ECE26400 Programming Exercise #5

In this exercise, you will learn to process a file that contains information pertaining to the tiling of a square using tiles of certain shape. **You are not allowed to store the contents of the file in an array. All operations must be performed on a file.** This exercise is related to PE06 and PA03.

The main learning goals are:

1. How to open and close a file using `fopen` and `fclose`.
2. How to use `fgetc` to obtain a character from a file.
3. How to use `fscanf` to perform conversions from a file.
4. How to use `fseek` to go to a particular location of a file.
5. How to use `ftell` to know where you are in a file.

1 Getting started

You should unzip on eceprog.ecn.purdue.edu the zip file `pe05_files.zip` using the following command:

```
unzip pe05_files.zip
```

The zip file `pe05_files.zip` contains a folder, named `PE05`, and one file within the folder and a subfolder containing some sample files:

1. `answer05.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 `5-16-4-7`.

In this exercise, you have to create a file called `answer05.c` to define all functions declared in `answer05.h` and a file called `pe05.c` for the `main` function.

To get started, read this document in its entirety. PE05, PE06, and PA03 are designed as follows: In PA03, you have to generate a solution to a tiling problem (see Section 2). How do you verify that your solution to PA03 is correct? PE05 and PE06 are designed for you to verify that indeed the solution you generated for a tiling problem is valid. Since you have completed PE05 and PE06 first, it means that as you work on PA03, you can actually evaluate your program, i.e., you can check whether your program generates valid or invalid solutions. This is in fact how I write programs. I first write test functions, or functions to evaluate my final solutions. Then I write the functions that would generate the actual solutions.

2 Tiling of a square floor

In this exercise (and PE06 and PA03), we consider the tiling of a floor that is in the shape of a square. Moreover, the length of every side of the square has to be $n = 2^k$ units long, with $k \geq 0$. In other words, the length of each side of the floor is a power of two ≥ 1. The floor could be 1×1, 2×2, 4×4, 8×8, and so on.

