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

Студент гр. 3341 _____ Самокрутов А.Р.

Преподаватель _____ Глазунов С.А.

Санкт-Петербург

2024

Цель работы

Целью работы является изучение основных механизмов языка C++ путем разработки структур данных стека и очереди на основе динамической памяти.

Для достижения поставленной цели требуется решить следующие задачи:

1. Ознакомиться со структурами данных стека и очереди, особенностями их реализации;
2. Изучить и использовать базовые механизмы языка C++, необходимые для реализации стека и очереди;
3. Реализовать индивидуальный вариант стека в виде C++ класса, его операции в виде функций этого класса, ввод и вывод данных программы.

Задание

Вариант 6

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>.

Стек (который потребуется для алгоритма проверки парности тегов) требуется реализовать самостоятельно на базе массива. Для этого необходимо:

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

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 массив данных
    char** mData;
};
```

Перечень методов класса стека, которые должны быть реализованы:

- void push(const char* val) - 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
- extend(int n) - расширяет исходный массив на n ячеек

Примечания:

Указатель на массив должен быть protected.

Подключать какие-то заголовочные файлы не требуется, всё необходимое подключено(<cstring> и <iostream>).

Предполагается, что пространство имен std уже доступно.

Использование ключевого слова using также не требуется.

Пример:

Входная строка:

<html><head><title>HTML Document</title></head><body> This text is bold,
this is bold and italics </body></html>

Результат:

correct

Выполнение работы

Создаётся класс *CustomStack* со следующими полями:

protected:

- *char **mData* — указатель на последний элемент стека, т.е. указатель на указатель на *char*.

private:

- *size_t mLen* — количество элементов в стеке.
- *size_t mCapacity* — размер памяти, выделенной в куче под массив данных.

Далее описан конструктор класса *CustomStack()*, инициализирующий поля класса, и деструктор *~CustomStack()*, освобождающий динамически выделенную память.

Публичный метод *void extend(int n)* расширяет выделенную динамически память на *n* ячеек. Выделяется *mCapacity* + *n* ячеек в куче, после чего в них копируется содержимое массива, значение *mCapacity* при этом увеличивается на *n*, а память, выделенная под старые данные, очищается.

Публичный метод *void push(const char *val)* добавляет строку *val* в стек, при необходимости расширяя его при помощи метода *extend()*.

Публичный метод *void pop()* с удаляет последний элемент в стеке, уменьшая массив на одну ячейку. Если стек пуст, то вызывается соответствующая ошибка и работа программы прерывается.

Публичный метод *char *top()* возвращает последний элемент в стеке, если он есть, а иначе вызывает ошибку и завершает программу.

Публичный метод *size_t size()* возвращает значение длины списка.

Публичный метод *bool empty()* возвращает значение *true*, если список пустой, иначе — *false*.

Приватный метод *throwError()* выводит сообщение об ошибке и завершает работу программы с помощью вызова функции *exit(0)*.

Далее описаны программы, необходимые для обработки *HTML*-кода. Функция *std::string getTag(const std::string html, size_t &pos)* принимает на вход

строку *html* с кодом и индекс конечного символа последнего считанного тэга, находит очередной тэг и возвращает его содержимое.

Функция *bool isClosingTag(const std::string tag)* возвращает значение *true*, если поданный ей тэг — закрывающий, иначе — *false*. Аналогично работает функция *bool isOpeningTag(const std::string tag)*. Функция *bool isPairTag(const std::string tag)* проверяет, является ли поданный тэг парным (одиночными считаются только тэги <*hr*> и <*br*>). Функция *bool checkTag(const std::string tag, CustomStack &stk)* проверяет, что поданный ей тэг является закрывающим для тэга, лежащего в верху стека.

Функция *void processTag(const std::string tag, CustomStack stk)* обрабатывает поданный ей тэг следующим образом. Каждый одиночный открывающий тэг она копирует в виде *C-style* строки и добавляет в конец стека, закрывающий — сверяет с тэгом, лежащим в верху стека, и, если они совпадают, удаляет его оттуда, а иначе выводит сообщение о том, что поданный программе код — неправильный.

Функция *void processHtml(const std::string html, CustomStack &stk)* последовательно обрабатывает с помощью функции *processTag()* все тэги в строке *html*, полученные с помощью функции *getTag()*. Если после обработки всех тэгов стек пуст, выводится сообщение об успехе, иначе — о том, что поданный код неправильный.

В функции *int main()* создаётся стек *CustomStack stk* и считывается строка кода *std::string html*, после чего вызывается функция *processHtml*.

Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	<tag></tag>	correct	Проверка корректной работы программы на одном из простейших случаев.
2.	<open></close>	wrong	Проверка корректной работы программы на одном из простейших случае.
3.	<tag> text </tag>	correct	Проверка работы программы в случае, если в коде есть что-то помимо тэгов.
