Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное   
учреждение высшего образования

Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

Направление подготовки: «Прикладная математика и информатика»

**Лабораторная работа №3**

**«Вычисление арифметических выражений»**

**Выполнил**:

студент группы 3822Б1ПМ2

Махнёв Р.Д.

**Научный руководитель**:   
асс. кафедры МОСТ

Воденеева А.А.

Нижний Новгород

2022

**Оглавление**

[Введение 4](#_Toc152521518)

[1. Постановка задачи 5](#_Toc152521519)

[2. Основная часть 6](#_Toc152521520)

[2.1. Стек 6](#_Toc152521521)

[2.2. Постфиксная запись 6](#_Toc152521522)

[3. Програмная реализация 7](#_Toc152521523)

[3.1. Стек 7](#_Toc152521524)

[3.2. Лексема 10](#_Toc152521525)

[3.3. Операнд 11](#_Toc152521526)

[3.4. Закрывающая скобка 13](#_Toc152521527)

[3.5. Открывающая скобка 15](#_Toc152521528)

[3.6. Деление 16](#_Toc152521529)

[3.7. Вычитание 17](#_Toc152521530)

[3.8. Умножение 18](#_Toc152521531)

[3.9. Сложение 19](#_Toc152521532)

[3.10. Унарный минус 20](#_Toc152521533)

[3.11. Постфикс 21](#_Toc152521534)

[3.12. Известный операнд 24](#_Toc152521535)

[3.13. Неизвестный операнд 27](#_Toc152521536)

[4. Тестирование 28](#_Toc152521537)

[4.1. Стек 28](#_Toc152521538)

[4.2. Постфикс 30](#_Toc152521539)

[4.3. Результаты тестирования 33](#_Toc152521540)

[5. Выводы 35](#_Toc152521541)

[Список литературы 36](#_Toc152521542)

[Приложение А. Фрагменты кода класса стек 37](#_Toc152521543)

[Приложение Б. Фрагменты кода класса постфикс 42](#_Toc152521544)

[Приложение В. Фрагменты кода класса лексема 50](#_Toc152521545)

[Приложение Г. Фрагменты кода класса оператор 51](#_Toc152521546)

[Приложение Д. Фрагменты кода класса операнд 52](#_Toc152521547)

[Приложение Е. Фрагменты кода класса известный оператор 53](#_Toc152521548)

[Приложение Ё. Фрагменты кода класса неизвестный оператор 56](#_Toc152521549)

[Приложение Ж. Фрагменты кода класса открывающая скобка 58](#_Toc152521550)

[Приложение З. Фрагменты кода класса закрывающая скобка 60](#_Toc152521551)

[Приложение И. Фрагменты кода класса сложение 62](#_Toc152521552)

[Приложение Й. Фрагменты кода класса вычитание 64](#_Toc152521553)

[Приложение К. Фрагменты кода класса унарный минус 66](#_Toc152521554)

[Приложение Л. Фрагменты кода класса умножение 68](#_Toc152521555)

[Приложение М. Фрагменты кода класса деление 70](#_Toc152521556)

# ****Введение****

Вычислительные системы используют различные типы структур данных, чтобы обеспечить эффективное выполнение операций и хранение информации. Стек - это тип структуры данных, которая позволяет выполнять операции добавления и удаления элементов в определенном порядке. В данной лабораторной работе мы разработаем структуру данных Стек, а также используем ее для расчета арифметических выражений в обратной польской нотации.

# Постановка задачи

Основные цели и задачи работы заключаются в следующем:

1. Разработка структуры данных TStack и его методов.
2. Реализация интерфейса и методов для работы с обратной польской записью (постфиксная форма).
3. Разработка и реализация тестов на основе Google Test для классов TStack и методов постфиксной записи.
4. Публикация исходных кодов в открытом репозитории GitHub.

# Основная часть

## Стек

Стек - это структура данных, которая хранит элементы в порядке “последним пришел - первым ушел” (LIFO, Last In First Out) [1]. Это значит, что последний добавленный элемент будет первым извлечен из стека. Стек часто используется для моделирования таких вещей, как очередь задач, которую нужно выполнить, или для хранения состояний во время вычисления. Основные операции со стеком - это добавление элементов (push) и извлечение элементов (pop). Также стек может быть пуст (empty), и у него есть вершина (top), которая указывает на последний добавленный элемент.

## Постфиксная запись

Постфиксная запись - это запись, в которой операнды и операции записываются в одну строку, причем операции идут после своих операндов [2]. Например, выражение “2 3 + \*” означает “взять два, взять еще три, сложить их, потом взять результат и умножить его на себя”.

Данная запись удобна для стековых вычислений, так как позволяет легко отделить уже обработанные операнды от тех, которые еще нужно обработать.

Для работы с постфиксной записью можно использовать структуру данных “стек”. Когда считается очередной символ из строки, если это операнд, то мы просто добавляем его в стек. Если это операция, то мы извлекаем из стека один или два верхних операнда, применяем к ним или ему операцию, результат кладем обратно в стек. После того, как мы обработали всю строку, в стеке останется один результат.

# Програмная реализация

## Стек

Класс TStack является шаблонным классом, реализующим стек с использованием динамического массива.

Поля:

T\* pMem – указатель на массив, в котором содержатся элементы стека.

size\_t size – размер массива, в котором содержатся элементы стека.

size\_t end – индекс последнего элемента в массиве.

Конструкторы

TStack(size\_t \_size)

*Входные параметры:* размер стека.

*Выходные параметры:* нет.

*Действие:* выделяет память под стек, инициализирует поля.

TStack(const TStack<T>& stack)

*Входные параметры:* ссылка на TStack<T>& stack

*Выходные параметры:* нет.

*Действие:* копирует стек.

TStack(TStack<T>&& stack) noexcept

*Входные параметры:* размер стека.

*Выходные параметры:* нет.

*Действие:* выделяет память под стек, инициализирует поля.

Деструктор:

~TStack()

деструктор класса, освобождает память, занятую массивом.

Методы класса:

void Push(const T& elem)

*Входные параметры:* ссылка на элемент.

*Выходные параметры:* нет.

*Действие:* добавляет элемент в стек. Если стек полон, увеличивает размер массива в 1.5 раза и копирует все элементы в новый массив, затем добавляет новый элемент в конец стека.

T& Top()

*Входные параметры:* нет.

*Выходные параметры:* ссылка на верхний элемент стека.

*Действие:* возвращает ссылку на верхний элемент стека. Если стек пуст, генерирует исключение std::out\_of\_range.

size\_t Size() const

*Входные параметры:* нет.

*Выходные параметры:* целое неотрицательное число.

*Действие:* возвращает размер массива.

bool IsEmpty() const noexcept

*Входные параметры:* нет

*Выходные параметры:* булево.

*Действие:* проверяет, пуст ли стек. Возвращает true, если стек пуст, и false в противном случае.

const T& Top() const:

*Входные параметры:* нет

*Выходные параметры:* ссылка на верхний элемент стека.

*Действие:* возвращает ссылку на верхний элемент стека. Если стек пуст, генерирует исключение std::out\_of\_range.

void Pop()

*Входные параметры:* нет

*Выходные параметры:* ссылка на верхний элемент стека.

*Действие:* удаляет элемент из стека. Если стек пуст, генерирует исключение std::out\_of\_range.

