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 3821Б1ПМ2

Соколов И.Д.

**Проверил**:

преподователь каф. МОСТ,

Волокитин В.Д.

Нижний Новгород

2022

Содержание

[Введение 3](#__RefHeading___Toc687_2800270851)

[1. Постановка задачи 4](#__RefHeading___Toc689_2800270851)

[2. Руководство пользователя 5](#__RefHeading___Toc691_2800270851)

[3.1. Описание структуры программы 7](#__RefHeading___Toc693_2800270851)

[3.2. Описание алгоритмов 9](#__RefHeading___Toc695_2800270851)

[4. Результаты экспериментов 10](#__RefHeading___Toc697_2800270851)

[Заключение 11](#__RefHeading___Toc699_2800270851)

[Литература 12](#__RefHeading___Toc701_2800270851)

# Введение

Обратная польская нотация (ОПН) была разработана австралийским философом и специалистом в области теории вычислительных машин Чарльзом Хэмблином в середине 1950-х на основе польской нотации, которая была предложена в 1920 году польским математиком Яном Лукасевичем. Работа Хэмблина была представлена на конференции в июне 1957, и издана в 1957 и 1962.

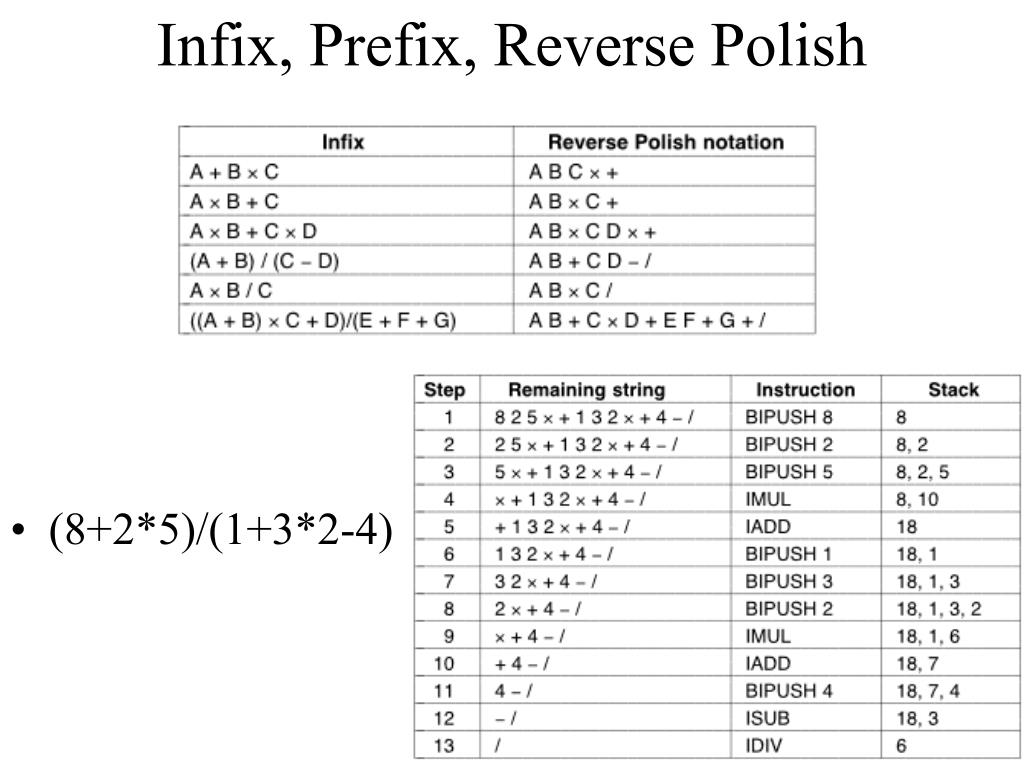
Чтобы дать индуктивное1 определение постфиксной нотации, обозначим выражения в инфиксной нотации ***E****,* ***E1****,* ***E2****,* эквивалентные им выражения в постфиксной нотации ***E`****,* ***E1`****,* ***E2`*** соответственно; ***o*** — произвольный бинарный оператор, тогда:

1. Если ***E*** — переменная или константа, то ***E`*** есть ***E****.*

2. Если ***E*** — выражение вида ***E1 o E2***, то ***E`*** есть ***E1` E2` o***.

3. Если ***E*** — выражение вида ***(E1)***, то ***E`*** есть ***E1`****.*

Для работы с постфиксной формой записи понадобятся функции, которые переводят арифметическое выражение в инфиксной форме в постфиксную форму записи, а также функции, которые высчитывают значение обратной польской записи.

Рисунок 1: Пример обратной польской записи

# Постановка задачи

Цель данной работы — разработка структуры данных Стек и ее использование для расчета арифметических выражений с использованием обратной польской записи (постфиксной формы).

