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 арифметический выражений»**

**Выполнил**:

студент группы 3821Б1ПМ2

Соколов В.Д.

**Проверил**:

Преподаватель каф. МОСТ,

Волокитин В.Д.

Нижний Новгород

2022

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

Данный документ представляет собой отчёт о проделанной работе. Отчёт написан для третьей лабораторной работы второго курса третьего семестра. Название лабораторной работы:

<<Вычисление арифметический выражений>>

Код программы, а также прилагающиеся к нему отдельные файлы лежат в папке с данным отчётом. При запуске программы в свойствах указать на сборку текущего проекта.

Ссылка на мой репозиторий: <https://github.com/Sokolov-Vladisl/mp2-lab3-Sokolov_Vlad>

Если есть вопросы, то мои контактные данные:

Почта: [vladsokolov2001@gmail.com](mailto:vladsokolov2001@gmail.com)

Skype login: live:.cid.69519504a5fbe14

Skype invite: <https://join.skype.com/invite/FBYFbm6ePTKm>

Номер телефона (лучше присылать СМС): +79307048350

VK: https://vk.com/id608399016

GitHub мой профиль: https://github.com/Sokolov-Vladisl

# Постановка задачи

Цель данной лабораторной работы – реализовать (на языке программирования C++, используя Visual Studio) программу, принимающую на вход выражение и на выводе дающую результат данного выражения. Создать свой шаблонный класс Стек. Изменение прямой формы записи выражения в постфиксную форму (обратная польская запись). Создать класс Лексем с наследуемыми классами. Создание тестов (Google Test), проверяющих корректность работы программы, для класса Стек и для функций перевода выражения в постфиксную форму.

# Руководство пользователя

При запуске программы первое, что пользователь увидит будет:

