**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №3**

**по дисциплине «Алгоритмы и структуры данных»**

**Тема: «Стеки и очереди»**

Студент гр. 7381 Минуллин М.А.

Преподаватель Фирсов М.А.

Санкт-Петербург

2018

**Задание.**

11д-в.

Рассматривается выражение следующего вида:

< выражение >::= < терм > | < терм > + < выражение > |

< терм > − < выражение >

< терм > ::= < множитель > | < множитель > \* < терм >

< множитель > ::= < число > | < переменная > | ( < выражение > ) |

< множитель > ^ < число >

< число > ::= < цифра >

< переменная > ::= < буква >

Такая форма записи выражения называется ***инфиксной***.

***Постфиксной (префиксной)*** формой записи выражения *aDb* называется запись, в которой знак операции размещен за (перед) операндами: *abD* (*Dab*).

*Примеры*

***Инфиксная***  ***Постфиксная***  ***Префиксная***

*a*−*b* *ab−* −*ab*

*a*\**b*+*c* *ab*\**c*+ +\**abc*

*a*\*(*b*+*c*) *abc*+\* \**a*+*bc*

*a*+*b*^*c*^*d*\**e* *abc*^*d*^*e*\*+ +*a*\*^*b*^*cde*.

Отметим, что постфиксная и префиксная формы записи выражений не содержат скобок.

Требуется:

Вывести в обычной (инфиксной) форме выражение, записанное в постфиксной форме в заданном текстовом файле *postfix* (рекурсивные процедуры не использовать и лишние скобки не выводить).

**Пояснение задания.**

Постфиксная нотация – форма записи математических и логических выражений, в которой операнды расположены перед знаками операций.

Отличительной особенностью обратной польской нотации является то, что все аргументы (или операнды) расположены перед знаком операции. В общем виде запись выглядит следующим образом:

* Запись набора операций состоит из последовательности операндов и знаков операций. Операнды в выражении при письменной записи разделяются пробелами.
* Выражение читается слева направо. Когда в выражении встречается знак операции, выполняется соответствующая операция над двумя последними встретившимися перед ним операндами в порядке их записи. Результат операции заменяет в выражении последовательность её операндов и её знак, после чего выражение вычисляется дальше по тому же правилу.
* Результатом вычисления выражения становится результат последней вычисленной операции.

Задачу перевода выражения из постфиксной в инфиксную, привычную для рядовых вычислений удобно решать с помощью стека. Стек, в свою очередь, будет написан на основе вектора.

Корректные входные данные можно поделить на *операнды* и *операции* над ними. *Операндом* считается цифра или буква латинского алфавита. *Операцией* – любой из символов сложения (+), вычитания (-), умножения (\*), деления (/) и возведение в степень (^). Нейтральными данными считаются пробельные символы, они игнорируются. Все остальные символы считаются недопустимыми для получения результаты, но корректно обрабатываемыми программой.

**Описание алгоритма.**

Рассмотрим перевод корректного выражения из постфиксной записи в инфиксную:

1. Если текущий элемент – операнд, то заносим его в стек, помечаем его минимальным приоритетом.
2. Если текущий элемент – знак операции, то
   1. Берём из стека 2 верхних элемента.
   2. Если приоритет операции первого элемента первого операнда меньше текущей операции и не равен минимальному приоритету, то обрамляем данный операнд скобками.
   3. Аналогично поступаем со вторым операндом.
   4. Возвращаем в стек склеенные второй операнд, знак операции и первый операнд, помечаем их приоритет текущей операцией.
3. Если строка кончилась, вытаскиваем из стека последний (и единственный) элемент – выражение в инфиксной нотации.

Возможно провести эмуляцию работы алгоритма, не используя стек в явном виде. Эта эмуляция позволяет за линейное время произвести проверку корректности входных данных и избежать дополнительных проверок внутри основного метода.

**Описание функций и структур данных.**

bool is\_digit(char c); – функция, определяющая, является ли данный символ цифрой.

char c – проверяемый символ.

*Возвращаемое значение:* логический тип, истина, если является, ложь – иначе.

bool is\_letter(char c); – функция, определяющая, является ли данный символ буквой.

char c – проверяемый символ.

*Возвращаемое значение:* логический тип, истина, если является, ложь – иначе.

bool is\_operation(char c); – функция, определяющая, является ли данный символ знаком операции.

char c – проверяемый символ.

