Advanced C Programming

Fall 2017 :: ECE 264 :: Purdue University

This is for Fall 2017 (2 years ago) only.

Maze #1 (E)

Due 10/20

Overview

This is the first step toward HW10 in which you will "solve" a maze problem. You will design a program that would allow an agent air-dropped into a rectangular maze to find the shortest path to reach some other destination in the maze. For this exercise, you will write functions to:

1. Determine the dimensions of a given maze.
2. Find the column location of the openings at the top/bottom of a given maze.
3. Count the number of path locations in a given maze.
4. Determine whether a given location in the maze is a wall or a path.
5. Create a new file containing a single-line representation of the maze.

You will be learning to program with **files**, including the `fopen(…)`, `fgetc(…)`, `fputc(…)`, and `fclose(…)` functions.

Input file format

A maze is a rectangle consisting of r rows and c columns. We number the rows from top to bottom as row 0 through row 6, and the columns from left to right as column 0 through column 8.

Example

The maze at right can be found in the file called sample.2.7x9. It has seven rows (`r=7`) and nine columns (`c=9`).

sample.2.7x9

```
XXXXX XXX\n
X     X\n
XXXXX X X\n
X     X X\n
XXXXXXX X\n
X       X\n
XXXXX XXX\n
```

The file contains exactly 7 lines, with exactly 10 characters per line: 9 visible characters (`'X'` or `' '`) plus a newline (`'\n'`) at the end of each line. It has a total of 25 `PATH` characters and 38 `WALL` characters. (`PATH` and `WALL` are constants defined in maze.h as `' '` and `'X'`, respectively.)

The character highlighted in orange (**x**) is a `WALL` character, and would be identified as row 0 (`r=0`) and column 2 (`c=2`). The two openings are highlighted in green ( ).

Creating new test mazes

An executable "amaze" is provided for you to generate other test cases. To generate a maze with `(2r + 1)` rows and `(2c+1)` columns, where `r` and `c` ≤ 69, issue the following command:

```
./amaze r c X > filename
```

…where *filename* is the name of the file to store the generated maze. Note that amaze outputs the maze to the stdout, but the redirection '>' redirects that output to a file with the name you specify.

Requirements

1. Your submission must contain each of the following files, as specified:

file	contents	
maze.c	functions	**`find_maze_dimensions`**`(FILE * fp, int * a_num_rows, int * a_num_cols)` → *return type:* void Determine the dimensions of the maze contain in file. • `fp` is an *open* file pointer for a file containing a maze *in the correct format*. • Write the number of rows to the location at `a_num_rows`, and likewise for `a_num_cols`. • You may assume `a_num_rows` and `a_num_cols` are safe to write to.
		`find_opening_location``(FILE * fp)` → *return type:* int Determine the column location of the top opening. • The top opening is the sole `PATH` character in the first row. • There will be exactly one opening in the top and one in the bottom (at the same column). • `fp` is an *open* file pointer for a file containing a maze *in the correct format*.
		`count_path_locations``(FILE * fp)` → *return type:* int Count the number of locations that are `PATH` (defined in maze.h). • `fp` is an *open* file pointer for a file containing a maze *in the correct format*. • Include the openings at the top and bottom of the maze. • A location is `PATH` if it equal to the `PATH` variable (currently defined as `' '` in maze.h). • Example: If `fp` refers to sample.2.7x9, then `count_path_locations(fp)` should return 25.
		`get_location_type``(FILE * fp, int row, int col)` → *return type:* char Return the type of location: `WALL` or `PATH`. • `row` and `col` are valid coordinates for the location of interest. • `fp` is an *open* file pointer for a file containing a maze *in the correct format*. • You simply return the character at the given location. For example, `get_location_type(fp, 0, 0)` will always return the first character in the file. • Example: If `fp` refers to sample.2.7x9, then `get_location_type(fp, 0, 2)` should return `WALL` (`'X'`).
		`represent_maze_in_one_line``(char * filename, FILE * fp)` → *return type:* int Write a copy of the maze, excluding newlines (`'\n'`), to a new file. • `fp` is an *open* file pointer for a file containing a maze *in the correct format*. • `filename` is a valid filename (i.e., not `""`, etc.). You will open this file and write to it. • Example: If `fp` refers to sample.2.7x9, then `represent_maze_in_one_line("newsample.2.7x9", fp)` should return write the following (below) to a new file called "newsample.2.7x9" and return 63. XXXXX XXXX XX XXX XXXX X X XX X XXXXXX XXXXXX XXX • If opening the new file or writing to it fails, return `MAZE_WRITE_FAILURE` (defined in maze.h). • Return the number of characters written into the the new file. • ⚠ The new file must not contain any newline characters.

