Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное   
учреждение высшего образования

«Национальный исследовательский

Нижегородский государственный университет им. Н.И. Лобачевского»

Институт информационных технологий математики и механики

**Отчет по учебной практике**

**на тему**

**Вычисление арифметических выражений**

**Выполнил**:студент группы 381703-2

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Долгополов Д.А.

Подпись

**Проверил**:

Доцент каф. МОСТ, к.т.н.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Сысоев А.В

Подпись

Нижний Новгород

2018

Содержание

[Введение 3](#_Toc529573303)

[Постановка задачи 4](#_Toc529573304)

[Руководство пользователя 5](#_Toc529573305)

[Руководство программиста 7](#_Toc529573306)

[Описание структуры программы 8](#_Toc529573307)

[Описание алгоритмов 8](#_Toc529573308)

[Заключение 10](#_Toc529573309)

[Литература 11](#_Toc529573310)

[Приложение 12](#_Toc529573311)

## 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

Требуется разработать систему классов, обеспечивающих поддержку стека и программных средств, производящих обработку арифметических выражений, перевод в из инфиксной в постфиксную форму и вычисление результата.

Также, осуществляется проверка выражения посредством проверки правильности расстановки скобок и знаков операций.

Перевод выражения в постфиксную форму осуществляется при корректном вводе, а вычисление - при корректном вводе численных значений.

# Руководство пользователя

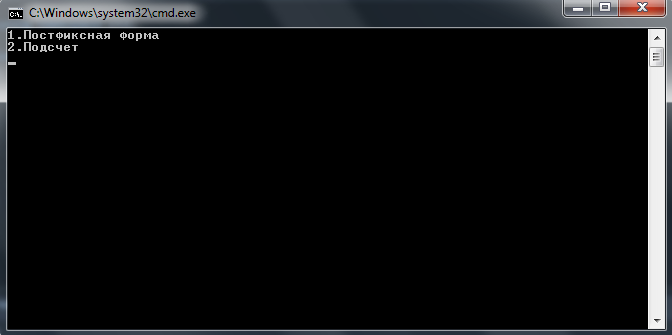
При запуске программы пользователь должен будет выбрать действие: перевести введенное выражение в постфиксную форму или посчитать его значение.

Рис. 1. Меню запуска программы

При выборе "1" пользователю будет предложено ввести арифметическое выражение, и если оно будет корректно, то будет выведена его постфиксная форма, где все элементы будут разделены запятыми.

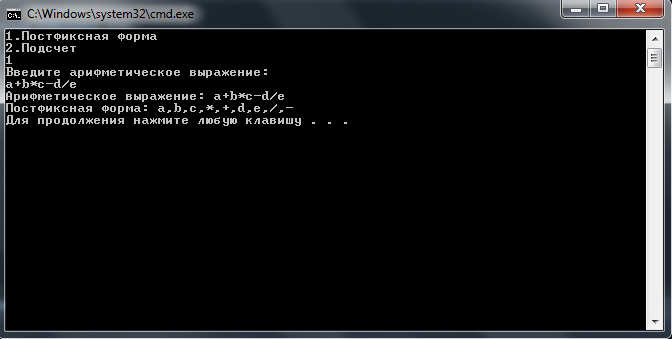


Рис. 2. Вывод постфиксной формы

При выборе "2" пользователю будет предложено ввести арифметическое выражение, и если оно будет корректно, то будет выведено его значение.

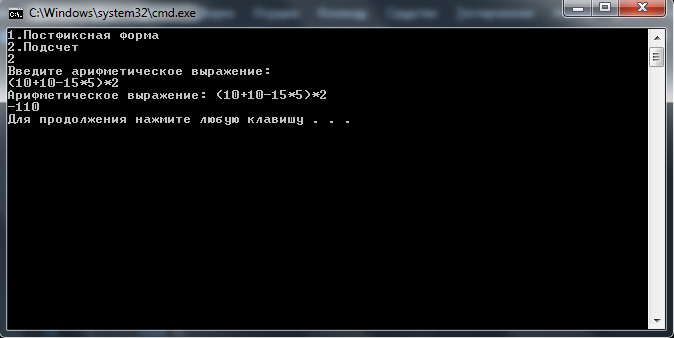


Рис. 3. Вывод результата подсчета

# Руководство программиста

В программе реализованы два класса под названием «**TStack**» и «**TPostfix**».