*Возвращаемое значение:* логический тип, истина, если является, ложь – иначе.

operation\_t get\_operation(char c); – функция, определяющая приоритет операции по её символу.

char c – проверяемый символ.

*Возвращаемое значение:* приоритет данной операции.

bool is\_valid\_postfix(const std::string& postfix); – функция, проверяющая, является ли заданная строка корректной в условиях данной задачи.

const std::string& postfix – строка выходных данных

*Возвращаемое значение:* логический тип: истина, если строка корректна.

std::string postfix\_to\_infix(const std::string& postfix); – функция, переводящая данное выражение из постфиксной формы в инфиксную.

const std::string& postfix – входная строка.

*Возвращаемое значение:* строка в инфиксной форме записи.

class Stack – шаблонный класс, реализующий абстрактный тип данных под названием стек, представляющий собой список элементов, организованных по принципу LIFO – Last In – First Out (первый вошёл – последний вышел). Стек реализован на базе динамического массива. Экземпляр класса хранит в себе:

size\_t \_top; – Количество элементов в структуре. Используется как индекс для вставки нового элемента.

size\_t \_size; – Количество элементов, под которые выделена память и которые стек может принять в себя без накладных расходов на увеличение размера стека.

Type \*\_data; – Динамический массив, в котором хранятся элементы стека. Его размер всегда соответствует значению \_size.

Так же используются значения, общие для всех экземпляров класса.

static const size\_t incr\_factor = 2; – Множитель объёма памяти при переполнении стек или при его избыточного запустения.

static const size\_t decr\_factor = 3; – Множитель для определения избыточного запустения стека. Если количество элементов, для которых выделена память больше, чем количество хранимых элементов в это число раз, то стек сжимается.

Методы для работы со стеком:

Stack(); – стандартный конструктор стека. Инициализирует все поля объекта стандартными значениями и выделяет память для одного элемента.

Type top(); – Метод, возвращающий элемент с вершины стека.

*Возвращаемое значение:* функция возвращает копию объекта на вершине стека.

void pop(); – Метод, удаляющий верхний элемент стека.

*Возвращаемое значение:* функция ничего не возвращает.

void push(const Type& value); – Метод, вставляющий в вершину стека новый элемент.

*Возвращаемое значение:* функция ничего не возвращает.

size\_t size() const; – Метод, определяющий количество элементов в коллекции.

*Возвращаемое значение:* целое беззнаковое число: количество элементов в коллекции.

bool empty() const; – Метод, определяющий, пуст ли стек.

*Возвращаемое значение:* логический тип: истина, если коллекция пуста.

~Stack(); – Деструктор класса, освобождает выделенную память под элементы стека.

**Тестирование.**

Для тестирования использовался скрипт, унаследованный из второй лабораторной работы, перенесённый практически без изменений. В директории проекта папка с тестами содержит в себе две папки: с корректными и некорректными тестами. Имена файлов выбраны для демонстрации тех или иных тестируемых случаев (empty\_string.txt, к примеру, пустой файл, а unknown\_symbols.txt содержит символы, недопустимые в постфиксной нотации). Ниже представлена таблица с результатами тестирования. Первые два теста указаны с полным выводом информации, последующие – с кратким.

|  |  |  |
| --- | --- | --- |
| № | Входные данные | Выходные данные |
| 1 | a | correct test: "easy1.txt"  input: a  result: a |
| 2 | 1 1 + | correct test: "easy2.txt"  input: 1 1 +  result: 1 + 1 |
| 3 | a 7 ^ 8 + c \* 9 / f - | result: (a ^ 7 + 8) \* c / 9 - f |
| 4 |  | error: input is empty |
| 5 | a b c | error: too little operations |
| 6 | a b c ^ / \* | error (index: 10, symbol: \*): operand expected |
| 7 | a b c () d | error (index: 6, symbol: (): not supported symbol |

**Выводы.**

В процессе выполнения лабораторной работы были изучены возможности мета-программирования в языке C++, программная реализация такой структуры данных, как стек, постфиксная нотация и методы перевода выражений из постфиксной в инфиксную нотацию. Закреплены навыки работы с системой контроля версий, мейк-файлами, bash-скриптами. Систематизированы навыки полуавтоматического тестирования программ.

