Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное   
учреждение высшего профессионального образования

Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

**Отчет по лабораторной работе №3**

**«Вычисление арифметических выражений (стеки)»**

**Выполнил**:студент группы 381703-2

Капустин Григорий Константинович

**Проверил**:

Доцент кафедры МОСТ, к.т.н.

Сысоев А.B.

Нижний Новгород

2018

Содержание

[Введение 3](#_Toc270962758)

[Постановка задачи 4](#_Toc270962759)

[Руководство пользователя 5](#_Toc270962760)

[Руководство программиста 6](#_Toc270962761)

[Описание структур программы 7](#_Toc270962762)

[Описание алгоритмов 8](#_Toc270962763)

[Заключение 10](#_Toc270962765)

[Литература 11](#_Toc270962766)

[Приложения 1](#_Toc270962767)2

# Введение

Лабораторная работа направлена на практическое освоение динамической структуры данных Стек. С этой целью в лабораторной работе изучаются различные варианты структуры хранения стеков и разрабатываются методы и программы решения ряда задач с использованием стеков. В качестве области приложений выбрана тема вычисления арифметических выражений, возникающей при трансляции программ на языке программирования высокого уровня в исполняемые программы.

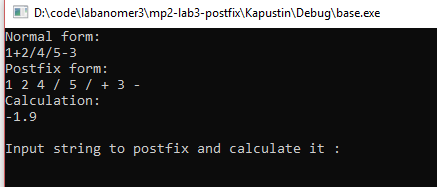
При вычислении произвольных арифметических выражений возникают две основные задачи: проверка корректности введённого выражения и выполнение операций в порядке, определяемом их приоритетами и расстановкой скобок. Существует алгоритм, позволяющий реализовать вычисление произвольного арифметического выражения за один просмотр без хранения промежуточных результатов. Для реализации данного алгоритма выражение должно быть представлено в постфиксной форме. Рассматриваемые в данной лабораторной работе алгоритмы являются начальным введением в область машинных вычислений.

# Постановка задачи

В рамках лабораторной работы ставится задача реализации программ, обеспечивающих поддержку стеков, и разработки программных средств, производящих обработку арифметических выражений, включая проверку правильности записи выражения, перевод в постфиксную форму и вычисление результата.

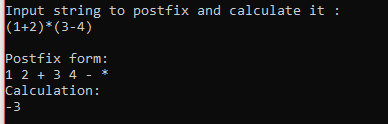
# Руководство пользователя

При запуске программы «sample» появляется консольное окно с примером работы алгоритма, а именно: перевод выражения из обычной формы в постфиксную, вычисление этого выражения. Также пользователю будет предложено ввести свое выражение.



**Рис. 1. Программа запрашивает ввод данных**

После ввода для выражения будет составлена постфиксная форма и будет выполнено его вычисление.



**Рис. 2. Вывод результатов**

# Руководство программиста

В программе реализованы два класса под названием «Стек» и «Постфикс».

**Класс «Стек» - шаблонный, имеющий три приватных поля:**

1. pMem: массив элементов шаблонного типа,
2. size: хранит размер стека,
3. topIndex: переменная, хранящая индекс верхнего элемента (-1, если стек пуст)

**В классе «Стек» имеются следующие методы:**

1. isEmpty: возвращает 1 – если стек не содержит элементов, 0 – если есть элементы,
2. IsFull: возвращает 1 – если стек полон, 0 – если не полон,
3. push\_back: в качестве аргумента принимает элемент шаблонного типа, добавляет его в стек, ничего не возвращает,
4. pop: возвращает верхний элемент, при этом удаляя его из стека,
5. top: возвращает верхний элемент, при этом не удаляя его из стека,
6. get\_size: возвращает актуальное количество элементов в стеке,

**Класс «Постфикс» имеет следующие приватные поля:**

1. infix: строка, содержащая исходное выражение,
2. postfix: строка, содержащая постфиксную форму,
3. operations: строка, содержащая все операции, поддерживаемые классом postfix,

