**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. 1304 _____ Крупин Н. С.

Преподаватель _____ Чайка К. В.

Санкт-Петербург

2022

**Цель работы**

Практика создания и использования динамических структур данных (а именно стека) в языке C++.

**Задание**

«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: <br>, <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:

1. Указатель на массив 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».

**Выполнение работы**

**I. Шаги работы программы**

Программа считывает строку в статическую память.

В цикле для каждого найденного с помощью стандартной функции strchr() символа „<“, означающего начало тега, программа анализирует данный тег.

Находит конец тега (символ „>“), запоминает адрес следующего символа и устанавливает символ конца тега в „\0“. Если конец тега не найден, строка автоматически считается некорректной.

По второму символу определяет, открывающий это тег или закрывающий.

Если открывающий тег не соответствует ни одному из набора тегов, не требующих закрывающего, адрес начала содержимого тега помещается в стек.

Если тег закрывающий, его содержимое сравнивается с верхним элементом стека: при совпадении элемент удаляется из стека, в противном случае (или если стек уже пуст) строка считается некорректной.

Если на момент завершения программы строка не признана некорректной, а стек пуст, то для каждого тега нашёлся свой закрывающий и строка считается корректной.

## II. Реализация стека

Для стека создан свой класс CustomStack, реализующий его работу на базе массива. Стек поддерживает саморасширение, но использует буфер для более редкого перевыделения памяти — чтобы после любого перераспределения у стека оставалась незанятая область (буфер).

Класс стека имеет поле protected mData — указатель на область памяти стека, и private поля:

mStackSize — число выделенных для стека ячеек памяти;

mDataSize — число ячеек, занятых элементами;

mReservedSize — память, заранее явным образом выделенная под стек извне, меньше которой стек не будет уменьшать память при работе;

mBufSize — число ячеек, а) добавляемое к памяти при нехватке, б) очищаемое при избытке, в) оставляемое свободными при очистке.

Публичные методы класса push(), pop(), top() и empty() реализуют основной функционал стека. Метод size() является вспомогательно информирующим и возвращает конкретное число элементов в стеке. Метод extend() позволяет менять поле mReservedSize. Также класс имеет private метод addStackSize(), вызываемый методами push(), pop() и extend() для перевыделения памяти, если это требуется.

При создании экземпляра класса без аргументов по умолчанию выбирается стек с нулевым полем mReservedSize и 4 ячейками буфера. Размер буфера и начальной памяти можно указать и вручную: CustomStack(mReservedSize, mBufSize), при этом установка буфера в 0 или указание только первого аргумента делает невозможным саморасширение стека, оставляя эту задачу на пользователя — с помощью метода extend(). При этом если размер зарезервированной памяти можно изменять после

создания объекта класса, размер буфера может быть задан только при инициализации.

В случае применения методов класса могут быть вызваны следующие ошибки (с помощью throw):

1 — Not enough memory (нехватка динамической памяти);

2 — Stack overflow (переполнение стека при нулевом буфере или принудительное уменьшение памяти стека меньше, чем число его элементов);

3 — Empty stack error (извлечение / удаление элемента пустого стека).

Разработанный программный код см. в приложении А.

**Тестирование**

1. Корректная строка

Ввод: <hr>bold b-i-text bold again

Вывод: correct

2. Пересекающиеся теги

Ввод: <hr>bold b-i-text not bold?

Вывод: wrong

3. Закрытие незакрываемого

Ввод: <hr>bold b-i-text bold again</hr>

Вывод: wrong

4. Незакрытие закрываемого

Ввод: bold? b-i-text? bold again?

Вывод: wrong

**Выводы**

Изучены базовые отличия языка C++ от C, понятия динамических структур данных (стек и очередь) и особенности их реализации на базе массива и списка.

Разработана программа, проверяющая строку текста и тегов html на корректность, а именно чтобы каждый открывающий тег (кроме заранее обозначенных) имел свой закрывающий и области их действия не пересекались. Для этого реализован класс саморасширяющегося стека на базе массива.

