Отчет по лабораторной работе №3

Предмет: «Алгоритмы и структуры данных»

Тема лабораторной работы: «Вычисление арифметических выражений»

Выполнил:

Студент

Курс: 2

Группа: 3824Б1ФИ2

ФИО: Большаков Владислав Владимирович

2025 г.

Содержание

1. Постановка задачи
2. Описание класса TStack, методов класса TStack
3. Описание класса TPostfix, методов класса TPostfix
4. Описание тестов Google Test

1. Постановка задачи

Задача: разработка структуры данных стек (класс **TStack**) и её использование для вычисления арифметических выражений. Разработка производится на языке программирования C++. В качестве структуры данных для хранения и обработки арифметических выражений используется класс **TPostfix** (в работе методов данного класса применяются методы класса TStack). Для проверки корректности работы реализованных классов производится написание автоматических тестов Google Test.

Условия и ограничения:

- Арифметическое выражение в качестве операндов может содержать переменные и вещественные числа.
- Допустимые операции: сложение (+), вычитание (-), умножение (*), деление (/). Также допустимо применение функции косинус (cos ()), и унарного минуса (в постфиксной форме обозначен знаком ~). Для функции косинус введено ограничение: для применения данной функции необходимо наличие круглых скобок, иначе запись воспринимается как имя переменной (например: cos(x) – это функция, cosx – имя переменной).
- Переменные обозначаются строчными и прописными буквами латинского алфавита.
- Программа должна выполнять предварительную проверку корректности выражения и сообщать пользователю вид ошибки и номер символа строки, в котором была найдена ошибка (первый символ, в котором была найдена ошибка).
- При вычислении арифметического выражения требуется ввод с консоли неизвестных переменных.

2. Описание класса TStack, методов класса TStack.

Класс TStack реализует структуру данных **стек** (**stack**). Основное предназначение данного класса – хранение набора элементов с доступом только к последнему помещенному элементу. Является шаблонным классом, использует такой контейнер языка C++ как вектор (std::vector).

Описание полей (спецификатор доступа **private**):
Содержит единственное поле pStack, которое является объектом класса std::vector.
Используется для хранения и работы с данными.

Описание методов (спецификатор доступа **public**):

Конструктор:
TStack() = default - используется конструктор по умолчанию (т.е. конструктор класса std::vector)

bool empty():

- Описание параметров и возвращаемых значений: возвращает значение типа bool, является константным методом.
- Описание реализуемого функционала: использует метод класса std::vector **empty**, который возвращает true, если вектор не содержит ни одного элемента.

size_t size():

- Описание параметров и возвращаемых значений: возвращает значение типа size_t, является константным методом.
- Описание реализуемого функционала: использует метод класса std::vector **size**, который возвращает количество элементов, содержащихся в векторе.

void push(const T& val):

- Описание параметров и возвращаемых значений: принимает ссылку на константу типа T (T – шаблонный тип), возвращает тип void.
- Описание реализуемого функционала: использует метод класса std::vector **push_back**, который добавляет в конец вектора передаваемое значение (в данном случае передаваемое значение – константа типа T).
- Оценка сложности: в худшем случае O(n) (если приходится использовать перепаковку для включения нового элемента; n – общее количество элементов).

void pop():

- Описание параметров и возвращаемых значений: возвращает тип void.
- Описание реализуемого функционала: удаляет элемент, находящийся на вершине стека. Используется метод класса std::vector **pop_back**, который удаляет последний элемент, содержащийся в векторе. В случае отсутствия элементов (стек пуст) выбрасывается исключение типа **runtime_error**.

T top():

- Описание параметров и возвращаемых значений: возвращает значение типа T, является константным методом.
- Описание реализуемого функционала: возвращает значение элемента, находящегося на вершине стека (сам элемент при этом не удаляется из стека). Используется метод класса std::vector **back** (возвращает значение последнего элемента в векторе). Если стек пуст, то бросается исключение **runtime_error**.

void swap(TStack<T>& other):

- Описание параметров и возвращаемых значений: принимает ссылку на объект данного класса, возвращает тип void.
- Описание реализуемого функционала: используется для обмена значениями между двумя стеками (даже если один из стеков пуст). Используется метод класса std::vector **swap** (обменивает значения двух векторов). Данный метод не задействовался при реализации класса TPostfix.

