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»**

**Выполнила**:

студентка группы 382003-1

Шамонина А.С.

**Проверил**:

преподаватель каф. МОСТ,

Волокитин В.Д.

Нижний Новгород

2021

**Содержание**

[Постановка задачи 3](#_Toc91333201)

[Метод решения 4](#_Toc91333202)

[Руководство пользователя 5](#_Toc91333203)

[Описание программной реализации 6](#_Toc91333204)

[Подтверждение корректности 7](#_Toc91333205)

[Заключение 8](#_Toc91333206)

[Приложение 9](#_Toc91333207)

# Постановка задачи

Требуется разработать программу, производящую вычисления арифметических выражений с вещественными числами. Выражение в качестве операндов может содержать переменные и вещественные числа. Допустимыми операциями являются: +, -, / и унарный - , наличие математических функций (sin, соs, ln, exp, и т.д.) опционально. Так же должна выполняться предварительная проверка корректности выражения с выводом вида ошибки. Необходимо, чтобы все функции тестировались с помощью Google-test

# Метод решения

**Class Lexem**

Данный класс предназначен для хранения отдельных элементов выражения (лексем) таких как число или операция.

1. Функции double GetValue() и string GetOperation() вращают значения операнда и операции
2. Функция bool GetType() возвращает true, если лексема является операндом, и false, если операцией
3. Так же переопределены операторы == и !=
4. Функция int priority() возвращает приоритет операции

**Class Arithmetic**

Данный класс предназначен для создания вектора из лексем, проверки выражения на корректность, перевода в обратную польскую запись и произведения вычислений по ней.

1. Функция void vectorLexem(string& str) переводит выражение в строке в вектор, состоящий из объектов класса Lexem по следующему алгоритму:

* Проходим по строке от начала до конца
* При встрече операций +,\*, / создаём объект класса Lexem и вставляем его в конец вектора лексем.
* При встрече - : если минус находится в начале строки, либо символ перед минусом это операция, но не ), то минус унарный и тогда создаем объект класса Lexem, в котором будем хранить \_, чтобы различать унарный и бинарный минусы, затем вставляем его в конец вектора лексем. Иначе минус бинарный и мы создаём объект класса Lexem, а после вставляем его в конец вектора лексем.
* При встрече цифры или точки проходим по строке, пока цифры и точки еще встречаются (если точек в числе несколько, выдается ошибка). Затем создаём объект класса Lexem с этим значением, далее вставляем его в конец вектора лексем.
* При встрече ) или ( создаём объект класса Lexem записыввя скобку как операцию, далее вставляем его в конец вектора лексем.
* Пробелы при этом удаляются, а при встрече недопустимого символа бросается исключение.

1. Функция void Postfix() производит перевод выражения, разложенного на вектор лексем, в обратную польскую запись по следующему алгоритму:

* Создаем дополнительный вектор лексем, чтобы использовать его как выходную строку.
* Проходим по каждому элементу вектора лексем.
* При встрече числа или переменной перемещаем их в выходную строку
* При встрече операции +, -, \*, / смотрим на её приоритет. Операции умножения и деления имеют приоритет 3. Операции сложения и вычитания имеют приоритет 2. Открывающая скобка и закрывающая соответственно 1 и 0. Унарный минус имеет приоритет 4(наивысший).
* Получив операцию производим проверку стека. Если стек пуст, или находящиеся в нем операции имеют меньший приоритет, чем приоритет текущей, то помещаем текущий символ в стек. Если символ на вершине стека имеет больший приоритет или равный приоритету текущего то извлекаем все из стека в выходную строку, пока выполняется это условие.
* При встрече ( кладём её в стек.
* При встрече ) извлекаем из стека все в выходную строку, пока не встретим в стеке (, которую следует удалить. Закрывающая скобка затем тоже удаляется.
* Если дошли до конца строки, а в стеке еще есть операции, извлекаем их в выходную строку.

