

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ "НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМ. Н.И. ЛОБАЧЕВСКОГО"

**Отчёт по лабораторной работе №3 по учебной
дисциплине «Алгоритмы и структуры данных»**

Выполнил:

Студент Щербаков Н.А.
группы 3824Б1ФИ2

1. Постановка задачи

Цель данной работы — разработка структуры данных Стек и ее использование для вычисления арифметических выражений. Выражение в качестве операндов может содержать переменные и вещественные числа. Допустимые операции известны: +, -, /, *. Допускается наличие унарного знака "-". Также допускается наличие математической функции exp. Программа должна выполнять предварительную проверку корректности выражения и сообщать пользователю вид ошибки и номера символов строки, в которых были найдены ошибки. При вычислении арифметического выражения требуется ввод с консоли неизвестных переменных..

То есть необходимо разработать интерфейсы и реализации классов TStack и Postfix, а также автоматические тесты Google Test.

2. Описание программной реализации

2.1. Класс TStack

Класс TStack представляет шаблонную структуру данных стек произвольного типа, предназначенную для хранения и обработки последовательности элементов. Класс обеспечивает выполнение базовых операций со стеком, включая добавление элемента в вершину стека, удаление верхнего элемента, получение значения верхнего элемента без его удаления, а также проверку стека на пустоту и определение текущего количества элементов. Кроме того, класс корректно обрабатывает ошибки, возникающие при попытке обращения к элементам пустого стека, с использованием механизма исключений.

Поля класса:

- `std::vector<T> mem` – память для представления элементов стека (вектор произвольного типа T).

Методы класса:

- 1) `TStack()`

Функционал: Конструктор по умолчанию. Стандартный конструктор класса.

- 2) `~TStack()`

Функционал: Деструктор. Стандартный деструктор класса.

- 3) `bool isEmpty() const`

Функционал: Проверяет стек на пустоту. Вызывает соответствующий метод проверки на пустоту вектора `mem` (`mem.empty()`).

Возвращаемое значение: `bool`, `true` - стек пуст, `false` - стек не пуст.

- 4) `size_t size() const`

Функционал: Возвращает размер стека (количество элементов в стеке). Вызывает соответствующий метод вектора `mem` (`mem.size()`).

Возвращаемое значение: `size_t`, размер стека.

- 5) `void push(const T& val)`

Функционал: Добавляет элемент типа T в вершину стека. Вызывает соответствующий метод вектора `mem` (`mem.push_back(val)`).

Параметры: val - элемент типа T.

6) **void pop()**

Функционал: Удаляет элемент типа T с вершины стека. Вызывает соответствующий метод вектора mem (mem.pop_back()). Есть проверка на пустоту стека.

7) **T top() const**

Функционал: Возвращает значение элемента типа T, который на вершине стека. Вызывает соответствующий метод вектора mem (mem.back()). Есть проверка на пустоту стека.

Возвращаемое значение: T, значение элемента на вершине стека.

2.2. Класс Postfix

Класс Postfix предназначен для обработки и вычисления арифметических выражений, заданных в инфиксной форме. Класс обеспечивает разбор входного выражения в инфиксной форме, проверку его корректности, преобразование инфиксной формы в постфиксную форму (обратнуюпольскую запись) и последующее вычисление результата выражения. Реализация поддерживает работу с вещественными числами, переменными, бинарными арифметическими операциями (+, -, /, *), унарным минусом и математической функцией exp. В процессе преобразования в постфиксную форму и вычисления результата выражения используется стек, что позволяет эффективно обрабатывать выражения произвольной сложности. Класс также предоставляет средства для получения списка используемых операндов и корректно обрабатывает ошибки ввода, выдавая диагностические сообщения при обнаружении некорректных выражений.

Поля класса:

- **std::string infix** – строка для хранения исходной инфиксной формы выражения.
- **std::string postfix** – строка для хранения полученной постфиксной формы выражения.
- **std::vector<std::string> lexems_infix** – вектор из строк, который хранит последовательность всех лексем, полученных из инфиксной формы выражения.
- **std::vector<std::string> lexems_postfix** – вектор из строк, который хранит последовательность всех лексем в постфиксной форме выражения.
- **std::vector<std::string> operands** – вектор из строк, который хранит имена всех операндов (переменных), которые присутствуют в инфиксной форме выражения.
- **std::vector<double> values** – вектор из вещественных чисел, который хранит значения всех операндов (переменных), которые присутствуют в инфиксной форме выражения.
- **std::vector<std::string> name_op_priority** – вектор из строк, который хранит имена всех операций, которые допустимы в выражении.
- **std::vector<int> op_priority** – вектор из целых чисел, который хранит приоритеты всех операций, которые допустимы в выражении.

