

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра САПР

ОТЧЕТ
по курсовой работе
по дисциплине «Алгоритмы и структуры данных»
Тема: «Преобразование алгебраических формул из инфиксной в
префиксную форму записи и вычисление значения выражения»
2 вариант

Студент гр. 9302

Кузнецов В.А.

Преподаватель

Тутуева А.В.

Санкт-Петербург

2020

Цель работы

Реализовать простейшую версию калькулятора. Который переводит выражение в префиксную форму, а затем решает.

Постановка задачи.

Необходимо реализовать простейшую версию калькулятора. Пользователю должен быть доступен ввод математического выражения, состоящего из чисел и арифметических знаков. Программа должна выполнить проверку корректности введенного выражения. В случае некорректного ввода необходимо вывести сообщение об ошибке с указанием позиции некорректного ввода. В противном выводится польская нотация введенного выражения, а также отображается результат вычисления.

Входные данные:

- арифметическое выражение
- поддерживаемый тип данных: вещественные числа (double)
- поддерживаемые знаки: +, -, *, /, ^, унарный "-", функции с одним аргументом (cos, sin, tg, ctg, ln, log, sqrt др. (хотя бы одну не из списка)), константы pi, e открывающая и закрывающая скобки

Выходные данные:

- префиксная ФЗ
- результат вычисления

Реализованные классы и функции

1. `template <class T> class Stack<T>` - класс реализующий стек:

Функции:

- 1.1 `void push(T elem)` – кладет элемент на стек.
 - 1.2 `T pop()` – удаляет последний добавленный элемент и возвращает его.
 - 1.3 `T top()` – возвращает последний добавленный элемент
 - 1.4 `int size()` – возвращает размер стека
 - 1.5 `Vector<T> toArray()` – возвращает стек в виде вектора
 - 1.6 `void resize()` – увеличивает максимальный размер вектора
 - 1.7 `bool isEmpty()` – возвращает пустой стек или нет
2. `template <class T> class Vector` – класс реализующий вектор

Функции:

2.1 `int length()` – возвращает размер вектора

2.2 `bool isEmpty()` - возвращает пустой вектор или нет

2.3 `void push_back(T elem)` – добавляет элемент в конец вектора

2.4 `T operator[](int i)` – перегрузка оператора, возвращает элемент по индексу

3. `class PrefixCalc` – класс реализующий перевод выражения в префиксную форму и вычисление результат

3.1 `void split()` – разделение введенной строки выражения на операторы и операнды

3.2 `bool isBinaryOperator(char ch)` – проверка символа на бинарный оператор

3.3 `bool isUnaryOperator(std::string el)` – проверка строки на унарный оператор

3.4 `bool isNumber(std::string el)` – проверка является ли строка числом

3.5 `double parseDouble(std::string number)` – перевод строки в число

3.6 `int binOpsPriority(std::string str)` – возвращает приоритет оператора

3.7 `std::string symbolToString(char ch)` – перевод символа в строку

3.8 `std::string prefixToString()` – возвращает выражение в префиксной форме в виде строки

3.9 `Vector<std::string> toPrefixForm()` – возвращает выражение в префиксной форме в виде вектора

3.10 `double doOperator(double a, std::string op)` – применяет унарный оператор к числу

3.11 `double doOperator(double a, double b, std::string op)` – применяет бинарный оператор к числу

3.12 `double calculate()` – подсчет значения выражения в префиксной форме

Код представлен в приложении А.

Описание алгоритма решения

Для перевода выражения в префиксную форму требуется создать два стека: первый является буфером, а второй – результатом перевода. Затем итерироваться по выражению с конца. Если встречается символ закрывающей скобки, добавить в буфер. Если встречается число или унарный оператор, то сразу добавить результирующий стек. Если встречается открывающая скобка, то доставать элементы из буфера и добавлять в результат, пока из буфера не будет удален символ открывающей скобки. Если был встречен бинарный оператор, то доставать из буфера операторы, пока не будет встречен оператор с большим приоритетом. Для вычисления значения выражения в префиксной форме, следует итерироваться по массиву выражения: все числа добавлять в стек, если встречается бинарный оператор, то применить его к двум верхним элементам стека и результат вернуть на стек, аналогично с унарным оператором.