Операторы:

TStack& operator= (TStack<T>&& stack) noexcept

*Входные параметры:* правая ссылка на TStack<T>

*Выходные параметры:* указатель на себя.

*Действие:* перемещает данные из другого стека в текущий стек и обнуляет указатель на массив другого стека.

bool operator== (const TStack<T>& stack) const

*Входные параметры:* ссылка на TStack<T>

*Выходные параметры:* булево.

*Действие:* сравнивает размеры стеков и все элементы массивов. Возвращает true, если стеки равны, и false в противном случае.

bool operator!= (const TStack<T>& stack) const

*Входные параметры:* ссылка на TStack<T>

*Выходные параметры:* булево.

*Действие:* сравнивает размеры стеков и все элементы массивов.

Возвращает true, если стеки не равны, и false в противном случае.

Класс TStack может использоваться для создания стеков различных типов данных, так как является шаблонным классом.

## Лексема

Методы класса Lexem описывают абстрактную лексему и имеют следующие свойства и функциональность:

void action(TStack<Lexem\*>& operands) const

*Входные параметры:* ссылка на стек операндов (operands) типа TStack<Lexem\*>

*Выходные параметры:* нет

*Действие:* этот метод выполняет операцию или действие, связанное с данной лексемой, используя операнды, хранящиеся в переданном стеке operands. Он изменяет содержимое стека переданных операндов в соответствии с лексемой.

std::string GetYourself() const

*Входные параметры:* нет

*Выходные параметры:* строка

*Действие:* этот метод возвращает строковое представление данной лексемы. Возвращаемая строка содержит информацию о конкретной лексеме.

size\_t GetPriority() const

*Входные параметры:* нет

*Выходные параметры:* целое число

*Действие:* этот метод возвращает приоритет данной лексемы. Приоритет используется для определения порядка выполнения операций или действий. Чем выше приоритет, тем раньше выполняется операция или действие.

## Операнд

Класс Operand представляет собой операнд.

Методы класса Operand:

void action(TStack<Lexem\*>& operands) const override

*Входные параметры:* ссылка на стек операндов (operands) типа

TStack<Lexem\*>.

*Выходные параметры:* нет.

*Действие:* добавляет себя в стек операндов.

std::string GetYourself() const override

*Входные параметры:* нет.

*Выходные параметры:* строка.

*Действие:* возвращает своё значение в строковом представлении.

size\_t GetPriority() const

*Входные параметры:* нет.

*Выходные параметры:* целое число.

*Действие:* возвращает приоритет данной лексемы. Приоритет используется для определения порядка выполнения операций или действий. Чем выше приоритет, тем раньше выполняется операция или действие.

Operand\* Plus(const Lexem\* operand) const

*Входные параметры:* указатель на Lexem.

*Выходные параметры:* указатель на Operand.

*Действие:* выполняет операцию сложения с другим операндом.

Operand\* Minus(const Lexem\* operand) const

*Входные параметры:* указатель на Lexem.

*Выходные параметры:* указатель на Operand.

*Действие:* выполняет операцию вычитания с другим операндом.

Operand\* UnaryMinus() const

*Входные параметры:* нет.

*Выходные параметры:* указатель на Operand.

*Действие:* выполняет операцию вычитания.

Operand\* Multiple(const Lexem\* operand) const

*Входные параметры:* указатель на Lexem.

*Выходные параметры:* указатель на Operand.

*Действие:* выполняет операцию умножения с другим операндом.

Operand\* Division(const Lexem\* operand) const

*Входные параметры:* указатель на Lexem.

*Выходные параметры:* указатель на Operand.

*Действие:* выполняет операцию деления с другим операндом.

Operand\* Copy() const

*Входные параметры:* нет.

*Выходные параметры:* ссылка на Operand.

*Действие:* копирование себя.

## Закрывающая скобка

Класс OperatorCloseBracket используется для обработки оператора закрывающей скобки в математических выражениях. Если в выражении встречается закрывающая скобка без открывающей скобки, то будет выброшено исключение с сообщением "miss open bracket 2".

OperatorCloseBracket наследуется от класса Operator.

Методы:

void action(TStack<Lexem\*>& operands) const

*Входные параметры:* ссылка на стек операндов (operands) типа TStack<Lexem\*>

*Выходные параметры:* нет

*Действие:* выбрасывает исключение, что потеряна открывающая скобка.

std::string GetYourself() const override:

*Входные параметры:* нет.

*Выходные параметры:* строка.

*Действие:* возвращает “)”.

Operator\* IsIt(std::string& current)

*Входные параметры:* ссылка на строку.

*Выходные параметры:* Oprator\*.

*Действие:* Если последний символ строки является закрывающей скобкой ")", то метод обрезает этот символ, создает объект класса OperatorCloseBracket и возвращает его. Если символ не является закрывающей скобкой, метод возвращает nullptr.

size\_t GetPriority() const

*Входные параметры:* нет.

*Выходные параметры:* целое неотрицательное число.

*Действие:* возвращает приоритет оператора. Для закрывающей скобки метод возвращает 0.

## Открывающая скобка

OperatorOpenBracket наследуется от класса Operator и представляет открывающую скобку "(" в математическом выражении.

Методы:

void action(TStack<Lexem\*>& operands) const

*Входные параметры:* ссылка на стек операндов (operands) типа TStack<Lexem\*>

*Выходные параметры:* нет

*Действие:* выбрасывает исключение, что потеряна закрывающая скобка.

std::string GetYourself() const override:

*Входные параметры:* нет.

*Выходные параметры:* строка.

*Действие:* возвращает “(”.

Operator\* IsIt(std::string& current)

*Входные параметры:* ссылка на строку.

*Выходные параметры:* Oprator\*.

*Действие:* если последний символ строки является открывающей скобкой "(", то метод обрезает этот символ, создает объект класса OperatorOpenBracket и возвращает его. Если символ не является закрывающей скобкой, метод возвращает nullptr.

size\_t GetPriority() const

*Входные параметры:* нет.

*Выходные параметры:* целое неотрицательное число.

*Действие:* возвращает приоритет оператора. Для открывающей скобки метод возвращает 0.

## Деление

OperatorDivision класс, представляющий оператор деления, который наследуется от класса Operator.

Методы:

void action(TStack<Lexem\*>& operands) const

*Входные параметры:* ссылка на стек операндов (operands) типа TStack<Lexem\*>.

*Выходные параметры:* нет.

*Действие:* достаёт 2 элемента из стека и вызывает Division у левого операнда, и предаёт правый. Результат возвращает в стек.

std::string GetYourself() const

Входные параметры: нет.

Выходные параметры: строка.

Действие: возвращает “/”.

Operator\* IsIt(std::string& current)

Входные параметры: ссылка на строку.

Выходные параметры: Oprator\*.

Действие: если последний символ строки является "/", то метод обрезает этот символ, создает объект класса OperatorDivision и возвращает его. Если символ не является “/”, метод возвращает nullptr.

size\_t GetPriority() const

*Входные параметры:* нет.

*Выходные параметры:* целое неотрицательное число.

*Действие:* возвращает приоритет оператора. Для деления метод возвращает 2.

## Вычитание

OperatorMinus класс, представляющий оператор вычитания, который наследует класс Operator.

Методы:

void action(TStack<Lexem\*>& operands) const

*Входные параметры:* ссылка на стек операндов (operands) типа TStack<Lexem\*>.

*Выходные параметры:* нет.

