Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное

образовательное учреждение высшего образования

«Национальный исследовательский Нижегородский

государственный университет им. Н.И. Лобачевского»

Институт информационных технологий, математики и механики

Отчёт по учебной практике

Вычисление арифметических выражений

Выполнил:

студент гр. 381806-1

Кизаев Д.Д.

Проверил:

к.т.н., доцент каф. МОСТ ИИТММ

Кустикова В.Д.

Нижний Новгород

2020 г.

Содержание

[Введение 3](#_Toc39321108)

[Постановка задачи 4](#_Toc39321109)

[Руководство пользователя 5](#_Toc39321110)

[Возможные ошибки ввода 7](#_Toc39321111)

[Руководство программиста 8](#_Toc39321113)

[Описание структуры программы 8](#_Toc39321114)

[**Описание структур данных** 8](#_Toc39321115)

[Шаблонный класс stack 8](#_Toc39321116)

[Класс variables и структура var 9](#_Toc39321117)

[Статический класс postfix 11](#_Toc39321118)

[Описание алгоритмов 13](#_Toc39321119)

[Заключение 15](#_Toc39321120)

[Приложения 16](#_Toc39321121)

[Приложение 1. Функция main 16](#_Toc39321122)

[Приложение 2. Класс stack 16](#_Toc39321123)

[Приложение 3. Класс postfix 18](#_Toc39321124)

[Приложение 4. Класс variables 26](#_Toc39321125)

# Введение

Одним из ключевых преимуществ компьютеров над человеком является их способность производить большой объем арифметических вычислений быстро и безошибочно. Однако, для того, чтобы человек мог эффективно использовать это преимущество, ему необходимо считаться и с недостатками компьютеров, таким как, например, отсутствие интуиции и необходимость в четком последовательном алгоритме. Для нас некоторые вещи могут казаться очевидными, в то время как компьютер нуждается в точном знании того, как и в какой последовательности вычисляются операторы.

Привычная нам запись арифметических выражений, подразумевающая положение знаков действий между операндами, называется инфиксной.

К способам записи выражения, гарантирующим отсутствие двусмысленности по отношению к порядку операций, можно отнести:

* выражения с полностью расставленными скобками;
* префиксный формат, подразумевающий положение оператора непосредственно перед операндами;
* постфиксный формат, подразумевающий положение оператора позади операндов.

Данная лабораторная работа посвящена третьему формату – постфиксной (также известной как обратная польская) нотации. В работе описаны механизмы перевода арифметических выражений из инфиксной нотации в постфиксную с помощью структуры данных стек, а также приложение-калькулятор, использующее данный вид записи арифметических выражений.

# Постановка задачи

**Задача:**

Разработать программу, способную конвертировать инфиксную запись арифметического выражения в постфиксную и выполнять вычисление данного выражения с использованием его постфиксной формы.

**Входные данные:**

1. Строка с арифметическим выражением в инфиксной форме (допустимы одно- и многобуквенные названия переменных, пробелы, а также константы)
2. Значения переменных, при условии наличия в строке (1) переменных.

**Выходные данные:**

1. Строка с постфиксной записью исходного выражения
2. Результат вычисления

# Руководство пользователя

Инструкция по использованию программы:

1. При запуске исполняемого файла программы интерфейс приглашает пользователя ввести строку с арифметическим выражением:

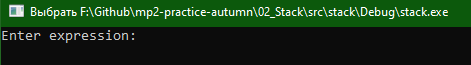


Рис. 1. Начальное окно

1. Введя необходимое выражение, нажмите Enter. Вы вольны использовать пробелы, но в случае ошибок в самом арифметическом выражении программа уведомит Вас об ошибке. Переменные могут содержать как одну, так и несколько букв и цифр, но действует стандартное ограничение: название переменной не должно начинаться с цифры.

Рисунок 2 демонстрирует пример ввода выражения «lets + find \* this / value55», где «lets», «find», «this» и «value55» - пользовательские переменные.