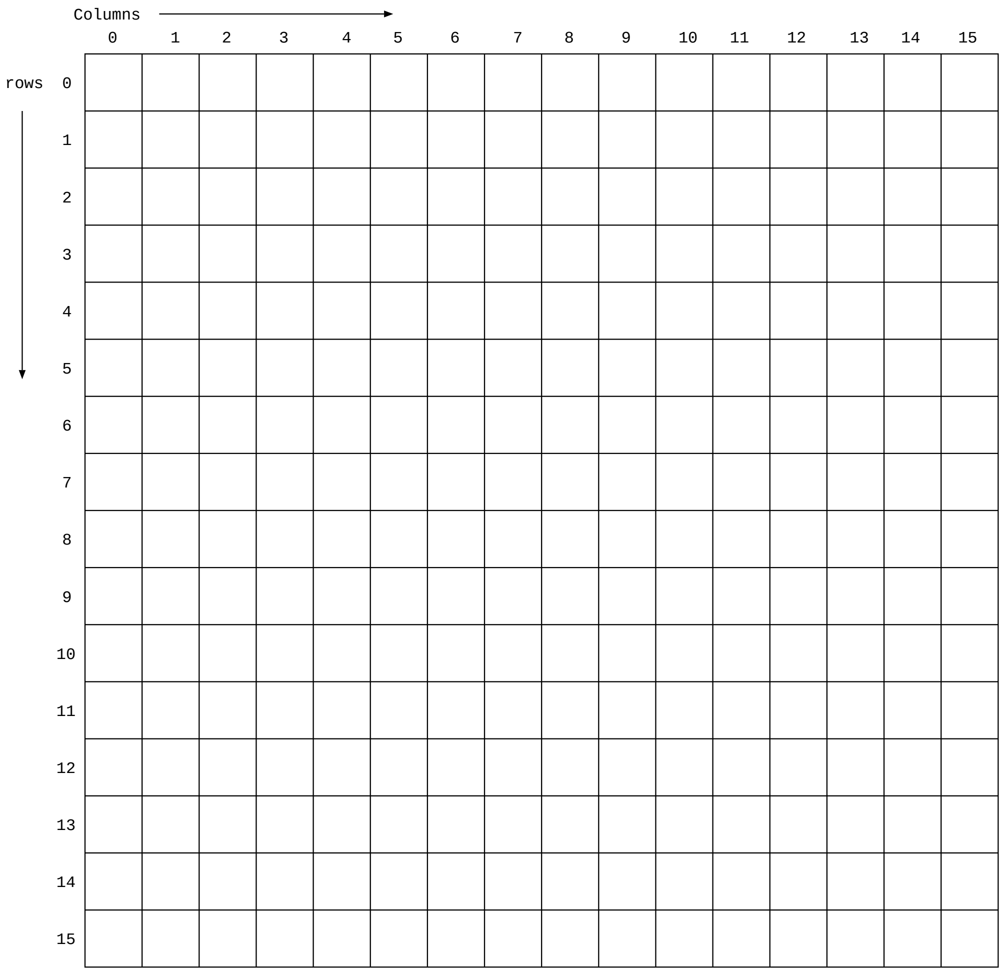


Figure 1: A 16 × 16 floor.

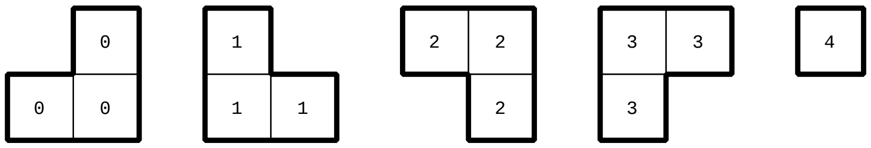
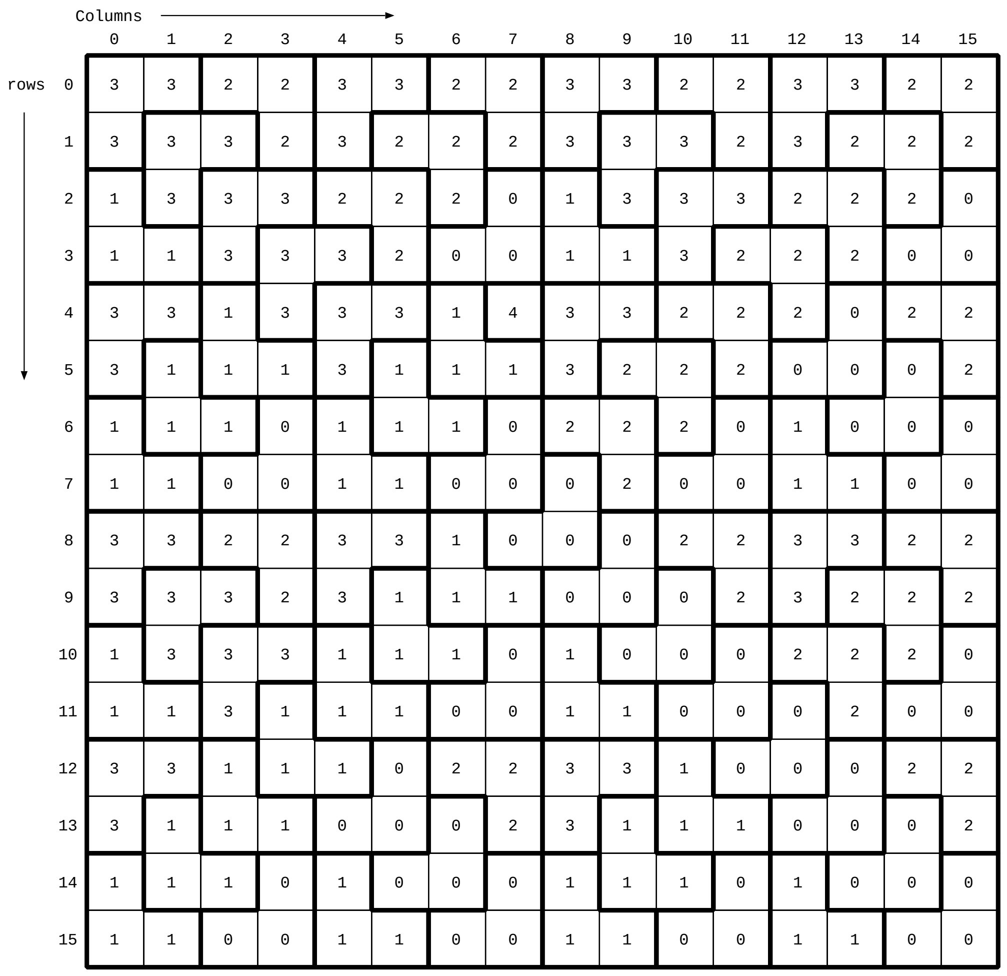


Figure 2: Four different *L*-shape tiles, each of 3 unit squares, and a 1-unit-square tile.