*Действие:* достаёт 2 элемента из стека и вызывает Minus у левого операнда, и предаёт правый. Результат возвращает в стек.

std::string GetYourself() const

Входные параметры: нет.

Выходные параметры: строка.

Действие: возвращает “-”.

Operator\* IsIt(std::string& current)

Входные параметры: ссылка на строку.

Выходные параметры: Oprator\*.

Действие: если последний символ строки является "-", то метод обрезает этот символ, создает объект класса OperatorMinus и возвращает его. Если символ не является “-”, метод возвращает nullptr.

size\_t GetPriority() const

*Входные параметры:* нет.

*Выходные параметры:* целое неотрицательное число.

*Действие:* возвращает приоритет оператора. Для вычитания метод возвращает 1.

## Умножение

OperatorMultiple класс, представляющий оператор умножения, который наследуется от класса Operator.

Методы:

void action(TStack<Lexem\*>& operands) const

*Входные параметры:* ссылка на стек операндов (operands) типа TStack<Lexem\*>.

*Выходные параметры:* нет.

*Действие:* достаёт 2 элемента из стека и вызывает Multiple у левого операнда, и предаёт правый. Результат возвращает в стек.

std::string GetYourself() const

Входные параметры: нет.

Выходные параметры: строка.

Действие: возвращает “\*”.

Operator\* IsIt(std::string& current)

Входные параметры: ссылка на строку.

Выходные параметры: Oprator\*.

Действие: если последний символ строки является "\*", то метод обрезает этот символ, создает объект класса OperatorMultiple и возвращает его. Если символ не является “\*”, метод возвращает nullptr.

size\_t GetPriority() const

*Входные параметры:* нет.

*Выходные параметры:* целое неотрицательное число.

*Действие:* возвращает приоритет оператора. Для умножения метод возвращает 2.

## Сложение

OperatorPlus класс, представляющий оператор сложения, наследуемый от класса Operator.

Методы:

void action(TStack<Lexem\*>& operands) const

*Входные параметры:* ссылка на стек операндов (operands) типа TStack<Lexem\*>.

*Выходные параметры:* нет.

*Действие:* достаёт 2 элемента из стека и вызывает Plus у левого операнда, и предаёт правый. Результат возвращает в стек.

std::string GetYourself() const

Входные параметры: нет.

Выходные параметры: строка.

Действие: возвращает “+”.

Operator\* IsIt(std::string& current)

Входные параметры: ссылка на строку.

Выходные параметры: Oprator\*.

Действие: если последний символ строки является "+", то метод обрезает этот символ, создает объект класса OperatorPlus и возвращает его. Если символ не является “+”, метод возвращает nullptr.

size\_t GetPriority() const

*Входные параметры:* нет.

*Выходные параметры:* целое неотрицательное число.

*Действие:* возвращает приоритет оператора. Для сложения метод возвращает 1.

## Унарный минус

OperatorUnaryMinus класс, представляющий оператор сложения, который наследуется от класса Operator.

Методы:

void action(TStack<Lexem\*>& operands) const

*Входные параметры:* ссылка на стек операндов (operands) типа TStack<Lexem\*>.

*Выходные параметры:* нет.

*Действие:* достаёт элемент из стека и вызывает UnaryMinus. Результат возвращает в стек.

std::string GetYourself() const

Входные параметры: нет.

Выходные параметры: строка.

Действие: возвращает “-u”.

Operator\* IsIt(std::string& current)

Входные параметры: ссылка на строку.

Выходные параметры: Oprator\*.

Действие: если последний символ строки является "-" и он перед ( или находится в начале, то метод обрезает этот символ, создает объект класса UnaryMinus и возвращает его. Если символ не является “-”, метод возвращает nullptr.

size\_t GetPriority() const

*Входные параметры:* нет.

*Выходные параметры:* целое неотрицательное число.

*Действие:* возвращает приоритет оператора. Для сложения метод возвращает 100.

## Постфикс

шаблонный класс TPostfix представляет собой класс для преобразования инфиксного выражения в постфиксную форму и вычисления результата. Тип T является параметром шаблона и представляет тип данных, с которым работает класс.

Поля:

std::string infix - строка, содержащая инфиксное выражение.

std::vector<Lexem\*> postfix - вектор указателей на объекты класса

Lexem, представляющих постфиксную форму выражения.

Конструкторы:

TPostfix()

*Входные параметры:* нет.

*Выходные параметры:* нет.

*Действие:* устанавливает значение infix равным "3+2\*4".

TPostfix(std::string \_infix)

*Входные параметры:* строка.

*Выходные параметры:* нет.

*Действие:* устанавливает значение infix равным \_infix

Деструктор:

~TPostfix()

*Входные параметры:* строка.

*Выходные параметры:* нет.

*Действие:* освобождает память, занятую объектами класса Lexem в векторе

postfix.

Методы:

std::string GetInfix()

*Входные параметры:* нет.

*Выходные параметры:* строка.

*Действие:* возвращает значение поля infix в виде строки.

std::string GetPostfix()

*Входные параметры:* нет.

*Выходные параметры:* строка.

*Действие:* возвращает значение поля postfix в виде строки, сформированной путем объединения строковых представлений объектов класса Lexem.

void ToPostfix()

*Входные параметры:* нет.

*Выходные параметры:* нет.

*Действие:* преобразует инфиксное выражение в постфиксную форму и сохраняет результат в поле postfix.

T Calculate()

*Входные параметры:* нет.

*Выходные параметры:* результат вычисления.

*Действие:* вычисляет результат выражения в постфиксной форме, используя значения переменных, и возвращает результат типа T.

void AddOperand()

*Входные параметры:* нет.

*Выходные параметры:* нет.

*Действие:* добавляет операнд в постфиксную форму.

void CheckOpenBracket()

*Входные параметры:* нет.

*Выходные параметры:* нет.

*Действие:* проверяет наличие открывающей скобки в стеке операторов, если есть вызывает исключение.

void InsertCloseBracket()

*Входные параметры:* нет.

*Выходные параметры:* нет.

*Действие:* вставляет закрывающую скобку в постфиксную форму.

void InsertOperator()

*Входные параметры:* нет.

*Выходные параметры:* нет.

*Действие:* вставляет оператор в постфиксную форму.

## Известный операнд

KnownOperand шаблонный класс с значением T, представляющий операнд, значение которого известно.

Поле:

T value – значение операнда.

Конструкторы:

KnownOperand(const T& \_value)

*Входные параметры:* ссылка типа T.

*Выходные параметры:* нет.

*Действие:* устанавливает значение value равным \_value

KnownOperand(std::string& \_value)

*Входные параметры:* ссылка на строку.

*Выходные параметры:* нет.

*Действие:* устанавливает значение value равным значению записанному в \_value.

KnownOperand(const KnownOperand<T>& op)

*Входные параметры:* ссылка на KnownOperand<T>.

*Выходные параметры:* нет.

*Действие:* присваивает значение value равным op.value.

Методы:

Operand\* Copy() const

*Входные параметры:* нет.

*Выходные параметры:* ссылка на Operand.

*Действие:* копирование себя.

void action(TStack<Lexem\*>& operands) const override

*Входные параметры:* ссылка на стек операндов (operands) типа

TStack<Lexem\*>.

*Выходные параметры:* нет.

*Действие:* добавляет себя в стек операндов.

std::string GetYourself() const override

*Входные параметры:* нет.

*Выходные параметры:* строка.

