**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 постфиксную форму записи и вычисление значения выражения»**

**Вариант №1**

|  |  |  |
| --- | --- | --- |
| Студент гр. 0302 |  | Приезжих Т.А. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

**2021**

### 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 обратная 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 и др. (хотя бы одну не из списка)), константы pi, e открывающая и закрывающая скобки

Выходные данные:

● постфиксная ФЗ

● результат вычисления

### Описание алгоритма

После ввода число переводится из инфиксной записи в постфиксную. Затем в постфиксной записи проходят вычисления (числа переходят в стек, операторы обрабатывают 1-2 числа в стеке (зависит от оператора) и выводят результат в стек). После всех вычислений программа выводит постфиксную запись и рассчитанное значение выражения.

### Пример работы

### 

### Листинг

main.cpp

#include "stack.cpp"

using namespace std;

double \_plus(Stack<double>& Stack){

if (Stack.get\_size() > 1){

double temp1 = Stack.get\_head();

Stack.pop\_front();

double temp2 = Stack.get\_head();

Stack.pop\_front();

Stack.push(temp1 + temp2);

return temp1 + temp2;

}

else if (Stack.get\_size()) return Stack.get\_head();

else throw exception("Invalid expression!");

}

double \_minus(Stack<double>& Stack){

if (Stack.get\_size() > 1){

double temp1 = Stack.get\_head();

Stack.pop\_front();

double temp2 = Stack.get\_head();

Stack.pop\_front();

Stack.push(temp2 - temp1);

return temp2 - temp1;

}

else if (Stack.get\_size()){

double temp1 = Stack.get\_head();

Stack.pop\_front();

Stack.push(-temp1);

return Stack.get\_head();

}

else throw exception("Invalid expression!");

}

double \_multiplication(Stack<double>& Stack){

if (Stack.get\_size() > 1){

double temp1 = Stack.get\_head();

Stack.pop\_front();

double temp2 = Stack.get\_head();

Stack.pop\_front();

Stack.push(temp2 \* temp1);

return temp2 \* temp1;

}

else throw exception("Invalid expression!");

}

double \_division(Stack<double>& Stack){

if (Stack.get\_size() > 1){

double temp1 = Stack.get\_head();

Stack.pop\_front();

double temp2 = Stack.get\_head();

Stack.pop\_front();

if (temp1 == 0) throw exception("Divide by zero is always a bad idea (error, obviosly)");

else {

Stack.push(temp2 / temp1);

return temp2 / temp1;

}

}

else throw exception("Invalid expression!");

}

double \_sqrt(Stack<double>& Stack) {

if (Stack.get\_size()) {

double temp1 = Stack.get\_head();

if (temp1 < 0)

throw exception("Negative numbers don't have square roots!");

Stack.pop\_front();

Stack.push(sqrt(temp1));

return sqrt(temp1);

}

else throw exception("Invalid expression!");

}

double \_exponentiation(Stack<double>& Stack){

if (Stack.get\_size() > 1){

double temp1 = Stack.get\_head();

Stack.pop\_front();

double temp2 = Stack.get\_head();

Stack.pop\_front();

Stack.push(pow(temp2, temp1));

return pow(temp2, temp1);

}

else throw exception("Invalid expression!");

}

double \_log(Stack<double>& Stack){

if (Stack.get\_size() > 1){

double temp1 = Stack.get\_head();

Stack.pop\_front();

double temp2 = Stack.get\_head();

Stack.pop\_front();

if (temp1 < 0 || temp2 < 0)

throw exception("Logarithm's argument and base must be positive!");

if (temp2 == 1)

throw exception("Logarithm's base must not be equal to one!");

Stack.push(log(temp1) / log(temp2));

return log(temp1) / log(temp2);

}

else throw exception("Invalid expression!");

}

double \_sin(Stack<double>& Stack){

if (Stack.get\_size()){

double temp1 = Stack.get\_head();

if (temp1 == 3.14159 || temp1 == 3.14159 \* 2){

Stack.pop\_front();

