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: Строки. Рекурсия, циклы, обход дерева

Студент гр. 3341 Мальцев К.Л.

Преподаватель Глазунов С.А.

Санкт-Петербург
2024

Цель работы

Цель работы заключается в разработке программы на языке программирования, которая осуществляет рекурсивный обход иерархии папок и файлов в заданной структуре, анализирует содержимое текстовых файлов, выполняет математические операции в соответствии с правилами задания и выводит на экран итоговый результат вычислений, основанный на содержимом файлов и вложенных папок.

Задание

Вариант 2

Задана иерархия папок и файлов по следующим правилам:

название папок может быть только "add" или "mul"

В папках могут находиться другие вложенные папки и/или текстовые файлы

Текстовые файлы имеют произвольное имя с расширением .txt

Содержимое текстовых файлов представляет собой строку, в которой через пробел записано некоторое количество целых чисел

Требуется написать программу, которая, запускается в корневой директории, содержащей одну папку с именем "add" или "mul" и вычисляет и выводит на экран результат выражения состоящего из чисел в поддиректориях по следующим правилам:

Если в папке находится один или несколько текстовых файлов, то математическая операция определяемая названием папки (add = сложение, mul = умножение) применяется ко всем числам всех файлов в этой папке

Если в папке находится еще одна или несколько папок, то сначала вычисляются значения выражений, определяемые ими, а после используются уже эти значения

Ваше решение должно находиться в директории /home/box, файл с решением должен называться solution.c. Результат работы программы должен быть записан в файл result.txt. Ваша программа должна обрабатывать директорию, которая называется tmp.

Основные теоретические положения

Основные теоретические положения для работы с файловой иерархией в C и использования рекурсии:

1. Работа с файловой иерархией: файловая иерархия представляет собой структуру, в которой файлы и директории организованы в виде дерева. Взаимодействие с файловой иерархией включает операции чтения, записи, создания, удаления файлов и директорий.

2. Работа с файлами и директориями: для работы с файловой иерархией в языке программирования C используются функции стандартной библиотеки языка, такие как fopen(), fclose(), fread(), fwrite(), opendir(), readdir(), closedir() и другие. Эти функции позволяют осуществлять доступ к файлам и директориям, выполнять чтение и запись данных.

3. Рекурсия: рекурсия в программировании — это прием, при котором функция вызывает саму себя. При работе с файловой иерархией рекурсия позволяет обходить все уровни директорий и файлов вложенных структур. Это особенно полезно при неопределенном количестве уровней вложенности или при необходимости выполнить однотипную операцию на каждом уровне.

4. Рекурсивное обход директорий: в рамках обработки файловой иерархии рекурсия часто используется для обхода всех элементов директории, включая поддиректории. Это позволяет пройти по всем уровням вложенности и обработать каждый файл или директорию в структуре.

5. Базовый и рекурсивный случаи: в рекурсивной функции для обхода директорий важно определить базовый случай, при котором рекурсия завершится, и рекурсивный случай, в котором функция вызывает саму себя для обработки следующего уровня директории.

6. Управление памятью: при работе с файловой иерархией и использовании рекурсии важно правильно управлять памятью. Необходимо освобождать ресурсы, выделенные для открытия файлов и директорий, чтобы избежать утечек памяти и повысить производительность программы.

Выполнение работы

1. В начале программы объявлены все необходимые функции и структуры данных.

2. В функции main() вызывается функция getResult(), которая получает результат обхода директорий "tmp/add" и "tmp/mul". Этот результат сохраняется в переменной result.

3. Результат выводится в файл result.txt с помощью функции outputDataToFile().

4. В функции getResult() поочередно проверяется наличие директорий "tmp/add" и "tmp/mul". Если они существуют, вызывается функция traversal(), которая рекурсивно обходит поддиректории и файлы внутри них.

5. В функции traversal() проверяется тип элемента (файл или директория) и соответствующим образом обрабатывается. Для директорий "add" и "mul" выполняются сложение и умножение результатов соответственно. Для файлов выполняются сложение или умножение чисел из файла в зависимости от операции.

