МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

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**

ЛАБОРАТОРНАЯ РАБОТА

на тему:

**«ВЫЧИСЛЕНИЕ АРИФМЕТИЧЕСКИХ ВЫРАЖЕНИЙ НА СТЕКАХ»**

**Выполнил(а):** студент(ка) группы 3822Б1ФИ1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Шпынов Н.А. /

Подпись

**Проверил:** к.т.н., доцент каф. ВВиСП

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Кустикова В.Д. /

Подпись

Нижний Новгород  
2023

**Оглавление**

[Введение 1](#_Toc3715)

[1 Постановка задачи 2](#_Toc15559)

[2 Руководство пользователя 3](#_Toc6449)

[2.1 Приложение для демонстрации работы стеков 3](#_Toc21185)

[2.2 Приложение для демонстрации работы постфиксной формы 4](#_Toc27738)

[3 Руководство программиста 6](#_Toc29860)

[3.1 Использованные алгоритмы 6](#_Toc9401)

[3.2 Описание классов 9](#_Toc8464)

[Заключение 15](#_Toc4085)

[Литература 16](#_Toc14963)

[Приложения 17](#_Toc11430)

# Введе**н**ие

Запись арифметического выражения в постфиксной форме - один из самых удобных способов его представления. Ее смысл заключается в размещении операторов перед операторами, что позволяет точно установить порядок выполнения операций, и, как следствие, дает возможность машине удобно и быстро работать с ним.

Для реализации представления арифметического выражения в постфиксной форме часто используется стек.

Стек - [абстрактный тип данных](https://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D1%8B%D0%B9_%D1%82%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85" \o "Абстрактный тип данных), представляющий собой [список элементов](https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)" \o "Список (информатика)), организованных по принципу [LIFO](https://ru.wikipedia.org/wiki/LIFO" \o ") (последний пришёл - первый вышел).

# Постановка задачи

Цель – реализовать представление механизма, для преобразования стандартной записи арифметического выражения в постфиксную с помощью стека.

Задачи:

1. Разработка структуры данных стека.
2. Разработка представления класса. Для представления арифметического выражения в постфиксной форме, следует создать соответствующий класс.
3. Определение операций. Разработать алгоритмы и функции для выполнения операций над арифметическим выражением, таких как преобразование в постфиксную форму, вычисление результата выражения и т.д.
4. Тестирование и отладка. Как и при любой разработке программного обеспечения, в конце необходимо провести тестирование и отладку кода. Следует обнаружить и исправить возможные ошибки и проблемы, а также убедиться в правильности функционирования.

# Руководство пользователя

## Приложение для демонстрации работы стеков

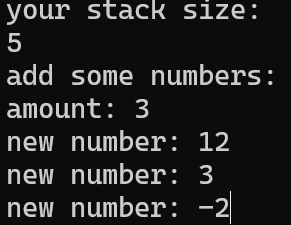
1. Запустить sample\_stack.exe. В результате появится следующее окно (рис. 1):



1. Основное окно приложения

На этом шаге потребуется ввести максимальный размер вашего стека

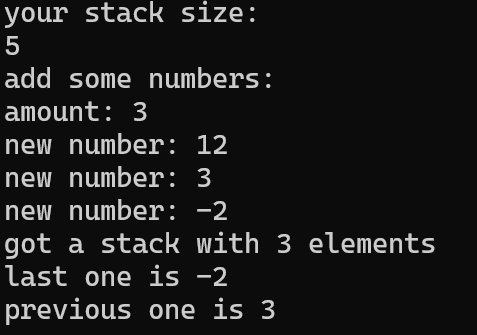
1. После ввода его размера, следует ввести несколько чисел (рис. 2):



1. Ввод длин

Вводить значения надо по порядку через пробел или Enter, предварительно указав, сколько их будет.

1. Программа выводит размер стека, последний и предпоследний полученные элементы (рис. 3):



1. Вывод

## Приложение для демонстрации работы постфиксной формы

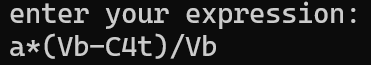
1. Запустить sample\_PN.exe. В результате появится следующее окно   
   (рис. 4):