Stack.push(0);

return 0;

}

Stack.pop\_front();

Stack.push(sin(temp1));

return sin(temp1);

}

else throw exception("Invalid expression!");

}

double \_cos(Stack<double>& Stack){

if (Stack.get\_size()){

double temp1 = Stack.get\_head();

if (temp1 == 3.14159 / 2 || temp1 == 3.14159 / 2 + 3.14159){

Stack.pop\_front();

Stack.push(0);

return 0;

}

Stack.pop\_front();

Stack.push(cos(temp1));

return cos(temp1);

}

else throw exception("Invalid expression!");

}

double \_tg(Stack<double>& Stack){

if (Stack.get\_size()){

double temp1 = Stack.get\_head();

if (temp1 == 2 \* 3.14159){

Stack.pop\_front();

Stack.push(0);

return 0;

}

if (temp1 == 3.14159 / 2)

throw exception("tg(pi/2) does not exist!");

Stack.pop\_front();

Stack.push(tan(temp1));

return tan(temp1);

}

else throw exception("Invalid expression!");

}

double \_ctg(Stack<double>& Stack){

if (Stack.get\_size()){

double temp1 = Stack.get\_head();

if (temp1 == 0)

throw exception("Divide by zero is always a bad idea (error, obviosly)");

Stack.pop\_front();

Stack.push(1 / tan(temp1));

return 1 / tan(temp1);

}

else throw exception("Invalid expression!");

}

double \_ln(Stack<double>& Stack){

if (Stack.get\_size()){

double temp1 = Stack.get\_head();

if (temp1 < 0)

throw exception("Logarithm's argument must be positive!");

Stack.pop\_front();

Stack.push(log(temp1));

return log(temp1);

}

else throw exception("Invalid expression!");

}

double \_max(Stack<double>& Stack) {

if (Stack.get\_size() > 1) {

double temp1 = Stack.get\_head();

Stack.pop\_front();

double temp2 = Stack.get\_head();

Stack.pop\_front();

if (temp1 >= temp2) {

Stack.push(temp1);

return temp1;

}

else {

Stack.push(temp2);

return temp2;

}

}

else throw exception("Invalid expression!");

}

bool IsOperator(string inp){

string variations[] = { "-","+","\*","/","^","sin","cos","tg","ln","ctg","sqrt","log","max","min" };

for (int i = 0; i < 14; i++)

if (inp == variations[i])

return true;

return false;

}

bool IsNumber(char letter){

if ((letter >= '0' && letter <= '9') || letter == '.')

return true;

else return false;

}

bool IsNumber(string letter){

if ((letter >= "0" && letter <= "9") || letter == "." || letter == "pi" || letter == "e")

return true;

else return false;

}

int Precedence(string letter){

if (letter == "^" || letter == "sin" || letter == "cos" || letter == "tg" || letter == "ln" || letter == "ctg"

|| letter == "sqrt" || letter == "log" || letter == "max" || letter == "min")

return 3;

else if (letter == "\*" || letter == "/")

return 2;

else if (letter == "+" || letter == "-")

return 1;

else return -1;

}

string InfToPost(string infix){

Stack<string> Stack;

string postfix, temp;

unsigned u = 0;

bool flag = 0;

for (unsigned i = 0; i < infix.length(); i++){

if (infix[i] == ' ')

continue;

u = 0;

temp = "";

flag = 0;

while (u < 2){

temp += infix[i + u];

if (IsNumber(temp)){

flag = 1;

break;

}

u++;

}

if (flag){

if (temp != "pi" && temp != "e"){

do{

postfix += infix[i];

i++;

}

while ((IsNumber(infix[i])) && i < infix.length());

postfix += ' ';

i--;

}

else{

postfix += temp;

postfix += ' ';

i += u;

}

}

else if (infix[i] == '(')

Stack.push(string(1, infix[i]));

else if (infix[i] == ')'){

while (Stack.get\_head() != "(" && (Stack.get\_size())){

postfix += Stack.get\_head();

postfix += ' ';