Обоснование использования структур данных

Для обработки выражения и перевода его в префиксную форму удобнее использовать контейнер, для этого был написан свой вектор, который динамически увеличивается при заполнении. Для обработки выражения и вычисления значения используется стек.

Оценка временной сложности каждого метода

Функция	Сложность
<code>void push(T elem)</code>	$O(1)$
<code>T pop()</code>	$O(1)$
<code>T top()</code>	$O(1)$
<code>int size()</code>	$O(1)$
<code>Vector<T> toArray()</code>	$O(n)$
<code>void resize()</code>	$O(1)$
<code>bool isEmpty()</code>	$O(1)$
<code>int length()</code>	$O(1)$

<code>bool isEmpty()</code>	<code>O(1)</code>
<code>void push_back(T elem)</code>	<code>O(1)</code>
<code>T operator[](int i)</code>	<code>O(1)</code>
<code>void split()</code>	<code>O(n)</code>
<code>bool isBinaryOperator(char ch)</code>	<code>O(1)</code>
<code>bool isUnaryOperator(std::string el)</code>	<code>O(1)</code>
<code>bool isNumber(std::string el)</code>	<code>O(1)</code>
<code>double parseDouble(std::string number)</code>	<code>O(1)</code>
<code>int binOpsPriority(std::string str)</code>	<code>O(1)</code>
<code>std::string symbolToString(char ch)</code>	<code>O(1)</code>
<code>std::string prefixToString()</code>	<code>O(n)</code>
<code>Vector<std::string> toPrefixForm()</code>	<code>O(n)</code>
<code>double doOperator(double a, std::string op)</code>	<code>O(1)</code>
<code>double doOperator(double a, double b, std::string op)</code>	<code>O(1)</code>
<code>double calculate()</code>	<code>O(n)</code>

Описание реализованных unit-тестов

- 1) TestTranslation(1 - 4) – Тестирует функцию перевода в выражения в префиксную форму
- 2) TestCalculation(1 – 3) – Тестирую правильное вычисления значения выражения

Пример работы

На рис. 1-3 представлены примеры работы программы.

```
Введите выражение для перевода в префиксную форму и вычисления
ln(log(33.3+sqrt(22+555.332/2))+22^2+sqrt(574^4))
Выражение в префиксной форме:
ln + log + 33.3 sqrt + 22 / 555.332 2 + ^ 22 2 sqrt ^ 574 4
Результат:
12.7067
```

Рисунок 1- пример работы программы

```
Введите выражение для перевода в префиксную форму и вычисления
(-(22+11-(11+ln(16)+cos(13)))+22)
Выражение в префиксной форме:
+ -- + 22 - 11 + 11 + ln 16 cos 13 22
Результат:
3.68004
```

Рисунок 2- пример работы программы

```
Введите выражение для перевода в префиксную форму и вычисления
e+pi*22-(11.11+14.1/22.5)/77.4
Выражение в префиксной форме:
+ e - * pi 22 / + 11.11 / 14.1 22.5 77.4
Результат:
71.6817
```

Рисунок 3 - пример работы программы

Вывод

В ходе лабораторной работы был изучен метод перевод математического выражения в префиксную форму, а затем вычисление результата в префиксной форме. Составлена и протестирована программа по выполнению поставленной задачи.

