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

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ №4
ПО ДИСЦИПЛИНЕ «ПРОГРАММИРОВАНИЕ»
ТЕМА: ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ.

Студентка гр. 1304 _____ Ярусова Т. В.

Преподаватель _____ Чайка К. В.

Санкт-Петербург
2022

Цель работы.

Изучить динамических структур данных. Освоение работы с классами.

Задание.

Требуется написать программу, получающую на вход строку, (без кириллических символов и не более 3000 символов) представляющую собой код "простой" html-страницы и проверяющую ее на валидность. Программа должна вывести **correct** если страница валидна или **wrong**.

Html-страница, состоит из тегов и их содержимого, заключенного в эти теги. Теги представляют собой некоторые ключевые слова, заданные в треугольных скобках. Например, <**tag**> (где tag - имя тега). Область действия данного тега распространяется до соответствующего закрывающего тега </**tag**> который отличается символом /. Теги могут иметь вложенный характер, но не могут пересекаться:

<tag1><tag2></tag2></tag1> - верно;

<tag1><tag2></tag1></tag2> - не верно;

Существуют теги, не требующие закрывающего тега.

Валидной является html-страница, в коде которой всякому открывающему тегу соответствует закрывающий (за исключением тегов, которым закрывающий тег не требуется)

Во входной строке могут встречаться любые парные теги, но гарантируется, что в тексте, кроме обозначения тегов, символы < и > не встречаются, аттрибутов у тегов также нет. Теги, которые не требуют закрывающего тега:
, <hr> .

Класс стека (который потребуется для алгоритма проверки парности тегов) требуется реализовать самостоятельно на базе **списка**. Для этого необходимо:

Реализовать **класс** CustomStack, который будет содержать перечисленные ниже методы. Стек должен иметь возможность хранить и работать с типом данных ***char****

Структура класса узла списка:

```
struct ListNode{
ListNode* mNext;
char* mData;
};
```

Объявление класса стека:

class CustomStack {

public:

// методы push, pop, size, empty, top + конструкторы, деструктор

private:

// поля класса, к которым не должно быть доступа извне

protected: // в этом блоке должен быть указатель на голову

ListNode* mHead;

};

Перечень методов класса стека, которые должны быть реализованы:

 void push(const char* tag) - добавляет новый элемент в стек

 void pop() - удаляет из стека последний элемент

 char* top() - доступ к верхнему элементу

 size_t size() - возвращает количество элементов в стеке

 bool empty() - проверяет отсутствие элементов в стеке

Примечание

1. Указатель на голову должен быть protected.
2. Подключать какие-то заголовочные файлы не требуется, все необходимое подключено(<cstring> и <iostream>).
3. Предполагается, что пространство имен std уже доступно.
4. Использование ключевого слова using также не требуется.

5. Структуру ListNode реализовывать самому не надо, она уже реализована.

Выполнение работы.

Реализация класса стека ***CustomStack***:

В блоке *protected* объявлен указатель *struct ListNode* mHead*.

В блоке *public mHead* присваивается значение *nullptr*.

В функции *void push(constchar*tag)* происходит добавление нового элемент в стек. Создается новый элемент списка, поле *mNext* будет указывать на *mHead*, полю *mData* присваивается содержимое переменной *tag*. Новый элемент становится на место головы списка.

В функции *void pop()*происходит удаление из стека последнего элемента. Производится проверка на то, является ли стек пустым. Если стек пустой, то выводится *wrong* и программа завершает работу. Иначе – головой списка становится второй элемент списка, а первый – удаляется.

В функции *char*top()* происходит возвращение данных, хранящихся в поле *mData* головы списка. Если стек не имеет элементов, выводится *wrong* и программа завершает работу.

В функции *size_t size()*происходит счет элементов списка, пока не встретится *nullptr* и из функции возвращается количество элементов стека.

В функции *bool empty()* производится проверка стека на пустоту. Если функция *size()*возвращает 0, то данная функция возвращает *true*. Иначе –*false*.

Функция *main()*:

Объявляется переменная *text*, с помощью функции *new* происходит выделение памяти и с помощью функции *fgets()* происходит считывание текста с консоли. Объявляется *CustomStackstack*.

Далее происходит поиск слов, заключенных в “<>”. Слово запоминается в переменную *buff*. Если первый символ не является “/”, то слово добавляется в стек *stack.push(buff)*. Иначе сравнивается содержимое *buff+1*(т.е.так мы сравниваем слово без “/”) и *stack.top()*. Если слова сходятся, то удаляем элемент стека *stack.pop()*. Иначе выводим *wrong* и завершаем

работу. Когда алгоритм проверит всю строку, выполняем проверку стека на пустоту *stack.empty()*. Если пустой –то выводим *correct*. Иначе -*wrong*.

Разработанный программный код см. в приложении А.

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные
1.	<html><head><title>HTML Document</title></head><body> This text is bold, this is bold and italics </body></html>	*correct*
2.	<head><html><head></html>	*wrong*

Выводы.

В ходе выполнения лабораторной работы были основные принципы работы с динамическими структурами данных на языке C++.

Был реализован стек на базе линейного списка, разработана программа, в которой выполнялась проверка на валидность строки, представляющею собой код html-страницы.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: Yarusova_Tatyana_lb4.c

```
class CustomStack{
        public:
            CustomStack(){
              mHead = nullptr;
            }

            void push(const char* tag){
                struct ListNode* new_el = new struct ListNode;
                new_el -> mData = new char[strlen(tag)];
                strcpy(new_el -> mData,tag);
                new_el -> mNext = mHead;
                mHead = new_el;
            }

            void pop(){
                if(size() == 0){
                    cout << "wrong" << endl;
                    exit(0);
                }

                struct ListNode* tmp = mHead;
                delete mHead -> mData;
                mHead = mHead -> mNext;
                delete tmp;
            }

            char* top(){
                if(size() == 0){
                    cout << "wrong" << endl;
                    exit(0);
                }
                return mHead -> mData;
            }

            size_t size(){
                size_t size = 0;
                struct ListNode* end = mHead;
                while(end != nullptr){
                    size++;
                    end = end -> mNext;
                }
                return size;
            }

            bool empty(){
                if(size() == 0){
                    return true;
                }
                return false;
            }

        protected:
```

```
            struct ListNode* mHead;
};

int main(){
    char* text = new char[3000];
    fgets(text, 3000, stdin);
    CustomStack stack;
    int i = 0;
    int help = 0;

    while(help <= strlen(text)){

        if(text[help] == '<'){
            char* buff = new char[30];
            while(text[help] != '>'){
                i++;
                help++;
            }
            strncpy(buff, text+help - i +1, i-1);
            if(strcmp(buff, "br") && strcmp(buff, "hr")){
              if(buff[0]!='/'){
                stack.push(buff);
              }
              else{
                if(!strcmp(stack.top(), buff+1)) stack.pop();
                else{
                  cout << "wrong" << endl;
                  return 0;
                }
              }
            }
            buff = nullptr;
            i = 0;
        }
        help++;
    }
    if(stack.empty()) cout << "correct" << endl;
    else cout << "wrong" << endl;

    return 0;
}
```