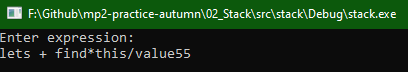


Рис. 2. Ввод арифметического выражения

1. Программа интерпретирует пользовательский ввод и возвращает постфиксную форму для заданного выражения. Во избежание визуального слияния переменных используются пробелы между операндами и операторами.

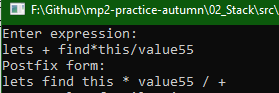


Рис. 3. Постфиксная форма выражения

1. Далее пользователю предлагается задать значения переменных. Поскольку среди операций доступно деление, используется тип double.

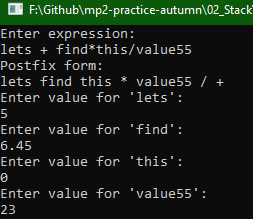


Рис. 4. Ввод значений переменных

1. После ввода последней переменной и нажатия Enter программа вычисляет и выводит результат.

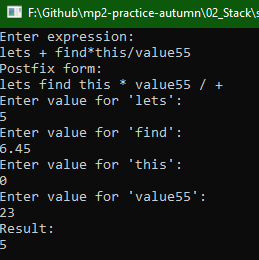


Рис. 5. Результат

# Возможные ошибки ввода

1. Некорректное количественное сочетание операторов и операндов:

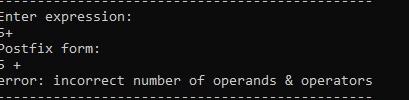


Рис. 1. Не хватает второго операнда

1. Некорректная расстановка скобок:

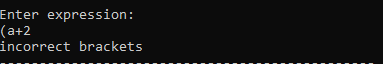


Рис 2. Количество открывающих и закрывающих скобок не совпадает

1. Некорректные названия переменных:

# 

Рис 3. Название переменных начинается с цифры

# Руководство программиста

## Описание структуры программы

Программа состоит из следующих файлов:

1. **stack.h** – объявление и реализация шаблонного класса stack;
2. **variables**.**h** – объявление класса для удобного хранения и обработки пользовательских переменных;
3. **variables**.**cpp** – описание класса для удобного хранения и обработки пользовательских переменных;
4. **postfix**.**h** – объявление класса с методами обработки арифметических выражений;
5. **postfix**.**cpp** – описание класса с методами обработки арифметических выражений;
6. **main**.**cpp –** основная функция работы постфиксного калькулятора.

**Описание структур данных**

### Шаблонный класс stack

Шаблонный класс stack реализует структуру данных стек на основе массивов.

**Объявление шаблонного класса stack:**

template <typename T>

class stack

{

private:

T\* elems;

int size;

int top;

public:

stack(int max\_size = 10);

stack(const stack<T>& s);

~stack();

T pop(); // достать верхний элемент

T peek() const; // посмотреть верхний элемент

void push(T new\_el); // вставка

bool is\_full() const;

bool is\_empty() const;

stack<T>& invert\_stack();

stack<T>& operator=(const stack<T>& other\_stack);

friend ostream& operator << (ostream& stream, const stack<T> \_stack)

{

stack<T> tmp = \_stack;

tmp.invert\_stack();

stream << "bottom -> ";

while (!(tmp.is\_empty()))

stream << "'" << tmp.pop() << "' -> ";

stream << "top" << endl;

return stream;

};

};

**Члены класса stack:**

T\* elems; - массив элементов стека шаблонного типа T

int size; - максимальный размер стека

int top; - индекс вершины стека

**Методы класса Stack:**

stack(int max\_size = 10); - конструктор с параметром, в качестве параметра принимает максимальный размер создаваемого стека;

stack(const stack<T>& s); - конструктор копирования для стеков, в качестве параметра принимает константную ссылку на объект класса stack;

~stack(); - деструктор для стека, не принимает параметров;

T pop(); - функция изъятия верхнего элемента стека, не принимает параметров, возвращает элемент шаблонного типа T, удаляя его из стека;

