Министерство науки и высшего образования 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 лабораторной работе**

**«Вычисление арифметических выражений»**

**Выполнил**:

студент группы 381903-2

Недошивин Д.С.

**Проверил**:

ассистент каф. МОСТ,

Волокитин В.Д.

Нижний Новгород

2020

**Содержание**

[Постановка задачи 3](#_Toc26962562)

[Метод решения 4](#_Toc26962563)

[Руководство пользователя 5](#_Toc26962564)

[Описание программной реализации 7](#_Toc26962565)

[Подтверждение корректности 8](#_Toc26962566)

[Заключение 9](#_Toc26962568)

[Приложение 10](#_Toc26962569)

# Постановка задачи

Разработать программу, выполняющую вычисление арифметического выражения с вещественными числами. Выражение в качестве операндов может содержать переменные и вещественные числа. Допустимые операции известны: +, -, /, \*. Допускается наличие знака "-" в начале выражения или после открывающей скобки. Опционально - наличие математических функций (sin, соs, ln, exp, и т.д.) Программа должна выполнять предварительную проверку корректности выражения и сообщать пользователю вид ошибки и номера символов строки, в которых были найдены ошибки.

**Этапы выполнения работы:**Разработка шаблонного класса TStack

1. Разбиение исходного арифметического выражения на лексемы (т.е. выделить операнды, операции, переменные)
2. Проверка корректности выражения:

* правильность расстановки скобок (с использованием стека (TStack) для хранения индексов скобок).
* пропущены ли операнды или знаки операций
* недопустимые символы

1. Перевод выражения в постфиксную (польскую) запись
2. Вычисление выражения по постфиксной записи
3. Создание консольного интерфейса пользователя
4. Написание отчета

Все функции должны тестироваться с помощью Google test-ов.

# Метод решения

***Класс Solver***

1. *void convert\_string\_to\_lexeme(string& s)*

Функция предназначена для формирования из строки массива лексем, удаляет пробелы из строки.

1. *void convert\_to\_RPN()*

Функция предназначена для перевода массива лексем в массив лексем, записанный в обратной польской нотации

* Проверка числа скобок
* Проверка синтаксиса при помощи функции can\_go\_next()
* Преобразование выражения в обратную польскую нотацию с использованием стека   
  Рассматриваем поочередно каждый символ:
  1. Если этот символ - число, то просто помещаем его в выходную строку.
  2. Если символ - знак операции (+, -, \*, /, ) ,( ), то проверяем приоритет данной операции. Операции умножения и деления имеют наивысший приоритет (3). Операции сложения и вычитания имеют меньший приоритет (2). Наименьший приоритет (равен 0) имеет открывающая скобка, закрывающаяся имеет приоритет 1.
  3. Получив один из этих символов, мы должны проверить стек:
     + - 1. Если стек все еще пуст, или находящиеся в нем символы   
            (находится в нем могут только знаки операций и открывающая скобка) имеют меньший приоритет, чем приоритет текущего символа, то помещаем текущий символ в стек.
         2. 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 – открывающаяся 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, которую следует просто извлечь из стека. Закрывающая скобка также извлекается из стека.
  4. Если вся входная строка разобрана, а в стеке еще остаются знаки операций, извлекаем их из стека в выходную строку.

1. double solve()

Функция предназначена для вычисления выражения, записанного в форме обратной польской нотации (алгоритм вычисления).

Для реализации этого алгоритма используется стек для чисел.   
В качестве входной строки мы рассматриваем выражение, записанное в обратной польской нотации:

1. Если очередной символ входной строки - число, то кладем его в стек.

2. Если очередной символ - знак операции, то извлекаем из стека два верхних числа (в случае унарного минуса - одно), используем их в качестве операндов для этой операции, затем кладем результат обратно в стек.

Когда вся входная строка будет разобрана в стеке должно остаться одно число, которое и будет результатом данного выражения. Это число функция и возвращает в качестве результата

# Руководство пользователя

Пользователю программы предоставлен для использования 1 исполняемый файл под названием ''Sample.exe''.

При открытии исполняемого файла ''Sample.exe'' пользователю будет предложено ввести арифметическое выражение для его последующего расчета.  
Числа в арифметическом выражении могут быть использованы как дробные, так и целые. Допустимые операции: “+”, ”-”, ”\*”, ”/”, ”(”, ”)”. Допустимы пробелы в арифметическом выражении.  
В случае синтаксической ошибки в арифметическом выражении программа прекратит вычисление и сообщит точный вид ошибки пользователю.  
После выполнения расчетов программа предложит повторный ввод или завершение работы программы.