6. Файлы с числовыми данными считываются с помощью функции getDataFromFile(), данные из них обрабатываются с помощью функции processData(), которая разделяет числа и сохраняет их в вектор.

7. Вектор чисел передается в функции sum() или mul() для выполнения соответствующей операции.

8. Рекурсивный обход директорий и файлов выполняется до тех пор, пока вся структура "tmp" не будет обработана.

9. Результат обработки итоговых данных возвращается из функции getResult() в функцию main() и выводится в файл result.txt.

Структуры данных представлены в коде для работы с вектором (Vector) и строкой (String).

1. Структура Vector:

- ll* array - указатель на массив элементов типа ll (вероятно, это определено в другом месте в коде);

- int cap - текущая емкость массива (capacity), т.е. сколько элементов может содержать массив без перераспределения памяти;

- int size - текущий размер массива, т.е. сколько элементов фактически содержит массив.

Функции для работы с вектором Vector:

- Vector* initVector() - функция инициализации вектора, выделяет память под структуру Vector.

- ll atV(Vector* v, int idx) - функция получения значения элемента по индексу idx в векторе v.

- void pushBackV(Vector* v, ll el) - функция добавления элемента el в конец вектора v. При необходимости увеличивает емкость и перераспределяет память.

- void printVector(Vector* v) - функция печати содержимого вектора.

2. Структура String:

- char* array - указатель на массив символов;

- int cap - текущая емкость массива символов;

- int size - текущий размер строки (количество символов).

Функции для работы со строкой String:

- String* initString() - функция инициализации строки, выделяет память под структуру String.

- char atS(String* s, int idx) - функция получения символа по индексу idx в строке s.

- void pushBackS(String* s, char el) - функция добавления символа el в конец строки s. При необходимости увеличивает емкость и перераспределяет память.

- void printString(String* s) - функция печати содержимого строки.

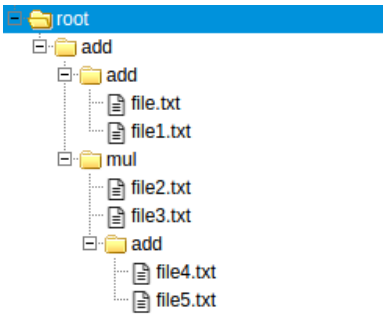
Обе структуры позволяют динамически увеличивать емкость и добавлять элементы (элементы вектора и символы в строку) при необходимости. Они предоставляют удобные методы для доступа к элементам и для печати содержимого.

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.	file.txt: 1 file1.txt: 1 file2.txt: 2 2 file3.txt: 7 file4.txt: 1 2 3 file5.txt: 3 -1		Тест с e.moevm

Выводы

В ходе выполнения данной работы были приобретены навыки эффективного использования рекурсивных методов для обхода сложных структур данных, а также работы с файловой системой, анализа содержимого текстовых файлов и выполнения математических операций в соответствии с заданными правилами. Разработка программы, способной автоматически обрабатывать информацию из различных файлов и директорий, позволила улучшить навыки программирования и решения сложных задач.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: solution.c

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

#define NULL_CH              '\0'
#define ENDL                 "\n"
#define NULL_CH_BUFFER_SIZE  1
#define MATCH_GROUPS_SIZE    1

const char* kOutputFileName = "result.txt";
const char* kPatternAdd = "^[a-zA-Z0-9/]*add$";
const char* kPatternMul = "^[a-zA-Z0-9/]*mul$";


void throwIndexError();

typedef struct VectorNum
{
    /** Vector implementation
     * A standard container which offers access to
     * individual elements in any order.
    */
    long long* array;
    int capacity;
    int size;
} VectorNum;

