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 лабораторной работе**

**«Вычисление арифметических выражений»**

**Выполнил**:

студент группы 3822Б1ПМ1

Ковалёв К.И.

**Проверил**:

преподаватель каф. МОСТ,

Волокитин В.Д.

Нижний Новгород

2023

Содержание

[Введение 3](#_Toc529541653)

[1. Постановка задачи 4](#_Toc529541654)

[2. Руководство пользователя 5](#_Toc529541655)

[3.1. Описание структуры программы 6](#_Toc529541656)

[3.2. Описание алгоритмов 6](#_Toc529541657)

[4. Результаты экспериментов 7](#_Toc529541658)

[Заключение 8](#_Toc529541659)

[Литература 9](#_Toc529541660)

[Приложение 10](#_Toc529541661)

# Введение

Разработка приложения, позволяющего вычислять арифметические выражения, требует также реализации структуры данных стек. Помимо этого, необходима реализация перевода в постфиксную форму. Наиболее удобным её вариантом является создание соответствующего класса. Все перечисленные соображения позволяют чётко сформулировать задачи проекта.

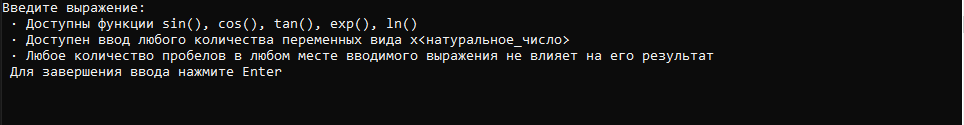
# Постановка задачи

Поставлены следующие задачи:

1. Разработать интерфейс шаблонного класса TStack.
2. Реализовать методы шаблонного класса TStack.
3. Разработать интерфейс класса TPostfix для работы с постфиксной формой.
4. Реализовать методы класса TPostfix.
5. Разработать и реализовать консольное приложение, позволяющее вычислять значения арифметических выражений, содержащих базовые арифметические операции, унарный минус, функции sin(), cos(), tan(), exp(), ln(), а также позволяющее вводить любое количество переменных и предоставляющее возможность пересчёта значения выражения для различных значений переменных.
6. Разработать и реализовать тесты для классов TStack и TPostfix на базе Google Test.
7. Опубликовать исходный код в личном репозитории на GitHub.

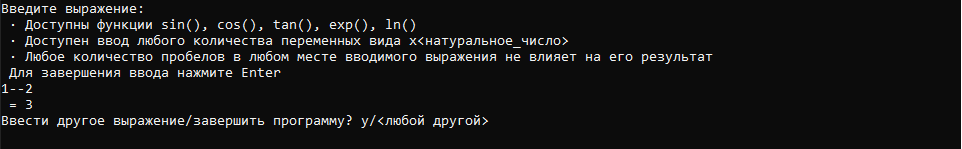
# Руководство пользователя

* + 1. При запуске консольного приложения пользователю будет предложено ввести арифметическое выражение для подсчёта, в нём доступны к использованию:
* функции sin(), cos(), tan(), exp(), ln();
* ввод любого количества переменных вида x<натуральное число>
* ввод любого количества пробелов в любом месте, выражение будет посчитано с игнорированием всех пробелов.



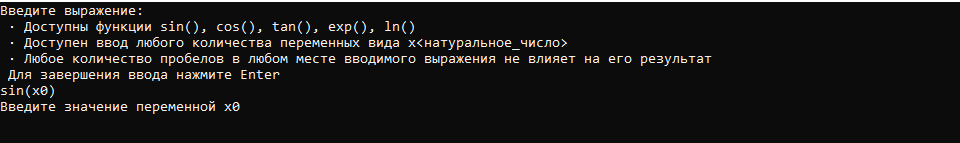
*Рис.1. Первоначальный ввод*

* 1. После ввода выражения, не содержащего переменных, оно будет посчитано, результат будет выведен в консоль. Затем пользователь сможет выбрать ввод другого выражения, т.е. возврат к п. 1 (если введёт y) или завершение работы программы (ввод любого другого символа).



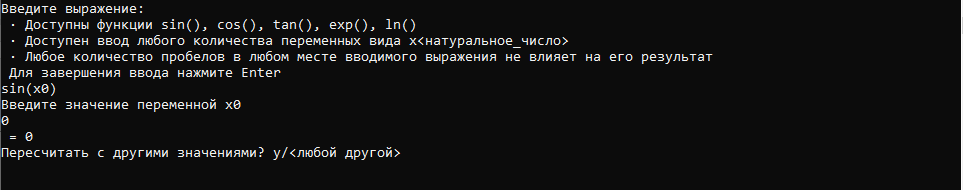
*Рис.2. Подсчёт значения введённого выражения*

2.2) После ввода выражения, содержащего хотя бы одну переменную, пользователю будет предложено ввести все значения для переменных в том порядке, в котором они появлялись в выражении.