Приложение А

Листинг программы

thermWork.cpp

```
#include <iostream>
#include <windows.h>
#include <string>
#include "Stack.h"
#include "PrefixCalc.h"
#include "Vector.h"

int main()
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    std::string expression = "";
    std::cout << "Введите выражение для перевода в префиксную форму и вычисления" <<
std::endl;
    std::cin >> expression;
    PrefixCalc calculator = PrefixCalc(expression);
    try {
        calculator.toPrefixForm();
        std::cout << "Выражение в префиксной форме: " << std::endl;
        std::cout << calculator.prefixToString() << std::endl;
        std::cout << "Результат:" << std::endl;
        std::cout << calculator.calculate() << std::endl;
    }
    catch (std::exception& e) {
        std::cout << e.what() << std::endl;
    }
}
```

prefixCalc.h

```
#pragma once
#include <iostream>
#include <string>
#include <cmath>
#include "Vector.h"
#include "Stack.h"

class PrefixCalc
{
private:
    std::string sourceExp;
    Vector<std::string> vec = Vector<std::string>();
    Vector<std::string> prefixFormExpVec = Vector<std::string>();

    void split();

    bool isBinaryOperator(char ch);

    bool isBinaryOperator(std::string ch);

    bool isUnaryOperator(std::string el);

    bool isNumber(std::string el);
}
```

```

double parseDouble(std::string number);

std::string symbolToString(char ch);

int binOpsPriority(std::string str);

double doOperator(double a, std::string op);

double doOperator(double a, double b, std::string op);

public:
    PrefixCalc(std::string sourceExp);

    std::string prefixToString();

    Vector<std::string> toPrefixForm();

    double calculate();
};

```

prefixCalc.cpp

```

#include "PrefixCalc.h"

void PrefixCalc::split() { //split inputed
string with expression
    std::string buffer = "";
    for (int i = 0; i < sourceExp.length(); i++)
    {
        if (isdigit(sourceExp[i])) { // if meet digit add to buffer
            if (buffer == "" || buffer[buffer.length() - 1] == '.')
                || isdigit(buffer[buffer.length() - 1])) {
                buffer += sourceExp[i];
            }
            else {
                throw std::runtime_error("Can't be digit on position " +
std::to_string(i));
            }
        }
        else if (sourceExp[i] == '.') { //if meet dot add
to buffer and check valid of buffer
            if (buffer.length() > 0 && std::isdigit(buffer[buffer.length() - 1])
                && std::count(buffer.begin(), buffer.end(), '.') < 1) {
                buffer += '.';
            }
            else {
                throw std::runtime_error("Can't be dot on position " +
std::to_string(i));
            }
        }
        else if (isalpha(sourceExp[i])) { //if meet alpha add to
buffer and check it
            if (buffer.length() == 0 || isalpha(buffer[buffer.length() - 1])) {
                buffer += sourceExp[i];
            }
            else {
                throw std::runtime_error("Can't be alpha symbol on position " +
std::to_string(i));
            }
        }
        else if (isBinaryOperator(sourceExp[i])) { //if meet binary
operator

```



```

        if (isNumber(buffer)) { //check if in buffer is
number add it to result vector and clear
            vec.push_back(buffer);
            buffer = "";
            vec.push_back(symbolToString(sourceExp[i]));
        }
        //if met symbol determine if it unary operaotor
        else if (sourceExp[i] == '-' && (i == 0 || sourceExp[i - 1] == '(')) {
            vec.push_back("--");
        }
        else {
            if (sourceExp[i - 1] == ')') { //binary operator can
be afer number and close bracket
                vec.push_back(symbolToString(sourceExp[i]));
            }
            else {
                throw std::runtime_error("Symbol on index: " +
std::to_string(i - 1) + "can't be before binary operator");
            }
        }
    }

    else if (sourceExp[i] == '(') { //if met open
bracket
        if (isNumber(buffer)) { //no numbers before
            throw std::runtime_error("Can't be number before open bracket at
index: " + std::to_string(i));
        }
        if (buffer.length() > 0) { //before can only be
operators
            if (isUnaryOperator(buffer)) {
                vec.push_back(buffer);
            }
            else {
                throw std::runtime_error("No such operator: " +
std::to_string(i - buffer.length()));
            }
        }
        buffer = "";
        vec.push_back("(");
    }

    else if (sourceExp[i] == ')') { //before close bracket can be
only numbers
        if (!isNumber(buffer) && sourceExp[i - 1] != ')') {
            throw std::runtime_error("Can't be operators before close bracket
at index: " + std::to_string(i));
        }
        if (buffer.length() > 0) {
            vec.push_back(buffer);
        }
        buffer = "";
        vec.push_back(")");
    }
}
//add last element of expression
if (buffer.length() > 0) {
    if (isUnaryOperator(buffer) || isNumber(buffer) || isBinaryOperator(buffer)) {
//add element at
        vec.push_back(buffer);
    }
    else {
        throw std::runtime_error("Error with last element of express");
    }
}