Выполнение работы предполагает решение следующих задач:

1.Разработка интерфейса шаблонного класса TStack.

2.Реализация методов шаблонного класса TStack.

3.Разработка интерфейса класса TPostfix для работы с постфиксной формой.

4.Реализация методов класса Tpostfix.

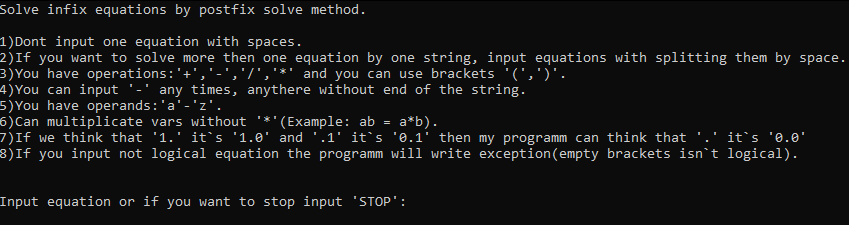
5.(Опцианально) Добавить обработку исключений.

6.Разработка и реализация тестов для классов TStack и TPostfix на базе Google Test.

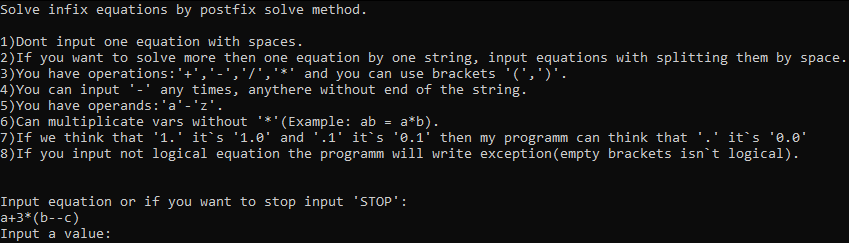
# Руководство пользователя

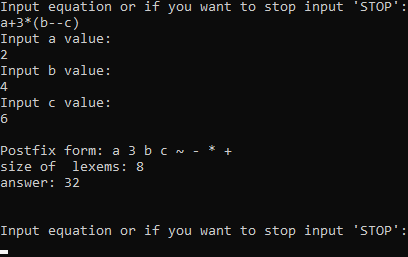
При запуске программы на экран будут выведены ограничения для данной программы, а также запись говорящая о том, что уже можно вводить арифметическое выражение(Рисунок 2).

При вводе арифметического выражения, если вы введете переменные, то дальше программа выведет на экран запись говорящую о том, что нужно проинициализировать переменную( будет появляться столько раз, сколько у вас различных переменных, поэтому если их нет, данная запись появляться не будет)(Рисунок 3).

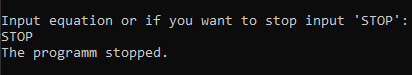
Рисунок 2

После инициализации всех переменных программа начнет считать значение выражение, после подсчета выведет на экран постфиксную форму записи выражения, количество лексем в этом выражении и ответ(Рисунок 4).

Рисунок 3

Рисунок 4

После этого можно вводить новое выражение. Так программа будет повторяться до бесконечности. Если захотите остановиться нужно ввести „STOP”.(Рисунок 5)

Рисунок 5

Важно отметить, что оператор „~”(тильда) в моей записи используется как обозначение унарного минуса и что лексемы это операнды или операции, не считая скобок. Также важно знать, что данная программа реализована для подсчета выражений только из даблов.

1. **Руководство программиста**

## Описание структуры программы

Хедер **stack.h**

В данном хедере реализуется шаблонный класс стека – ***Tstack***.

В нем реализованы следующие методы класса:

Конструктор ***Tstack()*** - создает пустой стек.

Метод ***void push\_back(T value)*** – добавляет в стек шаблонный элемент ***value***.

Метод ***T pop()*** – возвращает верхний элемент стека и убирает его из стека.

Метод ***T show()*** – возвращает верхний элемент стека, без его удаления из стека.

Метод ***bool IsEmpty()*** – возвращает 1, если в стеке нет элементов, иначе возвращает 0.

Метод ***bool IsFull()*** – возвращает 1, если стек заполнен, иначе возвращает 0.

метод ***size()*** – возвращает количество элементов в стеке.

деструктор ***~Tstack()*** – очищает стек.

Хедер **arithmetic.h**

В данном хедере написан класс **TPostfix** без реализации своих методов.

Данный класс нужен, чтобы хранить арифметическое выражение.

Он хранит его в двух видах, в инфиксной форме и в постфиксной, а также хранит количество лексем в постфиксной форме и значения всех переменных.