T peek() const; - функция просмотра верхнего элемента стека, не принимает параметров, возвращает элемент шаблонного типа T, не изменяя при этом сам стек;

void push(T new\_el); - функция вставки элемента на вершину стека;

bool is\_full() const; - функция, контролирующая, произошло ли переполнение стека;

bool is\_empty() const; - функция, контролирующая пустоту стека, необходима при проверке на возможность изъятия элементов;

stack<T>& invert\_stack(); - функция, позволяющая «перевернуть» стек, изменив порядок элементов в нем на обратный (применяется, например, при выводе);

stack<T>& operator=(const stack<T>& other\_stack); - перегрузка оператора присваивания для стека;

friend ostream& operator << (ostream& stream, const stack<T> \_stack) – дополнительный инструмент, перегрузка операции потокового вывода << для стека, позволяющая вывести его при необходимости в удобном для человека виде с обозначением начала и конца (вершины). В качестве параметров принимает ссылку на поток вывода и константную ссылку на выводимый стек.

### Класс variables и структура var

Структура var объединяет в себе названия введенных переменных и их числовые значения. Класс variables объединяет в себе массив структур var для хранения переменных, некоторые его параметры (такие как количество переменных) и методы работы с переменными (поиск переменных в постфиксной строке и ввод их значений)

**Объявление структуры var:**

struct var

{

string var\_name; - имя переменной

double var\_value; - значение переменной

};

**Объявление класса variables:**

class variables

{

private:

int var\_used;

var\* array;

public:

variables(int \_var\_used = 10);

~variables();

int get\_var\_used();

var get\_var(int i);

void register\_variables(string postfix);

void enter\_var\_values();

friend class postfix;

};

**Члены класса variables:**

int var\_used; - счетчик использованных переменных, выступает в роли размера массива переменных при инициализации объекта класса;

var\* array; - массив структур типа var для хранения переменных;

**Методы класса variables:**

variables(int \_var\_used = 10); - конструктор с параметром, в качестве параметра принимает число используемых переменных и создающий массив структур соответствующего размера;

~variables();- деструктор класса variables, в конце работы программы освобождает память от массива переменных array

int get\_var\_used();- метод-геттер для получения значения var\_used со спецификатором доступа private;

var get\_var(int i); - метод-геттер для получения из массива переменных экземпляра структуры var под номером I (массив переменных хранится со спецификатором доступа private);

void register\_variables(string postfix); - метод, позволяющий занести в массив переменных array имена переменных, встреченные в переданной на вход строке postfix. Строка должна быть уже сконвертирована в постфиксную нотацию.

void enter\_var\_values();- метод, позволяющий пользователю ввести значения переменных после получения их имен методом register\_variables.

### Статический класс postfix

Статический класс, включает в себя методы, необходимые программе для перевода выражения из инфиксной нотации в постфиксную.

**Объявление статического класса postfix:**

class postfix

{

public:

static bool is\_operator(char c);

static bool is\_operator(string s);

static bool is\_l\_bracket(char c);

static bool is\_l\_bracket(string s);

static bool is\_r\_bracket(char c);

static bool is\_r\_bracket(string s);

static bool is\_number(char c);

static bool is\_number(string s);

static bool is\_letter(char c);

static bool is\_letter(string s);

static int check\_priority(string s);

static string recognize\_number(string s);

static string recognize\_variable(string s);

static bool check\_correct(string s);

static int check\_correct\_postfix(string s);

static string convert\_to\_postfix(string s);

static double calculate(string s, variables& handler);

};

**Члены класса postfix:**

Класс postfix не содержит членов.