3. Описание класса TPostfix, методов класса TPostfix.

Класс TPostfix предназначен для хранения и работы с арифметическими выражениями. Данный класс хранит инфиксную и постфиксную записи выражения (соответственно преобразует исходное выражение в постфиксную форму), проверяет исходное выражение на корректность, производит вычисление арифметического выражение (возможен ввод неизвестных переменных с консоли, или же передача вектора значений неизвестных переменных в соответствующий метод).

Описание полей (спецификатор доступа **private**):

- infix – объект класса std::string, хранит исходное выражение в инфиксной форме
- postfix – объект класса std::string, хранит постфиксную форму исходного выражения
- lexems_infix – объект класса std::vector, который хранит элементы типа std::string. Данное поле хранит лексемы инфиксной формы выражения.
- lexems_postfix – аналогично полю lexems_infix, только данное поле хранит лексемы постфиксной формы выражения.

- operands - объект класса std::vector, который хранит элементы типа std::string. Данное поле хранит уникальные операнды инфиксной формы (операнды записаны в том порядке, в каком были в исходном выражении)
- value_operands - объект класса std::vector, который хранит элементы типа double. Поле предназначено для хранения значений операндов, содержащихся в поле operands.
- operations - объект класса std::vector, который хранит элементы типа std::string. Данное поле хранит допустимые операции.
- priority - объект класса std::vector, который хранит элементы типа int. Данное поле хранит приоритеты операций, описанных в поле operations.

Описание методов (спецификатор доступа **private**):

void InfCorrect (const std::string& inf):

- Описание параметров и возвращаемых значений: принимает ссылку на константу типа std::string, возвращает тип void.
- Описание реализуемого функционала: конструктор класса принимает арифметическое выражение в виде строки. Данное выражение нужно проверить на корректность, для чего и используется этот метод. С помощью цикла for метод посимвольно проходится по строке и обнаруживает ошибки. Запрещенные ситуации:
    - Использование некорректных символов (некорректные символы – символы отличные от знаков операций, круглых скобок, цифр, букв латинского алфавита (также запрещены пробелы));
    - Использование символов непосредственно перед открывающейся скобкой или непосредственно после закрывающейся (например: a(…), 2(…), (…)a, (…)2); Использование вложенных скобок разрешено (например: “((a+b))”);
    - Использование пустых скобок (“()”);
    - Некорректное число скобок в выражении (недостаточное количество открывающихся или закрывающихся скобок);
    - Некорректное использование точки (перед точкой и после неё могут быть только цифры. Пример некорректной записи: “a+b.1”). Также запрещено использование нескольких точек в числовом операнде (например: “a+1.34.56”), использование точки в самом начале выражения, в самом конце;
    - Слияние букв и цифр (например: a+b234);
    - Использование операций (+, *, /, -) в начале выражения (например: +a*3; не касается унарного минуса) или в конце, использование операции после другой операции (a+-b) (касается операций: +,-,*, /), использование операции сразу после открывающейся скобки (не касается унарного минуса) или перед закрывающейся скобкой.

  Если возникла запрещенная ситуация, то метод выбрасывает исключение **invalid_argument**, в сообщении указывает вид ошибки и номер символа строки, в котором она была обнаружена.
- Оценка сложности: цикл for по каждому символу строки, также в методе создается объект класса TStack, используется его метод push, используется цикл while, который проходит по всем символам, формирующим числовой операнд (если количество точек в числе достигло двух, прекращает работу). Итоговая сложность O(n), где n – количество символов в строке.