Замечание: поля **std::vector<std::string> operands** и **std::vector<double> values**, а также поля **std::vector<std::string> name_op_priority** и **std::vector<int> op_priority** связаны между собой по индексам, то есть элементы с одинаковыми индексами логически соответствуют друг другу.

Методы класса:

1) **void Parse()**

Функционал: Приватный метод класса Postfix, который реализует разбор инфиксной формы выражения на отдельные лексемы, при этом проверяя корректность ввода самой инфиксной формы. Бросает исключение, если инфиксная форма имеет некорректный вид.

Сложность: $O(n)$ - где $n = \text{size_infx}$, то есть n - длина входной строки, количество символов в инфиксной форме выражения; так как каждый символ обрабатывается не более одного раза (в цикле for).

2) **void ToPostfix()**

Функционал: Приватный метод класса Postfix, который преобразует инфиксную форму выражения в постфиксную форму (обратную польскую запись).

Сложность: $O(n*k)$ - где $n = \text{size_of_lexems}$, то есть n - количество лексем в инфиксной форме выражения, а $k = \text{size_of_operands}$, то есть k - количество операндов (переменных) в инфиксной форме выражения; так как каждая лексема обрабатывается не более одного раза (в цикле for) и при обработке каждой лексемы производится поиск operandов (в цикле for). Поскольку количество operandов (переменных) ограничено количеством лексем, а в типичных случаях число переменных значительно меньше общего числа лексем, сложность метода близка к линейной. Также в данном методе вызывается приватный метод Parse(), который имеет сложность - $O(n)$. Поэтому суммарная сложность не меняется (остаётся $O(n*k)$).

3) **Postfix(*std::string* infix)**

Функционал: Конструктор с параметром. Получает на вход инфиксную форму выражения. Задаёт имена и приоритеты допустимых операций выражения.

Параметры: *infix* - инфиксная форма выражения.

Сложность: $O(n*k)$ – так как вызывается приватный метод ToPostfix(), который имеет сложность $O(n*k)$.

4) **~Postfix()**

Функционал: Деструктор. Стандартный деструктор класса.

5) **std::string GetInfix() const**

Функционал: Возвращает инфиксную форму выражения.

Возвращаемое значение: *std::string*, инфиксная форма выражения.

6) **std::string GetPostfix() const**

Функционал: Возвращает постфиксную форму (обратную польскую запись) выражения.

Возвращаемое значение: *std::string*, постфиксная форма выражения.

7) **std::vector<std::string> GetOperands() const**

Функционал: Возвращает последовательность (набор) имён всех operandов (переменных), которые присутствуют в инфиксной форме выражения.

Возвращаемое значение: *std::vector<std::string>*, возвращает вектор строк, содержащий имена всех operandов (переменных) инфиксной формы выражения.

Сложность: $O(n)$ - где n является количеством операндов (переменных) инфиксной формы выражения; так как производим копирование каждого операнда (переменной) в новый вектор (`vector_op`) в цикле.

8) `double Calculate(const std::vector<double> values)`

Функционал: Вычисляет выражение по постфиксной форме (обратнойпольской записи). Бросает исключение при несовпадении количества значений операндов (переменных) в `values` с количеством имён операндов (переменных) в `operands`.

Параметры: `values` - последовательность (набор) значений всех операндов (переменных), которые присутствуют в инфиксной форме выражения.

Возвращаемое значение: `double`, возвращает результат вычисления выражения.

Сложность: $O(n*k)$ - где $n = \text{size_of_lexems}$, то есть n - количество лексем в постфиксной форме выражения, а $k = \text{size_of_operands}$, то есть k - количество операндов (переменных) в инфиксной форме выражения; так как каждая лексема обрабатывается не более одного раза (в цикле `for`) и при обработке каждой лексемы производится поиск операндов (в цикле `for`). Поскольку количество операндов (переменных) ограничено количеством лексем, а в типичных случаях число переменных значительно меньше общего числа лексем, сложность метода близка к линейной.

9) `double Calculate()`

Функционал: Перегрузка метода `Calculate(const std::vector<double> values)` для реализации ввода операндов (переменных) с консоли.

Возвращаемое значение: `double`, возвращает результат вычисления выражения.

Сложность: $O(n*k)$ - так как вызывается перегруженный метод `Calculate(const std::vector<double> values)`, который имеет данную сложность. Также в данном методе присутствует цикл `for` от 0 до `operands.size()`, что даёт сложность $O(k)$. Поэтому суммарная сложность не меняется (остаётся $O(n*k)$).