**ПРИЛОЖЕНИЯ**

Приложение А. Текст скрипта полуавтоматического тестирования.

#!/bin/bash

make

if [ -f "result.txt" ]; then

rm result.txt

fi

touch result.txt

for i in $(ls tests/correct); do

echo "running test: \"tests/correct/$i\" ";

sleep 0.11s;

echo "correct test: \"$i\"" >>result.txt;

./postfix\_to\_infix <tests/correct/$i >>result.txt;

done;

for i in $(ls tests/incorrect); do

echo "running test: \"tests/incorrect/$i\" ";

sleep 0.11s;

echo "incorrect test: \"$i\"" >>result.txt;

./postfix\_to\_infix <tests/incorrect/$i >>result.txt;

done;

make clean

Приложение Б. Текст заголовочного файла класса Stack.

#ifndef \_\_STACK\_HPP\_\_

#define \_\_STACK\_HPP\_\_

#include <stack>

#include <cstddef>

template <class Type>

class Stack {

private:

// Количество реально хранимых элементов

size\_t \_top;

// Под сколько элементов выделено памяти

size\_t \_size;

// Массив данных

Type \*\_data;

// Множитель объёма памяти при переполнении стэка

static const size\_t incr\_factor = 2;

// если \_top < \_size / decr\_factor (пороговое значение),

// то следует уменьшить объём выделенной памяти.

// для оптимального выделения памяти decr\_factor > incr\_factor

static const size\_t decr\_factor = 3;

public:

Stack();

Type top();

void pop();

void push(const Type& value);

size\_t size() const;

bool empty() const;

~Stack();

};

template <class Type>

Stack<Type>::Stack() {

\_top = 0;

\_size = 1;

\_data = new Type[\_size];

}

template <class Type>

Type Stack<Type>::top() {

return \_data[\_top - 1];

}

template <class Type>

void Stack<Type>::pop() {

--\_top;

// Если стэке хранится меньше, чем \_size / decr\_factor элементов

// то размер выделенной памяти вменьшается в incr\_factor раз

if (\_top \* decr\_factor < \_size && \_size > 1) {

size\_t new\_size = \_size / incr\_factor;

// Копирование содержимого стэка в новую область памяти

Type \*new\_data = new Type[new\_size];

for (size\_t index = 0; index < new\_size; ++index)

new\_data[index] = \_data[index];

// Освобождение старых ресурсов

delete[] \_data;

\_size = new\_size;

\_data = new\_data;

}

}

template <class Type>

void Stack<Type>::push(const Type& value) {

\_data[\_top] = value;

++\_top;

// Если стэк переполнен, то объём выделенной

// памяти увеличивается в incr\_factor раз

if (\_top == \_size) {

size\_t new\_size = \_size \* incr\_factor;

// Копирование содержимого стэка в новую область памяти

Type \*new\_data = new Type[new\_size];

for (size\_t index = 0; index < \_size; ++index)

new\_data[index] = \_data[index];

// Освобождение старых ресурсов

delete[] \_data;

\_size = new\_size;

\_data = new\_data;

}

}

template <class Type>

size\_t Stack<Type>::size() const {

return (\_top);

}

template <class Type>

bool Stack<Type>::empty() const {

return (!\_top);

}

template <class Type>

Stack<Type>::~Stack() {

// Память выделена даже для пустого стэка, поэтому

// дополнительная проверка не требуется

delete[] \_data;

}

#endif

Приложение В. Текст файла с классом Stack.

#include "stack.hpp"

Приложение Г. Текст основного кода программы.

#include <string>

#include <iostream>

#include <stack>

#include "stack.hpp"

// Приоритеты обрабатываемых операций

enum operation\_t { NOTHING = 0, ADDITION = 1, SUBSTRACTION = 1, MULTIPLICATION = 2, DIVIDING = 2, POWER = 3 };

bool is\_digit(char c);

bool is\_letter(char c);

bool is\_operation(char c);

operation\_t get\_operation(char c);

bool is\_valid\_postfix(const std::string& postfix);

std::string postfix\_to\_infix(const std::string& postfix);

int main() {

std::string postfix;

std::getline(std::cin, postfix);

std::cout << "input: " << postfix << std::endl;

if (is\_valid\_postfix(postfix))

std::cout << "result: " << postfix\_to\_infix(postfix) << std::endl;

std::cout << std::endl;

return 0;