1. Функция double calculation() производит вычисления по обратной польской записи по следующему алгоритму:
   * Создаём стек лексем

* Проходим по каждому элементу вектора обратной польской записи.
  + При встрече числа вставляем его в стек
* При встрече унарного минуса берем из стека последнее значение, умножаем его на -1 и возвращаем в стек
* Иначе берем из стека два операнда, сначала второй операнд, потом первый. Если операция +, то складываем операнды. Если – то вычитаем из первого операнда второй, если \*, то умножаем операнды, если /, то делим первый операнд на второй, проверяя, что второй операнд не равен нулю.
  + Затем помещаем в стек результат операции.
  + В итоге последним значением в стеке останется результат выражения

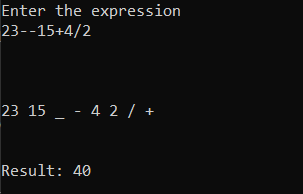
1. Функции bool checkBrackets() и bool Correct() осуществляют проверку корректности выражения по математическим правилам: бросают исключения при нахождении лишних скобок, пропущенных операций и операндов.
2. Функция void PutVariable(Arithmetic& s) используется для поиска переменных в выражении, запрашивает их ввод.

# Руководство пользователя

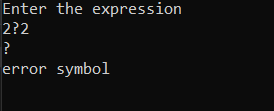
Запуская программу, пользователь получает предложение ввести арифметическое выражение по следующим правилам:

1. Не допускается использование символов, кроме «1234567890.+-\*/»
2. Не допускается запись нескольких операций подряд, кроме унарного минуса
3. Доступны три переменные x, y, z, которые в процессе вычисления программа предложит ввести

После выполнения расчетов выведется результат вычислений. В случае возникновения ошибки будет указан её вид или выведен недопустимый символ



пример корректного результата работы программы



пример работы программы при синтаксической ошибке

# Описание программной реализации

Программа использует стандартные библиотеки: iostream, cmath, string. Также используется библиотека vector для хранения лексем.

Программа представляет собой 4 проекта:

1. arithmetic:

arithmetic.h – объявление классов arithmetic и lexem, их полей и некоторых методов.

arithmetic.cpp – реализация методов класса Arithmetic и Lexem.

stack.h – объявление шаблонного класса TStack, реализация его методов,

1. gtest – библиотека Google test-ов
2. sample:

main\_arithmetic.cpp – реализация пользовательского интерфейса

1. tests:

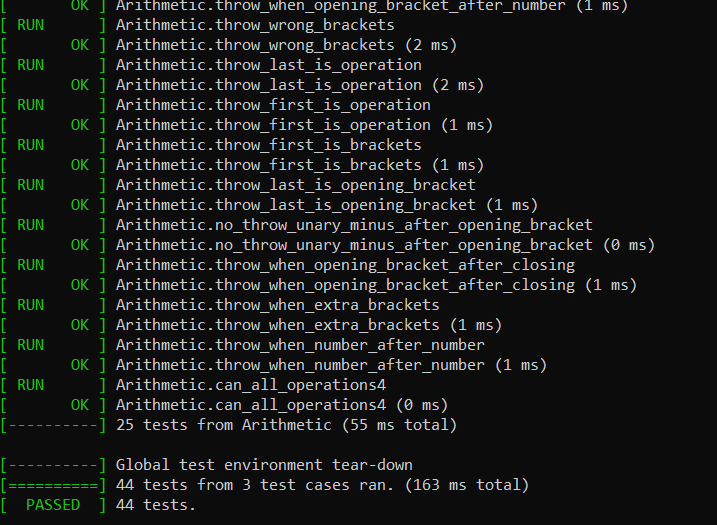
test\_stack.cpp – тесты для класса TStack.

test\_arithmetic.cpp – тесты для класса Arithmetic.

test\_main.cpp – запуск работы тестов.

# Подтверждение корректности

Для подтверждения корректности в программе используется функция Correct, которая также использует функцию checkBrackets, проверяющую правильность расстановок скобок. На вход подаётся вектор лексем, функция в цикле проверяет отсутствие пропущенных операндов, лишних операций, так же после операции не может стоять другая операция, число не может стоять перед скобкой или после них, во всех таких случаях выводится название ошибки. Так же реализованы Google-тесты, проверяющие методы на наличие ошибок.