Stack.pop\_front();

}

Stack.pop\_front();

}

else

{

u = 0;

temp = "";

flag = 0;

while (u < 4){

temp += infix[i + u];

if (IsOperator(temp))

{

flag = 1;

break;

}

u++;

}

if (flag){

if (!Stack.get\_size())

Stack.push(temp);

else{

if (Precedence(temp) > Precedence(Stack.get\_head()))

Stack.push(temp);

else if ((Precedence(temp) == Precedence(Stack.get\_head())) && (temp == "^" || temp == "sin" ||

temp == "cos" || temp == "tg" || temp == "ln" || temp == "ctg" || temp == "sqrt" || temp == "log" ||

temp == "max" || temp == "min"))

Stack.push(temp);

else{

while ((Stack.get\_size()) && (Precedence(temp)) <= Precedence(Stack.get\_head())){

postfix += Stack.get\_head();

postfix += ' ';

Stack.pop\_front();

}

Stack.push(temp);

}

}

flag = 0;

i += u;

}

else throw exception("Invalid expression!", i);

}

}

while (Stack.get\_size()){

postfix += Stack.get\_head();

postfix += ' ';

Stack.pop\_front();

}

return postfix;

}

double CalculatePostfix(string postfix){

bool flag = 0;

Stack<double> Stack;

string temp = "";

for (unsigned i = 0; i < postfix.length(); i++){

temp = "";

flag = 0;

while (postfix[i] != ' '){

temp += postfix[i];

if (IsNumber(temp))

flag = 1;

i++;

}

if (flag){

if (temp != "pi" && temp != "e")

Stack.push(atof(temp.c\_str()));

else if (temp == "pi")

Stack.push(3.14159);

else if (temp == "e")

Stack.push(2.71828);

flag = 0;

temp = "";

}

else if (IsOperator(temp))

{

if (temp == "+")

\_plus(Stack);

else if (temp == "-")

\_minus(Stack);

else if (temp == "\*")

\_multiplication(Stack);

else if (temp == "/")

\_division(Stack);

else if (temp == "^")

\_exponentiation(Stack);

else if (temp == "log")

\_log(Stack);

else if (temp == "max")

\_max(Stack);

else if (temp == "sin")

\_sin(Stack);

else if (temp == "cos")

\_cos(Stack);

else if (temp == "tg")

\_tg(Stack);

else if (temp == "ctg")

\_ctg(Stack);

else if (temp == "ln")

\_ln(Stack);

else if (temp == "sqrt")

\_sqrt(Stack);

}

}

double solution = Stack.get\_head();

Stack.pop\_front();

return solution;

}

int main()

{

try {

string infix, postfix;

cout << "Enter an expression you want to solve:" << "\n\n";

getline(cin, infix);

int check = 0;

for (unsigned i = 0; i < infix.length(); i++)

{

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

check++;

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

check--;

if (check < 0)

throw exception("Invalid expression!");

}

if (check)

throw exception("Invalid expression!");

postfix = InfToPost(infix);

cout << "\n" << "Postfix expression: " << "\n\n" << postfix;

double solution = CalculatePostfix(postfix);

cout << "\n\n" << "Answer :\n\n" << solution << "\n";

}

catch (exception ex) { cout << "\n\n" << ex.what() << "\n"; }

\_getch();

}

stack.cpp

#pragma once

#include <string>

#include <iostream>

#include <conio.h>

using namespace std;

template <class T>

class Stack

{

struct Node

{

Node\* pNext;

T data;

Node(T data = T(), Node\* pNext = nullptr){

this->data = data;

this->pNext = pNext;

}

};

int size;

Node\* head;

public:

Stack();

~Stack();

void push(T data);

T get\_head();

void pop\_front();

int get\_size() { return size; };

void clear();

friend std::ostream& operator<<(std::ostream& s, Stack& list)

{

unsigned count = 1;

Node\* cur = list.head;

while (count <= list.size) {

s << cur->data << '\t';

cur = cur->pNext;

count++;

}

return s;