3.Краткие комментарии к тестам

Для проверки правильности работы классов `TStack` и `Postfix` были созданы тесты с использованием фреймворка `Google Test`, охватывающие основные операции, граничные ситуации и обработку ошибок.

3.1. Тесты класса `TStack`

- **CreateEmptyStackIsCorrect** – Проверяет, что стек создаётся пустым: `isEmpty()` возвращает `true`, а `size()` возвращает 0 для двух независимых объектов класса `TStack`.
- **PushAndTopAndPopIsCorrectNoThrow** – Проверяет корректность методов `push()`, `top()` и `pop()`: после `push()` размер стека и элемент вершины стека обновляются, `top()` не бросает исключение и возвращает значение верхнего элемента стека, `pop()` корректно уменьшает размер стека, удаляя элемент с его вершины. Все методы не должны бросать исключений, когда стек не пуст.
- **PopAndTopAnyThrow** – Проверяет, что методы `pop()` и `top()` на пустом стеке бросают исключение `std::out_of_range`.

3.2. Тесты класса `Postfix`

- **SimpleExpressionsNoThrow** – Проверяет корректное преобразование простых инфиксных форм выражений в постфиксные формы (операции +, -, *, /; функция exp; унарный минус). Сравниваются полученная при преобразовании постфиксная форма и ожидаемая постфиксная форма, с помощью метода GetPostfix(). Также происходит проверка метода GetInfix(). Без выброса исключений.
- **SimpleExpressionsWithUnaryMinusNoThrow** – Проверяет корректное преобразование простых инфиксных форм выражений в постфиксные формы (операции +, -, *, /; функция exp; унарный минус) с унарным минусом к этим выражениям. Сравниваются полученная при преобразовании постфиксная форма и ожидаемая постфиксная форма, где унарный минус заменяется знаком «~». Без выброса исключений.
- **ComplexExpressionsWithUnaryMinusWithNumbersNoThrow** – Проверяет корректное преобразование более сложных инфиксных форм выражений в постфиксные формы, содержащие вещественные числа, функцию exp, унарный минус и различные комбинации операций. Сравниваются полученная при преобразовании постфиксная форма и ожидаемая постфиксная форма. Без выброса исключений.
- **ExpressionswithMultiLetterVariablesNoThrow** – Проверяет поддержку многобуквенных имён операндов (переменных) при разборе на лексемы и преобразовании инфиксной формы в постфиксную форму. Сравниваются полученная при преобразовании постфиксная форма и ожидаемая постфиксная форма. Без выброса исключений.
- **IncorrectExpressionsAnyThrow** – Проверяет различные некорректные сочетания в инфиксной форме, которые должны бросать исключение std::invalid_argument (некорректная расстановка или количество скобок, некорректный ввод вещественного числа, некорректный ввод переменных, некорректный ввод операций, ввод неизвестного символа и т.п.).
- **ObtainingSequenceOfOperands** – Проверяет, что метод GetOperands() возвращает корректный, упорядоченный, полный список (вектор из строк) обнаруженных в инфиксной форме выражения операндов (переменных). Приводится конкретный пример выражения: $a+b+c$.
- **SimpleExpressionsCalculateNoThrow** – Проверяет корректное вычисление простых выражений: в метод Calculate(const std::vector<double> values_) передаётся вектор значений переменных и проверяется, что метод возвращает ожидаемые числовые результаты вычисления выражения. Без выброса исключений.
- **SimpleExpressionsWithUnaryMinusCalculateNoThrow** – Проверяет корректное вычисление простых выражений с унарным минусом к этим выражениям: в метод Calculate(const std::vector<double> values_) передаётся вектор значений переменных и проверяется, что метод возвращает ожидаемые числовые результаты вычисления выражения. Без выброса исключений.
- **ComplexExpressionsWithUnaryMinusWithNumbersWithMultiLetterVariablesCalculateNoThrow** – Проверяет корректное вычисление более сложных выражений, содержащих унарный минус, вещественные числа, многобуквенные переменные, различные комбинации операций, функцию exp: в метод Calculate(const std::vector<double> values_) передаётся вектор значений переменных и проверяется, что метод возвращает ожидаемые числовые результаты вычисления выражения. Без выброса исключений.
- **ExpressionsCalculateAnyThrow** – Проверяет, что метод Calculate(const std::vector<double> values_) бросает исключение std::invalid_argument при передаче в него вектора значений переменных, размер которого отличен от числа переменных в выражении.
- **ExpressionsCalculateFromTheConsole** – Это визуальный тест. Проверяет работоспособность ручного ввода значений переменных для вычисления выражения. Есть проверка на ввод символа/строки, отличной от вещественного числа.