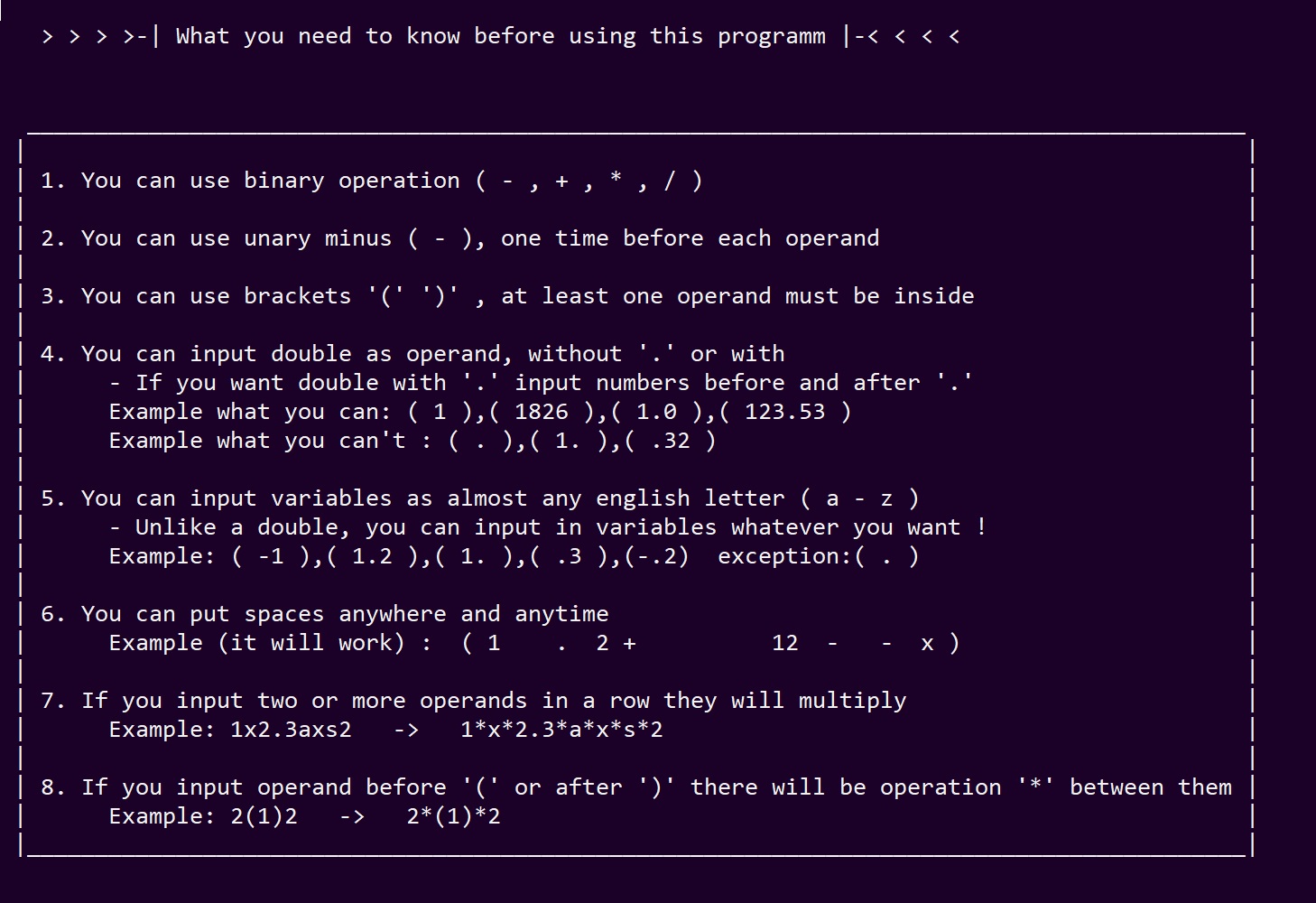


Рисунок Ознакомление

Это ознакомление. Здесь пользователь сможет ознакомиться с некоторыми правилами и особенностями программы. Ознакомившись со всеми восемью пунктами, пользователь уже будет иметь некоторое представление о программе.

Далее пользователю просто нужно ввести некоторое выражение в обычной форме записи. Пользователь может писать сколько угодно операций, операндов, скобок, но стоит помнить, что у самой строки есть ограничение на 256 символов (включая пробелы). Также пользователь может повторять уже использованные переменные (это уже было показано в пункте 7 на рисунке 1).



Рисунок Ввод выражения

Далее мы будем описывать работу программы на примере (рисунок 3):

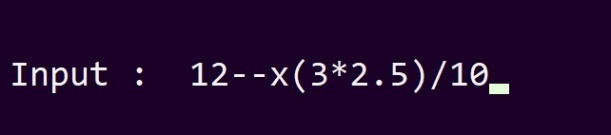


Рисунок Пример

Программа покажет нам вариант нашего ввода, но уже без пробелов. Затем нам предстоит важный выбор.

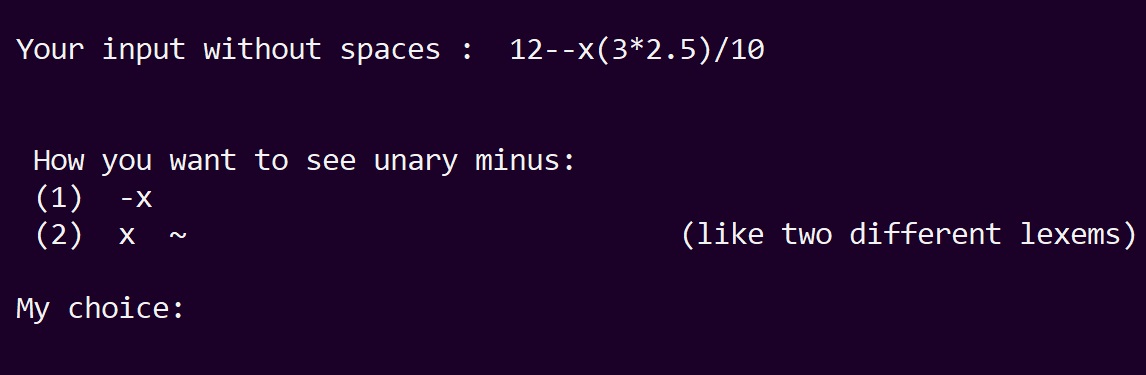


Рисунок Первый выбор

В связи с тем, что унарный минус это особая операция и она крайне неоднозначна для представления в постфиксной форме, пользователю предлагается выбор:

1. Минус в лексеме будет просто стоять слитно со своим операндом.
2. Представить минус как отдельную лексему, которая влияет на стоящий перед ней операнд.

Данный выбор будет в основном влиять на сопутствующие данные постфиксной формы (как например на размер). Пользователю не стоит заморачиваться и он может выбрать любой режим, который ему будет удобней.

В данном примере выбор: 2

После предыдущего выбора, программа начнёт работу и здесь могут вызваться исключения из-за каких-то ошибок. Но так как программа ничего плохого в нашей записи не нашла, она продолжает работать и пользователю будет предоставлен другой выбор.

