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 _____ Точилин А.Е.

Преподаватель _____ Тутуева А.В.

Санкт-Петербург
2020

Цель работы

Перевод простейшего математического выражения из инфиксной формы в префиксную и вычисление результата.

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:

- префиксная ФЗ
- результат вычисления

Описание реализуемых методов.

Оценка временной сложности методов.

Для реализации был набор функций и классов. Список функций каждого класса представлен в табл. 1.

Таблица 1 – Описание и оценка сложности функций поиска и сортировки.

Название метода	Описание	Оценка временной сложности
Calculator		
bool isBinary(char ch);	Проверка символа, возвращает true если символ знак операции	O(1)
bool checkCurrent(std::string str);	Проверка операторов и чисел на корректный ввод	O(n)
bool isOperand(std::string str);	Если строка операнд	O(n)
int getPriority(std::string str);	Возвращает приоритет бинарного оператора	O(1)
std::string* reverseStringArray(std::string* arr, int size);	Переворот массива строк	O(n)
std::string* toPrefixReserved();	Функция переводит выражение в инфиксной форме в префиксную форму(возвращает перевернутый массив)	O(n)
void tokenize();	Функция разделяет введенной выражение на токены (операторы и операнды)	O(1)
void processStack(Stack* stack);	Обработка стека вычисления выражения в префиксной форме	O(1)
double doBinarOperator(double value1, double value2, std::string oper);	Применяет переданный бинарный оператор к двум переданным значениям	O(1)
double doUnarOperator(double value, std::string oper);	Применяет переданный унарный оператор к переданному значению	O(1)
bool isNumber(std::string str);	Проверка строки на число	O(1)
bool isUnarOperator(std::string str);	Проверка является ли переданный строка унарным оператором	O(1)
double toDouble(std::string strVal);	Переводит переданную строку к double	O(1)
std::string toPrefixForm();	Функция переводит выражение в префиксную форму	O(n)
double calculatePrefix();	Счет выражения в префиксной форме	O(n)

Class Stack		
void push(std::string elem);	Добавление элемента	O(1)
std::string pop();	Снятие элемента	O(1)
bool isEmpty();	Проверка на пустату	O(1)
std::string top(n);	Возвращает n-ый элемент стека	O(1)
int getSize();	Возвращает размер стека	O(1)
void printStack();	Выводит стек (используется для отладки)	O(1)

Код всех функций представлен в приложении А.

Описание алгоритма решения

Сначала переданной выражение переводится в массив строк и при переводе проверяется корректность ввода. Затем производится перевод в префиксную форму. Для этого в цикле производится итерация с конца по массиву операторов и операндов. Если встречена закрывающая скобка, она кладется на стек. Если операнд либо бинарный оператор, то добавляется в результирующее выражение. Если открывающая скобка, то снимаем со стека элементы пока не найдем закрывающую. Если бинарный оператор, то снимаем со стека все операторы с большим либо равным приоритетом и добавляем в результирующее выражение, а встреченный оператор пушим на стек. После того как был пройден весь массив, со стека достаются все элементы и добавляются в результирующее выражение. После этого требуется перевернуть полученной выражение. Затем для вычисления результата выражения требуется, в обратном цикле обойти выражение в префиксной форме. Каждый элемент кладется на стек. Если на верху стека лежит бинарный оператор, а под ним число, то оператор применяется к числу и результат кладется на стек. Если лежит бинарный оператор и под ним два числа, то оператор применяется к этим двум числам и результат кладется на стек.

Обоснование использования структур данных

Для перевода выражения в префиксную форму и вычисления значения выражения в префиксной форме был использован,

собственный стек. Для хранения массива строк, состоящего из операторов и операндов был использован массив.

Описание реализованных unit-тестов

Для проверки работоспособности программы были реализованы unit-тесты. Бля этого был создан новый проект в Visual Studio, в котором написан класс Test_CW, методы класса и их описание представлены в табл. 2, код в приложении В.

