МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Алгоритмы и структуры данных»

Тема: Стеки и очереди

Студент гр. 7381 _____ Ильясов А.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2018

Задание

Вариант 11-г-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:

Перевести выражение, записанное в обычной (инфиксной) форме в заданном текстовом файле *infix*, в постфиксную форму и в таком виде записать его в текстовый файл *prefix*.

Пояснение задания

Польская нотация (запись), также известна как префиксная нотация (запись), это форма записи логических, арифметических и алгебраических выражений. Характерная черта такой записи — оператор располагается слева

от операндов. Если оператор имеет фиксированную арность, то в такой записи будут отсутствовать круглые скобки и она может быть интерпретирована без неоднозначности. Польский логик Ян Лукасевич изобрел эту запись примерно в 1920, чтобы упростить пропозициональную логику.

Описание алгоритма

Обработка входной строки, представляющей инфиксную запись выражения, производится с конца. Если последний символ не является буквой, цифрой или ‘)’, то выводится соответствующее сообщение об ошибке и программа завершает работу. Далее в цикле проверяется каждый символ от предпоследнего до первого: если символ – цифра или буква, то символ записывается в начало выходной строки, представляющей префиксную запись того же выражения. Сразу после этого проверяется символ стоящий перед только что проверенным, и если он также является цифрой или буквой, то выводится сообщение об ошибке, так как переменные и цифры должны быть разделены знаком операции или ‘)’. Если же проверяемый символ – знак математической операции, то проверяется предыдущий символ: если он также знак операции, то выводится сообщение об ошибке, так как не может стоять 2 знака операции подряд; если символ – не цифра, не буква и не ‘)’, то выводится сообщение об ошибке, что ожидался иной символ. Далее если стек пуст, то символ знака операции помещается в стек, если же стек не пуст, то до тех пор, пока приоритет операции, символ знака которой лежит в вершине стека, не будет ниже приоритета операции, символ знака которой является текущим просматриваемым, из стека извлекаются эти символы и помещаются в начало выходной строки, и затем текущий символ помещается в стек. Если проверяемый символ – ‘)’, то проверяется предыдущий символ, и если он является символом знака операции, то выводится сообщение об ошибке, иначе символ ‘)’ кладется в стек. Если проверяемый символ – ‘(’, то пока в стеке лежат символы знаков операций, они извлекаются и помещаются в начало выходной строки, и если после этого в вершине стека не лежит ‘)’, то это значит, что нарушена парность скобок, и выводится сообщение об ошибке. Ну

и если проверяемый символ не прошел ни одну из выше перечисленных проверок, значит, этот символ является недопустимым в данном формате, и выводится сообщение об ошибке. После того, как проверена вся входная строка, проверяется стек, и если он не пуст, то выводится сообщение об ошибке. Таким образом алгоритм работает за линейное время, одновременно и проверяя строку на корректность, и перезаписывая выражение в префиксную форму.

Описание функций и структур данных.

`size_t priority(char operation);` – функция, определяющая приоритет данной операции.

`char operation` – символ проверяемой операции.

Возвращаемое значение: целое беззнаковое число: приоритет математической операции.

`bool is_operation(char symbol);` – функция, определяющая, является ли данный символ знаком операции.

`char symbol` – проверяемый символ.

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

`void infix_to_prefix(std::string &infix_data_notation, std::string &prefix_data_notation);` – функция, переводящая данное выражение из инфиксной формы в префиксную.

`const std::string &infix_data_notation` – входная строка.

`std::string &prefix_data_notation` – выходная строка.

Возвращаемое значение: функция ничего не возвращает.

`class Stack` – шаблонный класс, реализующий абстрактный тип данных под названием стек. Стек реализован на базе динамического массива.

Экземпляр класса хранит в себе:

`size_t _top;` – количество элементов, находящихся в стеке.

`size_t _size;` – количество элементов, под которые выделена.

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

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 стеке.

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

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

Возвращаемое значение: логический тип: истина, если стек пуст.

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

Тестирование

Для проверки работоспособности программы был создан скрипт(см. ПРИЛОЖЕНИЕ В) для автоматического ввода и вывода тестовых данных. Результаты тестирования сохраняются в файл testsresult.txt.

