ECE26400 Programming Exercise #6

This exercise is related to PE05 and PA03. We continue to build on what you did in PE05. In particular, you will now have to determine whether a file contains a valid tiling solution. **You will be using a two-dimensional array to help you with that task.**

In addition to those outlined in PE05, the main learning goals are:

1. How to use `malloc` or `calloc` to allocate block(s) of memory for a two-dimensional array and use `free` to return the memory to the system.
2. How to iterate through a two-dimensional array.
3. How to output the content in a structure to a file.

1 Getting started

You should unzip on eceprog.ecn.purdue.edu the zip file `pe06_files.zip` using the following command:

```
unzip pe06_files.zip
```

The zip file `pe06_files.zip` contains a folder, named `PE06`, and one file within the folder and a subfolder containing some sample files:

1. `answer06.h`: This is a "header" file and it declares the functions you will be writing for this exercise.
2. `examples`: This is a subfolder with valid and invalid sample files: `0-16-4-7`, `2-16-4-7`, `3-16-4-7`, `4-16-4-7`, and `6-16-4-7`.

In this exercise, you have to create a file called `answer06.c` to define all functions declared in `answer06.h` and a file called `pe06.c` for the `main` function.

To get started, read this document in its entirety. First, you should be familiar with the tiling problem that we want you to solve in PA03 (see the description for PE05). You should also be familiar with the functions that you have written in PE05. In this exercise, you are going to build on what you did in PE05 and write four functions in `answer06.c` and the `main` function in `pe06.c`.

2 Functions you have to define

In PE05, the main purpose is to determine that a file given to you is of the correct format, and could potentially be a solution. In PE06, you determine whether the given file is indeed a solution.

2.1 Functions to be written in `answer06.c`

You have to define these functions in `answer06.c`, which you have to create. (The declarations of the prototypes are in `answer06.h`.)

The first two functions deal with memory allocation and returning the memory back to the system when your program no longer requires the memory.

```
char **allocate_2d_array(int n);

void free_2d_array(char **array, int n);
```

For the purpose of this exercise, you will create a two-dimensional array that is similar to the one shown in Figure 1.

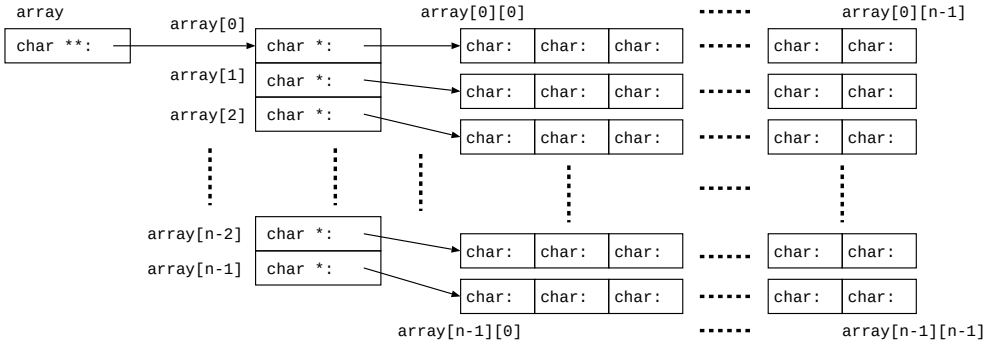

Figure 1: The expected structure of a two-dimensional array.

If the following statement is in the program:

```
char **array = allocate_2d_array(n);
```

where `n` has been declared and initialized with an appropriate floor length, you should allocate adequate space for a 2-dimensional array, and store appropriate addresses in `array`, and `array[0]`, `array[1]`, ... `array[n-1]`, as shown in the first figure. There are many ways to allocate (and correspondingly free) memory for a 2-dimensional array. If you find it useful to create a 2-dimensional array that uses more memory than what is shown in Figure 1, you are allowed to do that. Regardless of how you create the 2-dimensional array, **if** `array` **contains a valid address, other functions (written by you or the instructor) must be able to access** `array[i][j]` **successfully for valid values of** `i` **and** `j`**, i.e.** $0 \leq \mathtt{i}, \mathtt{j} < n$**.**

It is important that if the function could not obtain the necessary memory to set up such a 2-dimensional array, it returns `NULL`, which is then stored in `array`. Prior to returning `NULL`, memory that has been allocated in the function so far must be freed; otherwise, you will have memory leaks.