Now, consider a 16×16 floor in Figure 1. Here, $n = 2^4$, i.e., $k = 4$. We want to cover the area with some combinations of *L*-shape tiles of 3 unit squares in different orientations and a single 1×1 tile. We label the *L*-shape tiles in four different orientations as 0-tile, 1-tile, 2-tile, and 3-tile, and we refer to the 1×1 tile as a 4-tile. In Figure 2, we show a 0-tile, 1-tile, 2-tile, and 3-tile, as well as a 4-tile. We label all unit-squares in an *i*-tile, for $0 \leq i \leq 3$, with *i*.

A valid tiling solution is one where a single 4-tile is used, and some combinations of 0-tiles, 1-tiles, 2-tiles, and 3-tiles are used to cover all unit-squares of the 16×16 floor **without overlapping tiles**.

Figure 3 shows a solution to cover the 16×16 floor entirely when the 4-tile is at row 4 column 7. Here, we label the rows from top to bottom; we start with row number 0 and end with row number 15. We also label the columns from left to right; we start with column number 0 and end with column number 15.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	3	3	2	2	3	3	2	2	3	3	2	2	3	3	2	2
1	3	3	3	2	3	2	2	2	3	3	3	2	3	2	2	2
2	1	3	3	3	2	2	2	0	1	3	3	3	2	2	2	0
3	1	1	3	3	3	2	0	0	1	1	3	2	2	2	0	0
4	3	3	1	3	3	3	1	4	3	3	2	2	2	0	2	2
5	3	1	1	1	3	1	1	1	3	2	2	2	0	0	0	2
6	1	1	1	0	1	1	1	0	2	2	2	0	1	0	0	0
7	1	1	0	0	1	1	0	0	0	2	0	0	1	1	0	0
8	3	3	2	2	3	3	1	0	0	0	2	2	3	3	2	2
9	3	3	3	2	3	1	1	1	0	0	0	2	3	2	2	2
10	1	3	3	3	1	1	1	0	1	0	0	0	2	2	2	0
11	1	1	3	1	1	1	0	0	1	1	0	0	0	2	0	0
12	3	3	1	1	1	0	2	2	3	3	1	0	0	0	2	2
13	3	1	1	1	0	0	0	2	3	1	1	1	0	0	0	2
14	1	1	1	0	1	0	0	0	1	1	1	0	1	0	0	0
15	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0

Figure 3: A solution to cover the 16×16 floor entirely when the 4-tile is at row 4 column 7.

2.1 The tiling problem

This is the tiling problem that you will solve in PA03: You are given an $n \times n$ floor, where $n = 2^k$ with k being an integer ≥ 0. You are also given a valid location (in terms of the row-column coordinates) that a

4-tile has to reside. Your task is to generate the positions of 0-tiles, 1-tiles, 2-tiles, and 3-tiles to cover the entire floor, together with the 4-tile, without overlapping tiles.

Figure 4 shows the solutions for 1×1 floor and 2×2 floor (for four different locations of the 4-tile).

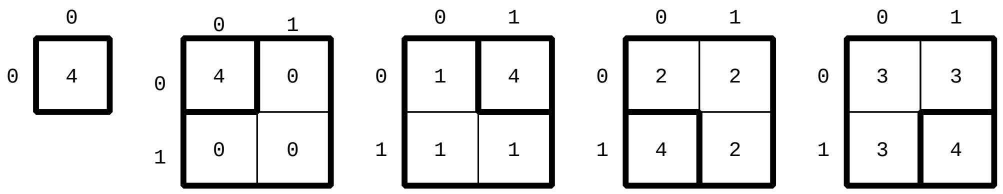


Figure 4: Tiling solutions for five different simple tiling problems: (1) 1×1 floor with the 4-tile at row 0, column 0; (2) 2×2 floor with the 4-tile at row 0, column 0, (3) 2×2 floor with the 4-tile at row 0, column 1, (4) 2×2 floor with the 4-tile at row 1, column 0, and (5) 2×2 floor with the 4-tile at row 1, column 1.

