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.

11-l-в. 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
    a. Берём из стека 2 верхних элемента.
    b. Если приоритет операции первого элемента первого операнда меньше текущей операции и не равен минимальному приоритету, то обрамляем данный операнд скобками.
    c. Аналогично поступаем со вторым операндом.
    d. 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.

```
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.

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)
```