Таким образом можно перечислить поля класса **Tpostfix**:

string infix – инфиксная форма выражения

***Lexem***\*\****postfix*** – постфиксная форма выражения

***size\_t*** ***size*** – количество лексем в постфиксной форме выражения

***map***<***char***,***double***> ***vars*** – значения переменных из выражения(first→ имя переменной, second → значение)

Также реализован класс **Lexem**, с его наследниками **Operation**, **Operand**, с его наследниками **Var** и **Value**.

**Lexem** реализован как абстрактный класс с одной чистой виртуальной функцией:

***virtual*** ***string*** ***show***() = 0; и единственным полем ***string*** ***value***.

Класс **Operand** нужен, чтобы ограничить понятия операции и операнда, поэтому в нем ничего не реализовано.

Методы классов **Operation**, **Var** и **Value**:

конструктор ***Var***(***char*** ***str***) – инициализирует ***value*** как ***str***.

конструктор ***Value***(***string*** ***dbl***) – инициализирует ***value*** как ***str***.

конструктор ***Operation***(***char*** ***str***) – инициализирует ***value*** как ***str***.

метод ***string*** ***show***() – возвращает значение поля ***value***.

Файл **arithmetic.cpp**

В данном файле реализуются все методы класса **TPostfix**:

конструктор ***TPostfix***::***TPostfix***(***string*** ***str***) – создает экземпляр класса, для которого выражение ***str*** будет записано в поле ***infix***.

***void*** ***Tpostfix***::***infix\_check***() – метод класса, который проверяет выражение записанное в поле infix на всевозможные логические ошибки и ошибки, которые идут против ограничений. Если такие есть бросает исключение.

***void*** ***Tpostfix***::***to\_postfix***() – метод класса, который превращает инфиксную форму записи в постфиксную(***infix*** → ***postfix***).

***string*** ***Tpostfix***::***showP***() – метод класса, который выводит на экран выражение записанное в постфиксной форме.

***void*** ***TPostfix***::***set\_infix***(***string*** ***infx***) – задает новое арифметическое выражение ***infx***.

***size\_t*** ***Tpostfix***::***get\_size***() – метод класса, который возвращает количество лексем в постфиксной форме выражения (***size***).

***double*** ***Tpostfix***::***CALCULATE***() – метод класса, который считает значение арифметического выражения в постфиксной форме и возвращает его.

Деструктор ***Tpostfix***::~***TPostfix***() – очищает ***postfix*** и ***vars*** при необходимости.

Также была реализована функция перевода числа из строки в дабл:

***double*** ***string\_to\_double***(***string*** ***s***) – функция, которая интерпретирует строку ***s*** как число, и возвращает его в виде ***double***. Также стоит отметить, что данная функция поддерживает экспоненциальную форму записи дабла.

## Описание алгоритмов

Алгоритм перехода из инфиксной формы записи в постфиксную выглядит сследующим образом:

1. Если это константа или переменная, то кладем в постфиксную форму.

1.1 Если это переменная, а до переменной была другая переменная без операции между ними, то кладем переменную в постфиксную форму и кладем умножение в стек.

2. Если это операция, то пока в стеке найдется операция не меньшая по приоритету, берем операцию из стека и кладем в постфиксную форму. Затем кладем текущую операцию в стек.

3. Если это открывающая скобка, кладем ее в стек.

4.Если это закрывающая скобка, пока в стеке верхним элементом не будет открывающая скобка, кладем элементы из стека в постфиксную форму.

5.Если в конце строки в стеке остались элементы, кладем их в постфиксную форму.

Важно уточнить приоритет операций, чем меньше число тем приоритетней операция:

‘~’ – 0

‘\*’ – 1

‘/’ – 1

‘+’ – 2

‘-’ – 2

Алгоритм подсчета выражения в постфиксной форме выглядит следующим образом:

1.Если это операнд, то кладем его в стек.

2.Если это операция, то то выполняем операцию AoB, где ‘o’ – операция, ‘B’ – верхний элемент стека, ‘A’ – второй верхний элемент стека, и результат выполнения кладем в стек.

2.1 Если это унарный минус, то берем верхний элемент стека, меняем знак, кладем в его в стек.

3. После выполнения алгоритма над последней лексемой, берем элемент из стека, он и будет ответом.

# Результаты экспериментов

Для подтверждения корректности программы, были подключены и реализованы Google Tests. В тестах были реализованы проверки на корректность работы всех функций, а также были написаны тесты подтверждающие корректное вычисление для самых различных арифметических выражений.

# Заключение

В результате проведенной лабораторной работы, были разработаны и реализованы методы шаблонного класса TStack, разработаны и реализованы методы класса TPostfix для работы с постфиксной формой. Для класса TPostfix была добавлен метод для обработки исключений. Были реализованы Google Tests для проверки корректности.

# Литература

*Т. Пратт, М. Зелковиц* – Языки программирования: разработка и реализация