/* Vector methods */
VectorNum* vectorNumInit();
void vectorNumDelete(VectorNum* vector_num);
long long vectorNumAt(VectorNum* vector_num, int index);
void    vectorNumPushBack(VectorNum*    vector_num,    long    long
new_element);
void printVector(VectorNum* vector_num);

typedef struct String
{
    /** String implementation
     * Managing sequences of characters.
    */
    char* array;
    int capacity;
    int size;
} String;

/* String methods */
String* stringInit();
void stringDelete(String* string);
```

```
    char stringAt(String* string, int index);
    void stringPushBack(String* string, char new_element);
    void stringPrint(String* string);

    long long getResult();
    void outputDataToFile(const char* file_path, long long result);
    long long fileTreeTraversal(char* dir_path, int type, char
operation,
                                regex_t*        regex_compiled_add_ptr,
regex_t* regex_compiled_mul_ptr);
    int dirPathValidation(char* dir_path, regex_t* regex_compiled);
    int isConsidered(char* dir_name);
    char* getSubPath(char* dir_path, char* sub_file_name);
    String* getDataFromFile(char* file_path);
    VectorNum* processData(String* string);
    long long vectorNumSum(VectorNum* vector_num);
    long long vectorNumProduct(VectorNum* vector_num);


    int main()
    {
        long long result = getResult();
        outputDataToFile(kOutputFileName, result);
        return 0;
    }

    void throwIndexError()
    {
        printf("Index out of range%s", ENDL);
    }

    long long getResult()
    {

        regex_t regex_compiled_add;
         regcomp(&regex_compiled_add, kPatternAdd, REG_EXTENDED);

        regex_t regex_compiled_mul;
         regcomp(&regex_compiled_mul, kPatternMul, REG_EXTENDED);

        DIR* dir;
        if ( dir = opendir("tmp/add") ) {
            return     fileTreeTraversal("tmp/add",     DT_DIR,     '+',
&regex_compiled_add, &regex_compiled_mul);
        }

        if ( dir = opendir("tmp/mul") ) {
            return     fileTreeTraversal("tmp/mul",     DT_DIR,     '-',
&regex_compiled_add, &regex_compiled_mul);
        }

        return 0;
    }

    void outputDataToFile(const char* file_path, long long result)
    {
        FILE* fout = fopen(file_path, "w");
        fprintf(fout, "%lld", result);
```

```
        fclose(fout);
    }

    long long fileTreeTraversal(char* dir_path, int type, char
operation,
                                regex_t*        regex_compiled_add_ptr,
regex_t* regex_compiled_mul_ptr)
    {
        if (type == DT_DIR) {
            DIR* dir = opendir(dir_path);

            if (dirPathValidation(dir_path, regex_compiled_add_ptr)) {
                long long sum = 0;
                struct dirent* de = readdir(dir);
                while (de) {
                    char* sub_path = getSubPath(dir_path, de->d_name);
                    if (isConsidered(de->d_name)) {
                        sum += fileTreeTraversal(sub_path, de->d_type,
'+', regex_compiled_add_ptr, regex_compiled_mul_ptr);
                    }
                    de = readdir(dir);
                }
                closedir(dir);
                return sum;
            }

            if (dirPathValidation(dir_path, regex_compiled_mul_ptr)) {
                long long product = 1;
                struct dirent* de = readdir(dir);
                while (de) {
                    char* sub_path = getSubPath(dir_path, de->d_name);
                    if (isConsidered(de->d_name)) {
                        product *= fileTreeTraversal(sub_path, de-
>d_type, '*', regex_compiled_add_ptr, regex_compiled_mul_ptr);
                    }
                    de = readdir(dir);
                }
                closedir(dir);
                return product;
            }

        }
        if (type == DT_REG) {
            VectorNum* numbers = processData(getDataFromFile(dir_path));
            if (operation == '+') {
                return vectorNumSum(numbers);
            }
            if (operation == '*') {
                return vectorNumProduct(numbers);
            }
        }
        return 0;
    }

    int dirPathValidation(char* dir_path, regex_t* regex_compiled_ptr)
    {
         regmatch_t match_groups[MATCH_GROUPS_SIZE];
```