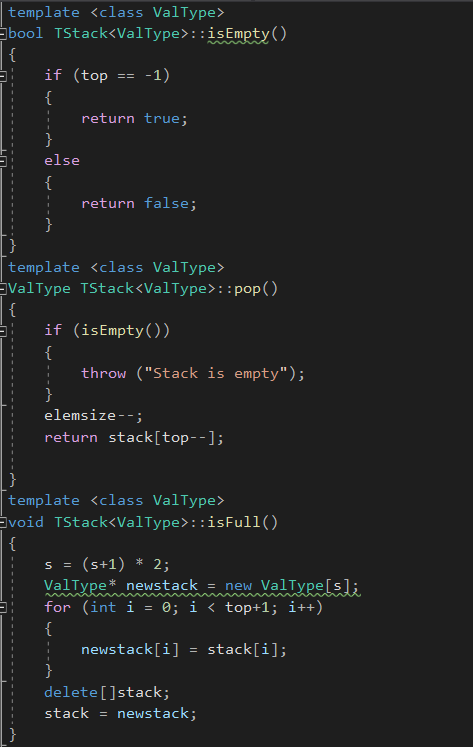


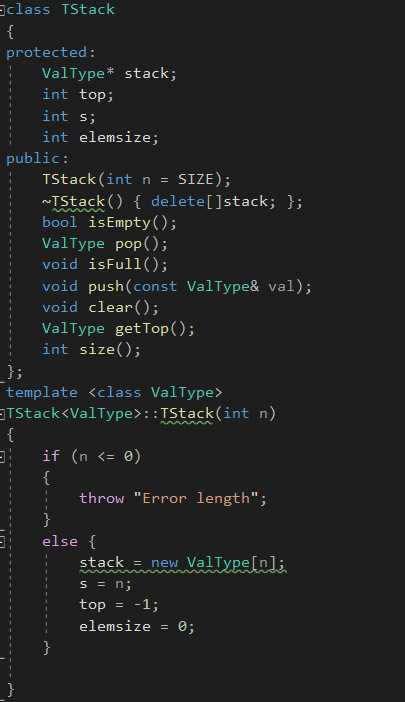
# Заключение

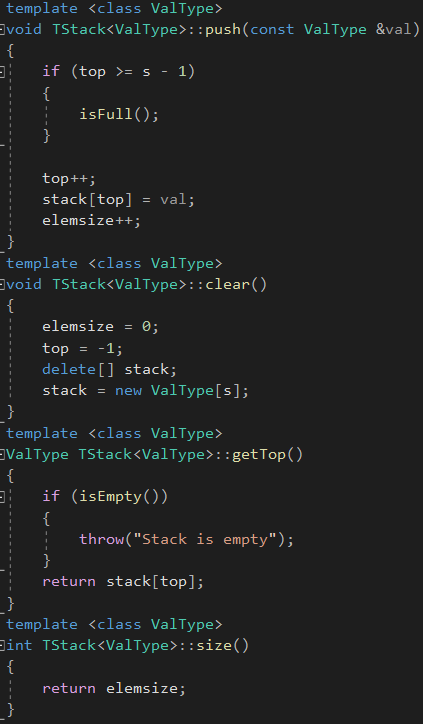
В ходе лабораторной работы была написана программа, поддерживающая вычисление арифметических выражений с вещественными числами. Программа также производит проверку входного выражения на корректность. Также были написаны Google-тесты для проверки работы реализованных методов.

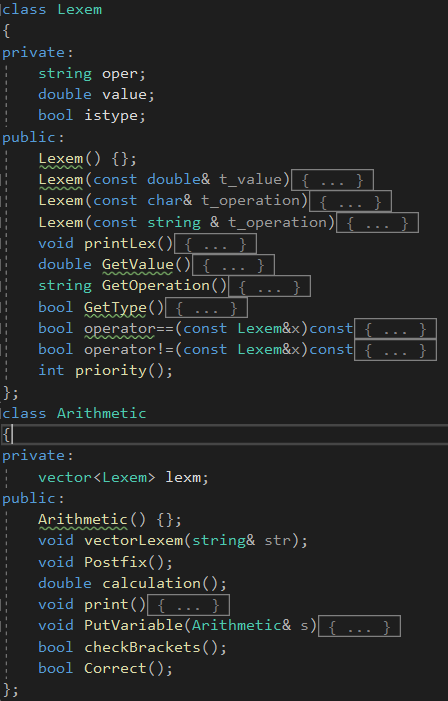
# Приложение

Класс TStack







Классы Lexem и Arithmetic

void Arithmetic::vectorLexem(string& str)//перевод в вектор лексем

{

string stringnum;

