МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ

ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №2 по

дисциплине «Программирование»

ТЕМА: Динамические структуры данных

Студент гр. 7303 Юсковец А.В.

Преподаватель Берленко Т.А.

Санкт-Петербург

2017

Оглавление

Цель работы....3
Стэк....4
· Директивы препроцессора....4
· Определение типов....4
· Инициализация стека....5
· push....5
· pop....5
· top....6
· isEmpty....6
· shrink_to_fit....6
Ход работы....7
· Исходный код....7
· Объявление и инициализации переменных....8
· Алгоритм....9
Вывод....10

Цель работы

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**.

Стэк

- **Директивы препроцессора**

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define INIT_SIZE 10
#define FACTOR 1.5
#define TAG_SIZE 10
```

Подключаются соответствующие библиотеки для использования функций: работы со строками, динамической памятью и стандартным вводом-выводом. Определяются макросы

INIT_SIZE — начальный размер стека,

FACTOR — множитель реаллокации,

HTML_SIZE — размер html-документа,

TAG_SIZE — начальный размер одного тега.

- **Определение типов**

```
typedef char* type;

typedef struct Stack {
    type* data;
    int size;
    int capacity;
} Stack;
```

type – тип содержимого стека.

Stack – тип стека.

Структура Stack содержит поля:

data – указатель на елемент в ячейке стека,

size – размер стека,

capacity – фактический размер стека в куче.

• **Инициализация стека**

```
Stack stack_init() {
    Stack stack;
    stack.data = (type*)malloc(INIT_SIZE*sizeof(type));
    stack.size = 0;
    stack.capacity = INIT_SIZE;

    return stack;
}
```

Выделяется память под стек и устанавливается его размер и вместимость.

• **push**

```
void push(Stack* stack, type el) {
    if (stack->size == stack->mem_size) {
        if (stack->data = realloc(stack->data, stack->capacity * FACTOR){
            stack->mem_size *= FACTOR;
            stack->data[stack->size++] = el;
        } else {
            fprintf(stderr, "Out of memory");
        }
    } else {
        stack->data[stack->size++] = el;
    }
}
```

Проверяется есть ли место для вставки очередного элемента (в обратном случае память перевыделяется) в ячейку записывается переданное значение и размер стека инкрементируется.

• **pop**

```
type pop(Stack* stack) {
    return stack->data[stack->size-- - 1];
}
```

Возвращается последний элемент стека и размер декрементируется.

· top

```
type top(Stack* stack) {
    return stack->data[stack->size - 1];
}
```

Возвращается последний элемент стека.

· isEmpty

```
int isEmpty(Stack* stack) {
    return !stack->size;
}
```

В случае, если размер стека ненулевой, возвращается 1, в обратном 0.

· shrink_to_fit

```
void shrink_to_fit(Stack* stack) {
    stack->data = realloc(stack->data, stack->size);
}
```

Функция освобождает всю, неиспользуемую стеком память.

Ход работы

· Исходный код

```
int main() {
    Stack stack = stack_init();

    char ch;
    char* tag = (char*)malloc(TAG_SIZE);
    int tag_capacity = TAG_SIZE;
    int tag_size = 0;
    while ((ch = getchar()) != '\n') {
        if (ch == '<') {
            while ((ch = getchar()) != '>' && ch != ' ') {
                if (tag_size == tag_capacity - 1) {
                    tag = realloc(tag, tag_capacity*FACTOR);
                    tag_capacity *= FACTOR;
                }

                tag[tag_size++] = ch;
            }

            tag[tag_size++] = '\0';

            if (tag[0] == '/') {
                if (isEmpty(&stack)) {
                    printf("wrong\n");
                    return 0;
                } else if (strcmp(pop(&stack), &tag[1]) == 0) {}
                else {
                    printf("wrong\n");
                    return 0;
                }
            } else {
                if (strcmp(tag, "br") != 0 && strcmp(tag, "hr") != 0) {
                    char* tag_copy = (char*)malloc(tag_size);
                    for (int i = 0; i != tag_size; ++i) {
                        tag_copy[i] = tag[i];
                    }

                    push(&stack, tag_copy);
                }
            }
        }

        tag_size = 0;
    }

    printf("correct\n");
    return 0;
}
```

· Объявление и инициализации переменных

```
Stack stack = stack_init();
char ch;
char* tag = (char*)malloc(TAG_SIZE);
int tag_capacity = TAG_SIZE;
int tag_size = 0;
```

stack — стек,
ch — переменная, с помощью которой будет производится считываение,
tag — строка с названием тега.

Так как максимально возможный размер тега неизвестен переменная является динамической строкой, в связи с этим заводится еще две переменные — размер тега, и количество памяти, выделенной под него.

· **Алгоритм**

```
    while ((ch = getchar()) != '\n') {
        if (ch == '<') {
            while ((ch = getchar()) != '>' && ch != ' ') {
                if (tag_size == tag_capacity - 1) {
                    tag = realloc(tag, tag_capacity*FACTOR);
                    tag_capacity *= FACTOR;
                }

                tag[tag_size++] = ch;
            }

            tag[tag_size++] = '\0';

            if (tag[0] == '/') {
                if (isEmpty(&stack)) {
                    printf("wrong\n");
                    return 0;
                } else if (strcmp(pop(&stack), &tag[1]) == 0) {}
                else {
                    printf("wrong\n");
                    return 0;
                }
            } else {
                if (strcmp(tag, "br") != 0 && strcmp(tag, "hr") != 0) {
                    char* tag_copy = (char*)malloc(tag_size);
                    for (int i = 0; i != tag_size; ++i) {
                        tag_copy[i] = tag[i];
                    }

                    push(&stack, tag_copy);
                }
            }
        }

        tag_size = 0;
    }

    printf("correct\n");
    return 0;
```

Считывание производится до тех пор пока не будет достигнут конец файла. Если встречается символ, открывающий тег то все его содержимое до закрывающей скобки или до пробела, записывается в tag. Далее в случае, если был передан закрывающий тег и до этого не было ни одного открывающего, программа завершается с «wrong». Если стек не пуст сравнивается последний тэг в стеке и закрывающий тэг, если они совпадают алгоритм продолжает работу. Если был передан открывающий тэг то он записывается в стек (за исключением "br" и "hr"). В конце каждой итерации внешнего цикла tag_size зануляется, чтобы можно было потом перезаписать содержимое tag.

Если на момент достижения конца документа не было завершения программы, то он выводит «correct».

Вывод

Была написана программа проверяющая валидность html-документа. Для реализации алгоритма проверки был написан стек на основе динамического массива. При компиляции и выполнении программы не возникает ошибок и предупреждений.