}

};

template <class T>

Stack<T>::Stack(){

size = 0;

head = nullptr;

}

template<class T>

Stack<T>::~Stack(){

clear();

}

template <class T>

void Stack<T>::push(T data){

head = new Node(data, head);

size++;

}

template<class T>

T Stack<T>::get\_head(){

T temp = head->data;

return temp;

}

template<class T>

void Stack<T>::pop\_front(){

if (this->size){

Node\* temp = head;

head = head->pNext;

delete temp;

size--;

}

}

template<class T>

void Stack<T>::clear(){

while (size)

pop\_front();

}

UnitTestKurs.cpp

#include "pch.h"

#include "CppUnitTest.h"

#include "..\Kurs\main.cpp"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTestKurs

{

TEST\_CLASS(UnitTestKurs)

{

public:

TEST\_METHOD(TestIsOperator)

{

string str = "+";

Assert::IsTrue(IsOperator(str));

str = "-";

Assert::IsTrue(IsOperator(str));

str = "\*";

Assert::IsTrue(IsOperator(str));

str = "/";

Assert::IsTrue(IsOperator(str));

str = "^";

Assert::IsTrue(IsOperator(str));

str = "sin";

Assert::IsTrue(IsOperator(str));

str = "cos";

Assert::IsTrue(IsOperator(str));

str = "tg";

Assert::IsTrue(IsOperator(str));

str = "ctg";

Assert::IsTrue(IsOperator(str));

str = "log";

Assert::IsTrue(IsOperator(str));

str = "ln";

Assert::IsTrue(IsOperator(str));

str = "min";

Assert::IsTrue(IsOperator(str));

str = "max";

Assert::IsTrue(IsOperator(str));

str = "sqrt";

Assert::IsTrue(IsOperator(str));

str = "ss";

Assert::IsTrue(!IsOperator(str));

str = "";

Assert::IsTrue(!IsOperator(str));

}

TEST\_METHOD(TestIsNumber)

{

string str = "1";

Assert::IsTrue(IsNumber(str));

str = "1.23";

Assert::IsTrue(IsNumber(str));

str = "0.12";

Assert::IsTrue(IsNumber(str));

str = "pi";

Assert::IsTrue(IsNumber(str));

str = "e";

Assert::IsTrue(IsNumber(str));

str = "sin";

Assert::IsTrue(!IsNumber(str));

str = "cos";

Assert::IsTrue(!IsNumber(str));

}

TEST\_METHOD(TestPlusMinus)

{

Stack<double> Stack;

Stack.push(2);

Assert::IsTrue(\_plus(Stack) == 2);

Stack.push(3);

Assert::IsTrue(\_plus(Stack) == 5);

Stack.push(5);

Assert::IsTrue(\_minus(Stack) == 0);

Stack.push(10);

Assert::IsTrue(\_minus(Stack) == -10);

}

TEST\_METHOD(TestMultiplicationDivision)

{

Stack<double> Stack;

Stack.push(5);

Stack.push(4);

Assert::IsTrue(\_multiplication(Stack) == 20);

try {

\_multiplication(Stack);

}

catch (exception ex) { Assert::IsTrue(ex.what()); }

Stack.push(10);

Assert::IsTrue(\_division(Stack) == 2);

try {

\_division(Stack);

}

catch (exception ex) { Assert::IsTrue(ex.what()); }

Stack.push(0);

try {

\_division(Stack);

}

catch (exception ex) { Assert::IsTrue(ex.what()); }

}

TEST\_METHOD(TestSqrt)

{

Stack<double> Stack;

Stack.push(64);

Assert::IsTrue(\_sqrt(Stack) == 8);

Stack.pop\_front();

try {

\_sqrt(Stack);

}

catch (exception ex) { Assert::IsTrue(ex.what()); }

try {

Stack.push(-1);

\_sqrt(Stack);

}

catch (exception ex) { Assert::IsTrue(ex.what()); }

}

TEST\_METHOD(TestExponenta)

{

Stack<double> Stack;