I would encourage you to find a solution to a 4×4 problem and a 8×8 problem shown in Figure 5. In the 4×4 problem, the 4-tile is placed at row 2, column 0. In the 8×8 problem, the 4-tile is placed at row 1, column 5. You may want to to place the 4-tile at different locations and solve the new 4×4 and 8×8 problems.

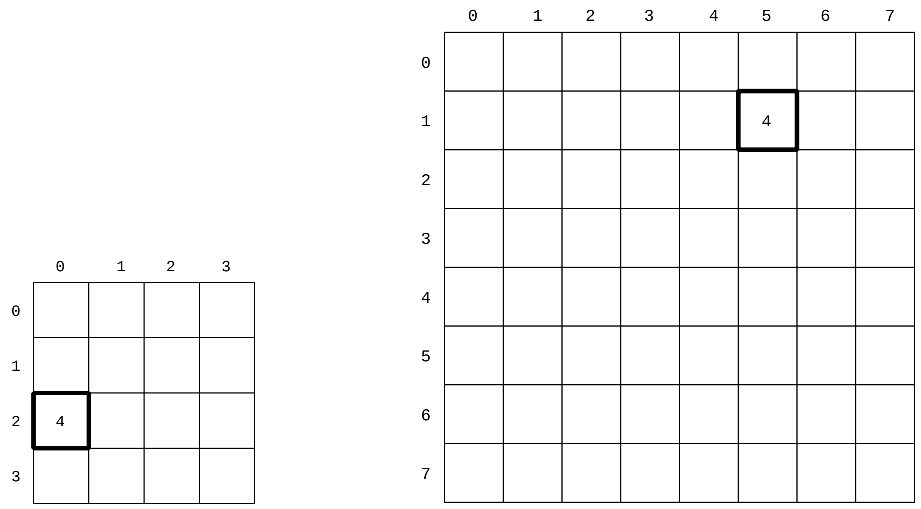


Figure 5: A 4×4 tiling problem and a 8×8 tiling problem.

One straightforward approach to solving this problem is to enumerate all possible ways to place 0-tiles, 1-tiles, 2-tiles, and 3-tiles and find one that is a valid solution.

I would like you to try a different approach, that of divide and conquer. Can you divide a $n \times n$ problem into $(n/2) \times (n/2)$ problems and solve (conquer) the smaller problems? In particular, can you place a 0-tile, 1-tile, 2-tile, or 3-tile at an appropriate location of the $n \times n$ problem and turn it into four smaller $(n/2) \times (n/2)$ problems.

2.2 File format for a solution

In this exercise, we will use a “text” file to store a solution. In this file, we first use the format `"%d\n"` to print the length of the floor. Then, we use the format `"%d,%d\n"` to print the row and column coordinates of the 4-tile.

We then have n lines, each with $n+1$ characters. We also label the lines from line 0 through line $n-1$. Each line corresponds to a row in the floor, starting from row 0 (and line 0). In each line, the first n characters in each row should be of value `'0'`, `'1'`, `'2'`, `'3'`, or `'4'`, and the line ends with `'\n'`. The j-th character, $0 \leq j < n$, at line i, $0 \leq i < n$, records the label of the tile at row i column j of the $n \times n$ floor. If the location is occupied by a 4-tile, that character is '4'. If the location is occupied by a 0-tile, 1-tile, 2-tile, or 3-tile, that character is `'0'`, `'1'`, `'2'`, or `'3'`, respectively.