*Действие:* возвращает своё значение в строковом представлении.

size\_t GetPriority() const

*Входные параметры:* нет.

*Выходные параметры:* целое число.

*Действие:* возвращает приоритет данной лексемы – 0.

Operand\* Plus(const Lexem\* operand) const

*Входные параметры:* указатель на Lexem.

*Выходные параметры:* указатель на Operand.

*Действие:* выполняет операцию сложения с другим операндом.

Operand\* Minus(const Lexem\* operand) const

*Входные параметры:* указатель на Lexem.

*Выходные параметры:* указатель на Operand.

*Действие:* выполняет операцию вычитания с другим операндом.

Operand\* UnaryMinus() const

*Входные параметры:* нет.

*Выходные параметры:* указатель на Operand.

*Действие:* выполняет операцию вычитания.

Operand\* Multiple(const Lexem\* operand) const

*Входные параметры:* указатель на Lexem.

*Выходные параметры:* указатель на Operand.

*Действие:* выполняет операцию умножения с другим операндом.

Operand\* Division(const Lexem\* operand) const

*Входные параметры:* указатель на Lexem.

*Выходные параметры:* указатель на Operand.

*Действие:* выполняет операцию деления с другим операндом.

## Неизвестный операнд

UknownOperand шаблонный класс с значением T, представляющий операнд, значение которого нам неизвестно на момент составления выражения, то есть переменную.

Поля:

std::map<std::string, Operand\*> known\_values – словарь каждое

имя переменной ассоциировано с её значением Operand\*.

std::string name – имя переменной.

Методы:

void action(TStack<Lexem\*>& operands) const

*Входные параметры:* ссылка на стек операндов (operands) типа TStack<Lexem\*>

*Выходные параметры:* нет

*Действие:* если в словаре есть значение переменной, то вместо неё подставляется её значение, иначе просит ввести значение переменной.

std::string GetYourself() const

*Входные параметры:* нет

*Выходные параметры:* строка

*Действие:* этот метод возвращает имя переменной.

size\_t GetPriority() const

*Входные параметры:* нет

*Выходные параметры:* целое число

*Действие:* этот метод возвращает приоритет данной лексемы 0.

# Тестирование

Для автоматических тестов был использован Google Test.

## Стек

TEST(TStack, can\_create\_stack\_with\_positive\_length)

Проверка возможности создания стека с положительной длиной. Ожидается, что для стека длиной 5 создание не вызовет исключений.

TEST(TStack, cant\_create\_stack\_with\_negative\_length)

Проверка невозможности создания стека с отрицательной длиной. Ожидается, что создание стека с длиной -5 вызовет исключение.

TEST(TStack, cant\_pop\_empty\_stack)

Проверка невозможности извлечения элемента из пустого стека с помощью метода Pop(). Ожидается, что вызов метода Pop() для пустого стека вызовет исключение.

TEST(TStack, cant\_top\_empty\_stack)

Проверка невозможности получения верхнего элемента из пустого стека с помощью метода Top(). Ожидается, что вызов метода Top() для пустого стека вызовет исключение.

TEST(TStack, can\_push\_element)

Проверка возможности добавления элемента в стек с помощью метода Push(). Ожидается, что добавление элемента (-1) не вызовет исключений.

TEST(TStack, can\_push\_many\_elements)

Проверка возможности добавления нескольких элементов в стек с помощью метода Push(). Ожидается, что добавление 8 элементов (-1) не вызовет исключений.

TEST(TStack, is\_pushed\_elements\_correct)

Проверка правильности добавления элементов в стек. Ожидается, что верхний элемент стека после добавления элемента будет соответствовать добавленному значению.

TEST(TStack, can\_stack\_copy)

Проверка возможности создания копии стека. Ожидается, что создание копии исходного стека не вызовет исключений.

TEST(TStack, is\_stack\_copy\_correct)

Проверка правильности создания копии стека. Ожидается, что содержимое исходного стека и его копии будут идентичны.

## Постфикс

TEST(TPostfix, can\_create\_postfix)

Проверка, что объект TPostfix можно создать без ошибок.

TEST(TPostfix, can\_get\_postfix)

проверка, что после вызова метода ToPostfix() нет ошибок.

TEST(TPostfix, can\_get\_postfix\_brackets)

проверка, что после вызова метода ToPostfix() нет ошибок, при использовании скобок.

TEST(TPostfix, can\_get\_postfix\_empty\_brackets)

Проверка, что после вызова метода ToPostfix() нет ошибок, при использовании пустых скобок.

TEST(TPostfix, cant\_get\_postfix\_no\_close\_bracket)

Проверка, что при отсутствии закрывающей скобки в выражении выбрасывается исключение.

TEST(TPostfix, cant\_get\_postfix\_no\_open\_bracket)

Проверка, что при отсутствии открывающей скобки в выражении выбрасывается исключение.

TEST(TPostfix, can\_get\_postfix\_unary\_minus)

Проверка, что после вызова метода ToPostfix() нет ошибок, при использовании унарного минуса.

TEST(TPostfix, can\_get\_postfix\_division)

Проверка, что после вызова метода ToPostfix() нет ошибок, при использовании деления.

TEST(TPostfix, can\_get\_postfix\_division\_and\_multiple)

Проверка, что после вызова метода ToPostfix() нет ошибок, при использовании умножения.

TEST(TPostfix, can\_get\_postfix\_unary\_minus\_and\_division)

Проверка, что после вызова метода ToPostfix() нет ошибок, при использовании унарного минуса и деления.

TEST(TPostfix, can\_get\_postfix\_unary\_minuses)

Проверка, что после вызова метода ToPostfix() нет ошибок, при использовании нескольких унарных минусов.

TEST(TPostfix, can\_calculate\_postfix)

Проверка, что после вызова метода Calculate() нет ошибок.

TEST(TPostfix, can\_calculate\_postfix\_unary\_minus)

Проверка, что после вызова метода Calculate() нет ошибок, при использовании унарного минуса.

TEST(TPostfix, can\_calculate\_postfix\_unary\_minus\_and\_brackets)

Проверка, что после вызова метода Calculate() нет ошибок, при использовании унарного минуса и скобок.

TEST(TPostfix, can\_calculate\_postfix\_division)

Проверка, что после вызова метода Calculate() нет ошибок, при использовании деления.

TEST(TPostfix, can\_calculate\_postfix\_multiple)

Проверка, что после вызова метода Calculate() нет ошибок, при использовании умножения.

TEST(TPostfix, can\_calculate\_postfix\_minus)

Проверка, что после вызова метода Calculate() нет ошибок, при использовании вычитания.

TEST(TPostfix, can\_calculate\_postfix\_large)

Проверка, что после вызова метода Calculate() нет ошибок, при использовании большого выражения.

TEST(TPostfix, is\_calculation\_postfix\_correct)

Проверка, что рассчет выражения в постфиксной форме выполняется правильно.

TEST(TPostfix, is\_calculation\_postfix\_unary\_minus\_correct)

Проверка, что рассчет выражения в постфиксной форме с унарным минусом выполняется правильно.

TEST(TPostfix,

is\_calculation\_postfix\_unary\_minus\_and\_bracket\_correct)

Проверка, что рассчет выражения в постфиксной форме с унарным минусом выполняется правильно.

TEST(TPostfix, is\_calculation\_postfix\_division\_correct)

Проверка, что рассчет выражения в постфиксной форме с делением выполняется правильно.

