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**

Направление подготовки: Прикладная математика и информатика

Профиль подготовки: Общий профиль

**ОТЧЕТ**

по

лабораторной работе

на тему:

**«Вычисление арифметических выражений»**

**Выполнил(а):** студент(ка) группы 3821Б1ПМ1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Ширшов М.С.

Подпись

Нижний Новгород  
2022-2023

Оглавление:

[Введение 3](#_Toc122385763)

[1. Постановка учебно – практической задачи 4](#_Toc122385764)

[2. Руководство пользователя 5](#_Toc122385765)

[3. Руководство программиста 6](#_Toc122385766)

[3.1. Лексический анализ (токенизация) 6](#_Toc122385767)

[3.2. Преобразование в обратную польскую запись (постфиксная) 7](#_Toc122385768)

[3.3. Вычисление выражения 8](#_Toc122385769)

[3.4. Реализация Лабораторной работы 9](#_Toc122385770)

[Класс Stack 9](#_Toc122385771)

[Класс TPostfix 9](#_Toc122385772)

[Заключение 10](#_Toc122385773)

[Список литературы 11](#_Toc122385774)

[Приложение 12](#_Toc122385775)

[Приложение 1(стек) 12](#_Toc122385776)

[Приложение 2(ToPostfix) 13](#_Toc122385777)

[Реализация postfix.h 13](#_Toc122385778)

[Реализация postfix.cpp 14](#_Toc122385779)

[Приложение 3(работа с пользователем) 20](#_Toc122385780)

# Введение

Вычисление арифметический выражений очень важная вещь в современном мире. Невозможно представить работу человека без калькулятора. Решение проблемы, на первый взгляд, не совсем очевидно. За всю история программирования была создано несколько способов представления выражений в удобный вид и дальнейший подсчёт. В ходе этой работы будет разобран самый удобный и понятный способ, с помощью алгоритма обратной Польской нотации.

Работа прежде всего будет интересна программистам, так как не содержит интерфейса для пользователя. Но даже взаимодействие с консолью удобнее, чем вводить цифры в домашний калькулятор, так как здесь присутствует приоритет операций и возможность пользоваться скобками и переменными.

Работа выполнена с помощью языка C++. В дальнейшем написанное можно дополнять, улучшать и использовать для себя, как личного помощника.

## Постановка учебно – практической задачи

Разработать класс для работы с арифметическими выражениями

Класс Арифметическое Выражение должен представлять следующие операции

- Получение, хранение и возврат исходной (инфиксной) формы

- Преобразование в постфиксную форму

- Возврат постфиксной формы

- Вычисление арифметического выражения при заданных значениях операндов

Условия и ограничения

- Операнды обозначаются буквами латинского алфавита или цифрами

- Исходное арифметическое выражение синтаксически корректно

- Список операций включает: + (сложение), - (вычитание), \* (умножение), / (деление)

- В выражении могут использоваться скобки вида ()

- Значение операндов представляет собой вещественные числа

## Руководство пользователя

Для работы с программой было бы неплохо знать C++. Но даже если знаний нет, то можно запустить проект и воспользоваться базовой работой с консолью.

string expression;

double res;

setlocale(LC\_ALL, "Russian");

cout << "Введите арифметическое выражение: ";

cin >> expression;

cout << expression << endl;

TPostfix postfix(expression);

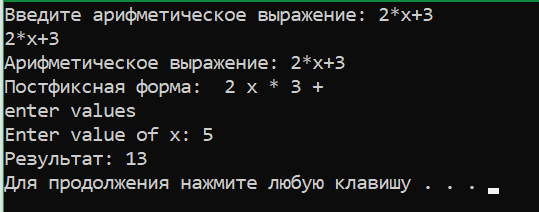
cout << "Арифметическое выражение: " << postfix.GetInfix() << endl;

cout << "Постфиксная форма: " << postfix.GetPostfix() << endl;

res = postfix.Calculate(cin,cout);

cout << res << endl;

Это код, который стоит по умолчанию. Пользователя просят ввести арифметическое выражение. Далее выводится инфиксная и постфиксная запись. Функция Calculate() попросит ввести значения переменных, а затем произведет подсчет. Итог в конце.