Класс «**TStack**» следующие поля:

Приватные:

1. pMem - указатель на объект шаблонного типа, отвечающий за хранение элементов в стеке.
2. size - переменная типа int, отвечающая за размер стека.
3. top - переменная типа int, отвечающая за индекс последнего элемента в стеке.
4. count - переменная типа int, отвечающая за количество элементов в стеке.

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

Публичные:

1. Необходимые конструкторы и деструктор.
2. Max\_size - метод, возвращающий максимальный размер стека.
3. Size - метод, возвращающий имеющееся количество элементов стека.
4. IsEmpty - метод, проверяющий стек на пустоту.
5. IsFull - метод, проверяющий стек на полноту.
6. push - метод, записывающий элемент в стек.
7. pop - метод, возвращающий значение последнего элемента в стеке и уменьшающий индекс.
8. Top\_el - метод, возвращающий значение последнего элемента в стеке и не уменьшающий индекс.
9. Prior - метод, определяющий приоритет операций в стеке.

Класс «**TPostfix**» имеет следующие поля:

Приватные:

1. infix - переменная, хранящая исходное арифметическое выражение.
2. postfix - переменная, хранящая выражение в постфиксной форме.
3. i - вспомогательная переменная.
4. prior - переменная, которая хранит приоритет операции в выражении.

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

Приватные:

1. NotOperation - метод, проверяющий является ли элемент операцией или операндом.
2. IsValidByBrackets - метод, проверяющий корректность расставленных скобок в выражении.
3. IsValidByOpers - метод, проверяющий корректность количества операций и операндов.
4. IsValidByPos - метод, проверяющий корректность записи выражения.
5. IsValidInTotal - метод, проверяющий правильность выражения в целом.
6. Helper - вспомогательный метод для проведения подсчета.

Публичные:

1. Конструктор.
2. GetInfix - метод, возвращающий изначально введенное арифметическое выражение.
3. GetPostfix - метод, возвращающий постфиксную форму введенного арифметического выражения.
4. ToPostfix - метод, выполняющий перевод введенного арифметического выражения из инфиксной формы в постфиксную.
5. Calculate - метод, выполняющий подсчет введенного арифметического выражения.

# Описание структуры программы

Программа состоит из трех проектов: base, base\_test и gtest.

В проекте base содержаться:

1. Заголовочный файл "stack.h", в котором объявлен и реализован класс "TStack".
2. Заголовочный файл "postfix.h", в котором объявлен класс "TPostfix".
3. Файл "postfix.cpp", в котором реализован класс "TPostfix".
4. Файл "sample\_postfix.cpp", в котором реализована функция main, представляющая пример пользовательского меню.

В проекте base\_test содержаться файлы "test\_main.cpp", "test\_tpostfix.cpp", "test\_tstack.cpp", в которых реализованы тесты для проверки работоспособности программы.

В проекте gtest содержаться необходимые файлы для поддержки работы Google Test.

# Описание алгоритмов

Алгоритмы реализации методов класса TStack:

1. Метод Max\_Size - возвращает максимальный размер стека.
2. Метод Size - возвращает число элементов в стеке.
3. Метод IsFull - сравнивает индекс верхнего элемента с size - 1.
4. Метод IsEmpty - сравнивает индекс верхнего элемента top с -1.
5. Метод push - заносит элемент в стек и увеличивает индекс верхнего элемента top на 1 и увеличивает количество элементов count на 1.
6. Метод pop - возвращает значение верхнего элемента и уменьшает индекс top на 1.
7. Метод Top\_el - возвращает значение верхнего элемента, но не уменьшает индекс top.
8. Метод Prior - возвращает значение приоритета для операций, "1" для "+" и "-", "2" для "\*" и "/".