TEST(TPostfix, is\_calculation\_postfix\_multiple\_correct)

Проверка, что рассчет выражения в постфиксной форме с умножением выполняется правильно.

TEST(TPostfix, is\_calculation\_postfix\_minus\_correct)

Проверка, что рассчет выражения в постфиксной форме со сложением выполняется правильно.

TEST(TPostfix, is\_calculation\_postfix\_large\_correct)

Проверка, что рассчет большого выражения в постфиксной форме выполняется правильно.

## Результаты тестирования

Все тесты прошли успешно (Рисунки 1, 2, 3)

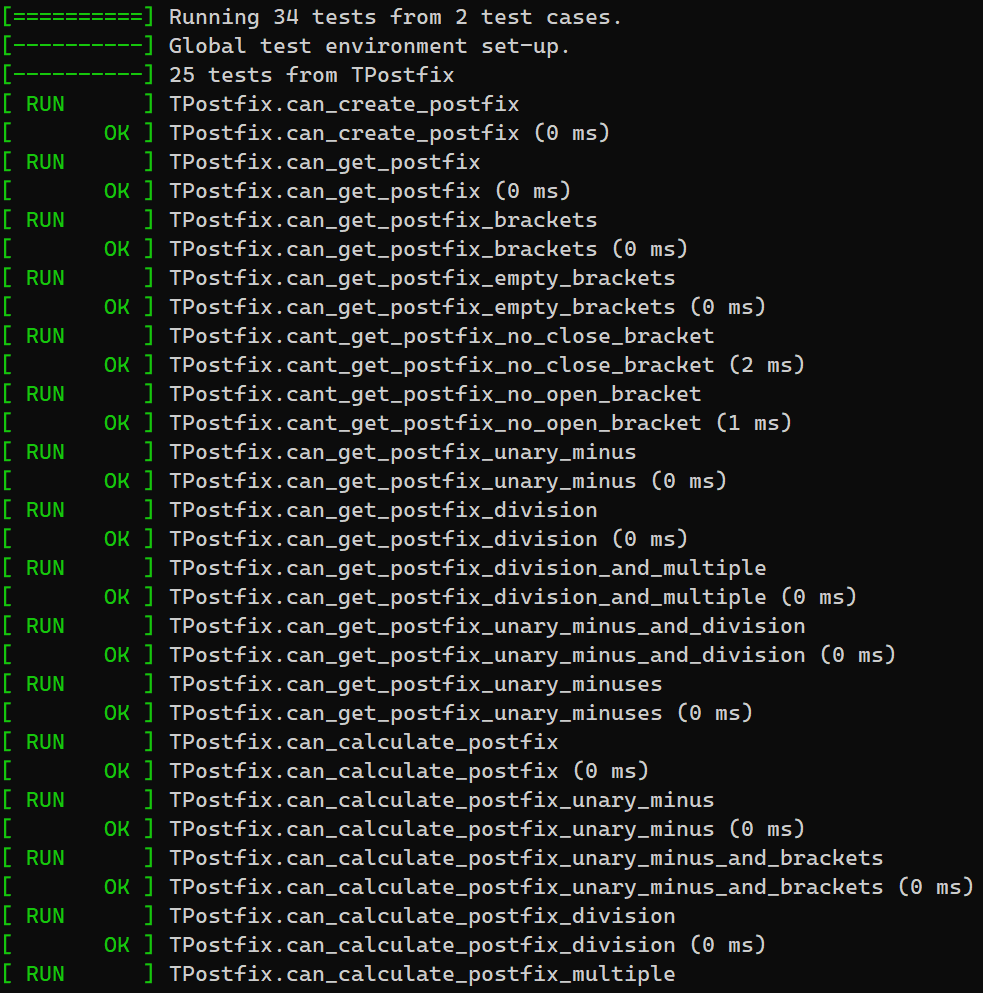


Рисунок 1. Начало тестов

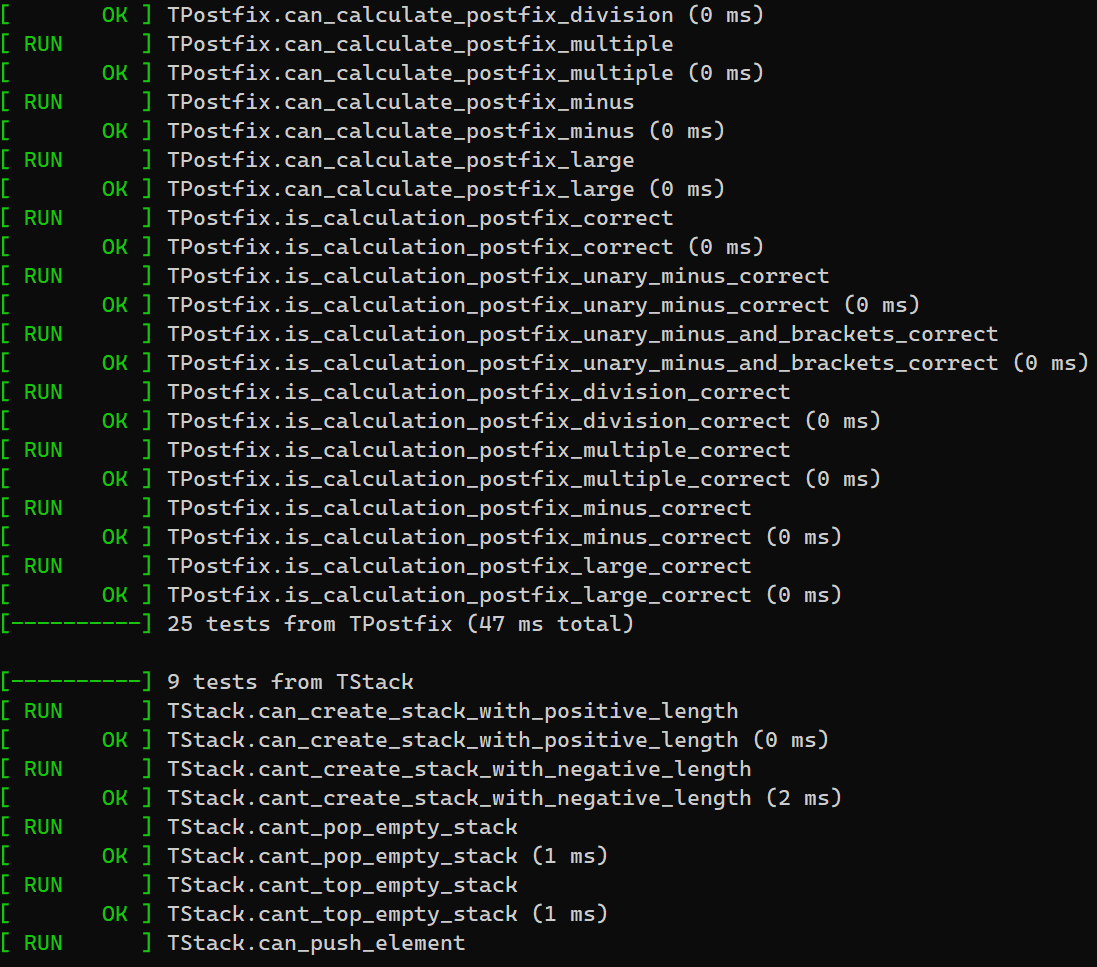


Рисунок 2. Середина тестов

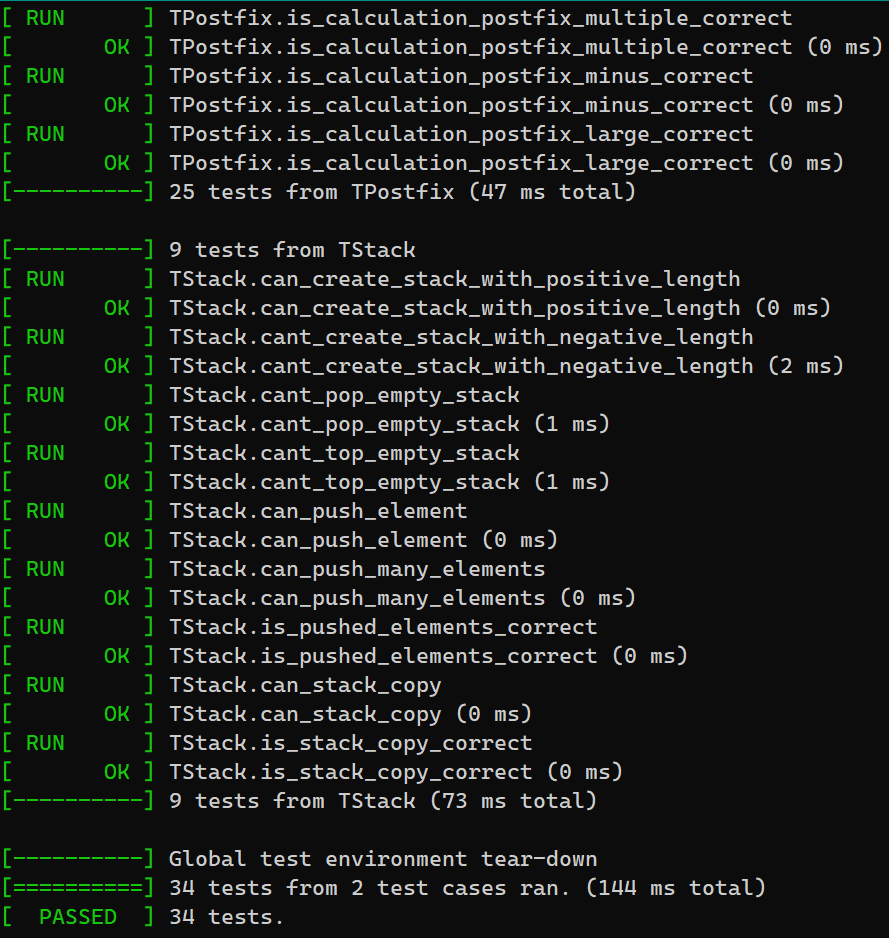


Рисунок 3. Конец тестов

# Выводы

После выполнения работы были достигнуты все цели и задачи:

1. Была разработана структура данных TStack и реализованы методы для ее работы.

2. Был реализован интерфейс для работы с обратной польской записью, а также методы для ее использования.

3. Были разработаны тесты на основе Google Test, которые позволили проверить корректность работы классов TStack и методов постфиксной записи.

4. Исходные коды были опубликованы в открытом репозитории на GitHub, что позволяет другим пользователям изучить и использовать эту работу.

# Список литературы

1. Википедия польская запись - <https://ru.wikipedia.org/wiki/Польская_запись>
2. Википедия стек - <https://ru.wikipedia.org/wiki/Стек>

# Приложение А. Фрагменты кода класса стек

#ifndef \_\_STACK\_H\_\_

#define \_\_STACK\_H\_\_

#include <stdexcept>

const int MaxStackSize = 100;

//#define DYNAMIC\_ARRAY

#ifdef DYNAMIC\_ARRAY

template <class T>

class TStack

{

T\* pMem;

size\_t size;

size\_t end;

public:

void Push(const T& elem);

explicit TStack(size\_t \_size = 2);

TStack(const TStack<T>& stack);

TStack(TStack<T>&& stack) noexcept;

TStack& operator= (TStack<T>&& stack) noexcept;

bool operator== (const TStack<T>& stack) const;

bool operator!= (const TStack<T>& stack) const;

~TStack() { delete[] pMem;}

T& Top();

size\_t Size() const;

bool IsEmpty() const noexcept;

const T& Top() const;

void Pop();

};

template<class T>

inline void TStack<T>::Push(const T& elem)

{

if (end >= size) {

T\* tmp = new T[size\_t(size \* 1.5)];

size = size\_t(size \* 1.5);

for (size\_t i = 0; i < size; i++)

{

tmp[i] = pMem[i];

}

delete[] pMem;

pMem = tmp;

}

pMem[end] = elem;

end++;

}

template<class T>

inline TStack<T>::TStack(size\_t \_size)

