

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

ОТЧЕТ
по лабораторной работе №3
по дисциплине «Программирование»
Тема: Обход файловой системы

Студент гр. 3342

Русанов А.В.

Преподаватель

Глазунов С.А.

Санкт-Петербург

2024

Цель работы

Ознакомление с рекурсией, а также с её применением для обхода файловой системы с помощью языка программирования С.

Задание

Вариант 3.

Дана некоторая корневая директория, в которой может находиться некоторое количество папок, в том числе вложенных. В этих папках хранятся некоторые текстовые файлы, имеющие имя вида <filename>.txt

В каждом текстовом файле хранится одна строка, начинающаяся с числа вида:

<число><пробел><латинские буквы, цифры, знаки препинания> ("124 string example!")

Требуется написать программу, которая, будучи запущенной в корневой директории, выведет строки из файлов всех поддиректорий в порядке возрастания числа, с которого строки начинаются

Выполнение работы

Функция `comparator` - функция сравнения двух строк, используется для сортировки массива строк в порядке возрастания чисел, содержащихся в строках.

Функция `memory_allocation_check` - функция, которая проверяет, выделена ли память успешно. Если указатель равен `NULL`, то выводится сообщение об ошибке и программа завершается.

Функция `file_is_valid` - функция проверяет, является ли файл допустимым для обработки. Возвращает 1, если файл имеет расширение `".txt"` и не имеет названия `"result.txt"`, иначе возвращает 0.

Функция `dir_is_valid` - функция проверяет, является ли директория допустимой для обработки. Возвращает 1, если директория не является текущей директорией (`"."`) или родительской директорией (`".."`), иначе возвращает 0.

Функция `get_full_path` - функция для получения полного пути к файлу или директории, объединяя две строки пути с помощью символа `"/"`.

Функция `read_file_data` - функция для чтения данных из файла. Функция открывает файл для чтения, считывает его содержимое и сохраняет каждую строку в массиве строк `lines`, увеличивая его размер при необходимости.

Функция `read_lines` - рекурсивная функция для чтения содержимого директории. Она открывает директорию, считывает содержимое и выполняет определенные действия для каждого файла и поддиректории внутри нее.

Функция `print_result` - функция для печати результатов в файл. Она принимает открытый файловый указатель `file`, массив строк `lines` и количество строк `count_lines`, и записывает содержимое массива строк в файл.

Функция `free_memory` - функция для освобождения памяти. Она освобождает память, выделенную для массива строк `lines` и для строки с полным путем `new_dir`.

Функция `main` - основная функция программы. Сначала она получает текущую директорию, затем создает переменные для хранения количества строк `count_lines` и массива строк `lines`. Затем она вызывает функцию `read_lines` для чтения данных из файлов в текущей директории (и поддиректориях), сортирует

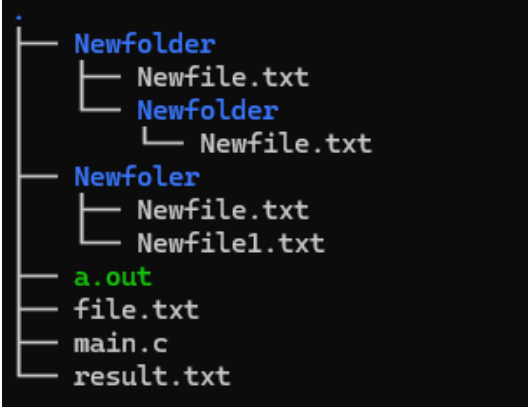
массив строк с помощью функции `qsort`, создает новый файл с именем "result.txt" и выводит результаты в него с помощью функции `print_result`. После этого память освобождается с помощью функции `free_memory`.

Разработанный программный код см. в приложении А.

Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.		1 Small text 2 Simple text 3 Wow? Text? 4 Where am I? 5 So much files!	Верное содержимое файла

Выводы

Было проведено ознакомление с рекурсией. Разработана программа на языке программирования С с использованием библиотеки `dirent.h` для реализации обхода файловой системы.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <dirent.h>
#include <regex.h>
#include <unistd.h>
#define MAX_PATH_SIZE 512

int comparator(const void* a, const void* b)
{
    char** line_a = (char**)a;
    char** line_b = (char**)b;
    long number_a = atol(*line_a);
    long number_b = atol(*line_b);
    if (number_a < number_b)
        return -1;
    if (number_a > number_b)
        return 1;
    return 0;
}

void memory_allocation_check(void* pointer)
{
    if(pointer == NULL)
    {
        printf("Memory allocation error!\n");
        exit(1);
    }
}

int file_is_valid(char *filename)
{
    return (strstr(filename, ".txt") && !strstr(filename,
"result.txt"));
}

int dir_is_valid(char *dir_name)
{
    return (strcmp(dir_name, ".") != 0 && strcmp(dir_name, "..") !=
0);
}

char *get_full_path(const char *path1, const char *path2)
{
    int res_path_len = strlen(path1) + strlen(path2) + 2;
    char *res_path = malloc(res_path_len * sizeof(char));
    memory_allocation_check(res_path);
    sprintf(res_path, "%s/%s", path1, path2);
    return res_path;
}
```



```

void read_file_data(char *filepath, char ***lines, int *count_lines)
{
    FILE *f = fopen(filepath, "r");
    if (f)
    {
        char *line = calloc(100, sizeof(char));
        memory_allocation_check(line);
        int len_line = 0;
        char sym;
        while ((sym = fgetc(f)) != EOF)
        {
            if (sym == '\n')
            {
                continue;
            }
            line[len_line++] = sym;
            if(len_line >= sizeof(line) / sizeof(char))
            {
                line = realloc(line, sizeof(char) * (len_line + 20));
                memory_allocation_check(line);
            }
        }
        line[len_line] = '\0';
        (*lines)[(*count_lines)++] = line;
        *lines = (char **)realloc(*lines, sizeof(char *) *
(*count_lines + 1));
        memory_allocation_check(lines);
    }
    fclose(f);
}

void read_lines(const char *dir_name, char ***lines, int *count_lines)
{
    DIR *dir = opendir(dir_name);
    if (dir)
    {
        struct dirent *de = readdir(dir);
        while (de)
        {
            if (de->d_type == DT_REG && file_is_valid(de->d_name))
            {
                char *new_dir = get_full_path(dir_name, de->d_name);
                read_file_data(new_dir, lines, count_lines);
                free(new_dir);
            }
            else if (de->d_type == DT_DIR && dir_is_valid(de->d_name))
            {
                char *new_dir = get_full_path(dir_name, de->d_name);
                read_lines(new_dir, lines, count_lines);
                free(new_dir);
            }
            de = readdir(dir);
        }
        closedir(dir);
    }
    else

```

```

        printf("Failed to open %s directory\n", dir_name);
    }

void print_result(FILE* file, char** lines, int* count_lines)
{
    fprintf(file, "%s", lines[0]);
    for (int i = 1; i < (*count_lines); i++)
    {
        fprintf(file, "\n%s", lines[i]);
    }
}

void free_memory(char* new_dir, char** lines, int* count_lines)
{
    free(new_dir);
    for (int i = 0; i < (*count_lines); i++)
    {
        free(lines[i]);
    }
    free(lines);
}

int main()
{
    char current_dir[MAX_PATH_SIZE];
    if (!getcwd(current_dir, sizeof(current_dir)))
    {
        perror("getcwd");
        exit(1);
    }

    int tmp = 0;
    int *count_lines = &tmp;
    char **lines = (char **)malloc(sizeof(char *));
    memory_allocation_check(lines);

    read_lines(current_dir, &lines, count_lines);

    qsort(lines, (*count_lines), sizeof(char*), comparator);

    char* new_dir = get_full_path(current_dir, "/result.txt");
    FILE *file = fopen(new_dir, "w+");
    if (!file){
        printf("Error:  cannot open result.txt\n");
        exit(1);
    }

    print_result(file, lines, count_lines);
    free_memory(new_dir, lines, count_lines);
    fclose(file);
    return 0;
}

```