**Методы класса Postfix:**

Группа методов is\_:

static bool is\_operator(char c);

static bool is\_operator(string s);

static bool is\_l\_bracket(char c);

static bool is\_l\_bracket(string s);

static bool is\_r\_bracket(char c);

static bool is\_r\_bracket(string s);

static bool is\_number(char c);

static bool is\_number(string s);

static bool is\_letter(char c);

static bool is\_letter(string s);

Каждый из методов перегружен для string и char, позволяет определить, относится ли очередной символ (char c для методов, принимающих char, и стартовый символ s[0] строки s для методов, принимающих string) к той или иной категории (скобки, операторы, числа, буквы). Используются в других методах при выборе дальнейшего алгоритма действий с очередным операндом или оператором, обнаруженным в строке.

static int check\_priority(string s); - метод определения  
приоритета оператора, предварительно обнаруженного в строке s на стартовой позиции [0].

static string recognize\_number(string s); - метод получения из строки последовательности символов, соответствующих числу.

static string recognize\_variable(string s); - метод получения из строки последовательности символов, соответствующих переменной.

static bool check\_correct(string s); - метод первичной проверки инфиксной записи на корректность ввода путем проверки равенства числа открывающих и закрывающих скобок.

static int check\_correct\_postfix(string s); - метод первичной проверки инфиксной записи на корректность ввода путем проверки равенства числа открывающих и закрывающих скобок.

static string convert\_to\_postfix(string s); - метод преобразования инфиксной строки в постфиксную. Принимает и возвращает строковый тип.

static double calculate(string s, variables& handler);

- метод вычисления арифметического выражения в постфиксной форме. Принимает строку в постфиксной форме и ссылку на объект класса variables, в котором хранятся сочетания имен и значений переменных. Возвращает результат вычислений – число типа double.

## Описание алгоритмов

В данной программе используется структура данных стек, реализованная на базе массива. Она работает по принципу Last In, First Out.

**Характеристики стека:**

1. максимальное количество элементов, которое может храниться в стеке
2. массив элементов;
3. вершина стека.

**Основные операции над стеком:**

1. вставка элемента;
2. изъятие элемента;
3. просмотр верхнего элемента;
4. проверка на пустоту;
5. проверка на полноту.

**Алгоритм преобразования арифметического выражения из инфиксной формы в постфиксную форму**:

Постфиксное выражение не содержит скобок. При этом каждый оператор в нем имеет свой приоритет.

Для реализации данного алгоритма применено 2 стека. Стек 1 – для хранения операций, стек 2 – для промежуточного хранения постфиксной формы и операндов.

*Входные данные***:** строка, содержащая арифметическое выражение в инфиксной форме.

*Алгоритм:*

1. Строка выражения просматривается слева направо. Каждый символ строки попадает либо в 1, либо во 2 стек.
2. Если пришёл операнд, то кладём его во 2 стек.
3. Если пришла операция с приоритетом 0:
   1. Если это «(» кладём её в стек 1
   2. Если это «)», то перекладываем все операции из стека 1 в стек 2, пока не встретим «(». «)» никуда не кладём, а «(» удаляем из стека 1.
4. Если пришла операция с другим приоритетом, то помещаем её в стек 1. Однако, перед этим выталкиваем любой из операторов, уже находящихся в стеке 1, если он имеет больший или равный приоритет, и кладём его на вершину стека 2.
5. Когда мы дошли до донца арифметического выражения, необходимо переместить все операции из стека 1 в стек 2.

*Выходные данные***:** строка, содержащая арифметическое выражение в постфиксной форме.

**Алгоритм вычисления значения арифметического выражения, записанного в постфиксной форме:**

*Входные данные***:** строка, содержащая арифметическое выражение в постфиксной форме.

*Алгоритм:*

1. Создаем пустой стек.
2. Просматриваем строчку постфиксной формы слева направо.
   1. Если встречается операнд, кладём его в стек.
   2. Если встречается операция, то производим изъятие двух верхних элементов из стека. Сначала берём второй операнд, затем – первый и выполняем арифметическую операцию над ними. Результат помещаем обратно в стек.
3. Когда мы завершим просмотр выражения, на верху стека будет лежать результат вычисления.

*Выходные данные***:** значение выражения, которое содержалось в строке.