4.	<a><c><d></d></c>	correct	Проверка работы программы при нескольких вложенных тэгах.
5.	<a><c></c><d></d>	correct	Проверка работы программы при нескольких вложенных тэгах.
6.	Pupa and Lupa are best friend no cap no tags	correct	Проверка работы программы при подаче кода без тэгов.

Выводы

Были изучены динамические структуры данных, такие как стек и очередь. Также были изучены различные способы их реализации: на основе массива и на основе связного списка.

Были изучены базовые механизмы языка программирования C++.

Полученные в ходе работы знания были применены на практике. Была написана программа на языке программирования C++, реализующая стек на базе массива и проверяющая с его помощью HTML-код на валидность.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
    #define CHUNK 1
    #define CAPACITY_ERR_MSG "New capacity is smaller than the size of the stack!"
    #define POP_EMPTY_STACK_ERR_MSG "You can't pop an empty stack, silly!!!"
    #define TOP_EMPTY_STACK_ERR_MSG "You can't peek at the top value since the stack is empty, what a fool you are..."
    #define ERROR_MSG "error"
    #define LEFT_FACING_CHEVRON '<'
    #define RIGHT_FACING_CHEVRON '>'
    #define EMPTY_STRING ""
    #define CLOSING_TAG_SLASH '/'
    #define BR_TAG "br"
    #define HR_TAG "hr"
    #define WRONG_RESULT_MSG "wrong"
    #define CORRECT_RESULT_MSG "correct"

    class CustomStack {
    public:
        CustomStack() {

            mCapacity = 1;
            mLen = 0;
            mData = new char *[mCapacity];
        }

        ~CustomStack() {
            delete[] mData;
        }

        void extend(int n) {
            char **tmpData = new char *[mCapacity + n];

            if (mCapacity + n < mLen)
                this->throwError(CAPACITY_ERR_MSG);

            mCapacity += n;
            memcpy(tmpData, mData, sizeof(char *) * mCapacity);
            delete[] mData;
            mData = tmpData;
        }

        void push(const char *val) {
            if (mLen >= mCapacity) {
                extend(CHUNK);
            }

            mData[mLen] = new char[strlen(val) + 1];
            strcpy(mData[mLen], val);
            mLen += 1;
        }
```

```
        void pop() {
            if (this->empty())
                this->throwError(POP_EMPTY_STACK_ERR_MSG);

            mData[--mLen] = nullptr;
            extend(-1);
        }

        char *top() {
            if (this->empty())
                this->throwError(TOP_EMPTY_STACK_ERR_MSG);

            return mData[mLen - 1];
        }

        size_t size() {
            return mLen;
        }

        bool empty() {
            return mLen <= 0;
        }

    private:
        size_t mLen;
        size_t mCapacity;

        void throwError() {
            std::cout << ERROR_MSG << std::endl;
            exit(0);
        }

        void throwError(const char *text) {
            std::cout << ERROR_MSG << ", " << text << std::endl;
            exit(0);
        }

    protected:
        char** mData;
    };

    std::string getTag(const std::string html, size_t &pos) {
        size_t start = html.find(LEFT_FACING_CHEVRON, pos);
        size_t end = html.find(RIGHT_FACING_CHEVRON, pos);
        size_t len;

        std::string rawTag;
          if (end > start && start != std::string::npos && end !=
std::string::npos) {
            len = end - start + 1;
            rawTag = html.substr(start, len);

            pos = end + 1;
        } else {
            pos = std::string::npos;

            return std::string(EMPTY_STRING);
        }
```

```
        std::string tag = rawTag.substr(1 , rawTag.length() - 2);
        return tag;
    }

    bool isClosingTag(const std::string tag) {
        return (tag.at(0) == CLOSING_TAG_SLASH);
    }

    bool isOpeningTag(const std::string tag) {
        return (tag.at(0) != CLOSING_TAG_SLASH);
    }

    bool isPairTag(const std::string tag) {
        return (tag != BR_TAG && tag != HR_TAG);
    }

    bool checkTag(const std::string closingTag, CustomStack &stk) {
             return  (CLOSING_TAG_SLASH  +  std::string(stk.top())  ==
closingTag);
    }

    void processTag(const std::string tag, CustomStack &stk) {
        if (!isPairTag(tag)) {
            return;
        }

        if (isOpeningTag(tag)) {
            char *arr = new char[tag.length() + 1];
            strcpy(arr, tag.c_str());
            stk.push(arr);
        }

        if (isClosingTag(tag)) {
            if (!checkTag(tag, stk)) {
                std::cout << WRONG_RESULT_MSG << std::endl;
                exit(0);
            }

            stk.pop();
        }
    }

    void processHtml(const std::string html, CustomStack &stk) {
        std::string tag;
        size_t buf = 0;

        tag = getTag(html, buf);
        while (tag != EMPTY_STRING) {
            processTag(tag, stk);

            tag = getTag(html, buf);
        }

        if (stk.empty())
            std::cout << CORRECT_RESULT_MSG << std::endl;
        else
            std::cout << WRONG_RESULT_MSG << std::endl;
```

```
}

int main() {
    CustomStack stk;

    std::string html;
    getline(std::cin, html);

    processHtml(html, stk);
    return 0;
}
```