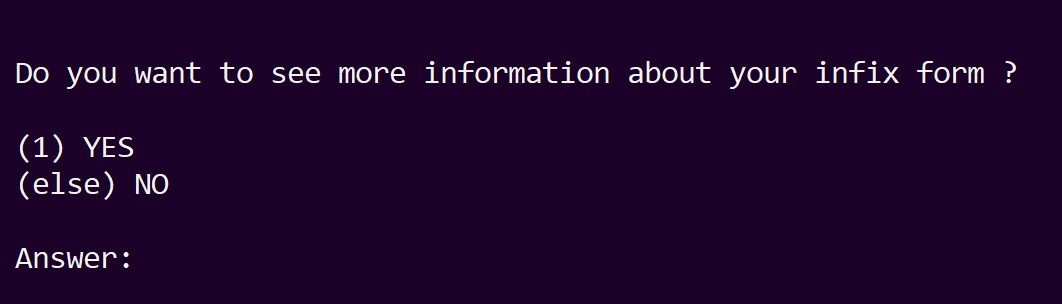


Рисунок Второй выбор

Программа уже успела выполнить огромную функцию разделяя наше выражение на лексемы, поэтому мы можем посмотреть на некоторые промежуточные результаты.

Выбор: 1

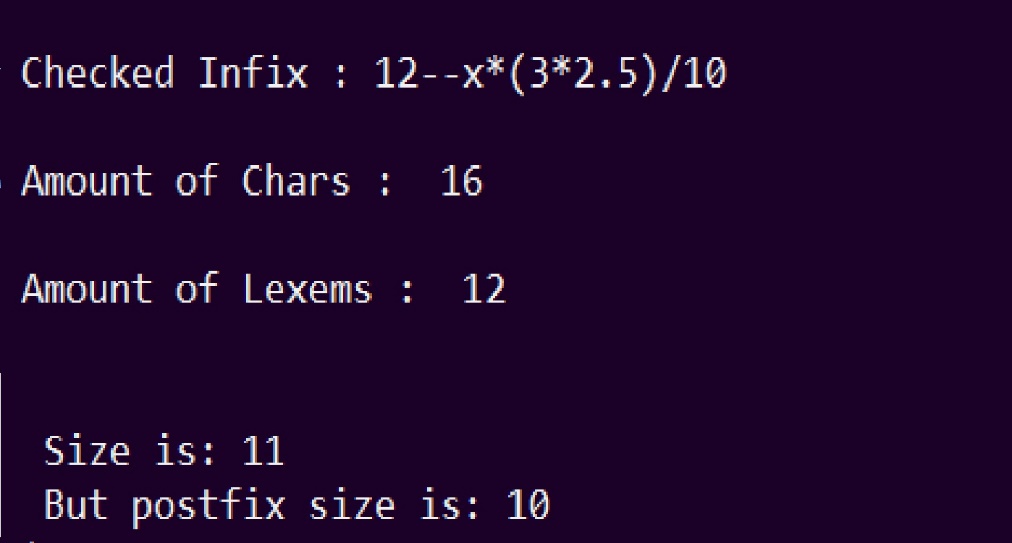


Рисунок Промежуточная информация

В первой строке будет выведена проверенное выражение. Хоть на первый взгляд и кажется, что ничего не изменилось, но на самом деле добавился новый знак. Если в исходном выражении, которое писал пользователь, умножение было слитно (пример: xy, x(2), 2.3x), то теперь программа показывает данную операцию.

На следующей строке выводится количество символов проверенного выражения. На другой строке - количество лексем.

Затем показаны размеры:

1)Показывает сколько элементов всего (здесь унарный минус и операнд - это один элемент)

2)Показывает размер постфиксной формы (без скобок, унарный минус - отдельный элемент)

Здесь заканчивается показ промежуточных результатов.

Далее пользователь видит саму постфиксную форму его выражения.

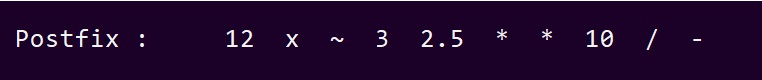


Рисунок Постфиксная форма 1

Результат выбора, если пользователь выбрал представление унарного минуса слитно:

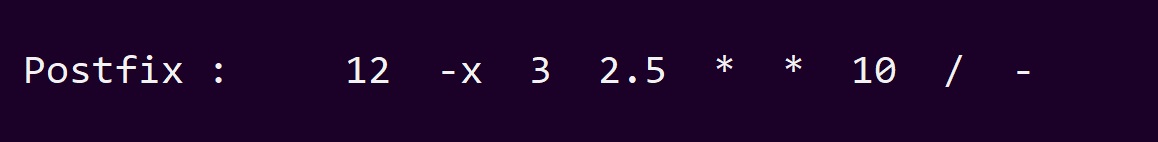


Рисунок Постфиксная форма 2

Здесь программа уже разложила лексемы в постфиксную форму и осталось лишь узнать значение выражения.