В качестве примера рассмотрим выражение A B C D - \* +, где A = 10, B = 2, C = -5, D = 1 (Таблица 1).

|  |  |
| --- | --- |
| **Обрабатываемый символ** | **Стек** |
| A (= 3) | 10 |
| B (= 2) | 10 | 2 |
| C (= -5) | 10 | 2 | -5 |
| D (=1) | 10 | 2 | -5 | 1 |
| - | 10 | 2 | - 6 |
| \* | 10 | -12 |
| + | -2 |
|  | ***Результат вычисления = -2*** |

Таблица 1. Вычисление значения выражения «A B C D - \* +»

# Заключение

Разработано приложение «постфиксный калькулятор», позволяющее осуществлять преобразование выражений в постфиксную нотацию и вычисление этих выражений. В процессе разработки изучена структуры данных стек, основанная на массивах, и механизмы вычисления выражений в обратной польской нотации.

# Приложения

## Приложение 1. Функция main

#pragma once

#include <iostream>

#include <string>

#include "stack.h"

#include "postfix.h"

#include "variables.h"

using namespace std;

int main()

{

while (true)

{

string input, converted;

cout << "Enter expression: " << endl;

getline(std::cin, input); // считываем всё вместе с пробелами, до конца строки

try

{

converted = postfix::convert\_to\_postfix(input);

cout << "Postfix form: " << endl << converted << endl;

int var\_num = postfix::check\_correct\_postfix(converted);

variables handler(var\_num);

if (var\_num > 0)

{

handler.register\_variables(converted);

handler.enter\_var\_values();

}

double final\_value = postfix::calculate(converted, handler);

cout << "Result:" << endl << final\_value << endl;

}

catch (const char\* err\_code)

{

cout << err\_code << endl;

}

cout << "-----------------------------------------------" << endl;

}

return 0;

}

## Приложение 2. Класс stack

#pragma once

#include <iostream>

using namespace std;

template <typename T>

class stack

{

private:

T\* elems;

int size;

int top;

public:

stack(int max\_size = 10);

stack(const stack<T>& s);

~stack();

T pop(); // достать верхний элемент

T peek() const; // посмотреть верхний элемент

void push(T new\_el); // вставка

bool is\_full() const;

bool is\_empty() const;

stack<T>& invert\_stack();

stack<T>& operator=(const stack<T>& other\_stack);

friend ostream& operator << (ostream& stream, const stack<T> \_stack)

{

stack<T> tmp = \_stack;

tmp.invert\_stack();

stream << "bottom -> ";

while (!(tmp.is\_empty()))

stream << "'" << tmp.pop() << "' -> ";

stream << "top" << endl;

return stream;

};

};

// конструкторы и деструкторы

template <typename T>

stack<T>::stack(int max\_size)

{

size = max\_size;

top = 0;

elems = new T[size];

// cout << "stack created size: " << size << endl;

};

template <typename T>

stack<T>::stack(const stack<T>& s)

{

size = s.size;

top = s.top;

elems = new T[size];

for (int i = 0; i < size; i++)

elems[i] = s.elems[i];

};

template <typename T>

stack<T>::~stack()

{

size = 0;

top = 0;

delete[] elems; // освобождаем не только указатель, а весь массив

};

// методы

template <typename T>

void stack<T>::push(T new\_el)

{

if (this->is\_full())

throw "error: stack is already full";

top++;

elems[top] = new\_el;

};

template <typename T>

T stack<T>::pop()

{

if (this->is\_empty())

throw "error: stack is already empty";

T tmp = elems[top];

top--;

return tmp;

};

template <typename T>

T stack<T>::peek() const

{

if (this->is\_empty())

throw "error: stack is already empty";

return elems[top];

};

template <typename T>

bool stack<T>::is\_full() const

{

return (top == size);

};

template <typename T>

bool stack<T>::is\_empty() const

{

return (top == 0);

};

template<typename T>

stack<T>& stack<T>::operator=(const stack<T>& other\_stack)

{

size = other\_stack.size;

top = other\_stack.top;

delete[] elems;

elems = new T[size];

for (int i = 0; i < size; i++)

elems[i] = other\_stack.elems[i];

return \*this;

};

template<typename T>

stack<T>& stack<T>::invert\_stack()