*Рисунок 1. Пример работы с программой*

Продублируем ограничения, наложенные на пользователя, чтобы программа работала корректно.

Условия и ограничения

- Операнды обозначаются буквами латинского алфавита или цифрами

- Исходное арифметическое выражение синтаксически корректно

- Список операций включает: + (сложение), - (вычитание), \* (умножение), / (деление)

- В выражении могут использоваться скобки вида ()

- Значение операндов представляет собой вещественные числа

## Руководство программиста

### Лексический анализ (токенизация)

Для реализации лексического анализа используется конечный автомат. Благодаря маленькому количеству операций, его можно не расписывать строго. Далее будут описаны правила введения выражения и простая таблица, по которой можно восстановить автомат при желании.

Арифметическое выражение, состоит из чисел, переменных, операндов и скобок.

-Пробелы не обрабатываются.

**-**Числа имеют вид:

<знак><целая часть>< . ><дробная часть>

Вариант <знак>< . ><дробная часть> считается ложным.

Знаком является только “-“, его может не быть.

Дробная и целая часть – набор цифр.

-Допустимые операнды: +, -, \*, /

Переменные состоят из букв латинского алфавита, символов “\_” . Имя переменной не может содержать цифры.

Допускается использование только круглых скобок. Количество открытых и закрытых скобок одинаково. Они должны так, чтобы выражение было логически верным.

Все правила составления арифметических выражений можно легко представить в виде таблицы, по которой построена программа.

|  |  |
| --- | --- |
| **Лексема** | **Следующая лексема** |
| Операнд | Операция, ) |
| Операция | Операнд, ( |
| ( | Число, ( |
| ) | Операция, ) |

*Рисунок 2. Правило написания арифметического выражения.*

Так, если после лексемы идет та, которой нет в таблицы, программа будет выдавать ошибку. "Invalid expression"

### Преобразование в обратную польскую запись (постфиксная)

После выполнения функции Parse(), программа проверила и составила набор лексем (инфиксную запись). Данная запись не дает возможности произвести подсчет выражения.

Рассмотрим алгоритм преобразования в обратную Польскую запись. Это форма записи [математических](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D0%B2%D1%8B%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5) и [логических](https://ru.wikipedia.org/wiki/%D0%9B%D0%BE%D0%B3%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D0%B2%D1%8B%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5) выражений, в которой [операнды](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D0%BD%D0%B4) расположены перед знаками [операций](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) польского логика Я. Лукасевича (1958), который изучал свойства этой записи. Обратная польская запись имеет ряд преимуществ перед инфиксной записью при выражении алгебраических формул. Во-первых, любая формула может быть выражена без скобок. Во-вторых, она [удобна для вычисления формул в машинах со стеками](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%80%D0%B0%D1%82%D0%BD%D0%B0%D1%8F_%D0%BF%D0%BE%D0%BB%D1%8C%D1%81%D0%BA%D0%B0%D1%8F_%D0%B7%D0%B0%D0%BF%D0%B8%D1%81%D1%8C#.D0.92.D1.8B.D1.87.D0.B8.D1.81.D0.BB.D0.B5.D0.BD.D0.B8.D1.8F_.D0.BD.D0.B0_.D1.81.D1.82.D0.B5.D0.BA.D0.B5). В-третьих, инфиксные операторы имеют приоритеты, которые произвольны и нежелательны. Например, мы знаем, что ab+c значит (ab)+c, а не a(b+c), поскольку произвольно было определено, что умножение имеет приоритет над сложением. Но имеет ли приоритет сдвиг влево перед операцией И? Кто знает? Обратная польская запись позволяет устранить такие недоразумения. Правда в нашей работе третье преимущество рассматриваться не будет.

Рассмотрим реализацию алгоритма ниже. В программе использовался стек.

*Цикл по массиву лексем*

*Если лексема R это*

1. *Число( операнд): R -> Res*
2. *“(“ : R -> Stack*
3. *“)” : Проход по элементам (Element) стека пока не “(“:*

*Element -> R1 ;R1 -> Res*

1. *Операция: Проход по элементам (Element) стека пока не “(“:*

*Если приоритет (R) <= приоритет (Element):*

*Element -> Res*

*R -> Stack*

*Elem of Stack -> Res*

Так, выражение 12\*(3+4 \*5) -7)) будет преобразовано в 12 3 4 5 \* + 7 - \*.

