# Advanced C Programming

*Fall 2017 :: ECE 264 :: Purdue University*



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

# Maze #2 (E)

Due 10/27



## Overview

This is the second step (after HW09) toward HW11, in which you will "solve" a maze problem. This time, you will do the following:

1. Read a maze in a multi-line file into a 2D array of characters.
2. Write a maze from a 2D array of characters into a multi-line file.
3. Expand an $h \times w$ maze into a $(2h - 1) \times w$ maze (h = # of rows; w = # of columns; both h and w are odd).
4. Expand an $h \times w$ maze into a $h \times (2w - 1)$ maze.

The remainder of the explanation for this assignment is in the Requirements table and Q&A (below).

## Requirements

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

| file | contents | |
|---|---|---|
| maze.c | functions | **`malloc_maze`**`(int num_rows, int num_cols)`<br>→ *return type:* Maze✶<br>Allocate heap memory for a maze, and initialize the `num_rows` and `num_cols` fields.<br>• Allocate space for the `Maze` *and* its `cells` field.<br>• You may assume `num_rows` and `num_cols` are >0.<br>• ~~Hint: This will call `malloc(…)` twice.~~<br>• If allocation fails, free any memory successfully allocated by this function prior to the failure, and then return NULL.<br>• Caller is responsible for freeing the new maze.<br>• ⚠ You cannot declare a variable of type `Maze` and then return its address. When the function returns, that variable's space will be deallocated (i.e., garbage). You must use the heap for this. |
| | | **`free_maze`**`(Maze✶ maze)`<br>→ *return type:* void<br>Free heap memory for a maze.<br>• You must free the `cells` and the `Maze` itself.<br>• If `maze` is `NULL`, do nothing. (This mimicks the behavior of `free(…)`.) |

| file | contents | |
|---|---|---|
| | | **`read_maze`**`(FILE* fp)`<br>→ *return type:* Maze*<br>Read a maze from a multi-line file.<br>• The file will be in the 2D format specified in HW09 and illustrated by sample.2.7x9.<br>• Return the address of a new `Maze` object on the heap.<br>• Initialize all fields of the new `Maze` object with the maze data in the file.<br>• Use your `malloc_maze(…)` to allocate the heap memory for the new `Maze` object.<br>• Caller is responsible for freeing the new maze. |
| | | **`write_maze`**`(const char* filename, const Maze* maze)`<br>→ *return type:* bool<br>Write a maze to a file.<br>• Write the contents of `maze` to a new file with the name specified by `filename`.<br>• Follow the 2D format specified in HW09 and illustrated by sample.2.7x9. The file created by this function will contain exactly `maze->num_rows` newline characters.<br>• ~~Do not open or close any file from within this function. You may, of course, write to the file referred to by `fp`, but do not close it or open any other file.~~<br>• ⚠ Be sure to close the file within this function.<br>• Return true if successful, or false if there was an error writing the file. |
| | | **`make_taller`**`(const Maze* orig)`<br>→ *return type:* Maze*<br>• Create a new `Maze` object maze based on `orig`, but with height `orig->num_rows * 2 - 1`.<br>• The width of the new maze should be the same as `orig` (i.e., `orig->num_cols`).<br>• The top half of the new maze should be the same as `orig`.<br>• The bottom half of the new maze should be a reflection of `orig`.<br>• Example:<br>`orig`<br>`XXXXX XXX`<br>`X X`<br>`X XXX XXX`<br>`X X X X`<br>`X X XXXXX`<br>`X X`<br>`XXXXX XXX`<br>`make_taller(orig)`<br>`XXXXX XXX`<br>`X X`<br>`X XXX XXX`<br>`X X X X`<br>`X X XXXXX`<br>`X X`<br>`XXXXX XXX`<br>`X X`<br>`X X XXXXX`<br>`X X X X`<br>`X XXX XXX`<br>`X X`<br>`XXXXX XXX`<br>• Do not modify `orig`.<br>• Use your `malloc_maze(…)` to allocate the heap memory for the new `Maze` object.<br>• Caller is responsible for freeing the new maze.<br>• ⚠Calling this function should not result in any calls to `fopen(…)` or `fclose(…)`, or `free(…)`.<br>• In case of allocation failure, return NULL. |