{

if (this->is\_empty())

return \*this;

stack <T> shadow(\*this);

this->top = 0;

while (!shadow.is\_empty())

this->push(shadow.pop());

return \*this;

};

## Приложение 3. Класс postfix

*Postfix.h*

#pragma once

#include <string>

using namespace std;

class variables;

class postfix

{

public:

// методы для преобразования инфиксной формы в постфиксную:

static bool is\_operator(char c);

static bool is\_operator(string s);

static bool is\_l\_bracket(char c);

static bool is\_l\_bracket(string s);

static bool is\_r\_bracket(char c);

static bool is\_r\_bracket(string s);

static bool is\_number(char c);

static bool is\_number(string s);

static bool is\_letter(char c);

static bool is\_letter(string s);

static int check\_priority(string s);

// получение из строки переменных и чисел целиком:

static string recognize\_number(string s);

static string recognize\_variable(string s);

// проверки на правильность

static bool check\_correct(string s); // проверка строки на правильность расстановки скобок

static int check\_correct\_postfix(string s); // возвр. кол-во переменных

// основной функционал

static string convert\_to\_postfix(string s);

static double calculate(string s, variables& handler);

*Postfix.cpp*

#pragma once

#include "stack.h"

#include "variables.h"

#include "postfix.h"

// действия из алгоритма: проверка значения полученного из стека

bool postfix::is\_operator(char c) // получили знак?

{

return ((c == '+') || (c == '-') || (c == '\*') || (c == '/'));

};

bool postfix::is\_operator(string s) // получили оператор?

{

return is\_operator(s[0]);

};

bool postfix::is\_l\_bracket(char c) // получили откр. скобку?

{

return (c == '(');

};

bool postfix::is\_l\_bracket(string s) // получили откр. скобку?

{

return is\_l\_bracket(s[0]);

};

bool postfix::is\_r\_bracket(char c) // получили закр. скобку?

{

return (c == ')');

};

bool postfix::is\_r\_bracket(string s) // получили откр. скобку?

{

return is\_r\_bracket(s[0]);

};

bool postfix::is\_number(char c) // получили число?

{

return (isdigit(c));

};

bool postfix::is\_number(string s) // получили число?

{

return (isdigit(char(s[0])));

};

bool postfix::is\_letter(char c) // получили переменную?

{

return (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z');

}

bool postfix::is\_letter(string s) // получили переменную?

{

return (s[0] >= 'a' && s[0] <= 'z') || (s[0] >= 'A' && s[0] <= 'Z');

}

string postfix::recognize\_number(string s)

{

int len = s.length();

string number = "";

// достаем число

for (int i = 0; i < len; i++)

{

if (isdigit(s[i]) || s[i] == '.')

number += s[i];

else if (s[i] != ' ' && !is\_operator(s[i]) && !is\_r\_bracket(s[i]))

throw "error: number ends with unexpected symbols (probably variable name starting from digit)";

else

return (number);

}

return (number);

};

string postfix::recognize\_variable(string s)

{

string name = "";

for (int i = 0; i < int(s.length()); i++)

{

// собираем имя переменной

if ((is\_operator(s[i])) or (s[i] == ' ') or (is\_r\_bracket(s[i])))

break;

else if (is\_l\_bracket(s[i]))

throw "error: variable name contains odd brackets";

else if (!(is\_letter(s[i]) or is\_number(s[i])))

throw "error: variable name contains wrong symbols";

else

name += s[i];

}

return name;

}

// работа со значением полученным из стека

bool postfix::check\_correct(string s) // проверка на правильность скобок

{

int lefts = 0, rights = 0;

for (int i = 0; i < int(s.length()); i++)

{

if (s[i] == '(')

lefts++;

if (s[i] == ')')

rights++;

if (rights > lefts)

return false;

}

if (rights != lefts)

return false;

return true;

};

int postfix::check\_priority(string s)

{

char c = s[0];

if (c == '\*')

return 5;

if (c == '/')

return 4;

if (c == '+')

return 3;

if (c == '-')

return 2;

if (c == '(')

return 1;

else

return 0;