Далее именно эта запись будет использоваться для подсчета выражения.

### Вычисление выражения

После функции ToPostfix() мы имеем набор лексем в наиболее удобной записи.

Проходимся по массиву этих лексем. Если нашли операцию, то считываем два оператора до неё и выполняем операцию. На их место ставим результат вычисления. Из-за того, что все процессы выполняются стеком, это сделать очень легко. Берем сверху, считаем, кидаем обратно наверх. После считывания всех лексем, результат выражения лежит в стеке.

12 3 4 5 \* + 7 - \*.

1. 4 5 \* 4\*5=20 новая запись 12 3 20 + 7 - \*
2. 3 20 + 3+20=23 новая запись 12 23 7 - \*
3. 23 7 - 23-7=16 новая запись 12 16 \*
4. 12\*16=192 в стеке 192

Исходное выражение 12\*(3+4 \*5) -7)). Ответ 192 верен.

### Реализация Лабораторной работы

Вся реализация выполнена в классе TPostfix. Так же для работы необходим стек, работающий на динамической памяти. Несколько слов про них ниже.

#### Класс Stack

Класс реализует базовый принцип стека First in Last out. Содержит все операции необходимые для работы с лабораторной. Можно взять и положить элемент в стек.

#### Класс TPostfix

Структура класса выглядит следующим образом:

string input\_expression

lexemType //тип лексем

vector<pair<lexemType, string>> infix; //набор лексем со значениями

vector<pair<lexemType, string>> postfix; // постфиксная запись

map<string, int> priority; //приоритет операций

map<string, double> operands; //набор переменных(variable) со значениями (по умолчанию в них нули)

Основной набор операций:

void Parse(); //разложить на лексемы

void ToPostfix();//сделать обратную польскую запись для подсчета

void GetValues(istream& input, ostream& output);

double Calculate(istream& input, ostream& output);// Ввод переменных, вычисление по постфиксной форме

Так же класс содержит конструктор, где идет считывание арифметического выражения. И два метода GetInfix(), GetPostfix(). Они выполняют возврат инфиксной и постфиксной записи соответсвенно.

Все методы можно подробнее изучить в приложении к работе.

# Заключение

Была разработана программа на языке C++ для вычисления арифметических выражений. Уникальность работы, в том, что выражения пользователь может вводить с приоритетом операций и переменными. Так же изучен алгоритм обратной польской записи. Вся реализация включена в классе TPostfix, который в дальнейшем можно дополнить и подключить в свои проекты.

# Список литературы

1. Hamblin C. L. Language and the Theory of Information : дис. – London School of Economics and Political Science (University of London), 1957.

# Приложение

## Приложение 1(стек)

Реализация Stack.h

#ifndef \_\_STACK\_H\_\_

#define \_\_STACK\_H\_\_

#include <vector>

#include <cstddef>

#include <stdexcept>

using namespace std;

template <typename T>

class Stack

{

vector<T> vec;

size\_t sz;

public:

Stack()

{

sz = 0;

}

Stack(size\_t size) : Stack()

{

vec.reserve(size);

}

Stack(size\_t size, const T\* arr) : Stack(size)

{

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

vec.push\_back(arr[i]);

sz = size;

}

size\_t size()

{

return sz;

}

bool empty()

{

return sz == 0;

}

bool operator==(const Stack& st)

{

if (this == &st)

return true;

return st.vec == vec;

}

void push(const T& elem)

{

vec.push\_back(elem);

sz++;

}

T top()

{

if (empty())

throw std::out\_of\_range("Empty stack");

return vec[sz - 1];}

void pop()

{

if (empty())

throw std::out\_of\_range("Empty stack");

vec.pop\_back();

sz--;

}

};

#endif

## Приложение 2(ToPostfix)

#### Реализация postfix.h

#include<iostream>

#include <string>

#include "stack.h"