The file `16-4-7` in the `examples` subfolder contains the solution in the third figure.

```
16
4,7
3322332233223322
3332322233323222
1333222013332220
1133320011322200
3313331433222022
3111311132220002
1110111022201000
1100110002001100
3322331000223322
3332311100023222
1333111010002220
1131110011000200
3311102233100022
3111000231110002
1110100011101000
1100110011001100
```

3 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. You will determine whether the given file is indeed a solution in PE06. Therefore, for the purpose of PE05, a given file is valid as long as:

- The length of the floor is in the correct format, and it is a power of two ≥ 1. By correct format, it means that the length is printed with the fewest digits possible (no leading zeros), it has no white space before it, and it is followed immediately by a `'\n'` or newline character.
- The row-column coordinates of the 4-tile are in the correct format, and the values of the row-column coordinates are within the correct bounds for the given floor length. By correct format, it means that the row coordinate is printed with the fewest digits possible (no leading zeros), it has no white space before it, and it is followed immediately by a `','`. Note that the single `'\n'` that immediately follows the floor length is the only character separating the row coordinate and the floor length.

The column coordinate is printed with the fewest digits possible (no leading zeros), it has no white space before it, and it is followed immediately by a `'\n'` or newline character.

- The size of the file is just enough to contain the length of the floor (in the right format), the row-column coordinates of the 4-tile (in the correct format), and the *n* lines of characters that correspond to the tiling solution. Here, *n* is the length of the floor, and there should be $n+1$ characters in each line.
- The *n* lines of characters are formed by valid characters. Each of the first *n* characters in each line should be `'0'`, `'1'`, `'2'`, `'3'`, or `'4'`. Moreover, `'4'` should occur only once, appearing at the correct location in the file as indicated by the row-column coordinates. After the *n* valid characters in each line, the line should end with a `'\n'` or newline character.

3.1 Functions to be written in `answer05.c`

You have to define these functions in `answer05.c`, which you have to create. (The declarations of the prototypes are in `answer05.h`.) In the first four functions described below, they all have a single parameter `FILE *fp`. You may assume that `fp` is not NULL.

In the following, for a function that takes in `FILE *fp` as a parameter, you may not assume that the file position index of `fp` is at the beginning of the file. In fact, the file position index of `fp` could be at any position in the file when the function is called.

```
bool is_floor_length_valid(FILE *fp);
```

You may not assume that the file position index of `fp` is at the beginning of the file.

This function should return `true` if the floor length is of the correct format and that it is a power of two that is ≥ 1; otherwise, it should return `false`.

The values `true` and `false` are of `bool` (Boolean) type, the definition of which is found in `stdbool.h`, which is included in `answer05.h`. The numerical values of `true` and `false` are respectively 1 and 0.

```
bool are_coordinates_valid(FILE *fp);
```

You may not assume that the file position index of `fp` is at the beginning of the file. You may not assume that the file position index of `fp` is at the appropriate position of the file for you to read in the row and column coordinates.

The function is called only if `is_floor_length_valid(fp)` returns `true`. In other words, the caller function has to ascertain that the floor length is valid in the file before calling this function. This function should not be called if the corresponding file does not have a valid floor length. This function should return `true` if the coordinates are of correct format and of suitable values; otherwise, it should return `false`.

```
bool is_file_size_valid(FILE *fp);
```

You may not assume that the file position index of `fp` is at the beginning of the file. You may not assume that the file position index of `fp` is at the appropriate position of the file for you to read in the floor content.

The function is called only if both `is_floor_length_valid(fp)` and `are_coordinates_valid(fp)` return `true`. This function should return `true` if the file size meets exactly the requirement for the file to contain a potentially valid solution; otherwise, it should return `false`.

```
bool is_floor_content_valid(FILE *fp);
```

You may not assume that the file position index of `fp` is at the beginning of the file. You may not assume that the file position index of `fp` is at the appropriate position of the file for you to read in the floor content.

The function is called only if all three functions `is_floor_length_valid(fp)`, `are_coordinates_valid(fp)`, and `is_file_size_valid(fp)` return `true`. This function should return `true` if the file content may correspond to a potentially valid solution; otherwise, it should return `false`.