};

string postfix::convert\_to\_postfix(string s)

{

if (!check\_correct(s))

throw "incorrect brackets";

stack <string> operators(s.length() + 1);

stack <string> results(s.length() + 1);

bool before\_was\_operand = false;

while (true)

{

// Пришёл пробел: частый случай, просто убираем

if (s[0] == ' ')

{

s = s.substr(1);

}

else

// Пришел оператор: смотрим приоритет операции на вершине стека операторов

// Перекладываем все операции приоритетнее нашей в стек результатов

// Помещаем нашу операцию в стек операторов

if (is\_operator(s))

{

string found = "";

found += s[0];

if (!operators.is\_empty())

{

int curr\_prior = check\_priority(s);

while ((!operators.is\_empty()) && (check\_priority(operators.peek()) >= curr\_prior))

{

results.push(operators.pop());

};

}

operators.push(found);

s = s.substr(1);

before\_was\_operand = false; // за последний вызов (этот) мы нашли оператор

}

else

// Пришла открывающая скобка: помещаем в стек операторов

if (is\_l\_bracket(s))

{

string found = "";

found += s[0];

operators.push(found);

s = s.substr(1);

before\_was\_operand = false; // за последний вызов (этот) мы нашли скобку

}

else

// Пришла закрывающая скобка:

// Перемещаем все из стека операторов в стек результатов, пока не встретим "(", ее удаляем

if (is\_r\_bracket(s))

{

while (!(is\_l\_bracket(operators.peek())))

{

results.push(operators.pop());

};

string dummy = operators.pop();

s = s.substr(1);

before\_was\_operand = false; // за последний вызов (этот) мы нашли скобку

}

else

// Пришел операнд-число:

// Преобразуем из строки в число, помещаем в стек результатов

if (is\_number(s))

{

if (before\_was\_operand == true)

throw "wrong input: two operands without a sign between";

string num = recognize\_number(s);

results.push(num);

s = s.substr(num.length());

before\_was\_operand = true; // нашли операнд-число

}

else

// Пришел операнд - переменная:

if (is\_letter(s))

{

if (before\_was\_operand == true)

throw "wrong input: two operands without a sign between";

string x = recognize\_variable(s);

results.push(x);

s = s.substr(x.length());

before\_was\_operand = true; // нашли операнд-переменную

}

else

// Другие варианты:

// Плохой. Раз ничего не сработало - в строке мусор, выбросим

if (s != "")

{

s = s.substr(1);

}

else

// Хороший. Конец строки: перекладываем из стека в стек

{

while (!operators.is\_empty())

results.push(operators.pop());

break;

};

}

string result = "";

results.invert\_stack();

while (!results.is\_empty())

{

result += results.pop() + " ";

}

return result;

}

double postfix::calculate(string s, variables& handler)

// на вход - строка в постфиксной форме, пробелы допустимы

{

stack <double> num\_stack;

while (true)

{

// Пришёл пробел: просто убираем

if (s[0] == ' ')

{

s = s.substr(1);

}

else

// Пришел оператор: достаем из стека 2 последних значения, совершаем операцию, кладем в стек

if (is\_operator(s))

{

if (num\_stack.is\_empty())

throw "incorrect postfix string";

double op\_2 = num\_stack.pop();

if (num\_stack.is\_empty())

throw "incorrect postfix string";

double op\_1 = num\_stack.pop();

switch (s[0])

{

case '+':

{

num\_stack.push(op\_1 + op\_2);

break;

}

case '-':

{

num\_stack.push(op\_1 - op\_2);

break;

}

case '\*':

{

num\_stack.push(op\_1 \* op\_2);

break;

}

case '/':

{

if (op\_2 == 0)

throw "error: division by zero";

else

num\_stack.push(op\_1 / op\_2);

break;

}

}

s = s.substr(1);

}

else

// Пришел операнд-число:

// Преобразуем из строки в число, помещаем в стек

if (is\_number(s))