Алгоритмы реализации методов класса TPostfix:

1. Метод NotOperation - сравнивает элемент с символами операций, возвращает true если элемент не является операцией, false если является.
2. Метод IsValidByBrackets - подсчитывает количество левых и правых скобок, возвращает true если их количество равно, false если не равно.
3. Метод IsValidByOpers - подсчитывает количество операций и операндов, возвращает true если количество операций на одну меньше, чем операндов, false если это не так.
4. Метод IsValidByPos - сравнивает текущий и предыдущий элементы, возвращает false если они оба являются операциями или левой и правой скобками, true если это не так.
5. Метод IsValidInTotal - проверяет на правильность с помощью трех предыдущих методов, возвращает true если они все вернули true, false если хотя бы один вернул false.
6. Метод Helper - совершает выбранную операцию между двумя выбранными элементами.
7. Метод GetInfix - возвращает введенную строку.
8. Метод GetPostfix - возвращает строку с постфиксной формой введенной строки.
9. Метод ToPostfix - проверяет поэлементно введенную строку и разделяет ее на операнды и операции. Все операнды попадают в постфиксную форму, операции сначала попадают в стек, откуда при добавлении операции большего приоритета или находясь между скобок или при конце введенной строки попадают в постфиксную форму. И операнды, и операции разделяются запятыми.
10. Метод Calcuate - проверяет поэлементно строку постфиксной формы, при встрече операнда - заносит его в стек, при встрече операции - выполняет ее с двумя верхними элементами стека с помощью метода Helper. Повторяется до тех пор, пока не кончится постфиксная форма и в стеке не останется один элемент, который метод возвращает.

# Заключение

Данная лабораторная работа познакомила нас со структурой данных стек, методами ее реализации и практическим применением. С помощью стека поставленная задача может быть решена более эффективно, чем без его использования.

# Литература

1. «C++. Руководство для начинающих», Герберт Шилдт.
2. «Лабораторный практикум. Учебно-методическое пособие», Барышева И.В., Мееров И.Б., Сысоев А.В., Шестакова Н.В. Под редакцией Гергеля В.П.

# Приложение

**"Stack.h"**

ifndef \_\_STACK\_H\_\_

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

const int MaxStackSize = 100;

template <class T>

class TStack

{

T \*pMem;

int size;

int top;

int count=0;

public:

TStack(int \_size)

{

size = \_size;

top = -1;

if ((size < 1) || (size > MaxStackSize))

throw size;

pMem = new T[size];

}

TStack(TStack &s)

{

size = s.size;

top = s.top;

pMem = new T[size];

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

pMem[i] = s.pMem[i];

}

~TStack()

{

delete [] pMem;

}

int Max\_size()

{

return size;

}

int Size()

{

return count;

}

bool IsFull()

{

return top == size -1;

}

bool IsEmpty()

{

return top == -1;

}

void push(const T &a)

{

if (IsFull())

throw "Error";

count++;

top++;

pMem[top] = a;

}

T pop()

{

if (IsEmpty())

throw "Error";

count--;

return pMem[top--];

}

T Top\_el()

{

if (IsEmpty())

throw "Error";

return pMem[top];

}

int Prior()

{

if (IsEmpty())

throw "Error";

if ((pMem[top] == '+') || (pMem[top] == '-'))

return 1;

if ((pMem[top] == '\*') || (pMem[top] == '/'))

return 2;

}

};

#endif

**"Postfix.h"**

ifndef \_\_POSTFIX\_H\_\_

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

#include <string>

#include "stack.h"

using namespace std;

class TPostfix

{

string infix;

string postfix;

int i = 0;

int prior;

bool NotOperation(char c);

bool IsValidByBrackets();

bool IsValidByOpers();

bool IsValidByPos();

bool IsValidInTotal();

double Helper(double a, double b, char c);

public:

TPostfix(string a = "a+b")

{

infix = a;

}

string GetInfix() { return infix; }

string GetPostfix() { return postfix; }

void ToPostfix();

double Calculate();

};

#endif

**"Postfix.cpp"**

#include "postfix.h"

#include "stack.h"

bool TPostfix::NotOperation(char c)

{

if ((c != '+') && (c != '-') && (c != '/') && (c != '\*'))

return true;

else

return false;

}

void TPostfix::ToPostfix()