Но перед этим пользователю нужно ввести значение переменных:

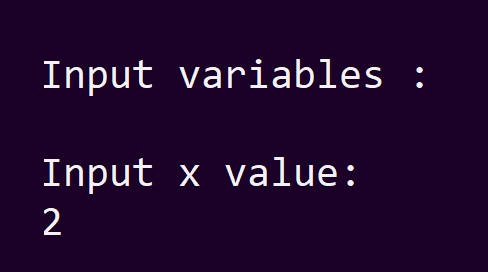


Рисунок Ввод переменных

Здесь пользователь вводит значение переменных в алфавитном порядке. Если переменных нет в выражении, то появится строка, в которой будет сказано об этом. Также, если пользователь введёт некорректное значение (например: 2g4), то ему будет предоставлена очередная попытка ввода.

В данном примере x=2

Далее пользователь сможет увидеть значения используемые в выражении

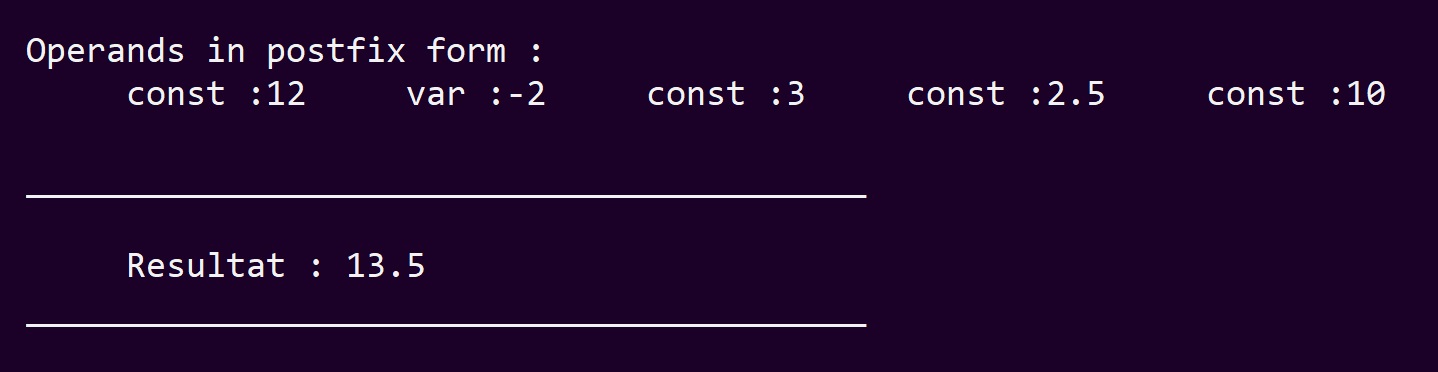


Рисунок Ответ

В одну строку будут выписаны все операнды и их значения, так как пользователь уже ввёл значения переменных. Порядок операндов будет идти как в постфиксной форме с лева на право.

Затем пользователь сможет увидеть результат. В случае, если в выражении при вычислении получится ситуация, где какой-то операнд делится на ноль, то будет вызвано исключение.

Но это ещё не конец. Теоретически (невозможно) программа может заработать и ответ будет неверным. Тогда на этот случай пользователь может посмотреть полную информацию о префиксной форме выражения.

Также пользователь может посмотреть эту информацию и по своему желанию.