Таблица 2 – Описание методов класса LR2tests

Методы	Описание метода
CheckWrongInput(1-7)	Проверяется что при неправильном вводе выражения приложение выдаст ошибку
CheckPrefixForm(1-4)	Проверяется правильный перевод выражения в префиксную форму
CheckResult(1-4)	Проверка правильного вычисления результата выражения

Пример работы

Пример работы программы представлен на рис 2., код функции main на рис. 1.

```
#include <iostream>
#include "Calculator.h"
#include <string>
#include <cmath>

int main()
{
    std::cout << "Input expression:" << std::endl;
    std::string expression;
    std::cin >> expression;
    Calculator calc = Calculator(expression);
    try {
        std::cout << "Expression in prefix form:" << std::endl;
        std::cout << calc.toPrefixForm() << std::endl;
        std::cout << "Result of exression:" << std::endl;
        std::cout << calc.calculatePrefix() << std::endl;
    }
    catch (std::exception& e) {
        std::cout << e.what() << std::endl;
    }
    return 0;
}
```

Рисунок 1 – код функции main

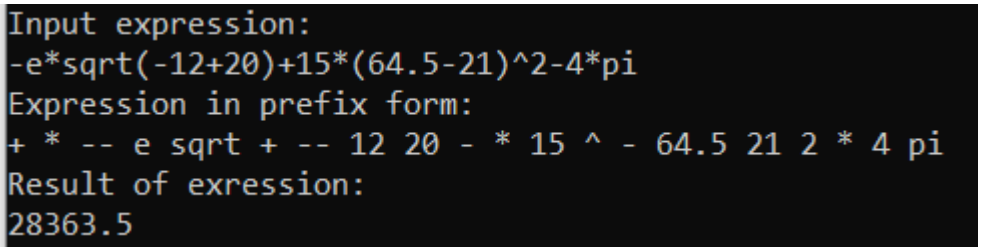

Рисунок 2 – результат запуска программы

Приложение А

Листинг файла Calculator.h

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

class Calculator
{
private:
	std::string sourceString;
	std::string* prefixForm;
	std::string* tokenizedArray = new std::string[1000]();
	int arrLength = 0;
	int prefixLength = 0;
	bool isInPrefix = false;

	bool isBinary(char ch);

	bool checkCurrent(std::string str);

	bool isOperand(std::string str);

	int getPriority(std::string str);

	std::string* reverseStringArray(std::string* arr, int size);

	std::string* toPrefixReserved();

	void tokenize();

	void processStack(Stack* stack);

	double doBinarOperator(double value1, double value2, std::string oper);

	double doUnarOperator(double value, std::string oper);

	bool isNumber(std::string str);

	bool isUnarOperator(std::string str);

	double toDouble(std::string strVal);

public:
	Calculator(std::string inputExpression);

	std::string toPrefixForm();

	double calculatePrefix();
};
```

Листинг файла Calculator.cpp

```
#include "Calculator.h"
//define math constans
#define M_PI 3.14159265358979323846   // pi
#define M_E 2.71828182845904523536 //e

Calculator::Calculator(std::string inputExpression) {
	this->sourceString = inputExpression;
}

//function make infix expression to prefix from
std::string Calculator::toPrefixForm() {
	// split input expression to array of operands and operators
	this->tokenize();
	/*for (int i = 0; i < arrLength; i++)
	{
		std::cout << this->tokenizedArray[i] << " ";
	}
	std::cout << std::endl;*/
	//convert to reversed prefix form
	std::string* ReversedPrefix = this->toPrefixReserved();
	//reverse array
	prefixForm = this->reverseStringArray(ReversedPrefix, prefixLength);
	//output expression in prefix form
	std::string result = "";
	for (int i = 0; i < prefixLength; i++)
	{
		result += prefixForm[i] + " ";
	}
	result.pop_back();
	this->isInPrefix = true;
	return result;
}

//convert string to double
double Calculator::toDouble(std::string strVal) {
	//check constants if not constant use function stod
	if (strVal == "e") {
		return M_E;
	}
	else if (strVal == "pi") {
		return M_PI;
	}
	else {
		return std::stod(strVal);
	}
}
//function which calculate
double Calculator::calculatePrefix() {
	if (!isInPrefix) {
		throw std::runtime_error("Equtation not converted!");
	}
	//using stack
	Stack* stack = new Stack();
	//go from and to start of prefix form equtation array
	for (int i = prefixLength - 1; i >= 0; i--) {
		//push to stack and use processStack function
		stack->push(prefixForm[i]);
		processStack(stack);
	}
	//print result
	return toDouble(stack->top());
```