{

string num\_s = recognize\_number(s);

double num = stof(num\_s);

num\_stack.push(num);

s = s.substr(num\_s.length());

}

else

// Пришел операнд - переменная:

// Находим значение, помещаем в стек

if (is\_letter(s))

{

string x = recognize\_variable(s);

for (int i = 0; i < handler.get\_var\_used(); i++)

if (x == handler.array[i].var\_name)

{

num\_stack.push(handler.array[i].var\_value);

break;

}

else

if (i == handler.get\_var\_used())

throw "error: value for variable not found";

s = s.substr(x.length());

}

else

if (s != "")

{

throw "error: unexpected symbols";

break;

}

else break;

}

double final = num\_stack.pop();

if (!num\_stack.is\_empty())

throw "calculation error: odd values in stack";

return final;

};

int postfix::check\_correct\_postfix(string s)

{

int numbers = 0;

int operators = 0;

int variables = 0;

while (true)

{

// Пришёл пробел: убираем

if (s[0] == ' ')

{

s = s.substr(1);

}

else

// Пришел оператор: считаем

if (is\_operator(s))

{

operators += 1;

s = s.substr(1);

}

else

// Пришел операнд-число: считаем

if (is\_number(s))

{

string num = recognize\_number(s);

s = s.substr(num.length());

numbers += 1;

}

else

// Пришел операнд-переменная:

if (is\_letter(s))

{

string x = recognize\_variable(s);

s = s.substr(x.length());

variables += 1;

}

else

// Пришла скобка: ошибка

if (is\_l\_bracket(s) || is\_r\_bracket(s))

{

throw "incorrect postfix string: brackets found";

break;

}

else

// Ничего не сработало - мусор, ошибка

if (s != "")

{

throw "error: unexpected symbols";

break;

}

else break;

};

if (numbers + variables != operators + 1)

throw "error: incorrect number of operands & operators";

return variables;

};

## Приложение 4. Класс variables

*Variables.h*

#pragma once

#include <string>

#include <iostream>

using namespace std;

struct var

{

string var\_name;

double var\_value;

};

class variables

{

private:

int var\_used; // счетчик использованных переменных

var\* array; // массив структур "имя-значение"

public:

variables(int \_var\_used = 10);

~variables();

int get\_var\_used();

var get\_var(int i);

void register\_variables(string postfix);

void enter\_var\_values();

friend class postfix;

};

*Variables.cpp*

#pragma once

#include "stack.h"

#include "variables.h"

#include "postfix.h"

variables::variables(int \_var\_limit)

{

this->var\_used = 0; // счетчик использованных переменных

this->array = new var[\_var\_limit];

}

variables::~variables()

{

delete[] this->array;

}

int variables::get\_var\_used()

{

return this->var\_used;

}

var variables::get\_var(int i)

{

if ((i < 0) || (i > var\_used))

throw "attempt to access non-existing variable";

else

return this->array[i];

}

// сбор из строки имен переменных

void variables::register\_variables(string s)

{

while (true)

{

// пришел операнд-переменная:

if (postfix::is\_letter(s))

{

string x = postfix::recognize\_variable(s);

bool exists = false;

for (int i = 0; i < this->var\_used; i++)

{

if (x == this->array[i].var\_name)

{

exists = true;

break; // уже встречалась

}

}

if (exists == false)

{

this->array[var\_used].var\_name = x;

this->var\_used++;

};

s = s.substr(x.length());

}

else

// не интересный нам символ, убираем

if (s != "")

{

s = s.substr(1);

}

else break;

}

};

// ввод значений

void variables::enter\_var\_values()

{

for (int i = 0; i < this->var\_used; i++)

{

cout << "Enter value for '" << this->array[i].var\_name << "':" << endl;

this->array[i].var\_value = userproof\_input();

}

}

double userproof\_input()

{

double x;

while (true)

{

cin >> x;

if (!cin.good())

{

cout << "incorrect format, please retry" << endl;

cin.clear();

while (cin.get() != '\n');

}

else

{

while (cin.get() != '\n');

return x;

}

}

}