***Пример исполнения***:

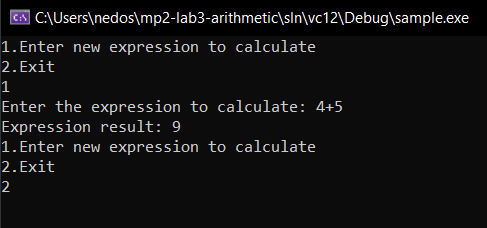


Рис. 1 (Пример 1)

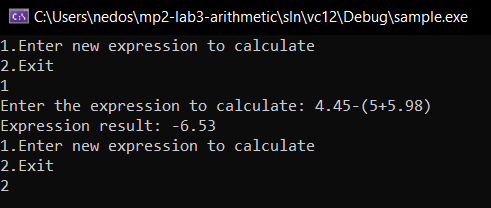


Рис. 2 (Пример 2)

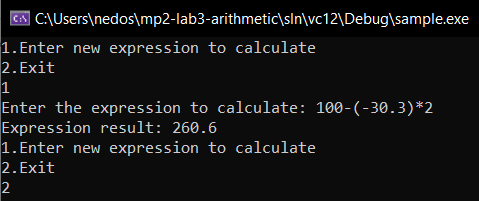


Рис. 3 (Пример 3)

# Описание программной реализации

Программа представлена 4 проектами, объединенными в одно решение

1. ***Arithmetic***

* *Arithmetic.h*Объявление классов *Lexeme* и *Solver*, необходимых для реализации алгоритма, а также их методов.
* *Stack.h*

Объявление шаблонного класса *TStack*, реализующего стек неограниченного размера, реализация методов класса *TStack*.

* *Arithmetic.cpp*

Реализация методов классов *Lexeme* и *Solver.*

1. ***Gtest***

Библиотека Google unit тестов.

1. ***Sample***

Реализация пользовательского интерфейса программы.

1. ***Tests***

* *Test\_arithmetic.cpp*

Unit тесты класса *Solver.*

* *Test\_main.cpp*

Запуск Unit тестов.

* *Test\_stack.cpp*

Unit тесты класса *TStack.*

# Подтверждение корректности

Для подтверждения корректности в программу включена библиотека Google тестов, которые проверяют функции на наличие ошибок.

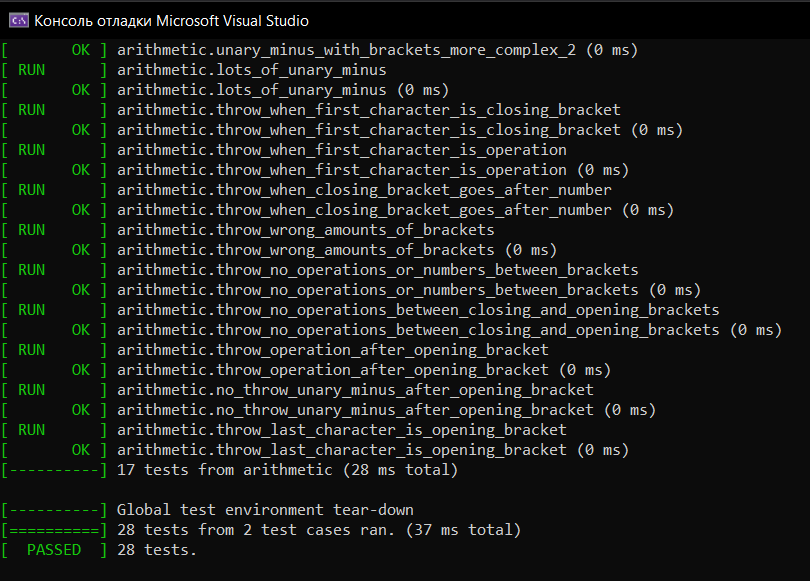


Рис. 4 (Подтверждение корректности)

# Заключение

В результате выполнения лабораторной работы мы разработали собственный класс TStack, изучили и реализовали алгоритм перевода изначально заданной строки в вид обратной польской нотации, алгоритм вычисления выражений, записанных в форме обратной польской нотации; реализовали обработку ошибок синтаксиса в заданном выражении.