```
}

//function to process stack while culculating
void Calculator::processStack(Stack* stack) {
	if (stack->isEmpty() || isNumber(stack->top())) {
		return;
	}
	//if meet unary operator and size of stack higher than 1 and after top elem stack has
number do operatorand push to array
	else if (isUnarOperator(stack->top())) {
		if (stack->getSize() > 1) {
			if (isNumber(stack->top(2))) {
				std::string oper = stack->pop();
				double val = doUnarOperator(toDouble(stack->pop()), oper);
				stack->push(std::to_string(val));
			}
		}
	}
	//if met binary operator and size of stack higher than 2 and after top elem stack has
two numbers do operator and push to array
	else {
		if (stack->getSize() > 2) {
			if (isNumber(stack->top(2)) && isNumber(stack->top(3))) {
				std::string oper = stack->pop();
				double val1 = toDouble(stack->pop());
				double val2 = toDouble(stack->pop());
				double val = doBinarOperator(val1, val2, oper);
				stack->push(std::to_string(val));
			}
		}
	}
}

//check isBinary operator symbol for tokenize function
bool Calculator::isBinary(char ch) {
	if (ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^') {
		return true;
	}
	return false;
}

//check inputed operators
bool Calculator::checkCurrent(std::string str) {
	if (std::isalpha(str[0])) {
		if (str == "cos" || str == "sin" || str == "tg" || str == "ctg" ||
			str == "ln" || str == "log" || str == "sqrt" || str == "cbrt" || str ==
"pi" || str == "e") {
			return true;
		}
		else return false;
	}
	//number should have less than 2 dots and no dot at and
	else if (std::isdigit(str[0]) || str[0] == '-') {
		if (std::count(str.begin(), str.end(), '.') > 1 || str[str.length() - 1] ==
'.') {
			return false;
		}
		return true;
	}
	else {
		return false;
	}
}
```

```
//check if input string operator or operand, single operators make operand
bool Calculator::isOperand(std::string str) {
	if (str == "e" || str == "pi" || str == "cos" || str == "sin" || str == "tg" || str
== "ctg" ||
		str == "ln" || str == "log" || str == "sqrt" || str == "cbrt" || str == "--")
{
		return true;
	}
	try
	{
		double value = std::stod(str);
		return true;
	}
	catch (std::exception& e)
	{
		return false;
	}
}

//get priority of operands

int Calculator::getPriority(std::string str) {
	if (str == "-" || str == "+") {
		return 1;
	}
	else if (str == "*" || str == "/") {
		return 2;
	}
	else if (str == "^") {
		return 3;
	}
	return 0;
}

//reverse string if also change "(" to ")" and ")" to "("
std::string* Calculator::reverseStringArray(std::string* arr, int size) {
	std::string* result = new std::string[1000]();
	for (int i = size - 1; i >= 0; i--)
	{
		if (arr[i] == ")") {
			result[size - 1 - i] = "(";
		}
		else if (arr[i] == "(") {
			result[size - 1 - i] = ")";
		}
		else {
			result[size - 1 - i] = arr[i];
		}
	}
	return result;
}

//function convert expression to prefix reversed form
std::string* Calculator::toPrefixReserved() {
	Stack operators = Stack();                                    //stack for operand
	std::string* res = new std::string[1000];
	int resLength = 0;
	for (int i = arrLength - 1; i >= 0; i--)    //iterate from and to start of tokeanized
array
	{
		if (tokenizedArray[i] == ")") {     //if met close bracket push to operators
stack
```

```
                        operators.push(")");
                }
                else if (isOperand(tokenizedArray[i])) {   //if met elem is operand add to
result array
                        res[resLength] = tokenizedArray[i];
                        resLength++;
                }
                else if (tokenizedArray[i] == "(") {    //if met close bracket pop stack ad
add to result while not met open bracket
                        while (operators.top() != ")") {
                                try {
                                        res[resLength] = operators.pop();
                                }
                                catch (const std::runtime_error& error) {
                                        throw std::runtime_error("No close bracket to open bracket
at index: " + std::to_string(i));
                                }
                                resLength++;
                        }
                        operators.pop();