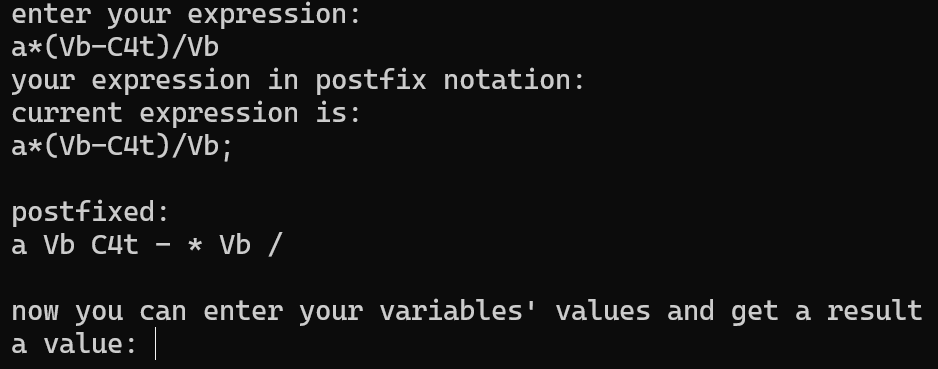
1. Основное окно приложения

Требуется ввести арифметическое выражение. Использовать можно операторы +, -, \*, /, скобки, а также константы и переменные, состоящие из латинских букв (с учетом заглавных) и цифр (не может стоять в начале переменной).

1. После ввода выражения (рис. 5):



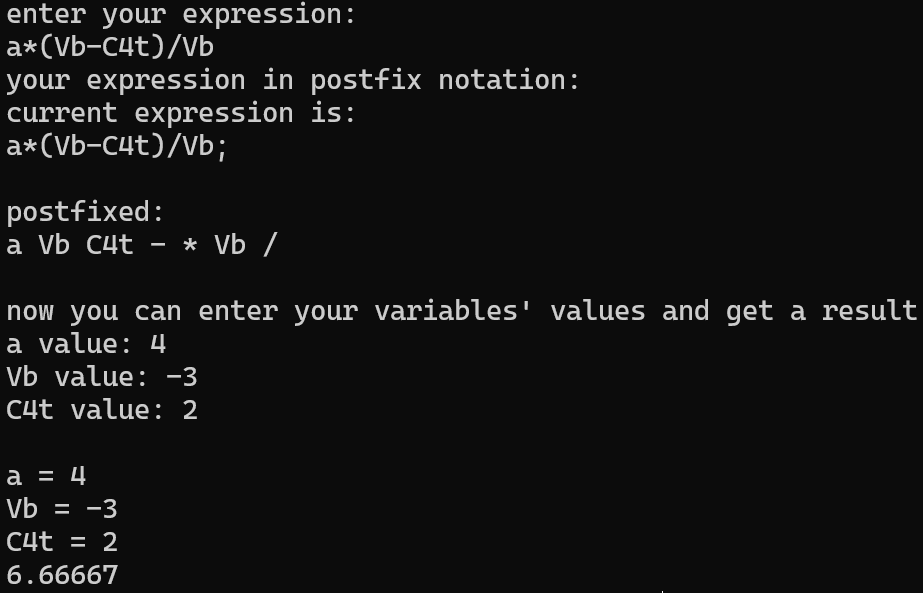
1. Ввод выражения
2. Далее программа для проверки выведет полученные ею данные и попросит значения переменных для того, чтобы посчитать частное значение выражения (рис. 6):



1. Проверка

Переменные вводятся по очереди, в порядке их появления в выражении.

1. Программа считает результат и выводит его (рис. 7).



1. Итог

# Руководство программиста

## Использованные алгоритмы

### Стек

Стек – [абстрактный тип данных](https://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D1%8B%D0%B9_%D1%82%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85" \o "Абстрактный тип данных), представляющий собой [список элементов](https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)" \o "Список (информатика)), организованных по принципу [LIFO](https://ru.wikipedia.org/wiki/LIFO" \o ") (последний пришёл - первый вышел).

Поддерживает операции возвращения значения верхнего элемента, его удаление, добавление элемента наверх, проверка стека на полноту и пустоту.

1. Верхний элемент

|  |
| --- |
| 3 |
| 4 |
| -2 |
| 2 |

Фунция .top() вернет 3

1. Удаление верхнего элемента

.del():

|  |  |  |
| --- | --- | --- |
| 3 |  |  |
| 4 | → | 4 |
| -2 |  | -2 |
| 2 |  | 2 |

1. Добавление элемента наверх

.add(3):

|  |  |  |
| --- | --- | --- |
|  |  | 3 |
| 4 | → | 4 |
| -2 |  | -2 |
| 2 |  | 2 |

4)

Проверка на пустоту/полноту

|  |
| --- |
|  |
|  |
|  |
|  |

.is\_empty() вернет true, .is\_full вернет false

|  |
| --- |
| 11 |
| 1 |
| 1 |
| 1 |

.is\_empty() вернет false, .is\_full вернет true

|  |
| --- |
|  |
| 1 |
| 1 |
| 1 |

.is\_empty() вернет false, .is\_full вернет false

### Постфиксная форма

Постфиксная форма - представление арифметического выражения так, что операторы размещаются перед операндами например выражение

(a\*b-c)/(d-a)-b

будет представлено как

ab\*c-da-/b-

Постфиксная форма позволяет точно установить порядок выполнения операций, и, как следствие, даёт возможность машине удобно и быстро работать с ним.