Forming_operands (const std::vector<std::string>& lexem_inf):

- Описание параметров и возвращаемых значений: принимает ссылку на константный объект типа std::vector, который содержит в себе элементы типа std::string; возвращает тип void.
- Описание реализуемого функционала: данный метод используется в методе Parse, который предназначен для формирования поля lexems_infix. Как только поле lexems_infix сформировано вызывается метод Forming_operands, который выделяет уникальные операнды из набора лексем и добавляет их в поле operands.
- Оценка сложности: выделение уникальных операндов производится с помощью двух циклов for. Первый цикл проходит по всем лексемам. Если лексема числовой операнд или переменная, то переходим ко второму циклу for. Данный цикл проходится по элементам поля operands (которое изначально пустое). Если выбранная лексема не содержится в поле operands она добавляется в данное поле. Итоговая сложность: O(n*m), n – количество элементов в поле lexems_infix, m - количество операндов.

Values(const std::vector<std::string>& operand):

- Описание параметров и возвращаемых значений: принимает ссылку на константный объект типа std::vector, который содержит в себе элементы типа std::string; возвращает тип void.
- Описание реализуемого функционала: с помощью цикла for метод проходит по элементам поля operands. Если строковый элемент представляет собой число, то далее формируется переменная типа double, которая добавляется в поле value_operands (строка «3.14» ─> double number = 3.14). Если элемент – это переменная, то соответствующая ячейка поля value_operands получает значение 0.
- Оценка сложности: используется два цикла for. Первый проходит по каждому элементу поля operands. Второй проходит по каждому символу выбранного элемента (для формирования переменной типа double). Итоговая сложность O (n*m), где n – количество элементов в поле operands, m – наибольшая длина строкового литерала, представляющего собой число.

Parse(const std::string& inf):

- Описание параметров и возвращаемых значений: принимает ссылку на константу типа std::string, возвращает тип void.
- Описание реализуемого функционала: метод предназначен для формирования поля lexems_infix. С помощью цикла for метод посимвольно проходит по переданной в метод строке.
    - Если символ представляет собой цифру, то с помощью цикла while формируется числовой литерал, который после заносится в поле lexems_infix.
    - Если это символ “c”, то с помощью функции substr (substr – функция класса std::string. Позволяет извлечь подстроку из заданной строки) проверяется, что совместно с последующими тремя символами не образуется строка “cos(“. Если да, то это функция косинус (в поле lexems_infix заносится строка “cos”, символы o и s игнорируются), иначе это имя переменной.
    - Если это символ латинского алфавита, который в последующем не образует функцию косинус, то с помощью цикла while формируется имя переменной, которая заносится в поле lexems_infix.
    - Если это операция “-“, то проверяется, что это может быть унарный минус (стоит в начале строки или после открывающейся скобки). Унарный минус заменяется на символ ~.

- Если это символ операции (+, - (бинарный), /, *) или скобка (открывающаяся или закрывающаяся), то данный символ заносится в поле lexems_infix.

После цикла for вызываются методы Forming_operands, Values (в данном порядке)

- Оценка сложности: в методе используются цикл for, проходящий по всем символам строки, Forming_operands, Values. Пусть длина строки, передаваемой в метод, равна n, количество лексем в строке m, количество операндов в поле lexems_infix k, число элементов в поле operands равно q, самый большой строковый литерал, представляющий собой число имеет длину p. Сложность цикла for: O(n), метода Forming_operands: O(m*k), метода Values (q*p). Тогда итоговая сложность O(n+m*k+q*p)

find_pos (const std::string& symbol, const std::vector<std::string>&vec):

- Описание параметров и возвращаемых значений: принимает два параметра. Первый: ссылка на константу типа std::string. Второй: ссылка на константный объект типа std::vector, который содержит элементы типа std::string. Возвращает тип int.
- Описание реализуемого функционала: первый параметр – это индекс элемента, который нужно найти. Второй – это вектор строк, в котором нужно найти переданный элемент. Если элемент содержится в векторе, то возвращается его индекс. Иначе возвращается -1. Данный метод используется в других методах (ToPostfix, Calculate).
- Оценка сложности: производится обход переданного вектора с помощью цикла for. Итоговая сложность: в худшем случае O(n), где n – количество элементов в векторе.