```

```

    }
}

bool PrefixCalc::isBinaryOperator(char ch) {
    if (ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^') {
        return true;
    }
    else return false;
}

bool PrefixCalc::isBinaryOperator(std::string ch) {
    if (ch == "+" || ch == "-" || ch == "*" || ch == "/" || ch == "^") {
        return true;
    }
    else return false;
}

bool PrefixCalc::isUnaryOperator(std::string el) {
    if (el == "sin" || el == "cos" || el == "tg" ||
        el == "ctg" || el == "arccos" || el == "arcsin" ||
        el == "arctg" || el == "arcctg" ||
        el == "sqrt" || el == "ln" || el == "log" || el == "--") {
        return true;
    }
    return false;
}

bool PrefixCalc::isNumber(std::string el) {
    if (el == "pi" || el == "e") { //strings of constants is numbers too
        return true;
    }
    try
    {
        double value = std::stod(el);
        return true;
    }
    catch (std::exception& e)
    {
        return false;
    }
}

double PrefixCalc::parseDouble(std::string number) { //parse numbers and constants
    if (number == "e") {
        return 2.71828182845904523536;
    }
    else if (number == "pi") {
        return 3.14159265358979323846;
    }
    else {
        return std::stod(number);
    }
}

std::string PrefixCalc::symbolToString(char ch) {
    std::string res = "";
    res += ch;
    return res;
}

int PrefixCalc::binOpsPriority(std::string str) {
    if (str == "-" || str == "+") {
        return 1;
    }
}

```

```

    }
    else if (str == "*" || str == "/") {
        return 2;
    }
    else if (str == "^") {
        return 3;
    }
    return 0;
}

double PrefixCalc::doOperator(double a, std::string op) {
    if (op == "sin") {
        return sin(a);
    }
    if (op == "cos") {
        return cos(a);
    }
    if (op == "tg") {
        return tan(a);
    }
    if (op == "ctg") {
        return 1.0 / tan(a);
    }
    if (op == "arcsin") {
        return asin(a);
    }
    if (op == "arccos") {
        return acos(a);
    }
    if (op == "arctg") {
        return atan(a);
    }
    if (op == "arcctg") {
        return atan(1 / a);
    }
    if (op == "sqrt") {
        return pow(a, 0.5);
    }
    if (op == "ln") {
        return log(a);
    }
    if (op == "log") {
        return log10(a);
    }
    if (op == "--") {
        return -a;
    }
    throw std::runtime_error("No such operator: " + op);
}

double PrefixCalc::doOperator(double a, double b, std::string op) {
    if (op == "+") {
        return a + b;
    }
    if (op == "-") {
        return a - b;
    }
    if (op == "*") {
        return a * b;
    }
    if (op == "/") {
        return a / b;
    }
}