# Приложение

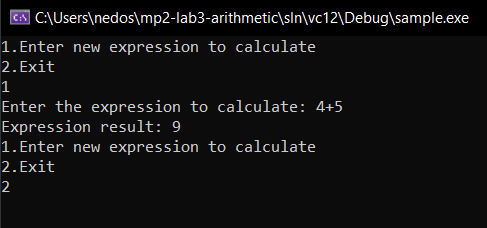


Рис. 1 (Пример 1)

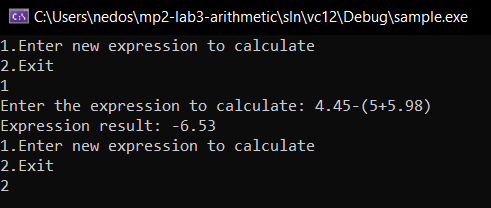


Рис. 2 (Пример 2)

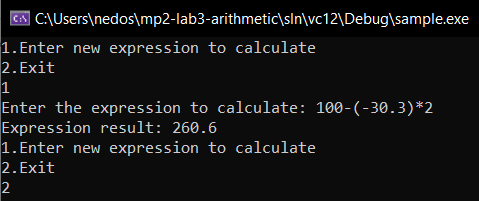


Рис. 3 (Пример 3)

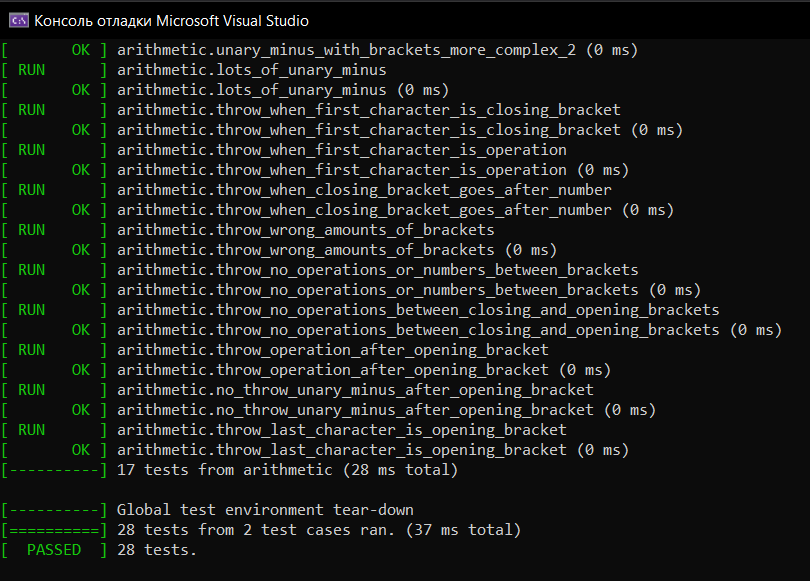


Рис. 4 (Подтверждение корректности)

**Главный код:**

class Solver

{

private:

vector <Lexeme> d;

public:

void convert\_string\_to\_lexeme(string& s)

{

int i = 0, count = 0;

string temp;

double num;

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

{

while (s[i] == ' ')

{

s.erase(i, 1);

}

}

while (i < s.size())

{

if (s[i] >= '0' && s[i] <= '9')

{

bool dot = false;

temp = "";

while ((i < s.size()) && ((s[i] >= '0' && s[i] <= '9') || (s[i] == '.')))

{

if (s[i] == '.' && dot)

{

string err;

err = "Number has two or more dots in it";

d.clear();

throw err;

}

if (s[i] == '.')

{

dot = true;

}

temp += s[i];

i++;

}

num = stod(temp, 0);

Lexeme a(num);

d.push\_back(a);

continue;

}

if (s[i] == '/')

{

Lexeme a('/');

d.push\_back(a);

i++;

continue;

}

if (s[i] == '\*')

{

Lexeme a('\*');

d.push\_back(a);

i++;

continue;

}

if (s[i] == '+')

{

Lexeme a('+');

d.push\_back(a);

i++;

continue;

}

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

{

Lexeme a('(');

d.push\_back(a);

i++;

continue;

}

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

{

Lexeme a(')');

d.push\_back(a);

i++;

continue;

}

if (s[i] == '-')

{

bool dot = false;

bool unary = false;

temp = "";

count = 0;

count++;

if (i == 0)

{

unary = true;

}

else if (s[i - 1] == '(' || s[i - 1] == '/' || s[i - 1] == '\*' || s[i - 1] == '+')

{

unary = true;

}

if (i + 1 < s.size())

{

while (i + 1 < s.size() && s[i + 1] == '-')

{

count++;

i++;

}

if (count % 2 == 1 && !unary)

{

Lexeme a('-');

d.push\_back(a);

i++;

continue;

}

else if (count % 2 == 0 && !unary)

{

Lexeme a('+');

d.push\_back(a);

i++;

continue;

}

else if (count % 2 == 1 && unary)

{

temp = "-";

}

if (i + 1 < s.size() && s[i + 1] == '(')

{

Lexeme a('-', 1);

d.push\_back(a);

i++;

continue;

}

if (i + 1 < s.size() && s[i + 1] >= '0' && s[i + 1] <= '9')

{

i++;

while ((i < s.size()) && ((s[i] >= '0' && s[i] <= '9') || (s[i] == '.')))