2. ⚠ For functions that take a file pointer `fp` as an argument:
   a. Assume `fp` is already open.
   b. Do not assume `fp` is currently at the beginning of the file.
   c. Do not close `fp` in the function that received it as an argument.
3. ⚠ Do not make any assumption about the order in which functions will be called. Each function must work as specified, even if we call them in some strange and unpredictable fashion.
4. ⚠ Any file pointers opened within any of the above functions must be closed before the function returns.
5. ⚠ Do not print error messages or anything else on `stdout` (i.e., with `printf(…)`).
6. ⚠ Do not modify maze.h.
7. ⚠ Refer to `PATH` and `WALL` only by those names, and not by the character constants `' '` and `'X'`.
8. ⚠ Do not include a `main(…)` in maze.c. (→ zero credit, since we won't be able to compile.)
9. Submissions must meet the code quality standards and the course policies on homework and academic integrity.

Submit

To submit HW09, type `264submit HW09 maze.c` from inside your hw09 directory.

In general, to submit any assignment for this course, you will use the following command:

```
264submit ASSIGNMENT FILES…
```

Submit often and early, even well before you are finished. Doing so creates a backup that you can retrieve in case of a problem (e.g., accidentally deleted your files).

To retrieve your most recent submission, type `264get --restore` *ASSIGNMENT* (e.g., `264get --restore hw09`).

To retrieve an earlier submission, first type `264get --list` *ASSIGNMENT* to view your past submissions and find the timestamp of the one you want to retrieve. Then, type `264get --restore -t` *TIMESTAMP* *ASSIGNMENT*.

Scores will be posted to the Scores page after the deadline for each assignment.

Update: You do not need to submit test_maze.c or Makefile for this exercise.

Pre-tester ●

The pre-tester for HW09 has been released and is ready to use.

Using the pretester

The pretester is a tool for checking your work after you believe you are done, and before we have scored it. It is not a substitute for your own checking, but it may help you avoid big surprises by letting you know if your checking was not adequate. To use the pre-tester, first submit your code. Then, type the following command. (*Do this only after you have submitted, and only after you believe your submission is perfect.*)

```
264test hw09
```

Do not ask TAs or instructors which tests you failed.

Keep in mind:

- Pre-testing is intended only for those who believe they are done and believe their submission is perfect.
- The pre-tester is not part of the requirements of this or any other assignment.
- You are responsible for reading the assignment carefully, and ensuring that your code meets all requirements.
- Feedback is limited, to ensure that everyone learns to test their own code.
- If your code is failing some tests, review *your tests* and make sure they are comprehensive enough to catch any bugs (deviations from requirements). Follow the tips given by the pre-tester.
- Code quality issues are not reported by the pre-tester; writing clean code is something you must learn to do from the start, not a clean-up step to do at the end.

Logistics:

- If we discover that we have not checked some significant part of the assignment requirements, we may add additional tests at any time up to the point when scores are released.
- The pre-tester will only be enabled after much of the class has submitted the assignment, and at least a few people have submitted perfect submissions. This is to allow us to test the pre-tester.
- The pre-tester checks your most recent submission. You must submit first.
- You may be limited to running the pre-tester ≤24 times in a 24-hour period. (This is not implemented yet but will be added.)

Q&A

1. **I get Permission Denied when I try to run ./amaze. How do I make it work?**
   From bash, run `chmod u+x amaze` to adjust the permissions so that it can be executed.

2. **Can we use `fseek(…)`/`feof(…)`/`EOF`/…?**
   Yes. You may use anything in the C99 standard library (subject to the base requirements in the Syllabus).

3. **Do I need to submit test_maze.c or Makefile?**
   No.

4. **Can I have a function that returns a `CodeDeck`?**
   Yes. A function may return a struct object.

5. **Does this line of code violate the code quality standards (multiple statements on line line)?**
   `*a_num_rows = *a_num_cols = 0;`
   No. That is one statement.

6. **Can we add helper functions to maze.c?**
   Yes, as long as ⓐ the name begins with `"_"`, and ⓑ they do not require any changes to maze.h.

Updates

10/17/2017 You do not need to turn in test_maze.c or Makefile. Added Q&A with Q1-Q5.

10/17/2017 The contents of "newsample.2.7x9" given in the description for the function represent_maze_in_one_line are updated to reflect what are really stored in the file. To test your function, you should write to a different file and perform a "diff" between "newsample.2.7x9" and your output file.

10/19/2017 Pre-tester was updated.