```

```

        if (op == "^") {
            return pow(a, b);
        }
        throw std::runtime_error("No such operator: " + op);
    }

    PrefixCalc::PrefixCalc(std::string sourceExp) {
        this->sourceExp = sourceExp;
    }

    std::string PrefixCalc::prefixToString() {                                     //join vector by
    " "
        std::string str = "";
        for (int i = 0; i < prefixFormExpVec.length() - 1; i++)
        {
            str += prefixFormExpVec[i] + " ";
        }
        str += prefixFormExpVec[prefixFormExpVec.length() - 1];
        return str;
    }

    Vector<std::string> PrefixCalc::toPrefixForm() {
        split();
        Stack<std::string> binOps = Stack<std::string>();
        Stack<std::string> expInPrefixForm = Stack<std::string>();
        for (int i = vec.length() - 1; i >= 0; i--)
        {
            if (vec[i] == ")") {
                binOps.push("(");
            }
            else if (isNumber(vec[i]) || isUnaryOperator(vec[i])) {
                expInPrefixForm.push(vec[i]);
            }

            else if (vec[i] == "(") {
                while (binOps.top() != ")") {
                    try {
                        expInPrefixForm.push(binOps.pop());
                    }
                    catch (const std::runtime_error& error) {
                        throw std::runtime_error("No close bracket to open bracket
at index: " + std::to_string(i));
                    }
                }
                binOps.pop();
            }

            else if (isBinaryOperator(vec[i])) {
                while (!binOps.isEmpty() && this->binOpsPriority(vec[i]) <= this-
>binOpsPriority(binOps.top())) {
                    expInPrefixForm.push(binOps.pop());
                }
                binOps.push(vec[i]);
            }
        }

        while (!binOps.isEmpty()) {
            expInPrefixForm.push(binOps.pop());
            if (vec[vec.length() - 1] == "(") {
                throw std::runtime_error("No open bracket");
            }
        }
        prefixFormExpVec = expInPrefixForm.toArray();
    }

```

```

        return prefixFormExpVec;
    }

    double PrefixCalc::calculate() {
        //calculating
        Stack<double> numbers = Stack<double>();
        for (int i = prefixFormExpVec.length() - 1; i >= 0; i--)    //iterating from end to
start of prefix form of expression
        {
            if (isNumber(prefixFormExpVec[i])) {                //numbers add to stack
                numbers.push(parseDouble(prefixFormExpVec[i]));
            }
            else {
                //do operators
                if (isUnaryOperator(prefixFormExpVec[i]) && numbers.size() > 0) {
                    double first = numbers.pop();
                    numbers.push(doOperator(first, prefixFormExpVec[i]));
                }
                else if (isBinaryOperator(prefixFormExpVec[i]) && numbers.size() > 1) {
                    double first = numbers.pop();
                    double second = numbers.pop();
                    numbers.push(doOperator(first, second, prefixFormExpVec[i]));
                }
            }
        }
        return numbers.top();
    }
}

```

Stack.h

```

#pragma once
#include <iostream>
#include "Vector.h"

template <class T>
class Stack
{
private:
    int head;
    int curMaxSize;
    T* arr;

    void resize() {
        //resizing dynamic array using memcpy
        if (this->head + 2 >= this->curMaxSize) {
            int newSize = this->curMaxSize * 2;
            T* newArr = new T[newSize];
            memcpy(newArr, arr, this->curMaxSize * sizeof(T));
            arr = newArr;
            curMaxSize = newSize;
        }
    };

public:
    Stack() {
        this->curMaxSize = 10;
        this->arr = new T[this->curMaxSize];
        this->head = -1;
    };

    bool isEmpty() {
        return head < 0;
    }
};

```

```

};

void push(T elem) {                                //before pushing resize array if it need
    resize();
    arr[++head] = elem;
};

T pop() {
    if (!isEmpty()) {
        return arr[head--];
    }
    else {
        throw std::runtime_error("Empty stack");
    }
}

T top() {
    return arr[head];
}

int size() {
    return head + 1;
}

Vector<T> toArray() {                               //add all
    Vector<T> resVec = Vector<T>();
    for (int i = head; i >= 0; i--)
    {
        resVec.push_back(arr[i]);
    }
    return resVec;
}
};