**Также содержатся следующие приватные методы:**

1. prior – принимает в качестве аргумента строку и возвращает её приоритет в соответствии с алгоритмом,,
2. splitInfix, splitPostfix: разбиение входного выражения (обычного и постфиксного) на подстроки с операциями и операндами,
3. calculateOperands: принимает в качестве аргументов две вещественных переменных и строку, являющейся операцией, возвращает результат вычисления между этими переменными,
4. inputVar: метод, считывающий переменные в арифметическом выражении,
5. isNum: принимает в качестве аргумента – символ, возвращает 1, если символ – цифра, 0, если нет,
6. isOperation: принимает в качестве аргумента символ, возвращает 1 если символ является операцией, 0, если не является таковой,

**Публичные методы класса «Постфикс»:**

1. isCorrect: принимает в качестве аргумента строку, возвращает 1, если строка – корректное выражение, 0 – если нет,
2. toPostfix: преобразование строки, полученной в качестве аргумента в постфиксную форму,
3. getPostfix: возвращает строку postfix,
4. calculate: вычисляет выражение, находящееся в поле postfix, возвращает результат вычислений в вещественном типе,

## Описание структуры программы

В решении содержатся:

1. проект base, содержащий классы постфикс, стек и программа-пример:
   1. postfix.h -
   2. postfix.cpp
   3. stack.h
   4. sample\_postfix.cpp
2. проект base\_test, содержащий модульные тесты для вышеперечисленных классов.
   1. Test\_tpostfix.cpp
   2. Test\_tstack.cpp

## Описание алгоритмов

**Стек** – динамическое множество, добавление элементов в котором происходит по принципу «последний вошел, первый вышел» (Last-In-First-Out или LIFO). Это значит, что мы будем иметь доступ только к последнему добавленному элементу.

Получим следующие методы:

1. void push\_back(uni bf)

Добавляет шаблонный элемент на вершину стека, за счет присваивания верхнего элемента bf, смещается на единицу вверх topIndex

1. uni pop()

Возвращается верхний элемент, верхний индекс уменьшается на единицу, за счет этого и происходит «удаление» элемента.

1. uni top()

Просто возвращает верхний элемент массива.

1. int getSize()

Возвращает количество элементов.

За счет того, что операции добавление и удаления элемента производятся без перевыделения памяти, а за счет переноса верхнего индекса в уже созданном массиве, сложность операций – O(1).

**Постфиксная запись** представляет собой такую запись арифметического выражения, в которой сначала записываются операнды, а затем – знак операции. Например, для выражения a + b \* c постфиксная запись будет a b c \* +. Здесь операндами операции \* будут b и c (два ближайших операнда), а операндами операции + будут а и составной операнд b c \*. Эта запись удобна тем, что она не требует скобок. Например, для выражения (a + b) \* c постфиксная запись будет a b + c \*. В этой записи не требуется ставить скобки для того, чтобы изменить порядок вычисления, зависящий от приоритета операций, как в исходном выражении, а значит возможно вычисление за один проход по строке.

Для реализации данной задачи потребуются следующие методы:

1. void toPosfix(string str)

На вход алгоритма поступает строка символов, на выходе должна быть получена строка с постфиксной формой. Каждой операции и скобкам приписывается приоритет.

**Таблица 3. Приоритет операций**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ( | ) | + | - | \* | / |
| 0 | 1 | 2 | 2 | 3 | 3 |

Если входная строка некорректна – то «бросается» исключение. Входная строка просматривается посимвольно слева направо до достижения конца строки. Операндами будем считать любую последовательность символов входной строки, не совпадающую со знаками определённых в строке operations(представлена в виде таблицы № 3) . Операнды по мере их появления переписываются в выходную строку. При появлении во входной строке операции, происходит вычисление приоритета данной операции. Знак данной операции помещается в стек, если:

* Приоритет операции равен 0 (это « ( » ),
* Приоритет операции строго больше приоритета операции, лежащей на вершине стека
* Стек пуст

В противном случае из стека извлекаются все знаки операций с приоритетом больше или равным приоритету текущей операции. Они переписываются в выходную строку, после чего знак текущей операции помещается в стек.

Имеется особенность в обработке закрывающей скобки. Появление закрывающей скобки во входной строке приводит к выталкиванию и записи в выходную строку всех знаков операций до появления открывающей скобки. Открывающая скобка из стека выталкивается, но в выходную строку не записывается. Таким образом, ни открывающая, ни закрывающая скобки в выходную строку не попадают.