The function `free_2d_array` must free all memory successfully allocated through `allocate_2d_array`. The function may assume that the first parameter `array` is not NULL. In other words, the caller function calls `free_2d_array` only if there is a valid 2-dimensional array to be freed. The function may also assume that the caller function will pass in the correct second parameter.

(Although the instructor will also write a `free_2d_array` function, the instructor will not use that function to free the memory allocated through the `allocate_2d_array` function written by a student. In other words, the `allocate_2d_array` and `free_2d_array` functions always go hand in hand. Keep this in mind when you write programs that require you to allocate and free memory. You typically want to write such functions as a pair.)

The `answer06.h` also define the following structure that would be used in the other two functions:

```
typedef struct _tiling_solution {
```

```
   int floor_length;      // length of square floor, power of two >= 1
   int row, column;       // valid row and column coordinates
   char **floor_content;  // address of 2-dimensional array for floor content
} tiling_solution;
```

The field `floor_length` should store the length of a square floor, and it should be a power of two ≥ 1. The fields `row` and `column` should store valid row and column coordinates, respectively, of the 4-tile in a tiling solution. The field `floor_content` should store the address of a 2-dimensional array for the storage of the floor content.

This is how we will use the structure in this exercise. We will declare a variable of type `tiling_solution` in the `main` function. The `main` function will process the arguments passed to the executable to fill in valid fields `floor_length`, `row`, and `column` in the variable of type `tiling_solution`. The address of this variable will be passed to the following two functions that you have to define in `answer06.c`. (The prototypes of these two functions are in `answer06.h`.)

```
int determine_tiling_solution_category(char *filename, tiling_solution *ts);

int save_tiling_solution(char *filename, tiling_solution *ts);
```

As in PE05, the function `determine_tiling_solution_category` in PE06 has to determine the category of the "tiling solution" contained in a file whose name (as a `char *`) is provided as the first parameter to the function. The determination of the category of the tiling solution has to rely on also the `floor_length`, `row`, and `column` fields of `*ts`. Upon the completion of the function, the field `floor_content` of `*ts` should be filled with a valid address pointing to a 2-dimensional array that stores the content of the tiling solution **if the tiling solution is valid.**

Given a valid tiling solution `*ts` that has all its fields properly filled, the `save_tiling_solution` print the tiling solution to an output file whose name is provided as the first parameter to the function.

We will now provide the details of these two functions. The function `determine_tiling_solution_category` determines the category to which the input file belongs. The following list shows a total of six possible categories, as in PE05, as well as the scenario when we could not allocate space for a 2-dimensional array to store the solution.

- `VALID_SOLUTION`
- `INVALID_FILENAME`
- `INVALID_FLOOR_LENGTH`
- `INVALID_COORDINATES`
- `INVALID_FILE_SIZE`
- `INSUFFICIENT_MEMORY`
- `INVALID_FLOOR_CONTENT`

They are defined in `answer06.h` as follows:

```
#define VALID_SOLUTION 0
#define INVALID_FILENAME 1
#define INVALID_FLOOR_LENGTH 2
#define INVALID_COORDINATES 3
#define INVALID_FILE_SIZE 4
#define INSUFFICIENT_MEMORY 5
#define INVALID_FLOOR_CONTENT 6
```

If the tiling solution file cannot be opened for reading, the function should return `INVALID_FILENAME`.

If the file contains invalid floor length, the function should return `INVALID_FLOOR_LENGTH`. The function must also return `INVALID_FLOOR_LENGTH` when the floor length in the file is in a wrong format or it does not match the floor length in `*ts`.