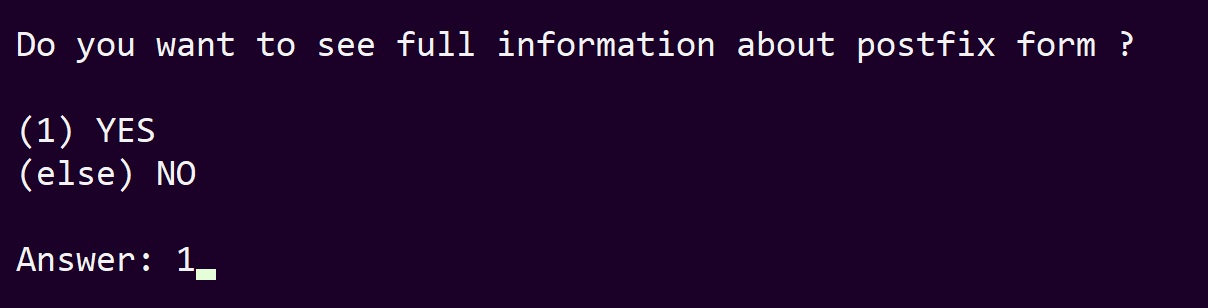


Рисунок Второй выбор

Выбор: 1

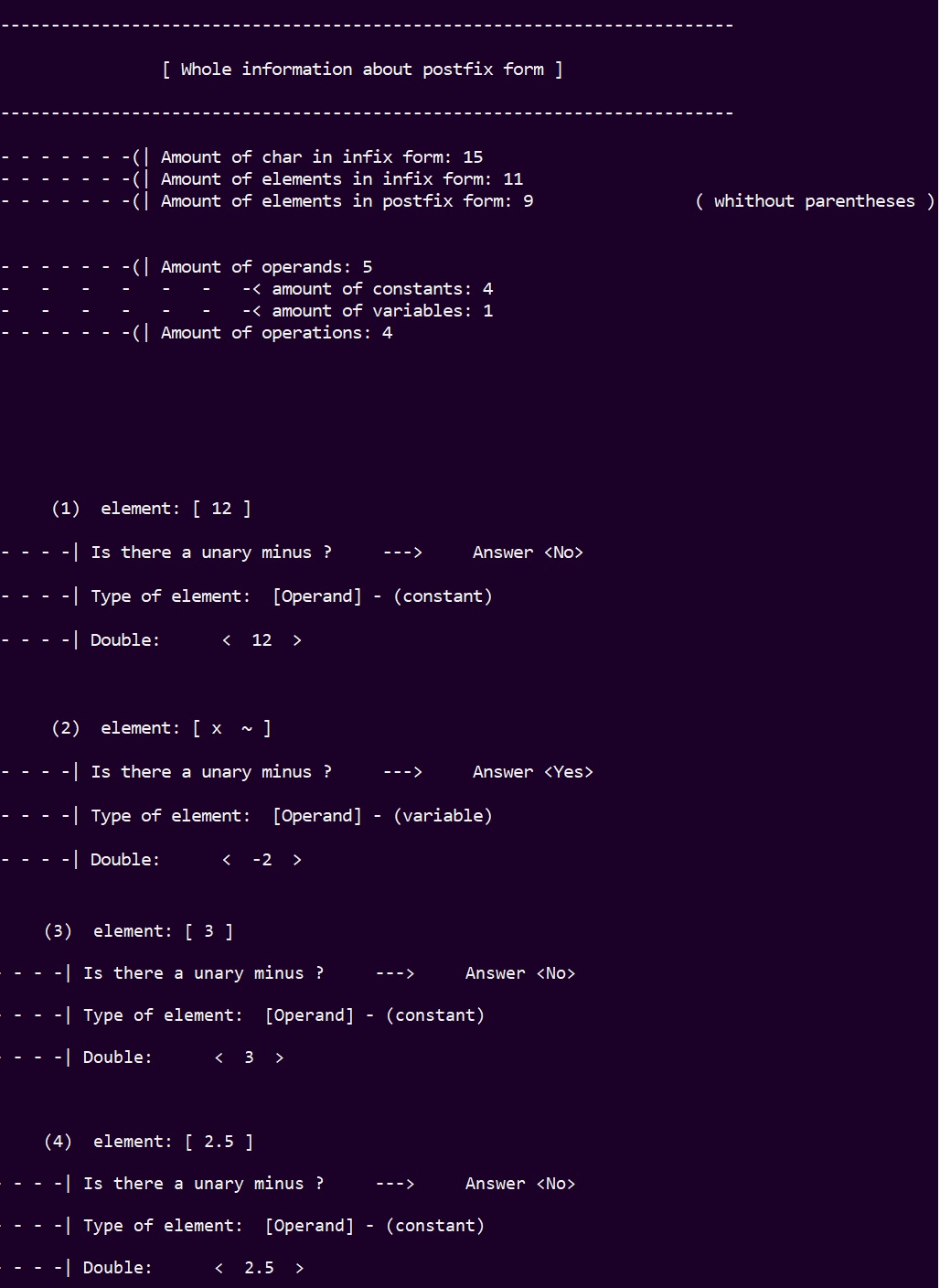


Рисунок Вся информация о постфиксной форме - 1

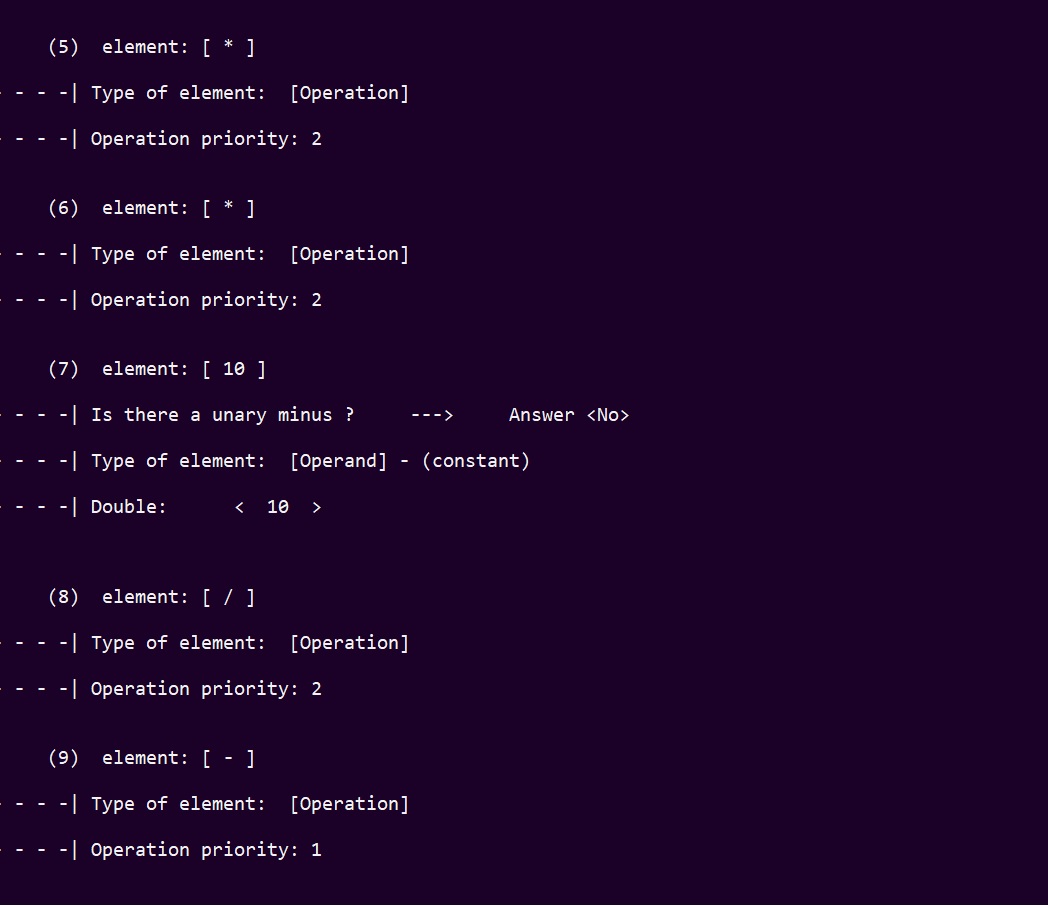


Рисунок Вся информация о постфиксной форме - 2

Операнд с унарным минусом пишется в одном элементе т.к. они напрямую связаны.

Напоминание! Пользователь может не выводить данную информацию, если она ему не нужна.

Программа завершает свою работу.

1. **Руководство программиста**

## Описание структуры программы

Самое важное!

В main:

TPostfix MainPart(infix, alternative,0);

MainPart.to\_postfix\_form(alternative,0);

MainPart.calculation(0);

Параметр 0 стоит по умолчанию. В каждой функции есть вывод строк на экран.

Заменив 0 на 1, мы избавляемся от вывода строк. Это было сделано, чтобы в google tests не выводились как раз эти строки. Если программист будет использовать этот альтернативный режим, то рекомендуется не пользоваться переменными. Если всё-таки переменные понадобятся, то смотри руководство пользователя приложение к рисунку 9.

В данной программе содержатся три основных класса:

1. Шаблонный класс Tstack
2. Класс LEXEM , который включает подклассы:
3. Operation
4. Operand

a)Const

b)Var

1. Класс TPostfix
2. **Tstack**

Класс Tstack имеет два конструктора. Первый принимает на вход целое число, которое будет являться размером стека, а второй не принимает на вход ничего – размер в нём равен единице. Также класс содержит в себе индекс вершины стека(int top) и размер стека (size\_t memSize). Содержится деструктор , очищающий поле pMem, в котором находятся значения стека.При вызове класса необходимо указать тип данных T : Tstack<T>.

Tstack содержит следующие методы:

bool isEmpty() - проверка на пустоту

bool isFull() – проверка на заполненность

T Top() – возвращает верхний элемент стека

Size\_t size() – показывает размер стека (сколько элементов)

T Pop() – возвращает и удаляет верхний элемент стека

Void Push(T a) – кладёт элемент на вершину стека

Перевыделение памяти происходит автоматически если вызван метод Push, а стек переполнен. Память для стека в этом случае увеличивается вдвое.

**2)LEXEM**

Класс **LEXEM** – это абстрактный класс. От него наследуются класс Operation и абстрактный класс Operand. От Operand наследуются класс **Const** и класс **Var**.

Так как класс **LEXEM** и класс Operand абстрактные, они не имеют своих конструкторов.