Ниже представлена таблица тестирования:

Входные данные	Выходные данные
f+2	+ f 2
(a+b)*2	* + a b 2
((((((a+1)*b)-2)/c)+3)*0)	* + / - * + a 1 b 2 c 3 0
q+e-2/e+4*E-4/R+3+s-f/8	+ q - e + / 2 e - * 4 E + / 4 R + 3 - s / f 8
(w) * (4) / (b - s)	* w / 4 - b s
)e+w(	--Error! At the beginning of the line expected '(', digit or letter, but was ')'--
(e*-w)	--Error! 2 operators in a row--
(((((((a+1)*b)-2)/c)+3)*0)	--Error! Missing ')'--
3+e-4*r+4-r/4*r+2/3r-4+*r/4*r+r*	--Error! At the end of the line expected ')', digit or letter, but was '*'--
((a*2)/(f-5))/((b/r)+(f&5))	Error! Extraneous symbol '&'--

Выводы

В процессе выполнения лабораторной работы были получены знания и навыки по реализации такой структуры данных, как стек, были изучены различные способы записи математических выражений и методы перевода выражений из инфиксной в префиксную нотацию. Закреплены навыки работы с системой контроля версий, мейк-файлами, bash-скриптами.

ПРИЛОЖЕНИЕ А

КОД MAIN.CPP

```
#include <iostream>
#include <string>
#include <fstream>
#include <cctype>
#include <cstdlib>
#include <algorithm>

#include "stack.hpp"

// Раскомментировать при запуске тестов
#define TEST

size_t priority(char operation);
bool is_operation (char symbol);
void infix_to_prefix(std::string &infix_data_notation, std::string
&prefix_data_notation);

int main() {
     std::string infix_data_notation;
     std::string prefix_data_notation;

     #ifndef TEST
          std::ifstream input_file("infix.txt");                //
Чтение данных
          std::getline(input_file, infix_data_notation);
     // происходит
          input_file.close();                           // с файла
infix.txt
     #else
          std::getline(std::cin, infix_data_notation);          //
Чтение происходит с
     #endif


infix_data_notation.erase(remove_if(infix_data_notation.begin(),
infix_data_notation.end(), isspace), infix_data_notation.end());
     infix_to_prefix(infix_data_notation, prefix_data_notation);

     #ifndef TEST
          std::ofstream output_file("prefix.txt");              //
Запись данных
          output_file << prefix_data_notation;                  //
просиходит
          output_file.close();                              // в
файл prefix.txt
     #else
```

```
          std::cout << prefix_data_notation << std::endl;
     // Вывод происходит в
     #endif

     prefix_data_notation.clear();
     infix_data_notation.clear();
exit(0);
}

size_t priority(char operation) {
     switch (operation) {
          case '+': case'-':
               return 1;
          case '*': case '/':
               return 2;
          case '^':
               return 3;
          default:
               return 0;
     }
}

bool is_operation (char symbol) {
     return (symbol == '^' || symbol == '*' || symbol == '/' ||
symbol == '+' || symbol == '-') ? true : false;
}

void infix_to_prefix(std::string &infix_data_notation, std::string
&prefix_data_notation) {
     Stack<char> stack;

     if (!isalpha(infix_data_notation[0]) && !
isdigit(infix_data_notation[0]) && infix_data_notation[0] != '(')
{
          std::cout << "--Error! At the beginning of the line
expected '(', digit or letter, but was '" <<
infix_data_notation[0] << "'--" << std::endl;
          exit(0);
     }

     int index = infix_data_notation.length() - 1;
     if (!isalpha(infix_data_notation[index]) && !
isdigit(infix_data_notation[index]) &&
infix_data_notation[index] != ')') {
          std::cout << "--Error! At the end of the line expected
')', digit or letter, but was '" << infix_data_notation[index] <<
"'--" << std::endl;
          exit(0);
     }

     while (index >= 0) {
```

```
     // Обход строки производится в обратном порядке
          if (isalpha(infix_data_notation[index]) ||
isdigit(infix_data_notation[index])) {
               prefix_data_notation.insert(0, 1,
infix_data_notation[index]);
               if (!isalpha(infix_data_notation[index-1]) && !
isdigit(infix_data_notation[index-1]))
                    prefix_data_notation.insert(0, 1, ' ');
               else {
                    std::cout << "--Error! 2 variables in a row--"
<< std::endl;
                    exit(0);
               }
          }
          else if (is_operation(infix_data_notation[index])) {
               if (is_operation(infix_data_notation[index-1])){
                    std::cout << "--Error! 2 operators in a row--"
<< std::endl;
                    exit(0);
               }
               if (!isalpha(infix_data_notation[index-1]) && !
isdigit(infix_data_notation[index-1]) &&
infix_data_notation[index-1] != ')') {
                    std::cout << "--Error! Expected ')', digit or
letter, but was '" << infix_data_notation[index-1] << "'--" <<
std::endl;
                    exit(0);
               }
               if (stack.empty())
                    stack.push(infix_data_notation[index]);
               else {
                    while (priority(infix_data_notation[index]) <=
priority(stack.top()) && !stack.empty()) {
                         prefix_data_notation.insert(0, 1,
stack.top());
                         stack.pop();
                         prefix_data_notation.insert(0, 1, ' ');
                    }
                    stack.push(infix_data_notation[index]);
               }
          }
          else if (infix_data_notation[index] == ')') {
               if (is_operation(infix_data_notation[index-1])) {
                    std::cout << "--Error! After ')' expected ')',
digit or letter, but was '" << infix_data_notation[index-1] <<
"'--" << std::endl;
                    exit(0);
               }
               stack.push(infix_data_notation[index]);
          }
          else if (infix_data_notation[index] == '(') {
```

```
                while (priority(stack.top())) {
                     prefix_data_notation.insert(0, 1, stack.top());
                     stack.pop();
                     prefix_data_notation.insert(0, 1, ' ');
                }
                if (stack.top() != ')') {
                     std::cout << "--Error! Missing ')'--" <<
std::endl;
                     exit(0);
                }
                stack.pop();
           }
           else {
                std::cout << "Error! Extraneous symbol '" <<
infix_data_notation[index] << "'--" << std::endl;
                exit(0);
           }
           index--;
     }
     while (!stack.empty()) {
           if (stack.top() == ')') {
                // Если ) больше, чем (, то в стеке должны остаться
лишние
                std::cout << "Error! ')' more than '('--" <<
std::endl;
                exit(0);
           }
           prefix_data_notation.insert(0, 1, stack.top());
           stack.pop();
           prefix_data_notation.insert(0, 1, ' ');
     }
}
```