Stack.push(2);

try {

\_exponentiation(Stack);

}

catch (exception ex) { Assert::IsTrue(ex.what()); }

Stack.push(2);

Assert::IsTrue(\_exponentiation(Stack) == 4);

}

TEST\_METHOD(TestLogarithm)

{

Stack<double> Stack;

Stack.push(2);

try {

\_log(Stack);

}

catch (exception ex) { Assert::IsTrue(ex.what()); }

Stack.push(16);

Assert::IsTrue(\_log(Stack) == 4);

}

TEST\_METHOD(TestSin)

{

Stack<double> Stack;

try {

\_sin(Stack);

}

catch (exception ex) { Assert::IsTrue(ex.what()); }

Stack.push(0);

Assert::IsTrue(\_sin(Stack) == 0);

}

TEST\_METHOD(TestCos)

{

Stack<double> Stack;

try {

\_cos(Stack);

}

catch (exception ex) { Assert::IsTrue(ex.what()); }

Stack.push(0);

Assert::IsTrue(\_cos(Stack) == 1);

}

TEST\_METHOD(TestTg)

{

Stack<double> Stack;

try {

\_tg(Stack);

}

catch (exception ex) { Assert::IsTrue(ex.what()); }

Stack.push(0);

Assert::IsTrue(\_tg(Stack) == 0);

}

TEST\_METHOD(TestCtg)

{

Stack<double> Stack;

try {

\_ctg(Stack);

}

catch (exception ex) { Assert::IsTrue(ex.what()); }

try {

Stack.push(0);

\_ctg(Stack);

}

catch (exception ex) { Assert::IsTrue(ex.what()); }

}

TEST\_METHOD(TestLn)

{

Stack<double> Stack;

try {

\_ln(Stack);

}

catch (exception ex) { Assert::IsTrue(ex.what()); }

Stack.push(1);

Assert::IsTrue(\_ln(Stack) == 0);

}

TEST\_METHOD(TestMax)

{

Stack<double> Stack;

Stack.push(2);

try {

\_max(Stack);

}

catch (exception ex) { Assert::IsTrue(ex.what()); }

Stack.push(4);

Assert::IsTrue(\_max(Stack) == 4);

}

TEST\_METHOD(TestPrecedence)

{

string str = "+";

Assert::IsTrue(Precedence(str) == 1);

str = "-";

Assert::IsTrue(Precedence(str) == 1);

str = "\*";

Assert::IsTrue(Precedence(str) == 2);

str = "/";

Assert::IsTrue(Precedence(str) == 2);

str = "sin";

Assert::IsTrue(Precedence(str) == 3);

str = "cos";

Assert::IsTrue(Precedence(str) == 3);

str = "tg";

Assert::IsTrue(Precedence(str) == 3);

str = "ctg";

Assert::IsTrue(Precedence(str) == 3);

str = "^";

Assert::IsTrue(Precedence(str) == 3);

str = "log";

Assert::IsTrue(Precedence(str) == 3);

str = "sqrt";

Assert::IsTrue(Precedence(str) == 3);

str = "ln";

Assert::IsTrue(Precedence(str) == 3);

str = "min";

Assert::IsTrue(Precedence(str) == 3);

str = "max";

Assert::IsTrue(Precedence(str) == 3);

}

TEST\_METHOD(TestInfToPost)

{

string str = "2+2";

Assert::IsTrue(InfToPost(str) == "2 2 + ");

str = "(6\*e-pi)/((pi^e-53)\*3)";

Assert::IsTrue(InfToPost(str) == "6 e \* pi - pi e ^ 53 - 3 \* / ");

}

TEST\_METHOD(TestCalculatePostfix)

{

string str = "2 2 + ";

Assert::IsTrue(CalculatePostfix(str) == 4);

str = "55 100 - 50 / ";

Assert::IsTrue(CalculatePostfix(str) == -0.9);

str = "16 sqrt 0 cos + 2 \* ";

Assert::IsTrue(CalculatePostfix(str) == 10);

}

};

}