                }
                else if (!(isOperand(tokenizedArray[i]))) {   //if met operator pop and add
operators with higher priority to result array
                        while (!operators.isEmpty() && this->getPriority(tokenizedArray[i]) <=
this->getPriority(operators.top())) {
                                res[resLength] = operators.pop();
                                resLength++;
                        }
                        operators.push(tokenizedArray[i]);   //add met operator to stack
                }
        }

        while (!(operators.isEmpty())) {  //pop all elems from stack and add to result
                res[resLength] = operators.pop();
                resLength++;
                if (res[resLength] == "(") {
                        throw std::runtime_error("No open bracket");
                }
        };

        this->prefixLength = resLength;   //change prefix array length
        return res;
}
//split input expression into array operators and operands
void Calculator::tokenize() {
        std::string result = "";
        std::string current = ""; //for current double or operator
        for (int i = 0; i < this->sourceString.length(); i++)
        {
                //before open bracket can only be operator
                if (this->sourceString[i] == '(') {
                        if (isNumber(current)) {
                                throw std::runtime_error("No numbers before open bracket: pos" +
std::to_string(i - 1));
                        }
                        if (current.length() > 0) {
                                if (checkCurrent(current)) {
                                        tokenizedArray[arrLength] = current;
                                        arrLength++;
                                }
                                else {
```

```
                                throw std::runtime_error("No such operator or error with
number, pos: " + std::to_string(i - current.length()));
                        }
                    }
                    current = "";
                    tokenizedArray[arrLength] = "(";
                    arrLength++;

                }
                //before close bracket can only be number
                else if (this->sourceString[i] == ')') {
                    if (!isNumber(current) && this->sourceString[i - 1] != ')') {
                        throw std::runtime_error("Before close bracket can only be number
pos: " + std::to_string(i));
                    }
                    if (current.length() > 0) {
                        tokenizedArray[arrLength] = current;
                        arrLength++;
                    }
                    current = "";
                    tokenizedArray[arrLength] = ")";
                    arrLength++;
                }
                // before binary operator can only be close bracket or number
                else if (isBinary(this->sourceString[i])) {
                    if (!isNumber(current)) {
                        if (i != 0) {
                            if (this->sourceString[i - 1] == '(' && this-
>sourceString[i] == '-') {
                                tokenizedArray[arrLength] = "--";
                                arrLength++;
                            }
                            else if (this->sourceString[i - 1] == ')') {
                                std::string symbStr = "";
                                symbStr += this->sourceString[i];
                                tokenizedArray[arrLength] = symbStr;
                                arrLength++;
                            }
                            else {
                                throw std::runtime_error("Before binary must be
number or bracket pos: " + std::to_string(i));
                            }
                        }
                        else {
                            if (this->sourceString[i] == '-') {
                                tokenizedArray[arrLength] = "--";
                                arrLength++;
                            }
                            else {
                                throw std::runtime_error("Before binary must be
number or bracket 22pos: " + std::to_string(i));
                            }
                        }
                    }
                    else {
                        tokenizedArray[arrLength] = current;
                        arrLength++;
                        current = "";
                        std::string symbStr = "";
                        symbStr += this->sourceString[i];
                        tokenizedArray[arrLength] = symbStr;
                        arrLength++;
                    }
```

```
		}
		else if (this->sourceString[i] == '.')
		{
			//if met dot prev symbol must be digit
			if (current.length() > 0 && std::isdigit(current[current.length() -
1])) {
				current += '.';
			}
			else {
				throw std::runtime_error("Dot not expexted, pos: " +
std::to_string(i));
			}
		}
		//if meet digit current must be zero length or prev elem shoul be digit or dot
		else if (std::isdigit(this->sourceString[i])) {
			if (current.length() == 0 || std::isdigit(current[current.length() -
1]) || current[current.length() - 1] == '.') {
				current += this->sourceString[i];
			}
			else {
				throw std::runtime_error("Digit not expexted, pos: " +
std::to_string(i));
			}
		}
		//if meet alpha current must be zero length of prev elem should be alpha
		else if (std::isalpha(this->sourceString[i]))
		{
			if (current.length() == 0 || std::isalpha(current[current.length() -
1])) {
				current += this->sourceString[i];
			}
			else {
				throw std::runtime_error("Error with symbol: " +
std::to_string(i));
			}
		}
		else {
			throw std::runtime_error("Error with symbol: " + std::to_string(i));
		}