For the floor content to be valid, the *n* (for a floor of length *n*) lines of characters representing a tiling solution must be formed by valid characters. Each of the first *n* characters in each line should be `'0'`, `'1'`, `'2'`, `'3'`, or `'4'`. Moreover, `'4'` should occur only once, appearing at the correct location in the file as indicated by the row-column coordinates. After the *n* valid characters in each line, the line should end with a `'\n'` or newline character.

```
int determine_tiling_solution_category(char *filename);
```

You have to determine the category of the "tiling solution" contained in a file whose name (as a `char *`) is provided as a parameter to the function. There are six possible categories:

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

They are defined in `answer05.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 INVALID_FLOOR_CONTENT 5
```

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`. If the file contains invalid coordinates, the function should return `INVALID_COORDINATES`. If the file size is invalid, the function should return `INVALID_FILE_SIZE`. If the floor content in the file is invalid, the function should return `INVALID_FLOOR_CONTENT`.

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 function should return the category `VALID_SOLUTION`.

Indeed, the only function that we would like to write is the `determine_tiling_solution_category` function. You may find this exercise somewhat repetitive because of all these other functions that you have to write. These other functions that you have to define are made to be a part of the exercise as a way to remind you that you typically want to work on a function a small chunk at a time. This exercise encourages you to test a feature added to a function (the `determine_tiling_solution_category` function) thoroughly before you add another feature to the function.

3.1.1 `fopen` and `fclose`

You should always check whether an attempt to open a file using `fopen` is successful. If the attempt to open a file is unsuccessful, the returned address of `fopen` is NULL or `(void *)0)`. Any attempt to access a `FILE` structure at location `NULL` or `(void *)0)` will result in a memory error, because that is a privileged location that your program should not access. In other words, any file operations (e.g., `fgetc`, `ftell`) involving `NULL` will cause your program to fail.

In fact, when `fopen` fails, you should not even perform a `fclose` on the returned address, because `fclose` performs some housekeeping operations that require it to access the `FILE` structure at location `NULL` or `(void *)0)`.

3.2 `main` function in `pe05.c`

The purpose of the executable `pe05` is fairly straightforward: It takes in an argument that corresponds to the name of the file containing a tiling solution. It determines the category of the file (or the tiling solution contained in the file) and prints to the `stdout` the category using the format string `"%d\n"`. In other words, the executable should print the category of the input file and return `EXIT_SUCCESS` when the `argc` is of correct value; otherwise, it should return `EXIT_FAILURE` (and print nothing to the `stdout`).

3.3 Printing, helper functions and macros

All debugging, error, or log statements in `answer05.c` and `pe05.c` should be printed to `stderr`.

You may define your own helper functions in `pe05.c` and `answer05.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.

What are good candidate helper functions? Think of a task that is needed in many of these functions that you have to write. You probably have to move the position index of a file stream to the beginning. Since that could be done with a simple statement, you probably do not have to define a function to do that. You probably have to read a number. In fact, you may have to read three numbers. Should you write a helper function to do that? What should you use to read in a number from a file? Should you use `fgetc`, `fscanf`, or `fread`? Should you use a combination of these functions?

Is the number delimited by a newline or a comma properly? Should you combine this check of delimiter with the function to read a number?

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.

4 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 pe05.c answer05.c -o pe05
```

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 pe05.c answer05.c -o pe05
```

You are recommended to copy the `Makefile` from PE01 and modify it appropriately for PE05.

5 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 `./pe05` with appropriate arguments under `valgrind`.

Please see PE01 description about `valgrind`.

6 Submission

You must submit a zip file called `PE05.zip`, which contains two files:

1. `answer05.c`
2. `pe05.c`

Assuming that you are in the folder that contains `answer05.c` and `pe05.c`, use the following command to zip your files:

```
zip PE05.zip answer05.c pe05.c
```

Make sure that you name the zip file as `PE05.zip`. Moreover, the zip file should not contain any folders. Submit `PE05.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.

7 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 `pe05.c`, it should be compilable with your `answer05.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 `is_floor_length_valid` function accounts for 10%. The `are_coordinates_valid` function accounts for 10%. The `is_file_size_valid` function accounts for 20%. The `is_floor_content_valid` function accounts for 20%. The `determine_tiling_solution_category` function accounts for 30%. 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.

8 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 `pe05.c` and `answer05.c`.

You are not allowed to use arrays to store the content of the input file in this exercise.

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.