После просмотра всей входной строки происходит последовательное извлечение всех элементов стека с одновременной записью знаков операций, извлекаемых из стека, в выходную строку.

1. double calculate()

Алгоритм вычисления арифметического выражения за один просмотр входной строки основан на использовании постфиксной формы записи выражения и работы со стеком. Входным данным служит строка символов, полученная в результате работы алгоритма из 2.2.3, выходным – результат вычисления выражения.

Выражение просматривается посимвольно слева направо. При обнаружении операнда производится перевод его в числовую форму и помещение в стек (если операнд не является числом, то вычисление прекращается с выдачей сообщения об ошибке.) При обнаружении знака операции происходит извлечение из стека двух значений, которые рассматриваются как операнд2 и операнд1 соответственно, и над ними производится обрабатываемая операция. Результат этой операции помещается в стек. По окончании просмотра всего выражения из стека извлекается окончательный результат.

# Заключение

Была изучена, реализована и применена динамическая структура данных стек. На примере постфиксной формы стала понятно очевидна выгода, в виде упрощения и удобства написания некоторых алгоритмов при использовании стека.

# Литература

1. Бьерн Страуструп. «Язык программирования С++»,
2. Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн, «Introduction to Algorithms »,
3. Барышева И.В., Мееров И.Б., Сысоев А.В., Шестакова Н.В. Под редакцией Гергеля В.П. «Лабораторный практикум. Учебно-методическое пособие».

# 

# Приложение

**Модуль “stack.h”**

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

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

const int MaxStackSize = 100;

const int MaxMemSize = 10;

template <class uni>

class TStack

{

uni \*pMem;

int size;

int topIndex;

public:

TStack(int \_size = MaxStackSize) :

size(\_size)

{

if (\_size<1 || \_size >MaxStackSize)

throw "incorrect size";

pMem = new uni[size];

topIndex = -1;

}

~TStack()

{

delete[] pMem;

}

bool isEmpty()

{

return topIndex == -1;

}

bool isFull()

{

return topIndex == (size - 1); // 4 elems, top index = 3... 3==(4-1)

}

void push\_back(uni bf)

{

if (!isFull())

{

topIndex++;

pMem[topIndex] = bf;

}

}

uni pop()

{

if (topIndex >= 0)

{

int tmp = topIndex;

topIndex--;

return pMem[tmp];

}

}

uni top()

{

if (topIndex >= 0)

return pMem[topIndex];

}

int getSize()

{

return topIndex + 1;

}

};

#endif

**Модуль “postfix.h”**

#ifndef \_\_POSTFIX\_H\_\_

#define \_\_POSTFIX\_H\_\_

#include <string>

#include <vector>

#include <iostream>

#include "stack.h"

using namespace std;

class TPostfix

{

const string operations = "+-\*/()";

short int prior(string str);

vector<string> splitInfix(const string& str);

vector<string> splitPostfix(const string& str);

double calculateOperands(string str, double op1, double op2);

double inputVar(const string& str);

bool isNum(const string& str);

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

string postfix;

string infix;

public:

TPostfix() {}

bool isCorrect(const string& str);

void toPostfix(const string& str);

string getPostfix();

double calculate();

};

#endif

**Модуль “postfix.cpp”**

#include "postfix.h"

#include "stack.h"

short int TPostfix::prior(string str)

{

short int prior = 0;

char c = str[0];

switch (c)

{

case '(':

prior = 0;

break;

case (')'):

prior = 1;

break;

case ('+'):

prior = 2;

break;

case '-':

prior = 2;

break;

case '\*':

prior = 3;

break;

case '/':

prior = 3;

break;

}

return prior;

}

vector<string>TPostfix::splitInfix(const string& str)

{

string buf;

vector<string> res;

size\_t npos = -1;

for (char c : str)

{

if (c == ' ')

continue;

if (operations.find(c) == npos)

buf += c;

else

{

if (!buf.empty())

res.push\_back(buf);

buf = c;

res.push\_back(buf);

buf.clear();

}

}

if (!buf.empty())

res.push\_back(buf);

return res;