}

bool is\_digit(char c) {

return ('0' <= c && c <= '9');

}

bool is\_letter(char c) {

return (('a' <= c && c <= 'z') || ('A' <= c && c <= 'Z'));

}

bool is\_operation(char c) {

return (c == '+' || c == '-' || c == '\*' || c == '/' || c == '^');

}

operation\_t get\_operation(char c) {

switch (c) {

case '+':

return ADDITION;

case '-':

return SUBSTRACTION;

case '\*':

return MULTIPLICATION;

case '/':

return DIVIDING;

case '^':

return POWER;

default:

return NOTHING;

}

}

bool is\_valid\_postfix(const std::string& postfix) {

// Имеет ли строка нулевую длину без пробельных символов,

bool empty = true;

for (size\_t index = 0; index < postfix.length(); ++index)

if (!isspace(postfix[index]))

empty = false;

if (empty) {

std::cout << "error: input is empty" << std::endl;

return false;

}

// Переменная, эмулирующая количество элементов в стэке

int operands = 0;

for (size\_t index = 0; index < postfix.length(); ++index) {

// Числа и буквы считаются операндами

if (is\_digit(postfix[index]) || is\_letter(postfix[index]))

++operands;

// Знаки "+", "-", "\*", "/", "^" считаются операциями

else if (is\_operation(postfix[index])) {

// Если в стэке меньше 2 элементов, входные данные некорректны

if (operands < 2) {

std::cout << "error (index: " << index << ", symbol: " << postfix[index] << "): operand expected" << std::endl;

return false;

}

// Извлечь 2 операнда, вернуть в стэк 1, итого: -1 операнд

--operands;

}

// Символ не входит в алфавит возможных символов входных данных

else if (!isspace(postfix[index])) {

std::cout << "error (index: " << index << ", symbol: " << postfix[index] << "): not supported symbol" << std::endl;

return false;

}

}

// В стэке должен остаться ровно 1 элемент

if (operands != 1) {

std::cout << "error: too little operations" << std::endl;

return false;

}

return true;

}

std::string postfix\_to\_infix(const std::string& postfix) {

Stack< std::pair<std::string, operation\_t> > stack;

for (size\_t index = 0; index < postfix.length(); ++index) {

if (is\_digit(postfix[index]) || is\_letter(postfix[index]))

stack.push(std::make\_pair(postfix.substr(index, 1), NOTHING));

else if (is\_operation(postfix[index])) {

// Вытаскиваем с вершины стека 2 операнда в обратном порядке:

// сначала - правый, затем - левый

std::pair<std::string, operation\_t> operand\_r = stack.top();

stack.pop();

std::pair<std::string, operation\_t> operand\_l = stack.top();

stack.pop();

operation\_t new\_operation = get\_operation(postfix[index]);

// Добавляем скобки, если приоритет новой операции выше

// приоритета sub operation левого операнда

if (operand\_l.second != NOTHING && operand\_l.second < new\_operation)

operand\_l.first = "(" + operand\_l.first + ")";

// Добавляем скобки, если приоритет новой операции выше

// приоритета sub operation правого операнда

if (operand\_r.second != NOTHING && operand\_r.second < new\_operation)

operand\_r.first = "(" + operand\_r.first + ")";

// Возвращаем в стек объединение 2 операндов и знак операции,

// обрамлённый двумя пробельными символами справа и слева

stack.push(std::make\_pair(operand\_l.first + " " + postfix.substr(index, 1) + " " + operand\_r.first, new\_operation));

}

}

// Возвращаем единственный оставший операнд стеке - решение задачи

return stack.top().first;

}

Приложение Д. Текст Make-файла.

CODE    = ./source/

OBJ     = main.o stack.o

EXE     = postfix\_to\_infix

CXX     = g++

CFLAGS  = -std=c++11 -Wall -Wextra -c

all: $(OBJ)

    $(CXX) $(OBJ) -o $(EXE)

main.o: $(CODE)main.cpp

    $(CXX) $(CFLAGS) $(CODE)main.cpp

stack.o: $(CODE)stack.hpp $(CODE)stack.cpp

    $(CXX) $(CFLAGS) $(CODE)stack.cpp

clean:

    rm $(OBJ)

cleanest:

    rm $(OBJ) $(EXE)