МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное
учреждение высшего образования
**«Национальный исследовательский
Нижегородский государственный университет им. Н.И. Лобачевского»
(ННГУ)**

**Институт информационных технологий, математики и механики**

Направление подготовки: «Программная инженерия»

**ОТЧЕТ**
по лабораторной работе

# СТЕК. ПОСТФИКСНАЯ ЗАПИСЬ

Выполнил: студент группы
3822Б1ПР2

_____ М.В.Фёдоров
Подпись

Проверила:

_____ Я.В. Силенко
Подпись

Нижний Новгород 2023

# Содержание

# Введение

Стек - это структура данных, которая работает по принципу **FILO** (first in - last out; первый пришел - последний ушел). Это означает, что доступ к элементам стека осуществляется в обратном порядке, то есть последний добавленный элемент будет первым удаленным. Стек часто сравнивают со стопкой тарелок: чтобы взять вторую сверху, нужно снять верхнюю.

Стеки широко используются в различных областях, включая:

1. Разработка программного обеспечения: Стеки используются для управления процессами и потоками данных в приложениях, таких как веб-серверы, базы данных, системы управления задачами и т.д.
2. Мобильные приложения: Стеки используются для управления задачами и уведомлениями в мобильных приложениях, таких как социальные сети, мессенджеры, игры и т.д.
3. Обработка данных: Стеки используются для обработки больших объемов данных, таких как логи, данные сенсоров и т.д.
4. Облачные вычисления: Стеки используются для управления задачами и ресурсами в облачных вычислениях, таких как обработка запросов, балансировка нагрузки и т.д.
5. Искусственный интеллект: Стеки используются для управления задачами и обработки данных в системах искусственного интеллекта, таких как машинное обучение, нейронные сети и т.д.
6.

# Постановка задачи

Требуется разработать программу для перевода математических выражений из инфиксной в постфиксную запись (обратную польскую запись). Соответственно, программа должна преобразовывать инфиксную запись выражения в постфиксную, проводить вычисления и выводить результат вычислений пользователю.

# Руководство пользователя

При запуске программы на экране появляется  консоль сообщением о вводе выражения. После ввода выражения программа выдаёт постфиксную запись этого выражения и сам ответ:

```
Введите математическое выражение
3+2*(4-1)
3 2 4 1-*+ = 9
```

# Руководство программиста

## Описание структуры программы

Программа состоит из следующих файлов:

1) TStack.h – содержит реализацию шаблонного класса TStack;
2) TCalculator.h – содержит реализацию класса постфиксного калькулятора TCalculator;
3) MyForm.h – содержит реализацию визуального интерфейса постфиксного калькулятора;
4) MyForm.h [конструктор] – форма для визуального редактирования;
5) Test_tstack.cpp, test_tcalc.cpp – содержат Гугл тесты для классов TStack и TCalculator;

## Описание структур данных

Class TStack:

Поля:

T* pMem – указательна на массив элементов стека;

int size – размер стека;

int top – вершина стека (индекс последнего элемента)

Конструкторы и деструктор:

TStack();

TStack(int _size = 0);

TStack(TStack<T>& stack);

~TStack();

Методы:

int GetSize() – возвращает текущий размер стека;

T GetTop() – возвращает значение элемента на вершине стека без удаления его из стека.

void Push(double elem) – добавляет элемент на вершину стека.

bool IsEmpty() – проверка стека на пустоту;

bool IsFull() – проверка стека на полноту;

void Clear() – очищает стек, зануляет все элементы;

T Get() – извлекает и возвращает элемент с вершины стека;

void copyInnards(const TStack& s) – копирует содержимое стека в текущий стек;

Операторы:

TStack& operator=(const TStack<T>& stack) – перегруженный оператор присваивания;

bool operator==(const TStack<T>& stack) – перегруженный оператор сравнения на равенство;

bool operator!=(const TStack<T>& stack) – перегруженный оператор сравнения на неравенство.

Class TCalculator:

Поля:

double answer – хранит результаты вычислений;

string infix – инфиксная запись выражения;

string postfix – постфиксная запись выражения:

TStack<double> stackDigits – хранит числовые значения;

TStack<char> stackOperators – хранит операторы;

int Priority(char elem) – возвращает приоритет оператора;

Конструкторы и деструктор():

TCalculator();

TCalculator(const string& exp);

~TCalculator() {}

Методы:

string GetExp() – возвращает входное инфиксное выражение;

string GetPostfix() – возвращает преобразованное постфиксное выражение;

string GetInfix() –  возвращает входное инфиксное выражение;

double GetAnswer() – возвращает результат вычислений;

void SetExp(const string& exp) – устанавливает новое входное выражение;

void ToPostfix() – преобразует инфиксное выражение в постфиксное;

double CalcPostfix() – вычисляет результат постфиксного выражения;

double Calc() – вычисления инфиксного выражения;

friend istream& operator>>(istream& istr, TCalculator& c) – перегрузка оператора ввода;

friend ostream& operator<<(ostream& ostr, const TCalculator& c) – перегрузка оператора вывода;

## Описание алгоритмов

Рассматриваем поочередно каждый символ:

1. Если этот символ - число (или переменная), то просто помещаем его в выходную строку.