#include <vector>

#include <map>

//#include <stack>

using namespace std;

class TPostfix

{

string input\_expression;

enum lexemType {

nothing,//начало выражения

number, // 4; -4; .4;

variable, //переменная\_\_\_\_ a;my\_argument - right\_\_\_\_\_-a;a1 - wrong!

operation, //+, -, \*, /

operator\_open, // (

operator\_close // )

};

vector<pair<lexemType, string>> infix; //набор лексем со значениями

vector<pair<lexemType, string>> postfix;

//static

map<char, int> priority;

map<string, double> operands; //набор переменных(variable) со значениями (по умолчанию в них нули)

bool isNumber(char c); // 0 ... 9

bool isLetter(char c); // a ... z, A ... Z, \_

bool isOperation(char c); // +, -, \*, /

bool isMinus(char c); // -

bool isPoint(char c); // .

bool isOperator\_open(char c); // (

bool isOperator\_close(char c); // )

void Parse(); //разложить на лексемы

void GetValues(istream& input, ostream& output);

public:

void ToPostfix();//сделать обратную польскую запись для подсчета

TPostfix(string \_input\_expression) :input\_expression(\_input\_expression) {

this->priority = { {'+',1},{'-',1},{'\*',2},{'/',2}};

if (input\_expression.empty())

throw invalid\_argument("Creating arithmetic expression from an empty string");

ToPostfix();

}

string GetInfix() { return input\_expression; }

string GetPostfix() {

string ans;

for (auto& i : postfix) {

ans += ' ';

ans += i.second;

}

return ans;

}

double Calculate(istream& input, ostream& output);// Ввод переменных, вычисление по постфиксной форме

};

#### Реализация postfix.cpp

#include "postfix.h"

bool TPostfix::isNumber(char c) {

return '0' <= c && c <= '9';

}

bool TPostfix::isLetter(char c) {

return 'a' <= c && c <= 'z' || 'A' <= c && c <= 'Z' || c == '\_';

}

bool TPostfix::isOperation(char c) {

return c == '+' || c == '-' || c == '\*' || c == '/';

}

bool TPostfix::isMinus(char c) {

return c == '-';

}

bool TPostfix::isOperator\_open(char c) {

return c == '(';

}

bool TPostfix::isOperator\_close(char c) {

return c == ')';

}

bool TPostfix::isPoint(char c) {

return c == '.';

}

void TPostfix::Parse() {

lexemType type = nothing;

int brackets = 0;// open++, close--

int left = 0;//левая граница лексемы, i-будет свегда правой границей

for (int i = 0; i < input\_expression.size(); i++) {

char c = input\_expression[i];

if (c == ' ')

continue;

switch (type) {

case (nothing):

// начало выражения

if (isOperator\_open(c)) {

left = i;

type = operator\_open;

}

else if (isNumber(c) || isPoint(c) || isMinus(c)) {

left = i;

type = number;

}

else if (isLetter(c)) {

left = i;

type = variable;

}

else {

throw invalid\_argument("Invalid expression");

}

break;

case (number):

if ((isPoint(c)) & (isPoint(input\_expression[i-1])))//две точки подряд нельзя

throw invalid\_argument("Invalid expression");

if ((isPoint(c) & (!isMinus(input\_expression[i - 1]))) || isNumber(c))

continue;//всё ещё число

infix.emplace\_back(number, input\_expression.substr(left, i - left));// добавили

if (isOperator\_close(c)) {

left = i;

type = operator\_close;

}

else if (isOperation(c)) {

left = i;

type = operation;

}

else {

throw invalid\_argument("Invalid expression");

}

break;

case (variable):

if (isLetter(c))

continue;

infix.emplace\_back(variable, input\_expression.substr(left, i - left));

operands.insert({ input\_expression.substr(left,i - left), 0.0 });

if (isOperator\_close(c)) {

left = i;

type = operator\_close;

}

else if (isOperation(c)) {

left = i;

type = operation;

}

else {

throw invalid\_argument("Invalid expression");

}

break;

case (operation):

infix.emplace\_back(operation, input\_expression.substr(left, i - left));

if (isOperator\_open(c)) {

left = i;

type = operator\_open;

}

else if (isNumber(c) || isPoint(c)) {

left = i;

type = number;

}

else if (isLetter(c)) {

left = i;

type = variable;