}

vector<string>TPostfix::splitPostfix(const string& str)

{

string buf;

vector<string> res;

size\_t npos = -1;

for (char c : str)

{

if (operations.find(c) == npos)

if (c == ' ')

{

if (!buf.empty())

res.push\_back(buf);

buf.clear();

}

else

buf += c;

else

{

if (!buf.empty())

res.push\_back(buf);

buf = c;

res.push\_back(buf);

buf.clear();

}

}

if (!buf.empty())

res.push\_back(buf);

return res;

}

double TPostfix::calculateOperands(string str, double op1, double op2)

{

char c = str[0];

switch (c)

{

case '+':

return op1 + op2;

case '-':

return op1 - op2;

case '\*':

return op1 \* op2;

case '/':

return op1 / op2;

}

}

double TPostfix::inputVar(const string& str)

{

double res;

cout << "\t" << str << "=";

cin >> res;

return res;

}

bool TPostfix::isNum(const string& str)

{

for (char c : str)

if (c < 48 || c>57)

return 0;

return 1;

}

bool TPostfix::isOperation(char c)

{

size\_t npos = -1;

if (operations.find(c) == npos || operations.find(c) == 4 || operations.find(c) == 5)

return 0;

return 1;

}

bool TPostfix::isCorrect(const string& str)

{

int counter = 0;

size\_t npos = -1;

if (isOperation(str[0]) || isOperation(str[str.size() - 1]))

return 0;

if (str[0] == '(')

counter++;

if (str[0] == ')')

counter--;

for (int i = 1; i < str.size(); i++)

{

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

counter++;

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

counter--;

if (isOperation(str[i]) && isOperation(str[i - 1]))

return 0;

}

if (counter == 0)

return 1;

else

return 0;

}

void TPostfix::toPostfix(const string& str)

{

if (!isCorrect(str))

throw "error";

postfix.clear();

infix = str;

vector<string> components = splitInfix(str);

TStack <string> St;

size\_t npos = -1;

for (string c : components)

if (operations.find(c) != npos)

{

if (c != ")")

{

if (St.isEmpty() || prior(c) == 0 || prior(c) > prior(St.top()));

else

{

while (!St.isEmpty())

if (prior(St.top()) >= prior(c))

{

postfix += St.pop();

postfix += ' ';

}

else

break;

}

St.push\_back(c);

}

else

{

int it = St.getSize();

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

if (St.top() != "(")

{

postfix += St.pop();

postfix += ' ';

}

else

{

St.pop();

break;

}

}

}

else

{

postfix += c;

postfix += ' ';

}

while (!St.isEmpty())

{

postfix += St.pop();

postfix += ' ';

}

}

double TPostfix::calculate()

{

if (postfix.empty())

throw "error";

vector<string> components = splitPostfix(postfix);

size\_t npos = -1;

double res = 0;

TStack<double> stack;

double op2;

double op1;

double var;

for (string c : components)

{

if (operations.find(c) == npos)

{

if (!isNum(c))

{

var = inputVar(c);

stack.push\_back(var);

}

else

stack.push\_back(stod(c));

}

else

{

op2 = stack.pop();

op1 = stack.pop();

stack.push\_back(calculateOperands(c, op1, op2));

}

}

return stack.pop();

}

string TPostfix::getPostfix()

{

return postfix;

}

**Модуль “sample\_postfix.cpp”**

#include <iostream>

#include <string>

#include "postfix.h"

using namespace std;

int main()

{

string expression;

TPostfix postfix;

vector<string> samples;

samples.push\_back("1+2/4/5-3");

for (string c : samples)

{

cout << "Normal form:" << endl << c << endl;

postfix.toPostfix(c);

cout << "Postfix form:" << endl << postfix.getPostfix() << endl;

cout << "Calculation:" << endl << postfix.calculate() << endl;

cout << endl;

}

cout << "Input string to postfix and calculate it :" << endl;

cin >> expression;

postfix.toPostfix(expression);

cout << endl << "Postfix form:" << endl << postfix.getPostfix() << endl;

cout << "Calculation:" << endl << postfix.calculate() << endl;

cout << endl;

system("pause");

return 0;

}