**LEXEM** содержит следующие поля:

int type - тип элемента (1-константа, 2-переменная, 3-операция, -1 это ‘(‘, -2 это ‘)’.

string lexem\_m - строка ( элемент )

string alter\_lexem\_m - строка ( элемент , только уже альтернативная версия )

**LEXEM** содержит следующие методы:

virtual char get\_char\_op() – возвращает char ( переменной или операции )

virtual bool isOperation() – проверка на операцию

virtual bool isOperand() – проверка на операнд

virtual double get\_double() – возвращает значение **Const** как числа, а также определяет знак Operation

virtual int get\_priority() – функция возвращающая приоритет операций ( “+,-“ = 1, “\*,/” = 2, “ ( , ) “ = 10 )

virtual void shop(bool alternative) – показывает lexem если 0 , alter\_lexem\_m если 1

string show() – возвращает копию lexem\_m ( вызывается чтобы показать скрытые операции )

string get\_lexem\_m() – возвращает lexem\_m

string get\_alter\_lexem\_m() – возвращает alter\_lexem\_m

int get\_type() – возвращает тип элемента

Класс **Operation** имеет два конструктора: один пустой, другой принимает на вход char операции. **Operation** содержит следующие поля:

char lexem\_Op – char операции

int priority = 10 – сразу равно 10, даётся скобкам

Класс **Const** имеет два конструктора: один пустой, другой принимает два одинаковых стека с char, знак (1 или -1) и режим ( -x или x ~ ). Принимаются два одинаковых стека, а не один, потому что из одного строят число (и он исчезает), а из другого строку. **Const** содержит ещё одно поле:

double lexem\_const – значение **Const**

Класс **Var** имеет два конструктора: один пустой, другой принимает на вход char переменной, знак( 1 или -1) и режим ( -x или x ~ ). **Var** содержит следующие поля:

char lexem\_var – char переменной

int var\_znak – знак переменной (по умолчанию = 1)

**2)TPostfix**

Класс **Tpostfix**, без преувеличения, основной класс в программе. Именно этот класс вызывается в main, его методы решают необходимую пользователю задачу. **TPostfix** имеет всего один гонструктор, в котором главную роль играет функция search. Деструктор удаляет массивы лексем( mas, postfix\_mas)

Класс **TPostfix** содержит следующие поля:

int secret\_operations – количество операций написанных в вводе слитно, но на деле имеющие свой char (пример: xz это на самом деле x\*z )

int infix\_len – длина строки с учётом secret\_operations

int c\_k1, c\_k2 – количество открывающихся и закрывающихся скобок.

LEXEM\*\* mas – массив лексем необходимых, для создания postfix\_mas

LEXEM\*\* postfix\_mas – массив лексем в постфиксной форме

int L\_Size – количество лексем

int true\_size – количество лексем (без учёта скобок), размер постфиксной формы

map<char, double> vars – значения переменных

Класс **TPostfix** содержит следующие методы:

void search(string infix,bool alternative,bool test) – нахождение арифметических ошибок, создание массива лексем. Принимает на вход выражение (string), режим( -x или x ~ ).

void to\_postfix\_form(bool alternative, bool test) – из первого массива лексем(mas) создаёт второй(postfix\_mas) в постфиксной форме. Принимает на вход режим( -x или x ~ ).

void checked\_infix(bool alternative) – показывает выражение, только теперь построенное из лексем. Показывает количество символов в выражении, количество лексем и элементов, размер постфиксной формы. Принимает на вход режим( -x или x ~ ).

double calculation(bool test) – высчитывает и возвращает результат выражения.

void get\_full\_information(bool alternative) – показывает общую информацию постфиксной формы. Принимает на вход режим( -x или x ~ ).

void get\_quick\_info(bool alternative) – показывает информацию о лексемах. Принимает на вход режим( -x или x ~ ).

int get\_infix\_len() – возвращает длину строки с учётом secret\_operations

int get\_c\_k1() – возвращает количество открывающихся скобок.

int get\_c\_k2() – возвращает количество закрывающихся скобок

int get\_true\_size() – возвращает количество лексем (без учёта скобок), размер постфиксной формы.

int get\_L\_Size() – возвращает количество лексем

Функции не являющиеся методами классов :

string without\_spaces(string infix) – возвращает строку без пробелов.

int get\_priority\_outside(char a) – возвращает приоритет char операции.

bool choice() – Выбор режима.

## Описание алгоритмов

В данной программе используется три основных алгоритма.

1. **“Перестройка” исходного выражения**. Выражение это string, что можно представить в массив из char. Не меняя последовательности символов, мы из одних символов собираем операнды, из других операции. В особых случаях мы можем добавить некоторые операции ( \* ). Таким образом получается алгоритм:

Создаём массив лексем:

Делаем пока не закончится строка:

1.Если минус , за ним число или переменная, а перед ним операция (или ничего), то это унарный минус ( - ).