	}
	//check last elem
	if (current.length() > 0) {
		if (checkCurrent(current)) {
			tokenizedArray[arrLength] = current;
			arrLength++;
		}
		else {
			throw std::runtime_error("Wrong last elem");
		}
	}
	if (!isNumber(tokenizedArray[arrLength - 1])) {
		if (tokenizedArray[arrLength - 1] != ")") {
			throw std::runtime_error("Last elem must be close bracket or number");
		}
	}
}

//apply binary operator
double Calculator::doBinarOperator(double value1, double value2, std::string oper) {
	if (oper == "+") {
		return value1 + value2;
	}
```

```
	else if (oper == "-") {
		return value1 - value2;
	}
	else if (oper == "*") {
		return value2 * value1;
	}
	else if (oper == "/") {
		return value1 * 1.0 / value2;
	}
	else if (oper == "^") {
		return pow(value1, value2);
	}
}

//apply unary operator
double Calculator::doUnarOperator(double value, std::string oper) {
	if (oper == "--") {
		return -value;
	}
	else if (oper == "cos") {
		return cos(value);
	}
	else if (oper == "sin") {
		return sin(value);
	}
	else if (oper == "tg") {
		return tan(value);
	}
	else if (oper == "ctg") {
		return 1.0 / tan(value);
	}
	else if (oper == "ln") {
		return log(value);
	}
	else if (oper == "log") {
		return log10(value);
	}
	else if (oper == "sqrt") {
		return sqrt(value);
	}
	else {
		return cbrt(value);
	}
}


//check is number for culculating
bool Calculator::isNumber(std::string str) {

	if (str == "pi" || str == "e") {
		return true;
	}
	try
	{
		double value = std::stod(str);
		return true && checkCurrent(str);
	}
	catch (std::exception& e)
	{
		return false;
	}
}
```

```
//check type of operator
bool Calculator::isUnarOperator(std::string str) {
	if (str == "*" || str == "-" || str == "+" || str == "/" || str == "^") {
		return false;
	}
	else return true;
}
```

Листинг файла Stack.h

```
#pragma once
#include <stdexcept>
#include <string>
#include <iostream>

class Stack
{
private:
	static const int MAX_SIZE = 500;
	std::string* stack;
	int indexHead;
public:
	Stack() {
		this->indexHead = -1;
		this->stack = new std::string[MAX_SIZE];
	};

	void push(std::string elem) { //add elem to array and change index of head
		indexHead++;
		stack[indexHead] = elem;
	};

	std::string pop() {  //if stack is't empty
		if (!isEmpty()) {
			indexHead--;      //reduce index of head
			return stack[indexHead + 1];  //and return popped element
		}
		else {
			throw std::runtime_error("Stack is empty in {function pop}");
		}
	}

	std::string top() { //get first elemetn
		if (isEmpty()) {
			throw std::runtime_error("Stack is empty in {function top}");
		}
		return stack[indexHead];
	}

	std::string top(int n) { //get n element of stack
		if (getSize() < n) {
			std::string err = "Stack size lower than " + std::to_string(n) + " {in
function top(int)}";
			throw std::runtime_error(err);
		}
		return stack[indexHead - n + 1];
	}

	int getSize() { //return size of array
		return indexHead + 1;
```

```
	}

	bool isEmpty() {
		return indexHead == -1;
	}

	void printStack() {
		if (indexHead == -1) {
			std::cout << "empty" << std::endl;
		}
		else {
			for (int i = indexHead; i >= 0; i--)
			{
				std::cout << stack[i] << "   ";
			}
		}
	}
};
```

Листинг файла Test_CW.cpp

```
#include "pch.h"
#include "CppUnitTest.h"
#include "../CW/Calculator.h"
#include "../CW/Calculator.cpp"
#include "../CW/Stack.h"
#include <cmath>

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace TestCW
{
	TEST_CLASS(TestCW)
	{
	public:

