Федеральное государственное автономное образовательное учреждение высшего образования "Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского"

Отчёт №3

по учебной дисциплине

«Алгоритмы и структуры данных»

Студента Ципина Д.Д.

группы 3824Б1ФИ2

Нижний Новгород – 2025 г.

Постановка задачи

Разработать структуру данных стек и использовать её для вычисления арифметических выражений. Выражение может содержать переменные и вещественные числа. Допустимые операции: +, -, /, *. Допускается наличие унарного знака «-». Также допускается наличие математической функции ln(x). 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.

Описание класса TStack

Класс TStack представляет собой шаблонную структуру данных, реализующую стек фиксированного размера, работающая по принципу LIFO. Используется для хранения операторов и операндов при разборе выражений.

Поля класса

- T* pMem — динамический массив элементов.
- int size — максимальная ёмкость стека.
- int top — индекс следующей свободной позиции (количество элементов в стеке).

Описание ключевых методов TStack

TStack(int _size = 100)

- **Параметры**: _size — максимальный размер стека.
- **Функционал**: выделяет динамическую память под массив из _size элементов, инициализирует стек пустым (top = 0).

TStack(const TStack<T>& s)

- **Параметры**: s — другой стек.

- **Функционал**: создаёт новый стек той же ёмкости и копирует элементы до текущей вершины.
- **Сложность**: O(n), где n — количество элементов в стеке. Требуется скопировать n элементов массива.

~TStack()

- **Функционал**: освобождает динамически выделенную память.

TStack<T>& operator=(const TStack<T>& s)

- **Параметры**: s — другой стек.
- **Функционал**: выполняет присваивание с использованием парадигмы copy-and-swap.
- **Сложность**: O(n), так как создаётся временная копия и копируются n элементов.

bool IsEmpty() const

- **Функционал**: проверяет, пуст ли стек (top == 0).

bool IsFull() const

- **Функционал**: проверяет, заполнен ли стек (top == size).

void Push(const T& val)

- **Параметры**: val — добавляемый элемент.
- **Функционал**: помещает элемент на вершину стека; при переполнении выбрасывает исключение.

T Pop()

- **Функционал**: удаляет и возвращает верхний элемент, при пустом стеке выбрасывает исключение.

T Top() const

- Функционал: возвращает верхний элемент без удаления, при пустом стеке выбрасывает исключение.

Описание класса TPostFix

TPostfix — класс для работы с арифметическими выражениями. Выполняет три основные задачи: проверка корректности инфиксного выражения, преобразование инфиксной формы в постфиксную, вычисление постфиксного выражения. Поддерживает вещественные числа, переменные (_x, var1, i), операции +, -, *, /, унарный минус, функцию ln(x).

Поля класса TPostFix

- **std::string infix** — исходное выражение.
- **std::string postfix** — постфиксная форма.

Описание ключевых методов TPostFix

TPostfix(const std::string& expr = "")

- **Параметры:** expr — инфиксное выражение.
- **Функционал:** сохраняет выражение во внутреннем поле infix.

const std::string& GetInfix() const

- **Функционал:** возвращает исходное инфиксное выражение.

const std::string& GetPostfix() const

- **Функционал:** возвращает постфиксную форму выражения.

bool IsLetter(char c) const

- **Параметры:** c — символ.
- **Функционал:** проверяет, является ли символ буквой латинского алфавита.

bool IsDigit(char c) const

- **Параметры:** c — символ.
- **Функционал:** проверяет, является ли символ цифрой.

bool IsOperator(char c) const

- **Параметры:** c — символ.
- **Функционал:** проверяет, является ли символ одним из операторов + - * /.

int Priority(const std::string& op) const

- **Параметры:** op — строка-оператор.
- **Функционал:** возвращает приоритет оператора (+ - → 1, * / → 2, ~ → 3, ln → 4).

void AddToken(const std::string& tok)

- **Параметры:** tok — токен (число, переменная, оператор).
- **Функционал:** добавляет токен в строку postfix с пробелом-разделителем.

void CheckExpression() const

- **Функционал:** выполняет полную синтаксическую проверку выражения:
    - корректность скобок,
    - отсутствие двух операторов подряд,
    - корректность унарного минуса,
    - корректность чисел (не больше одной точки, не начинается с точки),
    - запрет конструкций вида 1x,
    - корректность имён переменных (_x, var1),
    - проверка функции ln,
    - выражение не может заканчиваться оператором.