If the file contains invalid coordinates, the function should return `INVALID_COORDINATES`. The function must also return `INVALID_COORDINATES` when any of the row and column coordinates in the file is in a wrong format or they do not match row and column coordinates in `*ts`.

If the file size is invalid, the function should return `INVALID_FILE_SIZE`.

To determine whether the floor content in the file is valid, it is recommended that you allocate for a 2-dimensional array and store the floor content in the array such that `ts->floor_content[i][j]` corresponds to the label of tile at row `i` and column `j` of the square floor. Note that we expect you to store only characters `'0'`, `'1'`, `'2'`, `'3'`, or `'4'` at `ts->floor_content[i][j]` for valid row `i` and column `j`. For that reason, we have `INSUFFICIENT_MEMORY` to have a value that is lower than `INVALID_FLOOR_CONTENT`. If you cannot allocate sufficient memory for the 2-dimensional array, the function should return `INSUFFICIENT_MEMORY`.

After storing the floor content in the 2-dimensional array, you can check whether the content is a valid tiling solution. For a tiling solution to be valid, there should be a single 4-tile and it has to be at the specified row and column coordinates. Moreover, the remaining characters in the array must form complete 0-tiles, 1-tiles, 2-tiles, and/or 3-tiles without overlapping tiles. Unlike in PE05, you have to determine whether the floor content really form a legitimate tiling solution. If the floor content in the file is invalid, the function should return `INVALID_FLOOR_CONTENT`.

Your approach may require an additional 2-dimensional array. If you could not obtain sufficient memory to do that, the function again should return `INSUFFICIENT_MEMORY`.

A file may be invalid for several reasons. The function should return the category with the lowest number. For example, if the floor length and the file size are both invalid, the function should return the lowest-valued category, i.e., `INVALID_FLOOR_LENGTH`.

If the file is valid, the `floor_content` field of `*ts` should store a valid address pointing to a 2-dimensional array storing the floor content and the function should return the category `VALID_SOLUTION`. **If the function does not return the category** `VALID_SOLUTION`**, the** `floor_content` **field of** `*ts` **should store** `NULL`**.**

The `save_tiling_solution` function saves the **valid tiling solution** in `*ts` into an output file. It should assume that the fields `floor_length`, `row`, `column`, and `floor_content` are all correct. Moreover, it should assume that characters in the 2-dimensional array are `'0'`, `'1'`, `'2'`, `'3'`, or `'4'`. If the function can successfully open the (output) file for saving the tiling solution, the function should return 1 after printing successfully to the output file the floor length, row and column coordinates and the floor content in the format as outlined in PE05 for a tiling solution. Otherwise, the function should return 0.

2.1.1 `malloc`, `calloc`, and `free`

Both `malloc` and `calloc` returns a non-`NULL` address if the allocation of memory is successful. Assuming that you are storing the returned address in `ptr`, we typically use `sizeof(*ptr)` to specify the number of bytes taken up by an element of type `*ptr`.

For example, if `ptr` is of type `int *`, `*ptr` is of type `int`.

If we want to allocate memory for `n` such elements, where `n` is a non-negative number, we use either

```
ptr = malloc(sizeof(*ptr) * n);
```

or

```
ptr = calloc(n, sizeof(*ptr));
```

The difference between the two statements is that all bytes in the memory block allocated by `calloc` will be initialized to contain value 0.

Suppose the allocation succeeds, a non-`NULL` address is returned by `malloc` (or `calloc`) and stored in `ptr`. Then `free(ptr)` frees the memory block whose address is stored in `ptr`. It is important to free memory allocated through `malloc` or `calloc` when the memory is no longer needed. If you do not free such an allocated memory, it is a memory leak. If you try to free the allocated memory more than once, it is a memory error. If you try to free a subblock within an allocated memory block, that is also a memory error.

Suppose the allocation fails, `NULL` will be returned by `malloc` (or `calloc`) and stored in `ptr`. If you try to access `*ptr` or `ptr[0]`, your program is attempting to access a privileged location. Such an access will cause your program to fail. Therefore, it is important to check whether an allocation is successful.