void ToPostfix():

- Описание параметров и возвращаемых значений: возвращает тип void.
- Описание реализуемого функционала: данный метод используется для перевода исходного выражения в постфиксную форму. Алгоритм:
    - Создается объект класса TStack (объект содержит элементы типа std::string, а именно операции и открывающиеся круглые скобки)
    - С помощью цикла for метод проходит по полю lexems_infix.
    - Если лексема открывающая скобка, то она добавляется в стек (объект класса TStack)
    - Если лексема закрывающая скобка, то проверяется следующее условие: пока стек не пуст и на вершине стека не открывающая скобка, то извлекаем из стека лексемы и заносим их в постфиксную форму (одновременно формируем поле lexems_postfix, т.е. просто добавляем извлеченную лексему в данное поле). После удаляем с вершины стека открывающую скобку.
    - Если лексема – это операция, то пока стек не пуст проверяем, что приоритет лексемы меньше или равен приоритету операции, находящейся на вершине стека (для нахождения приоритета операции используется метод find_pos). Если условие выполняется, то в постфиксную форму добавляем лексему из стека (также добавляем данную лексему в поле lexems_postfix). Если стек пуст или приоритет лексемы больше приоритета операции, находящейся на вершине стека, то добавляем операцию в стек.
    - Если лексема является операндом, то заносим данную лексему в постфиксную форму и в поле lexems_postfix.
    - После завершения цикла for проверяем, что стек пуст. Если это не так, то с помощью цикла while переносим из стека оставшиеся лексемы (также добавляем их в поле lexems_postfix).

- Оценка сложности: пусть число элементов в поле lexems_infix равно n. Цикл for проходит по всем лексемам. Также в цикле for присутствуют циклы while, который используется для извлечения лексем из стека (стек содержит только операции и открывающие скобки. Каждая операция добавляется и извлекается из стека только один раз. Поэтому влияние циклов while не столь существенно). Также в методе используется метод find_pos. Но данный метод применяется к полю operations, размер которого равен 6 (количество допустимых операций). Итоговая сложность: O(n).

Спецификатор доступа **public**:

TPostfix(const std::string& inf) – конструктор:

- Описание параметров и возвращаемых значений: принимает ссылку на константу типа std::string.
- Описание реализуемого функционала: в списках инициализации поля infix и postfix инициализируются пустой строкой. Поля operations и priority получают соответствующие значения (унарный минус имеет приоритет выше, чем бинарные плюс и минус, но ниже остальных операций. Это необходимо для корректного перевода в постфиксную форму в методе ToPostfix (в силу реализации методов Parse и ToPostfix)). В теле конструктора вызывается метод InfCorrect, который проверяет переданную строку на корректность. Если в строке есть ошибки, то выбрасывается исключение. Если выброс исключения не произошел, то полю infix передается значение параметра. После вызывается метод Parse, который делит инфиксную форму на лексемы (также формирует поля operands, value_operands). Затем вызывается метод ToPostfix, который формирует поля postfix и lexems_postfix.
- Оценка сложности: O(n+m*k+q*p+n+m), где m – число лексем в поле lexems_infix, k – число операндов в поле lexems_infix, q – число операндов в поле operands, p – самый большой строковый литерал, представляющий собой число, n – длина строки (n+m*k+q*p – метод Parse, n – InfCorrect, m – ToPostfix). Т.к. длина строки превосходит остальные параметры, то итоговая сложность O(n).

GetInfix():

- Описание параметров и возвращаемых значений: возвращает значение типа std::string, является константным методом.
- Описание реализуемого функционала: возвращает исходную инфиксную форму выражения.

GetPostfix():

- Описание параметров и возвращаемых значений: возвращает значение типа std::string, является константным методом.
- Описание реализуемого функционала: возвращает постфиксную форму выражения.