}

else {

throw invalid\_argument("Invalid expression");

}

break;

case (operator\_open):

brackets++;

infix.emplace\_back(operator\_open, input\_expression.substr(left, i - left));

if (isOperator\_open(c)) {

left = i;

type = operator\_open;

}

else if (isNumber(c) || isPoint(c) || isMinus(c)) {

left = i;

type = number;

}

else if (isLetter(c)) {

left = i;

type = variable;

}

else {

throw invalid\_argument("Invalid expression");

}

break;

case (operator\_close):

brackets--;

infix.emplace\_back(operator\_close, input\_expression.substr(left, i - left));

if (isOperator\_close(c)) {

left = i;

type = operator\_close;

}

else if (isOperation(c)) {

left = i;

type = operation;

}

else {

throw invalid\_argument("Invalid expression");

}

break;

}

}

infix.emplace\_back(type, input\_expression.substr(left, input\_expression.size() - left));//последняя лексема

if (type == variable)

operands.insert({ input\_expression.substr(left,input\_expression.size() - left), 0.0 });

if (type == operator\_close)

brackets--;

if (type == operator\_open)

brackets++;

if (brackets != 0)

throw invalid\_argument("Invalid expression: troubles with brackets");

if ((type != operator\_close && type != number && type != variable)) {

throw invalid\_argument("Invalid expression: invalid ending");

}

}

void TPostfix::ToPostfix() {

Parse();

Stack<pair<lexemType, string>> st;

pair<lexemType, string> stelem;

for (auto& item : infix) {

switch (item.first) {

case operator\_open:

st.push(item);

break;

case operator\_close:

stelem = st.top();

st.pop();

while (stelem.first != operator\_open) {

postfix.emplace\_back(stelem);

stelem = st.top();

st.pop();

}

break;

case operation:

while (!st.empty()) {

stelem = st.top();

st.pop();

if (priority[item.second[0]] <= priority[stelem.second[0]])

postfix.emplace\_back(stelem);

else {

st.push(stelem);

break;

}

}

st.push(item);

break;

default:

postfix.emplace\_back(item);

}

}

while (!st.empty()) {

stelem = st.top();

st.pop();

postfix.emplace\_back(stelem);

}

}

void TPostfix::GetValues(istream& input, ostream& output) {

output << "enter values" << endl;

for (auto& op : operands) {

output << "Enter value of " << op.first << ": ";

//input>>op.second;

double val;

input >> val;

op.second = val;

}

}

double TPostfix::Calculate(istream& input, ostream& output) { // Ввод переменных, вычисление по постфиксной форме

GetValues(input, output);

Stack<double> st;

double leftoperand, rightoperand;

for (auto& lexem : postfix) {

switch (lexem.second[0]) {

case'+':

rightoperand = st.top();

st.pop();

leftoperand = st.top();

st.pop();

st.push(leftoperand + rightoperand);

break;

case '-':

rightoperand = st.top();

st.pop();

leftoperand = st.top();

st.pop();

st.push(leftoperand - rightoperand);

break;

case '\*':

rightoperand = st.top();

st.pop();

leftoperand = st.top();

st.pop();

st.push(leftoperand \* rightoperand);

break;

case '/':

rightoperand = st.top();

st.pop();

leftoperand = st.top();

st.pop();

st.push(leftoperand / rightoperand);

break;

default:

if (lexem.first == variable)

st.push(operands[lexem.second]);

else

st.push(stod(lexem.second));

}

}

return st.top();

}

## Приложение 3(работа с пользователем)

Реализация sample\_postfix.cpp

#include <iostream>

#include <string>

#include "postfix.h"

using namespace std;

int main()

{

string expression;

double res;

setlocale(LC\_ALL, "Russian");

cout << "Введите арифметическое выражение: ";

cin >> expression;

cout << expression << endl;

TPostfix postfix(expression);

cout << "Арифметическое выражение: " << postfix.GetInfix() << endl;

//postfix.ToPostfix();

cout << "Постфиксная форма: " << postfix.GetPostfix()<< endl;

res = postfix.Calculate(cin, cout);

cout <<"Результат: "<< res << endl;

return 0;

}