```
          return          (regexec(regex_compiled_ptr,          dir_path,
MATCH_GROUPS_SIZE, match_groups, 0) == 0);
      }

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

      char* getSubPath(char* dir_path, char* sub_file_name)
      {
          char*    sub_path    =    (char*)    calloc(strlen(dir_path)    +
strlen(sub_file_name) + 2 * NULL_CH_BUFFER_SIZE, sizeof(char));
          strcpy(sub_path, dir_path);
          sub_path[strlen(dir_path)] = '/';
          strcat(sub_path, sub_file_name);
          return sub_path;
      }

      String* getDataFromFile(char* file_path)
      {
          FILE* fin = fopen(file_path, "r");
          String* line = stringInit();

          signed char ch = 0;
          while ((ch = fgetc(fin)) != EOF) {
              stringPushBack(line, ch);
          }

          if (strchr(line->array, '\n')) {
              *strchr(line->array, '\n') = NULL_CH;
          }

          fclose(fin);

          return line;
      }

      VectorNum* processData(String* string)
      {
          VectorNum* processed_data = vectorNumInit();
          char* sep = " ";
          char* token;

          token = strtok(string->array, sep);
          while (token) {
              vectorNumPushBack(processed_data, atoll(token));
              token = strtok(NULL, sep);
          }

          stringDelete(string);
          return processed_data;
      }

      long long vectorNumSum(VectorNum* vector_num)
      {
          long long sum = 0;
```

```
        for (int i = 0; i < vector_num->size; i++) {
            sum += vectorNumAt(vector_num, i);
        }
        vectorNumDelete(vector_num);
        return sum;
    }

    long long vectorNumProduct(VectorNum* vector_num)
    {
        if (vector_num->size == 0) {
            return 0;
        }
        long long product = 1;
        for (int i = 0; i < vector_num->size; i++) {
            product *= vectorNumAt(vector_num, i);
        }
        vectorNumDelete(vector_num);
        return product;
    }



    VectorNum* vectorNumInit()
    {
        return (VectorNum*) calloc(1, sizeof(VectorNum));
    }

    void vectorNumDelete(VectorNum* vector_num)
    {
        free(vector_num->array);
        free(vector_num);
    }

    long long vectorNumAt(VectorNum* vector_num, int index)
    {
        if (index >= 0 && index < vector_num->size) {
            return vector_num->array[index];
        }
        throwIndexError();
        return 0;
    }

    void vectorNumPushBack(VectorNum* vector_num, long long new_element)
    {
        if (vector_num->size + 1 > vector_num->capacity) {
            if (vector_num->size == 0) {
                vector_num->capacity = 2;
            } else {
                vector_num->capacity    =    vector_num->capacity    *
vector_num->capacity;
            }
            vector_num->array  =  realloc(vector_num->array,  vector_num-
>capacity * sizeof(long long));
        }
        vector_num->array[vector_num->size++] = new_element;
    }
```

```
    void printVector(VectorNum* vector_num)
    {
        for (int i = 0; i < vector_num->size; i++) {
            printf("%lld ", vectorNumAt(vector_num, i));
        }
        printf(ENDL);
    }



    String* stringInit()
    {
        return (String*) calloc(1, sizeof(String));
    }

    void stringDelete(String* string)
    {
        free(string->array);
        free(string);
    }

    char stringAt(String* string, int index)
    {
        if (index >= 0 && index < string->size) {
            return string->array[index];
        }
        throwIndexError();
        return -1;
    }

    void stringPushBack(String* string, char new_element)
    {
        if (string->size + 1 + NULL_CH_BUFFER_SIZE > string->capacity)
{
            if (string->size == 0) {
                string->capacity = 2;
            } else {
                string->capacity = string->capacity * string->capacity;
            }
            string->array  =  realloc(string->array,  string->capacity  *
sizeof(char));
        }
        string->array[string->size++] = new_element;
        string->array[string->size] = '\0';
    }

    void stringPrint(String* string)
    {
        printf("%s", string->array);
    }
```