		TEST_METHOD(CheckWrongInput1)
		{
			Calculator calc = Calculator("-e1*sqrt(-12+20)+15*(64.5-21)^2");
			try {
				calc.toPrefixForm();
				Assert::Fail();
			}
			catch (std::exception& e) {}
		}
		TEST_METHOD(CheckWrongInput2)
		{
			Calculator calc = Calculator("*e*sqrt(-12+20)+15*(64.5-21)^2");
			try {
				calc.toPrefixForm();
				Assert::Fail();
			}
			catch (std::exception& e) {}
		}
		TEST_METHOD(CheckWrongInput3)
		{
			Calculator calc = Calculator("e*lm(-12+20)+15*(64.5-21)^2");
			try {
				calc.toPrefixForm();
				Assert::Fail();
			}
			catch (std::exception& e) {}
		}
```

```
TEST_METHOD(CheckWrongInput4)
{
	Calculator calc = Calculator("e*ln(-12+20)+15*(64.5-21)^2__");
	try {
		calc.toPrefixForm();
		Assert::Fail();
	}
	catch (std::exception& e) {}
}
TEST_METHOD(CheckWrongInput5)
{
	Calculator calc = Calculator("e*ln(-12+20)+15*(6.4.5-21)^2");
	try {
		calc.toPrefixForm();
		Assert::Fail();
	}
	catch (std::exception& e) {}
}
TEST_METHOD(CheckWrongInput6)
{
	Calculator calc = Calculator("ln)75.11*ln(22+ln(11)))");
	try {
		calc.toPrefixForm();
		Assert::Fail();
	}
	catch (std::exception& e) {}
}
TEST_METHOD(CheckWrongInput7)
{
	Calculator calc = Calculator(".3+3");
	try {
		calc.toPrefixForm();
		Assert::Fail();
	}
	catch (std::exception& e) {}
}
TEST_METHOD(CheckPrefixForm1)
{
	Calculator calc = Calculator("ln(75.11*ln(22+ln(11)))");
	std::string temp = calc.toPrefixForm();
	char tab[1024];
	strcpy_s(tab, temp.c_str());
	Assert::AreEqual("ln * 75.11 ln + 22 ln 11", tab);
}
TEST_METHOD(CheckPrefixForm2)
{
	Calculator calc = Calculator("(((5+3)/4)*6)");
	std::string temp = calc.toPrefixForm();
	char tab[1024];
	strcpy_s(tab, temp.c_str());
	Assert::AreEqual("* / + 5 3 4 6", tab);
}
TEST_METHOD(CheckPrefixForm3)
{
	Calculator calc = Calculator("-e*sqrt(-12+20)+15*(64.5-21)^2");
	std::string temp = calc.toPrefixForm();
	char tab[1024];
	strcpy_s(tab, temp.c_str());
	Assert::AreEqual("+ * -- e sqrt + -- 12 20 * 15 ^ - 64.5 21 2", tab);
}
TEST_METHOD(CheckPrefixForm4)
{
	Calculator calc = Calculator("ln(sqrt(cbrt(log(5))))");
```

```
			std::string temp = calc.toPrefixForm();
			char tab[1024];
			strcpy_s(tab, temp.c_str());
			Assert::AreEqual("ln sqrt cbrt log 5", tab);
		}
		TEST_METHOD(CheckResult1)
		{
			Calculator calc = Calculator("ln(75.11*ln(22+ln(11)))");
			calc.toPrefixForm();
			Assert::AreEqual(ceil(5.48038), ceil(calc.calculatePrefix()));
		}
		TEST_METHOD(CheckResult2)
		{
			Calculator calc = Calculator("(((5+3)/4)*6)");
			calc.toPrefixForm();
			Assert::AreEqual(12.0, calc.calculatePrefix());
		}
		TEST_METHOD(CheckResult3)
		{
			Calculator calc = Calculator("-e*sqrt(-12+20)+15*(64.5-21)^2");
			calc.toPrefixForm();
			Assert::AreEqual(ceil(28376.1), ceil(calc.calculatePrefix()));
		}
		TEST_METHOD(CheckResult4)
		{
			Calculator calc = Calculator("ln(sqrt(cbrt(log(5))))");
			calc.toPrefixForm();
			Assert::AreEqual(-0.059692, calc.calculatePrefix());
		}
	};
}
```

Вывод

В ходе выполнения работы был изучен метод перевода выражения из инфиксной формы в префиксную. И метод вычисления выражения в префиксной форме.