{

size = \_size;

end = 0;

if ((size < 1) || (size > MaxStackSize))

throw size;

pMem = new T[size];

}

template<class T>

inline TStack<T>::TStack(const TStack<T>& stack)

{

size = stack.end + 1;

pMem = new T[size];

end = stack.end;

for (size\_t i = 0; i < end; i++)

{

pMem[i] = stack.pMem[i];

}

}

template<class T>

inline TStack<T>::TStack(TStack<T>&& stack) noexcept

{

pMem = stack.pMem;

stack.pMem = nullptr;

size = stack.size;

end = stack.end;

}

template<class T>

inline TStack<T>& TStack<T>::operator=(TStack<T>&& stack) noexcept

{

std::swap(pMem, stack.pMem);

size = stack.size;

end = stack.end;

}

template<class T>

inline bool TStack<T>::operator==(const TStack<T>& stack) const

{

if (end != stack.end) return false;

for (size\_t i = 0; i < end; i++)

{

if (stack.pMem[i] != pMem[i]) return false;

}

return true;

}

template<class T>

inline bool TStack<T>::operator!=(const TStack<T>& stack) const

{

return !(\*this == stack);

}

template<class T>

inline T& TStack<T>::Top()

{

if (end == 0) throw std::out\_of\_range("stack is empty");

return pMem[end - 1];

}

template<class T>

inline const T& TStack<T>::Top() const

{

if (end == 0) throw std::out\_of\_range("stack is empty");

return pMem[end - 1];

}

template<class T>

inline void TStack<T>::Pop()

{

if (end == 0) throw std::out\_of\_range("stack is empty");

end--;

}

template<class T>

inline size\_t TStack<T>::Size() const

{

return size;

}

template<class T>

inline bool TStack<T>::IsEmpty() const noexcept

{

return end == 0;

}

#endif // DynamicArray

#endif

# Приложение Б. Фрагменты кода класса постфикс

#ifndef \_\_POSTFIX\_H\_\_

#define \_\_POSTFIX\_H\_\_

#include <vector>

#include <iostream>

#include <sstream>

#include <string>

//#define DYNAMIC\_ARRAY

//#include "stack.h"

#include "Lexem.h"

#include "Operand.h"

#include "OperatorDivision.h"

#include "OperatorMinus.h"

#include "OperatorMultiple.h"

#include "OperatorPlus.h"

#include "OperatorCloseBracket.h"

#include "OperatorOpenBracket.h"

#include "UnknownOperand.h"

using namespace std;

template <typename T>

class TPostfix

{

string infix;

vector<Lexem\*> postfix;

void AddOperand(std::string& current);

void CheckOpenBracket(TStack<Lexem\*>& operators);

void InsertCloseBracket(std::string& current, TStack<Lexem\*>& operators);

void InsertOperator(TStack<Lexem\*>& operators, Operator\*& tmp);

public:

