (c >= 47 && c <= 57) {//operand-number
                                if (lexeme.find(".") != -1) {
                                        operands.insert({ lexeme,Transform(lexeme) });
                                        postfix.push_back(lexeme);
                                        break;
                                }
                                operands.insert({ lexeme,stod(lexeme) });
                                postfix.push_back(lexeme);
                                break;
```

```
                        }
                        operands.insert({ lexeme, 0.0 });//operand-symbol(string)
                        postfix.push_back(lexeme);
                        break;
                }
}

        while (!stack_ops.IsEmpty()) {
                stack_op = stack_ops.Pop();
                string tmp_str;
                tmp_str = stack_op;
                postfix.push_back(tmp_str);
        }
}


vector<string> ArithmeticExpression::GetOperands()const {
        vector<string> tmp;
        auto it_begin{ operands.begin() };
        auto it_end{ operands.end() };
        while (it_begin != it_end) {
                tmp.push_back(it_begin->first);
                it_begin++;
        }
        return tmp;
}


map<string, double> ArithmeticExpression::SetOperands(const vector<string> operands) {
        map<string, double> tmp;
        double value;
        auto it_begin{operands.begin() };
        auto it_end{ operands.end() };
        while (it_begin != it_end) {
                if (this->operands.at(*it_begin) != 0 || *it_begin == "0") {
                        tmp.insert({ *it_begin, this->operands.at(*it_begin) });
                        it_begin++;
                        continue;
                }
                cout << "Enter value of operand " << *it_begin << ": ";
                cin >> value;
                tmp.insert({ *it_begin, value });
                it_begin++;
        }
        return tmp;
}


double ArithmeticExpression::Calculate(const map<string, double>& values) {
        Stack<double> expr_operands;
        string lexeme;
        char c;
        double left_op,right_op;
        auto it_begin = postfix.begin();
        auto it_end = postfix.end();
        while (it_begin != it_end) {
                lexeme = *it_begin;
                        c = lexeme[0];
                        switch (c) {
                        case '+':
                        {
                                right_op = expr_operands.Pop();
                                left_op = expr_operands.Pop();
                                expr_operands.Push(left_op + right_op);
                                break;
```

```
                        }
                        case '-':
                        {
                                right_op = expr_operands.Pop();
                                left_op = expr_operands.Pop();
                                expr_operands.Push(left_op - right_op);
                                break;
                        }
                        case '*':
                        {
                                right_op = expr_operands.Pop();
                                left_op = expr_operands.Pop();
                                expr_operands.Push(left_op * right_op);
                                break;
                        }
                        case '/':
                        {
                                right_op = expr_operands.Pop();
                                left_op = expr_operands.Pop();
                                if (right_op == 0) {
                                        throw "division by zero!";
                                }
                                expr_operands.Push(left_op / right_op);
                                break;
                        }
                        default:
                                expr_operands.Push(values.at(lexeme));
                                break;
                }
                it_begin++;
        }
        return expr_operands.Top();
}

int main()
{
   setlocale(LC_ALL, "rus");
   string expression_str;
   cout << "Enter expression:" << endl;
   cin >> expression_str;
   cout << endl;
   ArithmeticExpression expr(expression_str);
   vector<string> postfix = expr.GetPostfix();
   cout << "infix" << endl;
   cout << expr.GetInfix() << endl;
   cout << endl;
   cout << "postfix" << endl;
   auto iter_begin = postfix.begin();
   auto iter_end = postfix.end();
   while (iter_begin != iter_end) {
      cout << *iter_begin << " ";
      iter_begin++;
   }
   cout << '\n';
   vector<string> operands = expr.GetOperands();
   map<string, double> values_operands = expr.SetOperands(operands);
   cout << "Value of expression it`s " << expr.Calculate(values_operands);
   return 0;
}


int main()
{
   setlocale(LC_ALL, "rus");
   cout << "Создание стека..." << endl;
```

```
    Stack<double> st;


    cout << endl;
    cout << "Сколько элементов добавить в стек?" << endl;
    int count = 0;
    cin >> count;
    cout << "Добавьте элементы в стек: " << endl;
    for (int i = 0; i < count; i++) {
        cin >> st;
    }

    cout << "Вершина стека имеет значение равное " << st.Top() << " и номер в памяти " <<
st.GetTop() << endl;
    cout << "Текущее количество элементов в стеке равно " << st.Size() << endl;
    cout << endl;

    if (st.IsEmpty()) {
        cout << "Сработала операция проверки на полноту!" << endl;
        cout << "Стек пустой" << endl;
    }
    else {
        cout << "Сработала операция проверки на полноту!" << endl;
        cout << "Стек непустой" << endl;
    }
    cout << endl;
    if (st.IsFull()) {
        cout << "Сработала операция проверки на полноту!" << endl;
        cout << "Стек полный" << endl;
    }
    else {
        cout << "Сработала операция проверки на полноту!" << endl;
        cout << "Стек  неполный" << endl;
    }
    cout << endl;
    cout << "Извлечение некоторых элементов из стека..." << endl;
    double stack_item1 = st.Pop();
    double stack_item2 = st.Pop();
    double stack_item3 = st.Pop();
    cout << stack_item1 << " " << stack_item2 << " " << stack_item3 << endl;
    return 0;
}
```