{

TStack<char> a(MaxStackSize);

do

{

if (!IsValidInTotal())

throw "Error";

char c = infix[i];

if ((c == '+') || (c == '-'))

prior = 1;

if ((c == '\*') || (c == '/'))

prior = 2;

if ((c == '(') || (c == ')'))

prior = 0;

if (NotOperation(c) && (c != '(') && (c != ')'))

{

int j = i;

while (NotOperation(infix[j]) && (infix[j] != ')') && (j != infix.size()))

{

postfix.push\_back(infix[j]);

j++;

}

i = j - 1;

postfix.push\_back(',');

}

else

{

if (c == '(')

a.push(c);

if (a.IsEmpty())

a.push(c);

else

{

if (c == ')')

{

while ((!a.IsEmpty()) && (a.Top\_el() != '('))

{

postfix.push\_back(a.pop());

postfix.push\_back(',');

}

if (a.Top\_el() == '(')

a.pop();

}

while ((!a.IsEmpty()) && (prior <= a.Prior()) && (a.Top\_el() != '('))

{

postfix.push\_back(a.pop());

postfix.push\_back(',');

}

if ((c == '(') || (c == ')'))

;

else

a.push(c);

}

}

i++;

} while (infix[i] != NULL);

while ((infix[i] == NULL) && (!a.IsEmpty()))

{

postfix.push\_back(a.pop());

postfix.push\_back(',');

}

postfix.back() = ' ';

}

double TPostfix::Calculate()

{

ToPostfix();

string tmp;

TStack<double> b(infix.size());

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

{

if (NotOperation(postfix[i]))

{

while (postfix[i] != ',')

{

tmp += postfix[i];

i++;

}

b.push(stof(tmp));

tmp.clear();

}

else

{

if (b.IsEmpty())

throw;

else

{

b.push(Helper(b.pop(), b.pop(), postfix[i]));

i++;

}

}

}

return b.Top\_el();

}

double TPostfix::Helper(double a, double b, char c)

{

switch (c)

{

case '+':

{

return a + b;

}

case '-':

{

return a - b;

}

case '\*':

{

return a \* b;

}

case '/':

{

return a / b;

}

default:

throw;

}

}

bool TPostfix::IsValidByBrackets()

{

int left = 0, right = 0;

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

{

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

left++;

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

right++;

}

if (right == left)

return true;

else

return false;

}

bool TPostfix::IsValidByOpers()

{

int operands = 0, operations = 0;

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

{

if ((infix[i] == '(') || (infix[i] == ')'))

continue;

if (!NotOperation(infix[i]))

operations++;

else

if (NotOperation(infix[i]))

{

while (NotOperation(infix[i]) && (i != infix.size()))

i++;

i--;

operands++;

}

}

if (operands - 1 == operations)

return true;

else

return false;

}

bool TPostfix::IsValidByPos()

{

int i = 0;

string last = "operand", current;

if (!NotOperation(infix[i]))

return false;

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

return false;

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

last = "left bracket";

else

{

while (NotOperation(infix[i]) && (i != infix.size()))

i++;

last = "operation";

}

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

{

if (infix[j] == '(')

current = "left bracket";

else

if (infix[j] == ')')

current = "right bracket";

else

if (!NotOperation(infix[j]))

current = "operation";

else

if (NotOperation(infix[j]))

{

while (NotOperation(infix[j]) && (j != infix.size()))

j++;

j--;

current = "operand";

}

if ((current == last && current !="left bracket" && current != "right bracket") || (last == "left bracket" && current == "right bracket"))

return false;

last = current;

}

return true;

}

bool TPostfix::IsValidInTotal()

{

if (!IsValidByBrackets() || !IsValidByOpers() || !IsValidByPos())

return false;

return true;

}

**"Sample\_postfix.cpp"**

#include <iostream>

#include <string>

#include "postfix.h"

using namespace std;

int main()

{

string expression;

double res;

int enter = 0;

setlocale(LC\_ALL, "Russian");

cout << "1.Постфиксная форма" << endl;

cout << "2.Подсчет" << endl;

cin >> enter;

switch (enter)