Алгоритм преобразования выражения в постфиксную запись:

1. Для операций вводится приоритет: «\*», «/» – 3, «+», «-» – 2, «(» – 1. Для «)» приоритета нет.
2. Исходное выражение просматривается слева направо:
   1. Операнды по мере их появления помещаются в один стек
   2. Символы операций и левой скобки помещаются в другой стек следующим образом: если новая операция имеет меньший приоритет операция на верхушке второго стека, все операции в нем с большим или равным приоритетом перекладывается в первый стек, после чего текущая операция кладётся во второй. В случае «)» из второго стека изымаются элементы в первый до тех пор, пока не придёт левая скобка или стек не опустошится, левая скобка в стек не переносится, она удаляется.

Алгоритм вычисления значения

1. Читаются знаки слева направо до конца последовательности.
   1. Если пришёл операнд, его значение помещается в соответствующий стек.
   2. Если пришел оператор, из стека операндов берутся 2 верхних операнда, выполняется указанная операция, результат возвращается стек.
2. Когда будет достигнут конец строки, в стеке будет лежать только ответ

## Описание классов

### Класс TStack

template <typename T> class TStack {

private:

int max\_size;

int current\_amount;

T\* elements;

public:

TStack();

TStack(int size = 20);

~TStack();

bool is\_full() const;

bool is\_empty() const;

T& top();

void add(const T& elem);

void del() ;

};

*Поля:*

max\_size – максимально допустимый размер стека.

current\_amount – текущее количество элементов в стеке

elements – массив элементов стека

*Конструкторы:*

**TStack()** – конструктор по умолчанию.

**TStack(int** size = 20) – конструктор с параметром.

Назначение: создание стека с заданной максимальной длиной, 2 по умолчанию

Входные данные: size – длина вектора.

*Деструктор*:

~TStack()

*Методы:*

bool is\_full() const – проверка на полноту.

Назначение: проверяет стек, полон ли он.

Входные данные: нет.

Выходные данные: true если полон, false если нет.

bool is\_empty() const – проверка на пустоту.

Назначение: проверяет стек, пуст ли он.

Входные данные: нет.

Выходные данные: false если полон, true если нет.

T& top() – получение верхнего элемента.

Назначение: возвращает верхний элемент в стеке.

Входные данные: нет.

Выходные данные: элемент заданного типа данных.

void add(const T& elem) – добавление элемента в стек.

Назначение: добавление элемента наверх стека.

Входные данные: elem – элемент.

Выходные данные: нет.

void del() – удаление элемента из стека

Назначение: удаление из стека послежнего элемента.

Входные данные: нет.

Выходные данные: нет.

*Операторы:*

Нет

### Класс tPostfix

class tPostfix {

private:

string expression;

vector<operand> vars;

vector<string> expression\_refined;

vector<string> expression\_postfixed;

int get\_operator\_priority(const string op);

bool is\_operator(const int i) const;

bool is\_operator(char c) const;

bool is\_operator(const string c) const;

void divide\_string();

void add\_a\_new\_variable(const string n);

double operation(double a, char op, double b);

void create\_postfixed\_expression(TStack<string>& S);

public:

void get\_values\_for\_variables();

void postfix\_notation();

int get\_variables\_amount(){

return vars.size();

}

double calculate\_the\_result();

tPostfix(const string& S);

friend ostream& operator<<(ostream& out, const tPostfix& p) {

out << "current expression is:" << endl

<< p.expression << ";" << endl << endl;

out << "postfixed:" << endl;

for (int i = 0; i < p.expression\_postfixed.size(); i++)

out << p.expression\_postfixed[i] << " ";

out << endl;

return out;

}

};

*Поля*:

expression – полученное выражение.

vars – переменные в выражении.

expression\_refined – выражение, разделённое на операторы и операнды.

expression\_postfixed – выражение в постфиксной форме.

*Конструкторы*:

tPostfix(const string& S)– конструктор копирования

Назначение: создание элемента класса tPostfix, равного данному.

Входные данные: s – элемент.

*Методы*:

int get\_operator\_priority(const string op)– приоритет оператора.

Назначение: возвращает приоритет полученного на вход оператора.

Входные данные: op – оператор.