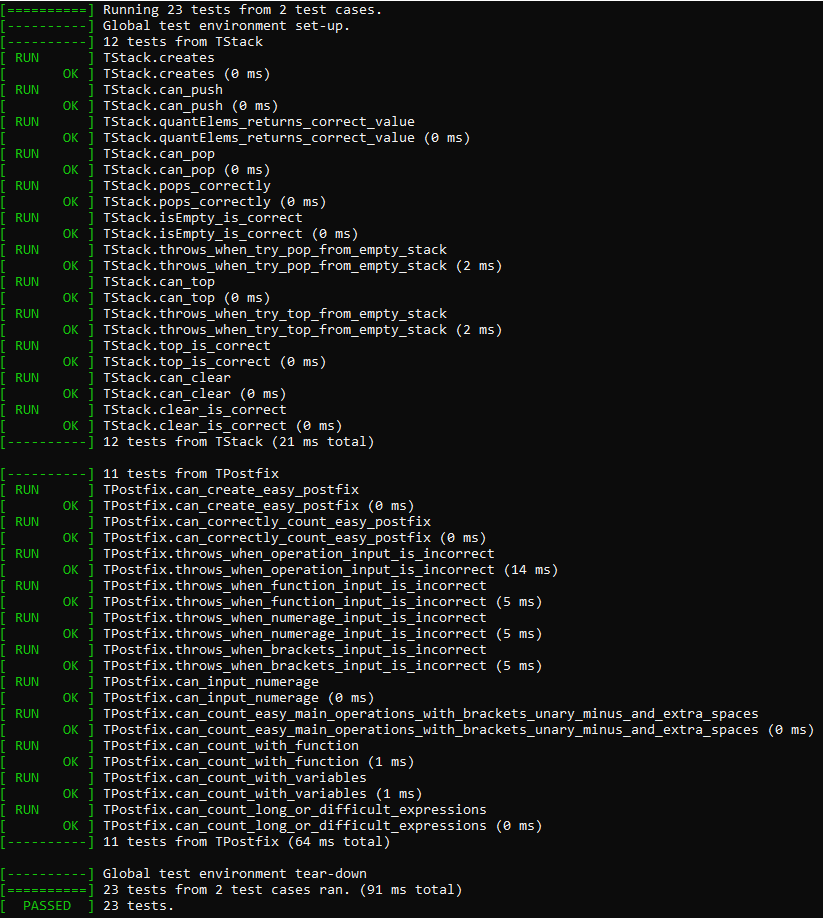
*Рис.3. Ввод значений переменных*

3) По завершении ввода выражение будет подсчитано для введённых значений переменных, результат будет выведен в консоль. Затем пользователь сможет выбрать пересчёт выражения для других значений переменных, т.е. возврат к п. 2.2 (если введёт y) или возврат к п. 2.1 (ввод любого другого символа).



*Рис.4. Выражение подсчитано для конкретных значений переменных*

При запуске проекта tests на экран будет выведено сообщение о количестве и составе тестов, их наименованиях и статусе прохождения:



*Рис.5. Запуск тестов*

1. **Руководство программиста**

## Описание структуры программы

## Программа состоит из четырёх проектов, в частности:

## arithmetic – содержит заголовочные файлы arithmetic.h, в котором объявлен класс TPostfix, и stack.h, в котором объявлен и реализован класс TStack, файл arithmetic.cpp, в котором реализован класс TPostfix с классом Lexeme в приватной области.

* gtest – библиотека Google Test.
* sample – содержит файл main\_arithmetic.cpp, в котором реализовано консольное приложение, выполняющее подсчёт арифметических выражений.
* tests – содержит файлы test\_arithmetic.cpp и test\_stack.cpp, в которых реализованы тесты программы и test\_main.cpp, необходимый для запуска тестов на базе Google Test.

## Описание алгоритмов

Класс TStack является шаблонным и реализует стек. Его прототип с указанием предназначения методов:

class TStack {

private:

T\* data;

size\_t size;

size\_t cap;

void resize(); //перевыделяет память, копирует значения из старой памяти

public:

TStack(); //конструктор, по умолчанию выделяет память на 10 эл-тов

size\_t quantElems();//возвращает количество элементов в стеке

bool isEmpty(); //возвращает true, если стек пуст, иначе false

void push(T el); //добавляет элемент в стек, при необходимости вызывает resize()

T pop(); //возвращает, а затем удаляет элемент с вершины стека, если стек не пуст

T& top(); //возвращает ссылку на элемент с вершины стека, если стек не пуст

void clear(); //очищает стек, перевыделяет память на 10 эл-тов

~TStack ()//деструктор

};