{

case 1:

{

cout << "Введите арифметическое выражение: " << endl;

cin >> expression;

TPostfix Postfix(expression);

cout << "Арифметическое выражение: " << expression << endl;

Postfix.ToPostfix();

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

break;

}

case 2:

{

cout << "Введите арифметическое выражение: " << endl;

cin >> expression;

TPostfix Postfix(expression);

cout << "Арифметическое выражение: " << expression << endl;

res = Postfix.Calculate();

cout << res << endl;

break;

}

default:

break;

}

return 0;

}

**"Test\_tpostfix.cpp"**

#include "postfix.h"

#include <gtest.h>

#include "postfix.cpp"

TEST(TPostfix, can\_create\_postfix)

{

ASSERT\_NO\_THROW(TPostfix p);

}

TEST(TPostfix, can\_get\_infix)

{

TPostfix p("a-b\*c");

EXPECT\_EQ("a-b\*c", p.GetInfix());

}

TEST(TPostfix, can\_get\_postfix)

{

TPostfix p("a-b\*c");

p.ToPostfix();

EXPECT\_EQ("a,b,c,\*,- ", p.GetPostfix());

}

TEST(TPostfix, can\_check\_true\_infix)

{

TPostfix p("a-b\*c");

ASSERT\_NO\_THROW(p.ToPostfix());

}

TEST(TPostfix, can\_check\_false\_brackets)

{

TPostfix p("(a-))b\*c");

ASSERT\_ANY\_THROW(p.ToPostfix());

}

TEST(TPostfix, can\_check\_false\_operations)

{

TPostfix p("a-+b\*c");

ASSERT\_ANY\_THROW(p.ToPostfix());

}

TEST(TPostfix, can\_check\_false\_positioning)

{

TPostfix p("a-+bc");

ASSERT\_ANY\_THROW(p.ToPostfix());

}

TEST(TPostfix, can\_calculate)

{

TPostfix p("(10+10)\*(15+15)");

EXPECT\_EQ(600, p.Calculate());

}

TEST(TPostfix, can\_calculate\_large\_infix)

{

TPostfix p("(10+10)\*(15+15)/20+(40-10)-50\*3/2+((75+15)/6)");

EXPECT\_EQ(0, p.Calculate());

}

TEST(TPostfix, cant\_calculate\_wrong\_infix)

{

TPostfix p("()10+10)\*(15+15)/20+(40-10)-50\*3/2+((75+15)/6)");

ASSERT\_ANY\_THROW(p.Calculate());

}

**"Test\_tstack.cpp"**

#include "stack.h"

#include <gtest.h>

TEST(TStack, can\_create\_stack\_with\_positive\_length)

{

ASSERT\_NO\_THROW(TStack<int> st(5));

}

TEST(TStack, cant\_create\_stack\_with\_too\_large\_length)

{

ASSERT\_ANY\_THROW(TStack<int> st(1000));

}

TEST(TStack, cant\_create\_stack\_with\_negative\_length)

{

ASSERT\_ANY\_THROW(TStack<int> st(-5));

}

TEST(TStack, can\_get\_max\_size)

{

TStack<int> st(5);

EXPECT\_EQ(5, st.max\_size());

}

TEST(TStack, can\_get\_size)

{

TStack<int> st(5);

st.push(1);

EXPECT\_EQ(1, st.Size());

}

TEST(TStack, can\_push)

{

TStack<int> st(5);

st.push(1);

EXPECT\_EQ(1, st.Top\_el());

}

TEST(TStack, cant\_push\_if\_full)

{

TStack<int> st(1);

st.push(1);

ASSERT\_ANY\_THROW(st.push(1));

}

TEST(TStack, cant\_pop\_if\_empty)

{

TStack<int> st(1);

ASSERT\_ANY\_THROW(st.pop());

}

TEST(TStack, cant\_get\_top\_element\_if\_empty)

{

TStack<int> st(1);

ASSERT\_ANY\_THROW(st.Top\_el());

}

TEST(TStack, cant\_get\_priority\_if\_empty)

{

TStack<int> st(1);

ASSERT\_ANY\_THROW(st.Prior());

}