- **Сложность: O(n)**, где n — длина выражения. Требуется пройти по каждому символу строки.

void ToPostfixInternal()

- **Функционал:** преобразует инфиксное выражение в постфиксное:
    - использует стек операторов,
    - учитывает приоритеты,
    - обрабатывает унарный минус,
    - обрабатывает функцию ln,
    - формирует итоговую строку postfix.
- **Сложность: O(n)**, так как каждый символ обрабатывается один раз.

void ToPostfix()

- **Функционал:** публичный метод, вызывающий CheckExpression() и затем ToPostfixInternal().
- **Сложность: O(n)** — определяется работой внутренних методов.

double Calculate()

- **Функционал:** вычисляет значение постфиксного выражения:
    - использует стек чисел,
    - выполняет операции + - * /,
    - обрабатывает унарный минус,
    - вычисляет ln(x),
    - запрашивает значения переменных у пользователя.
- **Сложность: O(n)**, где n — количество токенов в постфиксной форме.

Краткие комментарии к тестам

Для проверки корректности работы классов TStack и TPostfix были разработаны тесты с использованием фреймворка **Google Test**. Тесты охватывают базовые операции, граничные случаи и поведение при ошибках.

Тесты класса TStack

- **can_create_stack_with_positive_size** – проверяет корректность создания стека с положительным размером.
- **cannot_create_stack_with_negative_size** – проверяет выброс исключения при создании стека с отрицательным размером.
- **new_stack_is_empty** – убеждается, что новый стек пуст.
- **push_makes_stack_not_empty** – проверяет, что после Push стек перестаёт быть пустым.
- **pop_from_empty_throws** – проверяет выброс исключения при попытке Pop из пустого стека.
- **top_from_empty_throws** – проверяет выброс исключения при вызове Top на пустом стеке.
- **push_and_pop_work_correctly** – проверяет корректность работы Push и Pop, включая порядок LIFO.
- **top_works_correctly** – проверяет, что Top возвращает последний добавленный элемент без удаления.
- **stack_overflow_throws** – проверяет выброс исключения при попытке Push в заполненный стек.
- **stack_underflow_throws** – проверяет выброс исключения при Pop из пустого стека.
- **can_use_stack_with_different_types** – проверяет работу стека с разными типами данных (int, double, string).
- **copied_stack_is_equal** – проверяет корректность конструктора копирования: копия содержит те же элементы.
- **copied_stack_has_its_own_memory** – убеждается, что копия имеет независимую память.

- **assigned_stack_is_equal** – проверяет корректность оператора присваивания.
- **assigned_stack_has_its_own_memory** – убеждается, что присваивание создаёт независимую копию.
- **isfull_works_correctly** – проверяет корректность работы метода IsFull.

Тесты синтаксического анализа CheckExpression

- **valid_simple_expression** – проверяет, что корректное выражение проходит проверку.
- **invalid_double_operator** – проверяет обнаружение двух операторов подряд.
- **invalid_brackets_order** – проверяет ошибку при неправильном порядке скобок.
- **missing_closing_bracket** – проверяет обнаружение незакрытой скобки.
- **empty_brackets** – проверяет ошибку при пустых скобках.
- **invalid_number_two_points** – проверяет обнаружение числа с двумя точками.
- **number_can_end_with_point** – проверяет, что число может оканчиваться точкой.
- **number_cannot_start_with_point** – проверяет запрет числа, начинающегося с точки.
- **variable_can_start_with_underscore** – проверяет поддержку переменных, начинающихся с символа _.
- **function_ln_does_not_require_brackets** – проверяет, что ln может использоваться без скобок.
- **unary_minus_is_allowed** – проверяет корректность унарного минуса перед числом.
- **unary_minus_before_bracket** – проверяет корректность унарного минуса перед скобками.

Тесты преобразования в постфиксную форму (ToPostfix)

- **simple_expression** – проверяет корректность преобразования выражения 1+2*3.
- **brackets_change_priority** – проверяет, что скобки корректно меняют приоритет операций.
- **unary_minus** – проверяет преобразование унарного минуса в оператор ~.
- **unary_minus_before_bracket** – проверяет преобразование -(...) в постфиксную форму. • **ln_function** – проверяет корректную обработку ln 5.
- **ln_function_with_brackets** – проверяет обработку ln(5).
- **variable_expression** – проверяет корректное преобразование выражений с переменными.

Тесты вычисления выражений (Calculate)

- **simple_addition** – проверяет вычисление простого выражения 1+2.
- **priority_multiplication** – проверяет соблюдение приоритета операций.
- **brackets** – проверяет корректность вычисления выражения со скобками.
- **unary_minus** – проверяет вычисление выражения с унарным минусом.
- **unary_minus_before_bracket** – проверяет вычисление -(1+2).
- **ln_function** – проверяет вычисление функции ln(1).
- **division_by_zero_throws** – проверяет выброс исключения при делении на ноль.
- **complex_expression** – проверяет вычисление сложного выражения с функцией ln и унарным минусом.