TPostfix()

{

infix = "3+2\*4";

}

TPostfix(std::string \_infix)

{

infix = \_infix;

}

~TPostfix() {

for (auto lexem : postfix) {

delete lexem;

}

}

string GetInfix() { return infix; }

string GetPostfix() {

string result = "";

for (Lexem\* lexem : postfix) {

result += lexem->GetYourself();

}

return result;

}

void ToPostfix();

// Ввод переменных, вычисление по постфиксной форме

T Calculate();

};

#endif

template<typename T>

inline void TPostfix<T>::AddOperand(std::string& current)

{

if (!current.empty())

{

if (isdigit(current[0]))

postfix.push\_back(new KnownOperand<T>(current));

else

postfix.push\_back(new UnknownOperand(current));

current.clear();

}

}

template<typename T>

inline void TPostfix<T>::CheckOpenBracket(TStack<Lexem\*>& operators)

{

while (!operators.IsEmpty()) {

if (OperatorOpenBracket\* bracket = dynamic\_cast<OperatorOpenBracket\*>(operators.Top()))

throw exception("miss close bracket");

postfix.push\_back(operators.Top());

operators.Pop();

}

}

template<typename T>

inline void TPostfix<T>::InsertCloseBracket(std::string& current, TStack<Lexem\*>& operators)

{

AddOperand(current);

OperatorOpenBracket\* close = nullptr;

while ((!operators.IsEmpty()) && ((close = dynamic\_cast<OperatorOpenBracket\*>(operators.Top())) == nullptr))

{

postfix.push\_back(operators.Top());

operators.Pop();

}

if (close)

{

delete close;

operators.Pop();

}

else

{

throw exception("miss open bracket");

}

}

template<typename T>

inline void TPostfix<T>::InsertOperator(TStack<Lexem\*>& operators, Operator\*& tmp)

{

if (operators.IsEmpty() || operators.Top()->GetPriority() < tmp->GetPriority())

operators.Push(tmp);

else

{

do {

postfix.push\_back(operators.Top());

operators.Pop();

} while (!operators.IsEmpty() && operators.Top()->GetPriority() >= tmp->GetPriority());

operators.Push(tmp);

}

}

template<typename T>

inline void TPostfix<T>::ToPostfix() {

string current = "";

TStack<Lexem\*> operators;

bool isCloseBracketLast = false;

for (char symbol : infix)

{

current += symbol;

Operator\* tmp;

if (tmp = OperatorPlus::IsIt(current)) {

AddOperand(current);

InsertOperator(operators, tmp);

}

else if (tmp = OperatorMultiple::IsIt(current)) {

AddOperand(current);

InsertOperator(operators, tmp);

}

else if (tmp = OperatorOpenBracket::IsIt(current))

{

if (!current.empty()) throw exception("missing operator before open bracket");

operators.Push(tmp);

}

else if (tmp = OperatorCloseBracket::IsIt(current))

{

InsertCloseBracket(current, operators);

isCloseBracketLast = true;

continue;

}

else if (tmp = OperatorMinus::IsIt(current))

{

AddOperand(current);

if (isCloseBracketLast) {

delete tmp;

tmp = new OperatorMinus();

}

InsertOperator(operators, tmp);

}

else if (tmp = OperatorDivision::IsIt(current))

{

AddOperand(current);

InsertOperator(operators, tmp);

}

isCloseBracketLast = false;

//cout << GetPostfix() << std::endl;

}

AddOperand(current);

CheckOpenBracket(operators);

}

// Ввод переменных, вычисление по постфиксной форме

template<typename T>

inline T TPostfix<T>::Calculate() {

TStack<Lexem\*> stack;

for (Lexem\* lexem : postfix) {

lexem->action(stack);

}

KnownOperand<T>\* result = dynamic\_cast<KnownOperand<T>\*>(stack.Top());

T res = result->value;

delete result;

stack.Pop();

if (stack.IsEmpty())

return res;

else

{

while (!stack.IsEmpty())

{

std::cout << stack.Top()->GetYourself() << '\n';

delete stack.Top();

stack.Pop();

}

throw exception("something went wrong");

}

for (auto op : UnknownOperand::known\_values) {

delete op.second;

}

UnknownOperand::known\_values.clear();

}

# Приложение В. Фрагменты кода класса лексема

#pragma once

#define DYNAMIC\_ARRAY

#include "stack.h"

class Lexem {

public:

virtual void action(TStack<Lexem\*>& operands) const = 0;

virtual std::string GetYourself() const = 0;

virtual size\_t GetPriority() const = 0;

};

# Приложение Г. Фрагменты кода класса оператор

#pragma once

#include "Lexem.h"

class Operator : public Lexem

{

public:

static Lexem\* IsIt(std::string& current);

virtual size\_t GetPriority() const = 0;

};

# Приложение Д. Фрагменты кода класса операнд

#pragma once

#include <sstream>

#include <iostream>

#include "Lexem.h"

#include <map>

class Operand : public Lexem

{

public:

virtual void action(TStack<Lexem\*>& operands) const override = 0;

virtual std::string GetYourself() const override = 0;

virtual size\_t GetPriority() const override = 0;

virtual Operand\* Copy() const = 0;

virtual Operand\* Plus(const Lexem\* operand) const = 0;

virtual Operand\* Minus(const Lexem\* operand) const = 0;

virtual Operand\* UnaryMinus() const = 0;

virtual Operand\* Multiple(const Lexem\* operand) const = 0;

virtual Operand\* Division(const Lexem\* operand) const = 0;

};

# Приложение Е. Фрагменты кода класса известный оператор

#pragma once

#include "Operand.h"

template <typename T>

class KnownOperand : public Operand

{

public:

T value;

KnownOperand(const T& \_value) : value(\_value) {}

KnownOperand(std::string& \_value) {

std::stringstream geek;

geek << \_value;

geek >> value;

}

KnownOperand(const KnownOperand<T>& op) {

value = op.value;

}

virtual Operand\* Copy() const override {

return new KnownOperand(value);

}

virtual void action(TStack<Lexem\*>& operands) const override

{

operands.Push(new KnownOperand(\*this));

}

virtual std::string GetYourself() const override

{

return std::to\_string(value);

}

virtual Operand\* Plus(const Lexem\* operand) const override

{

const KnownOperand<T>\* op = dynamic\_cast<const KnownOperand<T>\*>(operand);

Operand\* result;

if (result = new KnownOperand(value + op->value))

return result;

throw "operator + is not defined";

}

Operand\* Minus (const Lexem\* operand) const override

{

const KnownOperand\* op = dynamic\_cast<const KnownOperand<T>\*>(operand);

Operand\* result;

if (result = new KnownOperand<T>(value - op->value))

return result;

throw "operator - is not defined";

}

Operand\* UnaryMinus () const override

{

KnownOperand\* result = nullptr;

if (result = new KnownOperand<T>(-value))

return result;

throw "operator - is not defined";

}

Operand\* Multiple (const Lexem\* operand) const override

{

const KnownOperand<T>\* op = dynamic\_cast<const KnownOperand<T>\*>(operand);

Operand\* result;

if (result = new KnownOperand(value \* op->value))

return result;

throw "operator \* is not defined";