It is fine to perform `free(ptr)` even when `ptr` stores `NULL`. (On the other hand, `fclose(NULL)` will result in a memory error and cause the program to fail.) In general, it is good practice to use `free(ptr)` only when `ptr` does not store `NULL`.

Now, suppose `ptr` is of the correct type, and you want to store the address of some allocated memory in `ptr[i][j]`, what should you use in the `sizeof` "function" in the `malloc` or `calloc` function call?

2.2 `main` function in `pe06.c`

The purpose of the executable `pe06` is fairly straightforward: It takes in five arguments, in the following order, that correspond to respectively:

1. length of square floor
2. row coordinate of the 4-tile
3. column coordinate of the 4-tile
4. name of the (input) file containing the tiling solution
5. name of the (output) file for you to store a valid tiling solution

Look at PE03, PE04, and PA02 on how you can convert the floor length, and row and column coordinates from the relevant `argv[i]`.

If these converted numbers are valid, the `main` function proceeds to determine the category of the input file. If the `main` function reaches here, the function should complete the tasks outlined in the next few

paragraphs and return EXIT_SUCCESS at the end. Otherwise, the function should return EXIT_FAILURE immediately.

To determine the category of the input file, the main function should declare a variable ts, for example, of type tiling_solution, and initialize the fields floor_length, row, and column properly. The name of the input file and the address of ts should be used to determine the category of the input file.

If the input file is not valid, the reason for the invalidity should be printed to stdout with the format "%d\n". If the input file is valid, the name of the output file and the address of ts should be used to save the tiling solution to the output file.

If the tiling solution could be saved to the output file, the main print VALID_SOLUTION to stdout with the format "%d\n". Otherwise, the main function should print INVALID_OUTPUT_FILE to stdout with the format "%d\n". Note that INVALID_OUTPUT_FILE is defined in answer06.h as follows:

```
#define INVALID_OUTPUT_FILE 7
```

2.3 Printing, helper functions and macros

All debugging, error, or log statements in answer06.c and pe06.c should be printed to stderr.

You may define your own helper functions in pe06.c and answer06.c. All these functions should be declared and defined as static functions. In other words, the scope of these functions are only within the files that they are found. Declaring and defining these functions as static eliminate the conflicts in function names.

You may want to modify some of the functions you have written in PE05 and turn them into (static) helper functions in pe06.c and answer06.c. Please note that you are submitting only pe06.c and answer06.c. **The files** pe06.c **and** answer06.c **must be self-contained and you should not make changes to** answer06.h**.**

You should not use macros that start with T_ (Upper case T and underscore) in their names. We will use such macros in the evaluation of your submissions. If you use such macros, we may not be able to evaluate your submission properly.

3 Compiling your program

We use the same flags introduced in PE01 to compile the program for debugging purpose.

```
gcc -std=c99 -Wall -Wshadow -Wvla -Werror -g -pedantic -fstack-protector-strong \
    --param ssp-buffer-size=1 pe06.c answer06.c -o pe06
```

When we evaluate your program, we also use the optimization flag -O3 (uppercase letter O and number three) instead of the -g flag as follows:

```
gcc -std=c99 -Wall -Wshadow -Wvla -Werror -O3 -pedantic -fstack-protector-strong \
    --param ssp-buffer-size=1 pe06.c answer06.c -o pe06
```

You are recommended to copy the Makefile from PE01 and modify it appropriately for PE06.

4 Writing, running and testing your program

We provide a few test cases for you. The first number (or "character") in the filename indicates the category of the tiling solution file if your program works as intended. Subsequently, the next three numbers in the filename correspond to the floor length, row coordinate and column coordinate of the 4-tile, respectively. You should be able to create more test cases by hand to test your program.

You should also run `./pe06` with appropriate arguments under `valgrind`.

Please see PE01 description about `valgrind`.

