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. Тест.

Студент гр. 3344 _____ Охрименко Д. И.

Преподаватель _____ Глазунов С. А.

Санкт-Петербург
2024

Цель работы

Освоение базовых принципов работы с языком C++, разработка и реализация собственного стека на основе односвязного списка, написание функции для проверки корректности HTML-кода страницы.

Задание

Вариант 3

Расстановка тегов.

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

Примечания:

1. Указатель на голову должен быть protected.
2. Подключать какие-то заголовочные файлы не требуется, всё 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

Стек:

Реализованы основные операции push, pop, size, empty и top, необходимые для работы со стеком. Метод push добавляет элемент в вершину стека. Метод pop удаляет элемент из вершины стека. Этот метод используется для извлечения элементов из стека. Метод size возвращает количество элементов в стеке. Этот метод полезен для определения текущего размера стека. Метод empty проверяет, пуст ли стек. Этот метод помогает определить, есть ли элементы в стеке. Метод top возвращает элемент, находящийся в вершине стека, без его удаления. Этот метод может быть полезен для просмотра верхнего элемента стека без его извлечения. Конструкторы и деконструктор также играют важную роль в работе со стеком. Конструкторы используются для создания нового экземпляра стека. При создании стека по умолчанию он считается пустым. Деструктор вызывается автоматически при уничтожении объекта стека. Он отвечает за освобождение всех ресурсов, занятых объектом. Эти методы и конструкторы обеспечивают гибкость и эффективность работы со стеком, позволяя легко добавлять, удалять и проверять элементы, а также создавать и уничтожать объекты стека.

Main:

В основной функции реализуем базовый алгоритм для решения «скобочной проблемы». В качестве «скобок» будем считать открывающиеся и закрывающиеся тэги, отсеивая прочую информацию из потока ввода. Для этого используем функцию strtok: каждый раз находя открывающуюся скобку «<», извлекаем всю подстроку, а на месте закрывающейся скобки «>» ставим символ ‘\0’ для того, чтобы убрать всё лишнее. В конце остаётся лишь проверить пустой стэк или нет. Если все «скобки» обрели себе пару — стэк останется пустым, и ответ будет «correct», иначе «wrong».

Тестирование

Результаты тестирования представлены в таблице 1.

Таблица 1 – Результаты тестирования

№	Входные данные	Выходные данные	Комментарии
1	<html><head><title>HTML Document</title></head><body> This text is bold, this is bold and italics </body></html>	correct	Верный вывод
2	<tag1><tag2></tag2></tag1><tag1><tag2></tag2></tag1><tag1><tag2></tag1></tag2><tag1><tag2></tag1></tag2>	wrong	Верный вывод

Выводы

Были изучены основы программирования на языке C++, подготовлен самодельный стек на основе однонаправленного списка и разработана функция для определения корректности HTML-кода страницы.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: lb4.cpp

```
class CustomStack {
    public:
        CustomStack() : mHead(nullptr) {};
        ~CustomStack()
        {
            while(!is_empty())
            {
                pop();
            }
        }

        int size()
        {
            int length = 0;
            ListNode* check = mHead;
            if(nullptr == check) return length;
            while(check->mNext != nullptr)
            {
                check = check->mNext;
                length++;
            }
            length++;
            return length;
        }

        void push(const char* x)
        {
         ListNode* node = new ListNode;
         node->mData = new char[strlen(x) + 1];
         strcpy(node->mData, x);
         node->mNext = mHead;
         mHead = node;
        }

        bool is_empty()
        {
            return (nullptr == mHead);
        }

        const char* top()
        {
            return mHead->mData;
        }

        char* pop()
        {
            if(is_empty())
            {
                throw std::length_error("stack is empty");
            }
```

```
        ListNode* delnode = mHead;
        char* x = delnode->mData;
        mHead = delnode->mNext;
        return x;
        }
    protected:
        ListNode* mHead;
    };

int main() {
    CustomStack vector;

    char str[3000];
    fgets(str, 3000, stdin);
    char* token;
    token = strtok(str, "<");
    while (token != NULL) { // <br>, <hr>.
        for(int i = 0; i < strlen(token); ++i){
            if(token[i] == '>'){
                token[i] = '\0';
                break;
            }
        }
        if(strcmp(token, "br") && strcmp(token, "hr") && strcmp(token,
"\n"))
        {
            if(vector.is_empty())
            {
                vector.push(token);
                // std::cout << "  input " << token << std::endl;
                token = strtok(NULL, "<");
                continue;
            }
            char closingword[100] = "/";
            char curtop[100];
            memcpy(curtop, vector.top(), 100);
            strcat(closingword, curtop);
            // std::cout << "close " << closingword << std::endl;

            if(!strcmp(token, closingword))
            {
                //   std::cout   <<   "delete   "   <<   vector.pop()   <<
std::endl;
                vector.pop();
            } else {
                vector.push(token);
                // std::cout << "  input " << token << std::endl;
            }
        }
        token = strtok(NULL, "<");
    }

    if(vector.is_empty())
        std::cout << "correct";
    else
        std::cout << "wrong";
```

```
    return 0;
}
```