2.Если число, за ним и перед ним нет переменной, то это double

3. Если переменная, за ней и перед ней нет числа, то это variables

4.Если бинарная операция, не в начале и не в конце, не перед другой операцией, если за операцией нет закрывающей скобки, то это Operation

5.Если скобки, внутри есть хотя бы один операнд, количество открывающих скобок больше закрывающих (или равно если это откр. Скобка), то это Operation.

6.Иначе неизвестный элемент – ошибка.

Если количество открывающих скобок не равно количеству закрывающих, то ошибка.

Найдя количество элементов, перераспределяем память массива лексем.

1. **Алгоритм перевода элементов из инфиксной формы в постфиксную.**

Создаём массив лексем – это будет постфиксная форма. Размер равен количеству элементов (не включая скобок).

Делаем пока элементов в инфиксной форме не останется :

1.Если операнд, то кладём в постфиксную форму

2.Унарный минус тоже кладём в постфиксную форму, т.к. он идёт сразу после операнда с которым связан

2.Если открывающая скобка, то кладём в стек

3.Если закрывающая скобка, то пока на вершине стека не будет открывающей скобки мы перемещаем элемент из стека в постфиксную форму. Затем (после while) убираем открывающую скобку из стека.

4. Если операция, то пока приоритет лексемы меньше или равен приоритета верхнего элемента стека мы перемещаем элемент из стека в постфиксную форму. Затем (после while) помещаем данную операцию в стек

Если лексем в инфиксной форме больше не останется, то перемещаем все элементы из стека в постфиксную форму.

1. **Алгоритм вычисления арифметических выражений.**

Сначала вводим значение переменных.

Цикл от нуля до количества лексем (без учёта скобок) :

1.Если операнд, то кладём его значение в стек

2.Если унарный минус, то умножаем последний элемент стека на -1.(вынимаем, умножаем, ставим)

2.Если операция то берём первый элемент из стека, второй, а затем используем операцию между вторым и первым. Кладём в стек результат.

(если деление на ноль, то ошибка)

Берём и стека единственный элемент и записываем его как результат.

Возвращаем результат.

# Результаты экспериментов

Для проведения экспериментов были разработаны двадцать пять тестов на базе Google Test.

1. Тесты на Tstack
2. can\_be\_created\_with\_different\_types - Можно создавать стек с различными типами данных ( int, double, float, char, unsigned int )
3. can\_be\_created\_with\_template\_types - Можно создавать стек со своим типом данных(Tstack<my\_class> st)
4. correct\_Push\_and\_Pop - проверка работы Push и Pop
5. correct\_check\_of\_Full\_and\_Empty – проверка на пустоту и на полноту
6. correct\_size – проверка на правильное количество элементов
7. create\_with\_already\_existed\_memory\_size – создать стек с уже определённой памятью
8. Тесты на Aritmetic
9. Infix\_form, empty\_string – ошибка если пустая строка
10. Infix\_form, infix\_witout\_spaces – правильно удаляет пробелы
11. Infix\_form, arithmetic\_check – находит арифметические ошибки
12. class\_Const, get\_double – правильно высчитывает double
13. class\_Const, get\_minus – правильно ставит минус (унарный)
14. class\_Const, is\_operand\_only – Const это операнд
15. class\_Const, is\_constanta\_type – Const имеет тип константа (1)
16. class\_Var, get\_minus – правильно ставит минус (унарный)
17. class\_Var, is\_operand\_only – Var это операнд
18. class\_Var, is\_variable\_type – Var имеет тип переменная (2)
19. class\_Operation, is\_operation\_only – Operation это операция
20. class\_Operation, get\_operation\_char – правильно берёт символ
21. class\_Operation, correct\_priority\_set – правильно расставляет приоритеты
22. class\_Operation, correct\_type\_set – правильно расставляет тип
23. class\_TPostfix, infix\_correct\_size – правильная длина строки
24. class\_TPostfix, lexem\_correct\_size – правильное количество лексем
25. class\_TPostfix, parentheses\_correct\_size – правильное количество скобок
26. class\_TPostfix, Postfix\_work – Вычисления дают правильный результат
27. functions, get\_priority\_outside\_work\_properly – правильно расставляет приоритет

# Заключение

В результате проведения лабораторной работы был реализован собственный шаблонный класс Стек. Также были реализованы абстрактный класс LEXEM с подклассами и класс TPostfix. Был построен алгоритм вычисления арифметических выражений с использованием стека. Были реализованы тесты на базе Google Test для проверки корректности работы программы.

# Литература

1.Лекции Сысоева А.В.

# Приложение

https://github.com/Sokolov-Vladisl/mp2-lab3-Sokolov\_Vlad