There are a few tips on how you may write and test your program. You should consider writing the allocation and free functions together. You can call these functions (in pair) in the `main` function and run to valgrind to see whether you allocated the right amount of memory in the right number of blocks and whether you free them successfully. For different values of `n`, you can calculate the total bytes required (each address has 8 bytes and a char occupies a byte).

You can make simple modifications to the `determine_tiling_solution_category` function from PE05 such that any "valid" solution from PE05 is still considered "valid" here. (All other functions from PE05 should be turned into static functions, perhaps with modifications, to be used by this modified function.) Now, you only have to add the features of allocating for the 2-dimensional array and storing the floor content into this 2-dimensional array.

This simplified version of `determine_tiling_solution_category` essentially reads from a file and store the content into memory. Now, you write the `save_tiling_solution` function. This function reads from memory and prints to a file.

Now, you can test the these two functions together from the `main` function. Any valid input files from PE05 would be saved into a output file. You can use the `diff` command at the terminal to check whether two files are different. For example, you can check whether `file1` is different from `file2` by using the following command.

```
diff file1 file2
```

Note that `file1` are `file2` are just examples. You should use actual file names.

If everything is fine so far, i.e., no memory issues and the input files match the corresponding output files, you can modify the `determine_tiling_solution_category` function to check for the correctness of a tiling solution.

If you have created all skeleton functions that you have to write, you can evaluate the correctness of the allocation and free functions at the correct functions. You would call the allocation function in the (skeleton) `determine_tiling_solution_category` function and the free function in the `main` function at appropriate locations. In other words, you may not have to write additional lines to test your functions.

5 Submission

You must submit a zip file called `PE06.zip`, which contains two files:

1. `answer06.c`
2. `pe06.c`

Assuming that you are in the folder that contains `answer06.c` and `pe06.c`, use the following command to zip your files:

```
zip PE06.zip answer06.c pe06.c
```

Make sure that you name the zip file as `PE06.zip`. Moreover, the zip file should not contain any folders. Submit `PE06.zip` through Brightspace. You may submit as many versions as you like. Brightspace will keep only the most recent submission, and we will grade only the final submission.

6 Grading

All debugging messages should be printed to `stderr`. We expect only a single `int` to be printed to `stdout`. It is important that if the instructor has a working version of `pe06.c`, it should be compilable with your `answer06.c` to produce an executable. For evaluation purpose, we will use different combinations of your submitted `.c` files and our `.c` files to generate different executables. If a particular combination does not allow an executable to be generated, you do not get any credit for the function(s) that the executable is supposed to evaluate. We use both `-g` and `-O3` flags (separately) to compile your submission. Your submission is evaluated only when compilation using each of the flags is successful.

It is important to note that the `.c` files from the instructor do not assume the presence of global variables that are declared by the students. Of course, the `.c` files from the instructor may use global variables that are in C libraries, such as `errno`, `stdout`, `stderr`, and so on. If your submission contains global variables that are declared by you, it is unlikely that your executable will work correctly.

Be aware that we set a time-limit for each test case based on the size of the test case. If your program does not complete its execution before the time limit for a test case, it is deemed to have failed the test case. We will not announce the time-limits that we will use. They will be very reasonable.

The `allocate_2d_array` function and the `free_2d_array` function together account for 20%. The `determine_tiling_solution_category` function accounts for 50%. The `save_tiling_solution` function accounts for 20%. The `main` function accounts for 10%.

The occurrence of any memory issues (memory errors or memory leaks flagged in a `valgrind` report) will result in 50-point penalty.

7 A few points to remember

All debugging messages should be printed to `stderr`. Other than the category of the input file, you should not be printing anything else to `stdout`. If the output of your program is not as expected, you get 0 for that test case.

You can declare and define additional static functions that you have to use in `pe06.c` and `answer06.c`.

You should not use macros that start with `T_`.

Grading of programming exercises and assignments is performed on machines with similar setup as eceprog.ecn.purdue.edu. You should perform testing of your work on eceprog.ecn.purdue.edu before submission. Correct output on your computer does not translate into correct output on eceprog.ecn.purdue.edu.