Класс TPostfix переводит строку в инфиксной форме в постфиксную, а также считает значение выражения в постфиксной записи. Его прототип:

class TPostfix

{

private:

class Lexeme

{

public:

int type\_of\_lex;

void\* lex;

size\_t pos=0;

Lexeme();

Lexeme(const Lexeme&& rv);

Lexeme(const Lexeme& rv);

template <typename T>

Lexeme(int t\_of\_l, T op, size\_t position=0);

Lexeme& operator=(const Lexeme & rv);

}

Lexeme\* Rev\_Pol\_Notation; //ОПЗ

int\* arr\_num\_variable; //массив с номерами переменных

double\* arr\_value\_variable; //массив со значениями переменных

size\_t i = 0, ind\_RPN = 0, cap\_RPN, ind\_variable = 0, cap\_variable;

TStack <Lexeme> ops; //стек лексем

public:

TPostfix(string& inp, bool need\_to\_check = true);

void duplicate\_RPN();

void insert\_result(const Lexeme& lex);

void insert\_variable(int xi);

void asker();

void input\_variables(double\* arr, size\_t ind);

size\_t Get\_ind\_variable();

double count();

};

Класс TPostfix содержит класс Lexeme в приватной области. Объект этого класса содержит флаг type\_of\_lex, который позволяет определить, какой лексемой является объект. Если значение type\_of\_lex равно 1, то объект является бинарной операцией, 2 - '(', 3 - ')', 4 - переменной, 5 - унарным минусом, 6 – функцией, 7 – числом. pos хранит индекс операции в строке, чтобы если на этапе вычисления станет ясно, что операция введена некорректно, можно было вывести позицию этой операции в исходной строке. lex является указателем на саму лексему, т.е. значение типа char в случае бинарной операции, скобок и унарного минуса, значение типа double в случае числа, значение типа size\_t в случае переменной (lex указывает на индекс этой переменной в массиве переменных) и значение типа int в случае функции (1 - sin, 2 - cos, 3 - tan, 4 - exp, 5 - ln). Для обработки этих случаев предусмотрен шаблонный конструктор преобразования типа, который выделяет память под тип данных, переданный при вызове конструктора. Конструктор копирования при помощи оператора switch() case создаёт копию объекта, не копируя указатели, а создавая одинаковые значения в разных участках памяти. Однако для конструктора перемещения копирование указателей допустимо.

Конструктор класса TPostfix переводит входную строку inp в запись в постфиксной форме. Это происходит в два прохода по строке: за первый удаляются лишние пробелы: в случае, если они действительно присутствовали, в консоль выводится вариант строки без пробелов; за второй с помощью оператора switch() case обрабатываются все возможные случаи ввода. Переменные и числа сразу при помощи метода insert\_result() заносятся в Rev\_Pol\_Notation, операции же, функции и скобки сначала попадают в стек лексем ops, откуда могут быть вытеснены в Rev\_Pol\_Notation операцией ниже или равной по приоритету либо вводом ‘)’. По завершении прохода про строке inp все операции из стека вытесняются в строку. Случаи неверного ввода функций, чисел, скобок обрабатываются в конструкторе.

Метод asker() выводит в консоль запрос на ввод значений переменных, вводит их и их значениями заполняет массив.

Метод count() по обратной польской записи вычисляет значение выражения. Для этого в один проход по массиву лексем с помощью оператора switch() case в стек помещаются числа (в т.ч. переменные, вместо которых подставлено соответствующее значение). Если же встречена операция или функция, то два последних значения из стека изымаются как операнды для этой операции/аргументы для функции. Результат вычислений помещается в стек. Во время прохода по массиву проверяется правильность ввода аргументов и операндов, в случае ошибки выполняется throw std::invalid\_argument с описанием ошибки.

Метод input\_variables(double\* arr, size\_t ind) вызывается в тестах при введении известного числа переменных.

Файл test\_stack.cpp содержит тесты класса TStack: по 2 теста на метод, один из которых проверяет на работоспособность (программа не завершилась преждевременно), а второй – на корректность.

Файл test\_arithmetic.cpp содержит тесты класса TPostfix: по 2 теста на основные методы, несколько тестов на различные ситуации ошибочного (и корректного) ввода лексем, в каждом тесте по несколько проверок.