{

if (s[i] == '.' && dot)

{

string err;

err = "Number has two or more dots in it";

d.clear();

throw err;

}

if (s[i] == '.')

{

dot = true;

}

temp += s[i];

i++;

}

num = stod(temp, 0);

Lexeme a(num);

d.push\_back(a);

continue;

}

else

{

string err;

err = "Wrong operation order after minus (no number or bracket)";

d.clear();

throw err;

}

}

else

{

Lexeme a('-');

d.push\_back(a);

i++;

continue;

}

}

else

{

string err;

err = "Incorrect symbol";

d.clear();

throw err;

}

}

}

void convert\_to\_RPN()

{

int count = 0;

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

{

if (d[i].is\_op\_br())

{

count++;

}

if (d[i].is\_cl\_br())

{

count--;

}

}

if (count != 0)

{

string err;

err = "Wrong amount of brackets";

d.clear();

throw err;

}

if (d[0].is\_op() && !d[0].is\_op\_br() && !d[0].is\_unary())

{

string err;

err = "Expression can not start with operations such as '+','\*','/',')'";

d.clear();

throw err;

}

if (d.back().is\_op())

{

if (!d.back().is\_cl\_br())

{

string err;

err = "Binary operation in the end of the expression";

d.clear();

throw err;

}

}

for (int i = 0; i < d.size() - 1; i++)

{

if (!d[i].can\_go\_next(d, i))

{

string err = "";

if (d[i].is\_number())

{

err += d[i].get\_number();

err += " can't go before ";

err += d[i + 1].get\_oper();

d.clear();

throw err;

}

else if (d[i].is\_op())

{

err += d[i].get\_oper();

err += " can't go before ";

if (d[i + 1].is\_number())

{

err += d[i + 1].get\_number();

}

else if (d[i + 1].is\_op())

{

err += d[i + 1].get\_oper();

}

d.clear();

throw err;

}

}

}

vector <Lexeme> input;

TStack <Lexeme> b;

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

{

if (d[i].is\_number())

{

input.push\_back(d[i]);

}

else if (d[i].is\_op())

{

if (d[i].is\_op\_br())

{

b.push(d[i]);

continue;

}

if (d[i].is\_cl\_br())

{

while (!b.front().is\_op\_br())

{

input.push\_back(b.pop());

}

b.pop();

}

gh:

if (b.is\_empty() || b.front().get\_prior() < d[i].get\_prior() || d[i].is\_unary())

{

if (!d[i].is\_cl\_br())

{

b.push(d[i]);

}

}

else

{

while (!b.is\_empty() && b.front().get\_prior() >= d[i].get\_prior())

{

input.push\_back(b.pop());

}

goto gh;

}

}

}

while (!b.is\_empty())

{

input.push\_back(b.pop());

}

d = input;

}

double solve()

{

TStack <Lexeme> b;

Lexeme a, c;

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

{

if (d[i].is\_number())

{

b.push(d[i]);

}

else

{

if (d[i].is\_unary())

{

a = b.pop();

}

else

{

a = b.pop();

c = b.pop();

}

switch (d[i].get\_oper())

{

case '+':

{

Lexeme temp(c.get\_number() + a.get\_number());

b.push(temp);

break;

}

case '-':

{

if (d[i].is\_unary())

{

Lexeme temp(-a.get\_number());

b.push(temp);

break;

}

else

{

Lexeme temp(c.get\_number() - a.get\_number());

b.push(temp);

break;

}

}

case '\*':

{

Lexeme temp(c.get\_number() \* a.get\_number());

b.push(temp);

break;

}

case '/':

{

if (a.get\_number() == 0.0)

{

string err;

err = "Division by zero";

d.clear();

throw err;

}

else

{

Lexeme temp(c.get\_number() / a.get\_number());

b.push(temp);

break;

}

}

}

}

}

d.clear();

return b.pop().get\_number();

}

};