2. Если символ - знак операции (+, -, *, / ), то проверяем приоритет данной операции. Операции умножения и деления имеют наивысший приоритет (допустим он равен 3). Операции сложения и вычитания имеют меньший приоритет (равен 2). Наименьший приоритет (равен 1) имеет открывающая скобка.

Получив один из этих символов, мы должны проверить стек:

а) Если стек все еще пуст, или находящиеся в нем символы (а находится в нем могут только знаки операций и открывающая скобка) имеют меньший приоритет, чем приоритет текущего символа, то помещаем текущий символ в стек.

б) Если символ, находящийся на вершине стека, имеет приоритет, больший или равный приоритету текущего символа, то извлекаем символы из стека в выходную строку до тех пор, пока выполняется это условие; затем переходим к пункту а).

3. Если текущий символ - открывающая скобка, то помещаем ее в стек.

4. Если текущий символ - закрывающая скобка, то извлекаем символы из стека в выходную строку до тех пор, пока не встретим в стеке открывающую скобку (т.е. символ с приоритетом, равным 1), которую следует просто уничтожить. Закрывающая скобка также уничтожается.

Если вся входная строка разобрана, а в стеке еще остаются знаки операций, извлекаем их из стека в выходную строку.

# Результаты

Рассмотрим выражение 12+(2*3-1)+15/3: по правилам вычислений, сначала рассмотрим скобку и первым действием умножим 2 на 3. Затем из этого произведения вычтем 1. В итоге получаем в скобках 5. Третьим действием сложим 12 и 5. Получаем: 17 + 15/3. Далее делим 15 на 3 и получаем 5. Просуммировав всё, получаем конечный ответ: 22.

Теперь введём данное выражение в нашей программе. Программа выдаёт постфиксную запись выражения и сам ответ:

```
Введите математическое выражение
12+(2*3-1)+15/3
12 2 3 * 1- + 15 3/+ = 22
```

Из постфиксной записи, произведённой программой, мы видим, что шаги её вычислений идентичны нашим: сначала 2*3, затем вычитаем из этого произведения 1, после складываем с 12 и переходим к 15/3, а затем переходим к суммированию всех членов выражения. А это говорит о том, что программа работает корректно, выдавая верное решение и верную постфиксную запись выражения.

# Заключение

Мною разработана программа для перевода математических выражений из инфиксной в постфиксную запись. Как показали результаты вычислений, программа верно переводит вводимое пользователем выражение из инфиксной формы в постфиксную, а также выводит верное решение выражения.

# Литература


1) Преобразование выражения в ОПЗ с использованием стека: https://trubetskoy1.narod.ru/ppn.html
2) Стек: https://codelessons.dev/ru/realizaciya-steka-stack-v-c/

# Приложение

TStack.h

```
#pragma once

template <class T>
class TStack
{
protected:
	T* pMem;
	int size;
	int top;
public:
	TStack() {
		size = 0;
		top = -1;
		pMem = nullptr;
	}
	TStack(int _size = 0) {
		if (_size < 0) throw "error";
		top = -1;
		size = _size;
		pMem = new T[size];
		for (int i = 0; i < size; i++) pMem[i] = 0;
	}
	//TStack(int _size = 10, int _top = 0, T* pMem);

	//TStack(int _size, int _top);
	TStack(TStack<T>& stack) {
```

```
        top = stack.top;
        size = stack.size;
        pMem = new T[size];
        for (int i = 0; i < size; i++) pMem[i] = stack.pMem[i];
}
~TStack() { delete pMem; }

int GetSize()
{
        return top + 1;
}

T GetTop()
{
        if (top < 0) throw "error";
        return pMem[top];
}

T Pop() {
        if (IsEmpty()) {
                throw "Stack is empty";
        }
        return pMem[top--];
        //return mas[top--];
}
void Push(double elem) {
        if (IsFull()) {
                throw - 1;
        }
```

```
        //pMem[top] = elem;
        //top++;
        pMem[++top] = elem;
}
bool IsEmpty() { return top + 1 == 0; }


bool IsFull() { return top + 1 == size; }

void Clear() { for (int i = 0; i < size; i++) pMem[i] = 0; }

T Get() {
        if (IsEmpty()) {
                throw - 1;
        }
        top--;
        return pMem[top];
}

void copyInnards(const TStack& s) {
        if (size != s.size) throw "error";
        for (int i = 0; i < size; i++) pMem[i] = s.pMem[i];
}

TStack& operator=(const TStack<T>& stack) {
        if (&stack == this) return *this;
        size = stack.size;
        top = stack.top;
        delete[] pMem;
```

```
            pMem = new T[size];
            for (size_t i = 0; i < size; i++) pMem[i] = stack.pMem[i];
            return *this;
        }

        bool operator==(const TStack<T>& stack) {
            if (size != stack.size) return false;
            for (int i = 0; i < size; i++) {
                if (pMem[i] != stack.pMem[i]) return false;
            }
            return true;
        }
        bool operator!=(const TStack<T>& stack) {
            if (size != stack.size) return true;
            for (int i = 0; i < size; i++) {
                if (pMem[i] != stack.pMem[i]) return true;
            }
            return false;
        }
};
```