double doublenum;

int i = 0;

while (i < str.length()) {

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

if (str[i] == '+' || str[i] == '\*' || str[i] == '/')

{

Lexem operation(str[i]);

lexm.push\_back(operation);

i++;

}

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

{

if (i == 0) {

Lexem unar\_minus('\_');

lexm.push\_back(unar\_minus);

i++;

continue;

}

if (i > 0)

{

if (lexm[lexm.size()-1].GetOperation() != ")" && lexm[lexm.size() - 1].GetType() == false && lexm[lexm.size() - 1].GetOperation() != "x" && lexm[lexm.size() - 1].GetOperation() != "y" && lexm[lexm.size() - 1].GetOperation() != "z") {

Lexem unar\_minus('\_');

lexm.push\_back(unar\_minus);

i++;

}

else {

Lexem minus('-');

lexm.push\_back(minus);

i++;

}

}

}

else if (str[i] >= '0' && str[i] <= '9' || (str[i] == '.'))

{

int pointCounter = 0;

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

{

stringnum += str[i];

i++;

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

pointCounter++;

}

if (pointCounter > 1) {

throw "You have more than one point in the number";

}

}

doublenum = stof(stringnum);

Lexem number(doublenum);

lexm.push\_back(number);

stringnum.clear();

}

else if ((str[i] == 'x') || (str[i] == 'y')|| (str[i] == 'z'))

{

Lexem A(str[i]);

lexm.push\_back(A);

i++;

}

else if (str[i] == ')' || str[i] == '(') {

Lexem bracket(str[i]);

lexm.push\_back(bracket);

i++;

}

else { cout << str[i] << endl; throw "error symbol"; break; }

}

}

void Arithmetic::Postfix()//перевод в опз

{

vector <Lexem> postfix;

TStack <Lexem> operations;

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

{

if (lexm[i].GetType() == true|| (lexm[i].GetOperation() == "x") || (lexm[i].GetOperation() == "y") || (lexm[i].GetOperation() == "z"))

{

postfix.push\_back(lexm[i]);

}

else

{

if (lexm[i].GetOperation() == "(")

{

operations.push(lexm[i]);

continue;

}

else if (lexm[i].GetOperation() == ")")

{

while (operations.getTop().GetOperation() != "(")

{

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

}

operations.pop();

continue;

}

if (i > 0 && lexm[i].GetOperation() == "\_")

{

operations.push(lexm[i]);

}

else {

while (!operations.isEmpty() && operations.getTop().priority() >= lexm[i].priority())

{

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

}

operations.push(lexm[i]);

}

}

}

while (!operations.isEmpty())

{

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

}

lexm = postfix;

}

double Arithmetic::calculation()//вычисления

{

TStack <Lexem> stack;

Lexem lexOp1, lexOp2;

double op1, op2, res;

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

if (lexm[i].GetType() == true) {

stack.push(lexm[i].GetValue());

}

else if (lexm[i].GetOperation() == "\_") {

res = (-1.0) \* stack.pop().GetValue();

stack.push(res);

continue;

}

else {

lexOp2 = stack.pop();

lexOp1 = stack.pop();

op1 = lexOp1.GetValue();

op2 = lexOp2.GetValue();

if (lexm[i].GetOperation() == "+")

res = op1 + op2;

if (lexm[i].GetOperation() == "-")

res = op1 - op2;

if (lexm[i].GetOperation() == "\*")

res = op1 \* op2;

if (lexm[i].GetOperation() == "/") {

if (op2 == 0) throw "You can't divide by zero";

res = op1 / op2;

}

stack.push(res);

}

}

Lexem result = stack.pop();

return result.GetValue();

}