Выходные данные: приоритет оператора.

bool is\_operator(const int i) const – проверка знака на оператор.

Назначение: проверяет символ из expression, является ли он каким либо оператором.

Входные данные: **i** – номер элемента в строке.

Выходные данные: true или false в зависимости от того, является ли оператором этот символ.

bool is\_operator(char c) const – проверка знака на оператор.

Назначение: проверяет полученный символ, является ли он каким либо оператором.

Входные данные: с – элемент.

Выходные данные: true или false в зависимости от того, является ли оператором этот символ.

bool is\_operator(const string c) const – проверка знака на оператор.

Назначение: проверяет полученный символ, является ли он каким либо оператором, если по какой то причине он был передан в формате строки.

Входные данные: s – строка.

Выходные данные: true или false в зависимости от того, является ли оператором этот символ.

void divide\_string()– деление строки на операторы и операнды.

Назначение: делит полученную строку, разделяя константы, операторы и операнды.

Входные данные: нет.

Выходные данные: нет.

void add\_a\_new\_variable(const string n)– добавление константы.

Назначение: добавляет константу в **vars.**

Входные данные: **n** – строка с именем константы.

Выходные данные: нет.

double operation(double a, char op, double b)– проведение операции.

Назначение: проводит заданную операцию с двумя операндами.

Входные данные: **a, b** – операнды, op – оператор.

Выходные данные: результат подсчета.

void create\_postfixed\_expression(TStack<string>&S) – разворачивает стек постфиксного выражения, полученного задом наперед.

Назначение: получает стек постфиксного выражения, вычисленного задом наперед, разворачивает его и записывает результат.

Входные данные: **s** – стек.

Выходные данные: нет.

void get\_values\_for\_variables() – получает значения переменных.

Назначение: получает от пользователя значения переменных в выражении.

Входные данные: нет.

Выходные данные: нет.

void postfix\_notation() – создает постфиксную форму выражения.

Назначение: получает уже разделенное выражение и создаёт из него постфиксную форму.

Входные данные: нет.

Выходные данные: нет.

int get\_variables\_amount() – получение количества переменных.

Назначение: считает текущее количество переменных в выражении.

Входные данные: нет.

Выходные данные: количество переменных.

double calculate\_the\_result() – считает значение выражения.

Назначение: после получения программой значения всех переменных, считает значение выражения.

Входные данные: нет.

Выходные данные: результат вычислений.

*Операторы:*

friend ostream& operator<<(ostream& out, const tPostfix& p)

Назначение: вывод постфиксной формы выражения

Входные данные: **out** – выходной поток, **p** – элемент класса **tPostfix** на вывод

Выходные данные: постфиксная форма выражения в консоли

# Заключение

Реализация работы с выражениями при помощи постфиксной формы оказалась эффективным подходом, позволяющим удобно хранить и оперировать выражениями и считать их частные значения. Этот метод легко был создан на основе стеков, функционал которых оказался крайне полезным для выполнения задачи.

# Литература

1. Стек в c++

[https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B5%D0%BA].

1. Алгоритм преобразования в постфиксную форму

[https://www.cyberforum.ru/cpp-beginners/thread444330.html]

1. Справочник по С++ [[https://prog-cpp.ru/cpp](https://prog-cpp.ru/cpp/)].
2. Справочник по С++ [[https://en.cppreference.com/w](https://en.cppreference.com/w/)].