TCalculator.h

```
#include "TCalculator.h"
#include <iostream>
#include <cmath>

TCalculator::TCalculator() : stackOperators(0), stackDigits(0)
{
```

```
	infix = "";
	postfix = "";
	answer = 0;
}

TCalculator::TCalculator(const string& exp) : stackOperators(exp.length()*3),
stackDigits(exp.length()*3)
{
	infix = exp;
	postfix = "";
	answer = 0;
}

int TCalculator::Priority(char elem) {
	switch (elem) {
	case'(':
	case')':return 1;
	case'+':
	case'-':return 2;
	case'*':
	case's':
	case'c':
	case'/':return 3;
	default: throw "error";
	}
}

void TCalculator::ToPostfix() {
	postfix = "";
	string src = "(" + infix + ")";
```

```
	char elem = '!';
	unsigned int i = 0;
	while (i < src.size()) {
		//postfix += " ";
		if (src[i] >= '0' && src[i] <= '9') {
			postfix += src[i];
		}
		if (src[i] == ',' || src[i] == '+' || src[i] == '-' || src[i] == '*' || src[i] == '/')
{
			postfix += " ";
			elem = stackOperators.Pop();
			while (Priority(elem) >= Priority(src[i])) {
				postfix += elem;
				postfix += " ";
				elem = stackOperators.Pop();
			}
			stackOperators.Push(elem);
			stackOperators.Push(src[i]);
		}
		if (src[i] == 's' || src[i] == 'c')
			stackOperators.Push(src[i]);
		if (src[i] == '(') {
			stackOperators.Push(src[i]);
		}
		if (src[i] == ')') {
			elem = stackOperators.Pop();
			while (elem != '(') {
				postfix += elem;
				elem = stackOperators.Pop();
			}
```

```
            }
            i++;
        }
}

void TCalculator::SetExp(const string& exp) {
        infix = exp;
}

double TCalculator::Calc() {
        try {
            ToPostfix();
            return CalcPostfix();
        }
        catch (const char* error) {
            cout << "Error: " << error << endl;
            return 0.0;
        }
}

double TCalculator::CalcPostfix() {
        for (size_t i = 0; i < postfix.size(); i++)
        {
            if (postfix[i] == '+' || postfix[i] == '-' || postfix[i] == '*' || postfix[i] == '/'
|| postfix[i] == 's' || postfix[i] == 'c') {
                double d1, d2;



                switch (postfix[i]) {
                case '+':
```

```
            d1 = stackDigits.Pop();
            d2 = stackDigits.Pop();

            stackDigits.Push(d1 + d2);
            break;
        case '-':
            d2 = stackDigits.Pop();
            d1 = stackDigits.Pop();
            stackDigits.Push(d1 - d2);
            break;
        case '*':
            d1 = stackDigits.Pop();
            d2 = stackDigits.Pop();
            stackDigits.Push(d1 * d2);
            break;
        case '/':
            d1 = stackDigits.Pop();
            d2 = stackDigits.Pop();
            //double tmp = -1;

            stackDigits.Push(d2/d1);
            break;
        case 's':
            d1 = stackDigits.Pop();
            stackDigits.Push(sin(d1));
            break;
        case 'c':
            d1 = stackDigits.Pop();
            stackDigits.Push(cos(d1));
```

```
                        break;
                default: throw - 1;
                }
        }
        if (postfix[i] <= '9' && postfix[i] >= '0') {
                size_t c;
                double t = std::stod(&postfix[i], &c);
                stackDigits.Push(t);
                i = i + (c - 1);
        }
    }
    answer = stackDigits.GetTop();

}

istream& operator>>(istream& istr, TCalculator& calc)
{
    cout << "Enter your expression:";
    string exp;
    istr >> exp;
    calc.infix = exp;
    return istr;
}

ostream& operator<<(ostream& ostr, const TCalculator& c)
{
    if (c.postfix.size() == 0) throw "error";
    ostr << "Infix expr -- " << c.infix << endl;
    ostr << "Postfix expr  -- " << c.postfix << endl;
```

```
        ostr << "Answer is -- " << c.answer << endl;
        return ostr;
}
```