Программа имеет ограничение на число символов и не распознаёт теги с параметрами, а также требует отсутствия символов „<“ и „>“ среди простого текста. Стек реализован с использованием буфера, что позволяет реже перевыделять память.

# ПРИЛОЖЕНИЕ А
# ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <iostream>
#include <cstring>

using namespace std;

/*
Using:
CustomStack(); - саморасширяющийся стек с буфером 4 без нижнего предела ёмкости.
CustomStack(n); - стек, ограниченный ёмкостью n.
CustomStack(n, k); - если k > 0: саморасширяющийся стек с буфером k и нижним пределом ёмкости n;
                                  если k = 0: стек, ограниченный ёмкостью n.
push(item); - добавление элемента item.
pop(); - удаление верхнего элемента.
size(); - количество элементов в стеке.
empty(); - проверка на пустоту.
top(); - возврат значения верхнего элемента.
extend(n); - увеличение нижнего предела ёмкости на n, для уменьшения может быть отрицательным.

Error codes:
1 - Not enough memory.
2 - Stack overflow.
3 - Empty stack error.
*/
class CustomStack{
public:
	CustomStack(size_t init_size, size_t buf_size):
		mStackSize(init_size), mReservedSize(init_size), mBufSize(buf_size){
		mData = new char*[init_size];
		if (init_size && !mData) throw 1; // Not enough memory.
	}
	CustomStack(size_t init_size): CustomStack(init_size, 0) {}
	CustomStack(): CustomStack(0, 4) {}

	void push(const char* val){
		if (mBufSize && mDataSize >= mStackSize)
			addStackSize(mBufSize);
		if (mDataSize >= mStackSize) throw 2; // Stack overflow.
		mData[mDataSize++] = (char*)val;
	}

	void pop(){
		if (empty()) throw 3; // Empty stack error.
		mDataSize--;
		if (mBufSize && mDataSize+2*mBufSize < mStackSize)
			if (mStackSize >= mReservedSize + mBufSize)
				addStackSize((int)mReservedSize-(int)mStackSize);
```

```
                        else addStackSize((int)mBufSize*(-1));
                }

                size_t size(){
                        return mDataSize;
                }

                bool empty(){
                        return !mDataSize;
                }

                char* top(){
                        if (empty()) throw 3; // Empty stack error.
                        return mData[mDataSize-1];
                }

                void extend(int n){
                        if ((int)mStackSize+n < (int)mDataSize) throw 2; // Stack
overflow.
                        mReservedSize += n;
                        if (mStackSize < mReservedSize)
                                addStackSize((int)mReservedSize-(int)mStackSize);
                }

                ~CustomStack(){
                        delete[] mData;
                }

        private:
                size_t mStackSize, mDataSize = 0;
                size_t mReservedSize, mBufSize;

                void addStackSize(int n){
                        char** tmpData;
                        mStackSize += n;
                        tmpData = new char*[mStackSize];
                        if (mStackSize && !tmpData) throw 1; // Not enough memory.
                        memcpy(tmpData, mData, mDataSize*sizeof(char*));
                        delete[] mData;
                        mData = tmpData;
                        tmpData = NULL;
                }

        protected:
                char** mData;
        };

        int main() {
                CustomStack stack = CustomStack();
                char str[3001], *s = str, *f;
                int errStatus = 0;

                cin.getline(str, 3001);

                while(s = strchr(s, '<')){
                        if (!(f = strchr(s, '>'))){
                                errStatus = 1; break;
                        }
                        *f = '\0';
                        if (*(s+1) == '/')
                                if (stack.empty() || strcmp(stack.top(), s+2)){
```

```
                        errStatus = 1; break;
                    } else stack.pop();
                else if (strcmp("br", s+1) && strcmp("hr", s+1))
                    stack.push(s+1);
                s = f+1;
        }

        if (!errStatus && stack.empty()) cout << "correct";
        else cout << "wrong";
        return 0;
}
```