Calculate(const std::vector<double>& values):

- Описание параметров и возвращаемых значений: принимает ссылку на константный объект типа std::vector, который хранит значения типа double; возвращает значение типа double.
- Описание реализуемого функционала: метод предназначен для вычисления арифметического выражения (значения неизвестных переменных передаются в метод в виде вектора элементов типа double). Алгоритм:
    - Подсчет количества неизвестных переменных в поле operands (хранит уникальные операнды выражения). Если результат подсчета не совпадает с размером переданного вектора (пользователь однократно вводит значения неизвестных переменных), то

бросается исключение **invalid_argument**.

- Дальше идет второй цикл for. Данный цикл предназначен для переноса значений неизвестных переменных из переданного вектора в поле value_operands. Если i-ый элемент данного поля равен нулю (нечисловые операнды получали значение 0 в методе Values), то в него переносится соответствующий элемент параметра values.
- Создается объект класса TStack, который хранит значения типа double. Далее идет цикл for, который проходит по элементам поля lexems_postfix.
- Если лексема — это бинарная операция, то из стека берутся два значения («правый операнд» - значение, находящееся на вершине стека, «левый операнд» - второе значение после «правого операнда»). В соответствии с операцией производится вычисление (например: лексема – операция “*”, тогда левый операнд умножается на правый), и результат помещается в стек. Если лексема – это операция деления, то сначала проверяется, что правый операнд не ноль. Если он равен нулю, то бросается исключение типа **invalid_argument**.
- Если лексема – унарная операция (унарный минус или функция косинус), то из стека извлекается только одно значение. К данному значение применяется соответствующая операция и результат заносится в стек.
- Если лексема – не операция, то это операнд (переменная или число). Соответствующее значение данного операнда заносится в стек.
- После работы цикла for в стеке хранится одно значение – результат вычисления арифметического выражения. Оно возвращается в качестве результата работы метода.

- Оценка сложности: пусть количество элементов в поле lexems_postfix равно n, количество элементов в поле operands равно m (m<n, причем в поле operands хранятся только уникальные операнды). Цикл for проходится по всем элементам поля lexems_postfix (сложность O(n)). Если лексема (элемент) – операнд, то ищется позиция данного элемента в поле operands (в худшем случае O(m)). Итоговая сложность O(n*m).

Calculate_keyboard_input():

- Описание параметров и возвращаемых значений: возвращает значение типа double.
- Описание реализуемого функционала: метод также используется для вычисления выражения, но ввод значений операндов производится с консоли. Алгоритм:
  - Создается объект класса std::vector, который хранит элементы типа std::string (op – имя объекта). Цикл for проходит по полю operands. Если элемент поля – имя неизвестной переменной, то она добавляется в op.
  - Создается объект класса std::vector, который хранит элементы типа double (val – имя данного объекта).
  - С помощью цикла for метод проходится по элементам вектора op. В данном цикле пользователю предлагается ввести значение определенного операнда. Если пользователь вводит некорректное значение, то выводится сообщение о некорректном вводе. Пользователю предлагается повторно ввести значение данного операнда. Если ввод корректен, то значение операнда добавляется вектор val.
  - После завершения работы цикла for вектор val хранит значения неизвестных переменных. Вызывается метод Calculate, в который передается вектор val. Метод возвращает результат вычисления выражения.
- Оценка сложности: пусть количество элементов в поле lexems_infix равно n, количество элементов в поле operands равно m, а количество неизвестных переменных в поле operands равно p. Метод проходится по полю operands с помощью цикла for (сложность O(m)), после

идет цикл for, который проходится по вектору op (в лучшем случае сложность O(p), т.к. пользователь может ввести некорректное значение и цикл продолжит работу), также в конце вызывается метод Calculate (его сложность O(n*m)). Т.к. основную нагрузку дает именно метод Calculate, то итоговая сложность: O(n*m).