# Результаты экспериментов

В качестве эксперимента операция унарный минус была реализована двумя способами: первая реализация не предполагала существенных различий между бинарным и унарным минусом:

case '-': {

Lexeme tmp(5, '-', i);

ops.push(move(tmp));

}

А вторая реализация ещё на этапе перевода в ОПЗ отделяла унарный минус от бинарного:

case '-': {

if (i == 0 || inp[i - 1] == '+' || inp[i - 1] == '\*' ||

inp[i - 1] == '/' || inp[i - 1] == '-' || inp[i - 1] == '(') //если это унарный минус, то

{

if ((inp[i + 1] <= '9' && inp[i + 1] >= '0') ||

inp[i + 1] == '.' || inp[i + 1] == ',' || inp[i + 1] == 'x' ||

inp[i + 1] == 'X' || inp[i + 1] == '-' || inp[i + 1] == '(' ||

inp[i + 1] == 'e' || inp[i + 1] == 'c' || inp[i + 1] == 's' ||

inp[i + 1] == 'l' || inp[i + 1] == 't') //если за минусом следует операнд или ещё один минус

{

Lexeme tmp(5, '-', i); //определи его унарным минусом

ops.push(move(tmp));

}

else

{

string except("Incorrect minus input at position ");

except += to\_string(i);

throw invalid\_argument(except);

}

}

else

{//иначе определи его бинарным

Lexeme tmp(1, '-', i);

while (!ops.isEmpty() && (ops.top().type\_of\_lex == 6 ||

ops.top().type\_of\_lex == 5 ||

ops.top().type\_of\_lex == 1 &&

( \*(char\*)ops.top().lex == '\*' || \*(char\*)ops.top().lex == '-' || \*(char\*)ops.top().lex == '/')))

insert\_result(ops.pop());

ops.push(move(tmp));

}

break;

}

Однако первой реализации на этапе подсчёта выражения требовалось обращение к ячейкам pos операндов (т.е. требовалось их заполнять!), а поиск ошибок становился практически невозможным. Можно представить это в виде блок-схемы:

Для каждого из соседних минусов необходима такая проверка, тогда, если не возникло противоречий, минус унарный

Текущий элемент в ОПЗ == ‘-’

Найти соседние с ним лексемы по позиции в исходной строке

Минус бинарный

Есть рядом минусы?

**ДА**

Оба операнды?

Хотя бы один из них – операнд?

**ДА**

**НЕТ**

**НЕТ**

**НЕТ**

Ошибка

Минус унарный

(необходима проверка корректности ввода)

Очевидно, эта серия проверок намного сложнее и длиннее в реализации, чем второй вариант реализации, требующий сложной проверки на этапе ввода. По этой причине был выбран второй вариант.

# Заключение

1. Разработан интерфейс шаблонного класса TStack.
2. Реализованы методы шаблонного класса TStack.
3. Разработан интерфейс класса TPostfix для работы с постфиксной формой.
4. Реализованы методы класса TPostfix.
5. Разработано и реализовано консольное приложение, позволяющее вычислять значения арифметических выражений, содержащих базовые арифметические операции, унарный минус, функции sin(), cos(), tan(), exp(), ln(), а также позволяющее вводить любое количество переменных и предоставляющее возможность пересчёта значения выражения для различных значений переменных.
6. Разработаны и реализованы тесты для классов TStack и TPostfix на базе Google Test.
7. Исходный код опубликован в личном репозитории на GitHub.

# Литература

1. Сайт Habr. Польская нотация или как легко распарсить алгебраическое выражение – <https://habr.com/ru/articles/596925/>
2. Сайт Habr. Преобразование инфиксной нотации в постфиксную – <https://habr.com/ru/articles/489744/>

# Приложение

case 'x': {//если введена переменная

int res = 0;

i++;

while (i < inp.length() && inp[i] <= '9' && inp[i] >= '0')

{

res \*= 10;

res += inp[i] - '0'; //скопируй её имя во временную вспомогательную переменную.

i++;

}

i--;

size\_t j = 0;

while (j < ind\_variable && arr\_num\_variable[j] != res)

j++;

if (j == ind\_variable)//если такая переменная ещё не встречалась ранее

insert\_variable(res); //добавь её в массив имён переменных,

Lexeme tmp(4, j); //в соответствующую ей лексему в ОПЗ положи индекс этой переменной в массиве

insert\_result(tmp);

break;

}

if (quan\_open\_braсkets != 0) //сообщение об ошибке в случае если остались лишние скобки в стеке после завершения прохода по строке

{

string except("There are ");

except += to\_string(quan\_open\_braсkets);

except += " extra '(' in line: at positions ";

bool first = true;

while (!ops.isEmpty())

{

if (ops.top().type\_of\_lex == 2)

if (first)

{

except += to\_string(ops.pop().pos);

first = false;

}

else

except += ", " + to\_string(ops.pop().pos);

else

insert\_result(ops.pop());

}

throw invalid\_argument(except);

}

else while (!ops.isEmpty())

insert\_result(ops.pop());

arr\_value\_variable = new double[ind\_variable];