```

Vector.h

```

#pragma once
#include <iostream>

template <class T>
class Vector
{
private:
    int last;
    int size;
    T* arr;

    void resize() {
        //resizing dynamic array using memcpy
        if (this->last + 2 >= this->size) {
            int newSize = this->size * 2;
            T* newArr = new T[newSize];
            memcpy(newArr, arr, this->size * sizeof(T));
            arr = newArr;
            size = newSize;
        }
    };

public:
    Vector() {
        this->size = 10;
        this->arr = new T[this->size];
        this->last = -1;
    }
};

```

```

};

int length() {
    return this->last + 1;
};

bool isEmpty() {
    return last < 0;
};

void push_back(T elem) {
    resize();
    arr[++last] = elem;
};

T operator[](int i) {
    return arr[i];
}
};

```

Tests.cpp

```

#include "pch.h"
#include "CppUnitTest.h"
#include "../thermWork/PrefixCalc.h"
#include "../thermWork/PrefixCalc.cpp"
#include "../thermWork/Vector.h"
#include "../thermWork/Stack.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace Tests
{
    TEST_CLASS(Tests)
    {
    public:
        TEST_METHOD(TestTranslation1)
        {
            PrefixCalc calc =
PrefixCalc("ln(log(33.3+sqrt(22+555.332/2))+22^2+sqrt(574^4))");
            calc.toPrefixForm();
            char tab[1024];
            strcpy_s(tab, calc.prefixToString().c_str());
            Assert::AreEqual("ln + log + 33.3 sqrt + 22 / 555.332 2 + ^ 22 2 sqrt ^
574 4", tab);
        }
        TEST_METHOD(TestTranslation2)
        {
            PrefixCalc calc = PrefixCalc("(-(22+11-(11+ln(16)+cos(13)))+22)");
            calc.toPrefixForm();
            char tab[1024];
            strcpy_s(tab, calc.prefixToString().c_str());
            Assert::AreEqual("+ -- + 22 - 11 + 11 + ln 16 cos 13 22", tab);
        }
        TEST_METHOD(TestTranslation3)
        {
            PrefixCalc calc = PrefixCalc("e+pi*22-(11.11+14.1/22.5)/77.4");
            calc.toPrefixForm();
            char tab[1024];
            strcpy_s(tab, calc.prefixToString().c_str());
            Assert::AreEqual("+ e - * pi 22 / + 11.11 / 14.1 22.5 77.4", tab);
        }
        TEST_METHOD(TestTranslation4)

```

```

    {
        PrefixCalc calc = PrefixCalc("ln(sqrt((log(e))))");
        calc.toPrefixForm();
        char tab[1024];
        strcpy_s(tab, calc.prefixToString().c_str());
        Assert::AreEqual("ln sqrt log e", tab);
    }
    TEST_METHOD(TestCalculation1)
    {
        PrefixCalc calc =
PrefixCalc("ln(log(33.3+sqrt(22+555.332/2))+22^2+sqrt(574^4))");
        calc.toPrefixForm();
        double number = 12.0;
        Assert::AreEqual(number, floor(calc.calculate()));
    }
    TEST_METHOD(TestCalculation2)
    {
        PrefixCalc calc = PrefixCalc("(-(22+11-(11+ln(16)+cos(13)))+22)");
        calc.toPrefixForm();
        double number = 3.0;
        Assert::AreEqual(number, floor(calc.calculate()));
    }
    TEST_METHOD(TestCalculation3)
    {
        PrefixCalc calc = PrefixCalc("e+pi*22-(11.11+14.1/22.5)/77.4");
        calc.toPrefixForm();
        double number = 71.0;
        Assert::AreEqual(number, floor(calc.calculate()));
    }
};
}

```