ПРИЛОЖЕНИЕ Б
ФАЙЛ STACK.HPP

```
#ifndef __STACK_HPP__
#define __STACK_HPP__

#include <cstddef>

template <class Type>
class Stack {
     private:
          size_t _top;
          size_t _size;
          Type *_data;
     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;
}

template <class Type>
void Stack<Type>::push(const Type &value) {
     _data[_top] = value;
     ++_top;

     if (_top == _size) {
          size_t new_size = _size * 2;
          Type *new_data = new Type[new_size];
          for (size_t index = 0; index < _size; index++)
               new_data[index] = _data[index];
          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() {
     delete[] _data;
}

#endif
```

ПРИЛОЖЕНИЕ В
ФАЙЛ RUNTESTS.SH

```
#!/bin/bash

if test ! -f "lab3" ; then
     g++ ./Source/main.cpp -o lab3
fi
if test -f "testsresult.txt"; then
     rm testsresult.txt
fi

touch testsresult.txt

for i in $(ls ./Tests/correct); do
     echo "running correct $i: $(cat Tests/correct/$i | more)"
          sleep 0.1s
     echo "correct "$i"" >> testsresult.txt
     echo "test data: $(cat Tests/correct/$i | more)" >>
testsresult.txt
     echo "result: " >> testsresult.txt
     ./lab3 < ./Tests/correct/$i >> testsresult.txt
     echo -e >> testsresult.txt
     echo "##################################################" >>
testsresult.txt
done
for i in $(ls ./Tests/incorrect); do
     echo "running incorrect $i: $(cat Tests/incorrect/$i | more)"
          sleep 0.1s
     echo "incorrect "$i"" >> testsresult.txt
     echo "test data: $(cat Tests/incorrect/$i | more)" >>
testsresult.txt
     echo "result: " >> testsresult.txt
     ./lab3 < ./Tests/incorrect/$i >> testsresult.txt
     echo -e >> testsresult.txt
     echo "##################################################" >>
testsresult.txt
done

sleep 0.2s
echo -e
```

```
echo  "test results are saved in testsresult.txt"

rm lab3
```

ПРИЛОЖЕНИЕ Г
ФАЙЛ RUN.SH

```
#!/bin/bash

if test ! -f "infix.txt"; then
     echo "Infix.txt not found"
     exit
fi
if test -f "prefix.txt"; then
     rm prefix.txt
fi

touch prefix.txt

g++ ./Source/main.cpp -o lab3
./lab3

echo "The expression is translated in the prefix form and written
in the prefix.txt"

rm lab3
```