Привязываться только к одному способу получения неизвестных переменных не лучший вариант. Поэтому было введено два метода для вычисления арифметического выражения. При реализации класса TPostfix использовались данные библиотеки: <iostream>, <vector>, <string> (для перевода выражения в строку (использовалась функция to_string). Метод InfCorrect), <cmath> (для применения функции cos. Метод Calculate).

4. Описание тестов Google Test

Тесты для класса TStack

Для проверки корректности методов класса TStack использовались три теста:

- В тесте test_emp проверялись методы: push, pop, top, empty, size.
- В тесте test_error проверялась корректность выброса исключений при попытке использования методов pop, top, если стек пуст.
- В тесте test_swap проверялась корректность работы метода swap.

Тесты для класса TPostfix

В группе тестов **TPostfix_test0** проверялась корректность работы конструктора, в основном корректность работы метода **InfCorrect** (обнаружение ошибки и выброс соответствующего исключения с указанием некорректного символа). Также в одном тесте проверялась корректность работы метода **GetInfix**.

Проверенные ситуации:

- Использование недопустимых символов в выражении (пробелов, знака “!”)
- Ошибки, связанные с использованием скобок (использование пустых скобок, недостаточное количество закрывающих скобок и т.д.)
- Некорректное использование точки (например: “a.1”, или использование точки в самом начале выражения и др.)
- Ошибки, связанные со «слиянием» цифр и букв латинского алфавита (например: “b+q1”)
- Некорректное использование знаков операций (возможность использования унарного минуса (исключение не выбрасывается), использование операции после символа другой операции (“a/-b”) и т.д.)

В группе тестов **TPostfix_test1** проверялась корректность работы метода **ToPostfix** и метода **GetPostfix**.

В тестах test_get_postfix – test_get_postfix_5 проверялся перевод в постфиксную форму простых выражений (выражений, содержащих одну операцию, например: “a+b”, “a*b”, “cos(x)” и т.д.). В тестах test_get_postfix – test_6 – test_get_postfix_13 проверялся перевод в постфиксную форму более сложных выражений (например: “-(a+b)*k”, “(-a+0.8)*(-h)” и т.д.).

В группе тестов **TPostfxi_test2** проверялась корректность работы метода Calculate.

В тестах calc – calc_6 проверялось вычисление простых выражений (данные выражения содержали одну операцию, например: “a+b, где a=1, b=1”, “-h, где h=1” и т.д.)

В тестах calc_7 – calc_17 проверялось вычисление более сложных выражений (например: “ (-a+0.8)*(-h), где a=13.2, h= -78” и т.д.)

В тестах calc_error и calc_error_1 проверялась работа метода при исключительных ситуациях: выброс исключения при неверном количестве значений переменных, передаваемых в метод Calculate, выброс исключения при делении на 0 (например: “a/(b+1), где a=1, b= -1”)

В группе тестов **TPostfix_test3** проверялась корректность работы метода Calculate_keyboard_input. Для этого использовалась функция **std::rdbuf** из библиотеки **<sstream>**. Метод Calculate_keyboard_input использует объекты библиотеки <iostream>: cout (объект типа ostream) для вывода на консоль, cin (объект типа istream) для ввода с консоли.

Объект типа ostream получает определенные значения, переводит данные значения в последовательность символов и передает их через буфер (временное хранилище данных) в определенное место (например: на консоль).

Объект типа istream получает последовательности символов из буфера, который получает их из определенного места (из консоли, файла), затем преобразует данные последовательности в определенные значения и передает значения переменным.

Функция std::rdbuf позволяет изменить буфер объектов ostream, istream, тем самым перенаправить потоки ввода/вывода в другое место. Изменив буфер объекта cout, мы не получим вывод на консоль, а изменив буфер объекта cin программа не запросит ввод данных с консоли (std::cin переданы заранее сформированные данные).
Данная функция была применена для автоматизации тестов.

В самих тестах производилось вычисление арифметических выражений (например: “a+c+(-c), где a=10, b=5, c= -1”).