}

Operand\* Division(const Lexem\* operand) const override

{

const KnownOperand<T>\* op = dynamic\_cast<const KnownOperand<T>\*>(operand);

Operand\* result;

if (result = new KnownOperand(value / op->value))

return result;

throw "operator / is not defined";

}

protected:

virtual size\_t GetPriority() const override

{

return 0;

}

};

# Приложение Ё. Фрагменты кода класса неизвестный оператор

#pragma once

#include <sstream>

#include <iostream>

#include "KnownOperand.h"

#include <map>

class UnknownOperand : public Lexem

{

std::string name;

public:

static std::map<std::string, Operand\*> known\_values;

UnknownOperand(const std::string& \_name) : name(\_name) {}

virtual void action(TStack<Lexem\*>& operands) const override

{

try {

operands.Push(known\_values.at(name)->Copy());

}

catch (std::out\_of\_range exception) {

int value;

std::cout << "enter " << name << std::endl;

std::cin >> value;

KnownOperand<int>\* tmp = new KnownOperand<int>(value);

known\_values[name] = tmp;

operands.Push(tmp);

}

}

virtual std::string GetYourself() const override

{

return name;

}

protected:

virtual size\_t GetPriority() const override

{

return 0;

}

};

std::map<std::string, Operand\*> UnknownOperand::known\_values;

# Приложение Ж. Фрагменты кода класса открывающая скобка

#pragma once

#include <exception>

#include "KnownOperand.h"

class OperatorOpenBracket : public Operator {

public:

virtual void action(TStack<Lexem\*>& operands) const override

{

throw std::exception("miss close bracket 2");

}

virtual std::string GetYourself() const override

{

return "(";

}

static Operator\* IsIt(std::string& current)

{

if (current[current.size() - 1] == '(') {

current.erase(current.size() - 1);

return new OperatorOpenBracket();

}

return nullptr;

}

virtual size\_t GetPriority() const override { return 0; }

};

# Приложение З. Фрагменты кода класса закрывающая скобка

#pragma once

#include <exception>

#include "Operator.h"

class OperatorCloseBracket : public Operator {

public:

virtual void action(TStack<Lexem\*>& operands) const override

{

throw std::exception("miss open braket 2");

}

virtual std::string GetYourself() const override

{

return "(";

}

static Operator\* IsIt(std::string& current)

{

if (current[current.size() - 1] == ')') {

current.erase(current.size() - 1);

return new OperatorCloseBracket();

}

return nullptr;

}

virtual size\_t GetPriority() const { return 0; }

};

};

# Приложение И. Фрагменты кода класса сложение

#pragma once

#include <exception>

#include "Operator.h"

#include "KnownOperand.h"

class OperatorPlus : public Operator {

public:

virtual void action(TStack<Lexem\*>& operands) const override

{

Lexem\* rOperand = operands.Top();

operands.Pop();

Lexem\* lOperand = operands.Top();

operands.Pop();

if (Operand\* tmp = dynamic\_cast<Operand\*>(lOperand)) {

Operand\* result = tmp->Plus(rOperand);

operands.Push(result);

delete tmp;

delete rOperand;

return;

}

delete lOperand;

delete rOperand;

throw std::exception("Operator + is not defined");

}

virtual std::string GetYourself() const override

{

return "+";

}

static Operator\* IsIt(std::string& current)

{

if (current[current.size() - 1] == '+') {

current.erase(current.size() - 1);

return new OperatorPlus();

}

return nullptr;

}

virtual size\_t GetPriority() const override { return 1; }

};

# Приложение Й. Фрагменты кода класса вычитание

#pragma once

#include <exception>

#include "Operator.h"

#include "KnownOperand.h"

#include "OperatorUnaryMinus.h"

class OperatorMinus : public Operator {

public:

virtual void action(TStack<Lexem\*>& operands) const override

{

Lexem\* rOperand = operands.Top();

operands.Pop();

Lexem\* lOperand = operands.Top();

operands.Pop();

if (Operand\* tmp = dynamic\_cast<Operand\*>(lOperand)) {

Operand\* result = tmp->Minus(rOperand);

operands.Push(result);

delete tmp;

delete rOperand;

return;

}

delete lOperand;

delete rOperand;

throw std::exception("operator - is not defined");

}

virtual std::string GetYourself() const override

{

return "-";

}

static Operator\* IsIt(std::string& current)

{

if (current[current.size() - 1] == '-') {

//if (current.size() == 1)

// return nullptr;

current.erase(current.size() - 1);

if (!current.empty())

return new OperatorMinus();

else

return new OperatorUnaryMinus();

}

return nullptr;

}

virtual size\_t GetPriority() const { return 1; }

};

# Приложение К. Фрагменты кода класса унарный минус

#pragma once

#include "OperatorMinus.h"

class OperatorUnaryMinus : public Operator {

public:

virtual void action(TStack<Lexem\*>& operands) const override

{

Lexem\* rOperand = operands.Top();

operands.Pop();

//Lexem\* lOperand = operands.Top();

//operands.Pop();

if (Operand\* tmp = dynamic\_cast<Operand\*>(rOperand)) {

Operand\* result = tmp->UnaryMinus();

operands.Push(result);

delete tmp;

return;

}

delete rOperand;

throw std::exception("unary minus is not defined");

}

virtual std::string GetYourself() const override

{

return "-u";

}

static Operator\* IsIt(std::string& current)

{

return nullptr;

}

virtual size\_t GetPriority() const { return 100; }

};

# Приложение Л. Фрагменты кода класса умножение

#pragma once

#include <exception>

#include "KnownOperand.h"

#include "Operator.h"

class OperatorMultiple : public Operator {

public:

virtual void action(TStack<Lexem\*>& operands) const override

{

Lexem\* rOperand = operands.Top();

operands.Pop();

Lexem\* lOperand = operands.Top();

operands.Pop();

if (Operand\* tmp = dynamic\_cast<Operand\*>(lOperand)) {

Operand\* result = tmp->Multiple(rOperand);

operands.Push(result);

delete tmp;

delete rOperand;

return;

}

delete lOperand;

delete rOperand;

throw std::exception("operator \* is not defined");

}

virtual std::string GetYourself() const override

{

return "\*";

}

static Operator\* IsIt(std::string& current)

{

if (current[current.size() - 1] == '\*') {

current.erase(current.size() - 1);

return new OperatorMultiple();

}

return nullptr;

}

virtual size\_t GetPriority() const { return 2; }

};

# Приложение М. Фрагменты кода класса деление

#pragma once

#include <exception>

#include "Operator.h"

#include "KnownOperand.h"

class OperatorDivision : public Operator {

public:

virtual void action(TStack<Lexem\*>& operands) const override

{

Lexem\* rOperand = operands.Top();

operands.Pop();

Lexem\* lOperand = operands.Top();

operands.Pop();

//Operator\* result = lOperand->Division(rOperand);

if (Operand\* tmp = dynamic\_cast<Operand\*>(lOperand)) {

Operand\* result = tmp->Division(rOperand);

operands.Push(result);

delete tmp;

delete rOperand;

return;

}

delete lOperand;

delete rOperand;

throw std::exception("operator / is not defined");

}

virtual std::string GetYourself() const override

{

return "/";

}

static Operator\* IsIt(std::string& current)

{

if (current[current.size() - 1] == '/') {

current.erase(current.size() - 1);

return new OperatorDivision();

}

return nullptr;

}

virtual size\_t GetPriority() const { return 2; }

};