# Приложения

**Реализация полей класса tPostfix**

**tPostfix::tPostfix(const string& S){**

**expression = S;**

**postfix\_notation();**

**}**

**int tPostfix::get\_operator\_priority(const string op){**

**switch(op[0]) {**

**case '(':**

**case ')':**

**return 1;**

**break;**

**case '+':**

**case '-':**

**return 2;**

**break;**

**case '\*':**

**case '/':**

**return 3;**

**break;**

**default:**

**throw "not an operator";**

**break;**

**}**

**}**

**bool tPostfix::is\_operator(const int i) const{**

**char TMP[6] = { '+', '-', '(', ')', '\*', '/' };**

**for (int j = 0; j < 6; j++)**

**if (TMP[j] == expression[i])**

**return true;**

**return false;**

**}**

**bool tPostfix::is\_operator(char c) const{**

**char TMP[6] = { '+', '-', '(', ')', '\*', '/'};**

**for (int j = 0; j < 6; j++)**

**if (TMP[j] == c)**

**return true;**

**return false;**

**}**

**bool tPostfix::is\_operator(const string c) const {**

**string TMP[6] = { "+", "-", "\*", "/", "(", ")" };**

**for (int j = 0; j < 6; j++)**

**if (TMP[j] == c)**

**return true;**

**return false;**

**}**

**void tPostfix::divide\_string(){**

**string tmp = "";**

**string tmpOp = "";**

**for (int i = 0; i < expression.size(); i++){**

**if(!is\_operator(expression[i]))**

**tmp += expression[i];**

**else{**

**if (tmp != "") {**

**expression\_refined.push\_back(tmp);**

**add\_a\_new\_variable(tmp);**

**}**

**tmpOp += expression[i];**

**expression\_refined.push\_back(tmpOp);**

**tmp = "";**

**tmpOp = "";**

**}**

**}**

**if (tmp != "") {**

**expression\_refined.push\_back(tmp);**

**add\_a\_new\_variable(tmp);**

**}**

**if (vars.size() <= 1)**

**throw "less than two operators";**

**}**

**void tPostfix::add\_a\_new\_variable(const string S){**

**operand tmp(S);**

**for (int i = 0; i < vars.size(); i++)**

**if (tmp.name == vars[i].name)**

**return;**

**vars.push\_back(tmp);**

**}**

**void tPostfix::get\_values\_for\_variables(){**

**double tmp;**

**for (int i = 0; i < vars.size(); i++)**

**{**

**try {**

**vars[i].value = stoi(vars[i].name);**

**}**

**catch (...) {**

**cout << vars[i].name << " value: ";**

**cin >> tmp;**

**vars[i].value = tmp;**

**}**

**}**

**cout << endl;**

**for (int i = 0; i < vars.size(); i++)**

**{**

**try {**

**double tmp = stoi(vars[i].name);**

**}**

**catch (...) {**

**cout << vars[i].name << " = " << vars[i].value << endl;**

**}**

**}**

**}**

**void tPostfix::postfix\_notation(){**

**divide\_string();**

**TStack<string> VariableStack(expression\_refined.size()+1);**

**TStack<string> OperatorStack(expression\_refined.size()+1);**

**for (int i = 0; i < expression\_refined.size(); i++) {**

**if(!is\_operator(expression\_refined[i]))**

**VariableStack.add(expression\_refined[i]);**

**else {**

**if((OperatorStack.is\_empty()) || (expression\_refined[i] == "(")){**

**OperatorStack.add(expression\_refined[i]);**

**continue;**

**}**

**if (expression\_refined[i] == ")"){**

**while(!OperatorStack.is\_empty()){**

**string op = OperatorStack.top();**

**OperatorStack.del();**

**if (op == "(")**

**break;**

**VariableStack.add(op);**

**}**

**} else {**

**if (get\_operator\_priority(expression\_refined[i]) <= get\_operator\_priority(OperatorStack.top())){**

**while(!OperatorStack.is\_empty()){**

**if (get\_operator\_priority(expression\_refined[i])<= get\_operator\_priority(OperatorStack.top())){**

**VariableStack.add(OperatorStack.top());**

**OperatorStack.del();**

**} else break;**

**}**

**}**

**OperatorStack.add(expression\_refined[i]);**

**}**

**}**

**}**

**while (!OperatorStack.is\_empty()){**

**VariableStack.add(OperatorStack.top());**

**OperatorStack.del();**

**}**

**create\_postfixed\_expression(VariableStack);**

**}**

**void tPostfix::create\_postfixed\_expression(TStack<string>& S) {**

**TStack<string> Reversed(40);**

**while (!S.is\_empty()){**

**Reversed.add(S.top());**

**S.del();**

**}**

**while (!Reversed.is\_empty()){**

**expression\_postfixed.push\_back(Reversed.top());**

**Reversed.del();**

**}**

**}**

**double tPostfix::calculate\_the\_result(){**

**TStack<double> ValueStack(expression\_postfixed.size());**

**for (int i = 0; i < expression\_postfixed.size(); i++) {**

**if (is\_operator(expression\_postfixed[i])) {**

**double st = ValueStack.top();**

**ValueStack.del();**

**double nd = ValueStack.top();**

**ValueStack.del();**

**ValueStack.add(operation(nd, expression\_postfixed[i][0], st));**

**}**

**else {**

**string A;**

**for (int j = 0; j < vars.size(); j++)**

**if (expression\_postfixed[i] == vars[j].name)**

**ValueStack.add(vars[j].value);**

**}**

**}**

**return ValueStack.top();**

**}**

**double tPostfix::operation (double a, char op, double b){**

**switch (op){**

**case '+':**

**return a + b;**

**case '-':**

**return a - b;**

**case '\*':**

**return a \* b;**

**case '/':**

**if (b == 0)**

**throw "can't divide by zero";**

**return a/b;**

**default:**

**throw "smth";**

**}**

**}**