<table>
<tr><th>file</th><th>contents</th><th></th></tr>
<tr><td></td><td></td><td>
make_wider(const Maze* orig)<br>
→ return type: Maze*
<ul>
<li>This is very similar to <code>make_taller(…)</code> but you will need some extra steps to ensure the paths in the left and right hemispheres of the new maze are connected as one maze.</li>
<li>Create a new <code>Maze</code> object maze based on <code>orig</code>, but with width <code>orig->num_cols * 2 - 1</code>.</li>
<li>The height of the new maze should be the same as <code>orig</code> (i.e., <code>orig->num_rows</code>).</li>
<li>The left half of the new maze should be the same as <code>orig</code>.</li>
<li>The right half of the new maze should be a reflection of <code>orig</code>.</li>
<li>Create an opening between the left and right halves at the middle row of the new maze.
<ul>
<li>Ensure that this opening is accessible from paths in both halves of your new maze by horizontal and vertical movements.</li>
<li>If needed, you may convert <code>WALL</code> locations to <code>PATH</code> locations, but only within the middle row.</li>
<li>Do not convert more <code>WALL</code> locations to <code>PATH</code> locations than necessary to ensure that the paths in the two halves are connected. This means you can't just change an entire row to PATH as a shortcut.</li>
<li>You may assume that all paths in <code>orig</code> are contiguous (i.e., no bubbles).</li>
</ul></li>
<li>Example:
<table>
<tr><th>orig</th><th>make_wider(orig)</th></tr>
<tr><td><pre>
XXXXX XXX
X       X
X XXX XXX
X X X   X
X X XXXXX
X       X
XXXXX XXX
</pre></td><td><pre>
XXXXX XXXXX XXXXX
X       X       X
X XXX XXXXX XXX X
X X X       X X X
X X XXXXXXXXX X X
X       X       X
XXXXX XXXXX XXXXX
</pre></td></tr>
</table></li>
<li>Do not modify <code>orig</code>.</li>
<li>Use your <code>malloc_maze(…)</code> to allocate the heap memory for the new <code>Maze</code> object.</li>
<li>Caller is responsible for freeing the new maze.</li>
<li>⚠Calling this function should not result in any calls to <code>fopen(…)</code> or <code>fclose(…)</code>, or <code>free(…)</code>.</li>
<li>In case of allocation failure, return NULL.</li>
</ul>
</td></tr>
<tr><td>test_maze.c</td><td>functions</td><td>
main(int argc, char* argv[])<br>
→ return type: int<br>
Test all of the six required functions in your maze.c.
<ul>
<li>Running your <code>main(…)</code> should cover all of your code, except for the parts that deal with allocation and file errors. (Testing error handling code is a standard practice, but takes more setup than we expect from you at this point.)</li>
</ul>
</td></tr>
<tr><td>expected.txt</td><td>output</td><td>Expected output from running your test_maze.c.</td></tr>
</table>

2. If you choose to reuse any of your functions from HW09, they should be converted to helper functions. That means the name should begin with `'_'` and they should not be listed in maze.h. (Hint: You probably won't need any of them.)
3. ⚠ Newline characters should not be stored in the `cells` field of `Maze` objects *in memory*.
4. ⚠ Make no assumptions 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.
5. ⚠ Do not print error messages or anything else on `stdout` (i.e., with `printf(…)`)—from your maze.c. Your test_maze.c may print to the terminal.
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 HW10, type `264submit HW10 maze.c test_maze.c expected.txt` from inside your hw10 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 hw10`).
>
> 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.

# Pre-tester ●

The pre-tester for HW10 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 hw10
> ```
>
> **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

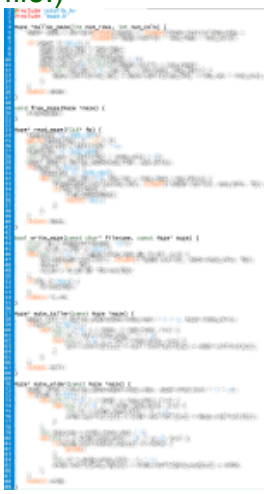
the tips given by the pre-tester.
- Code quality issues are not reported by the pre-tester; writing clean code is something you must learn to do from the start, not a clean-up step to do at the end.

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. **How much work is this?**
   Here's a tight solution, to illustrate the relative complexity of the functions in this exercise. Note that your `free_maze(…)` will most likely be longer. (cloc reports 83 sloc for the whole file.)



   Click to expand. In case of any discrepency, the requirements table above takes precedence over anything in this image. You may copy anything you see in this image.

2. **How should I approach `make_wider(…)`?**
   Suppose you start with the following maze:

```
XXXXX XXX
X       X
XXXXX X X
X     XXX
X XXXXX X
X       X
XXXXX XXX
```

   If you simply create the expansion in the manner used for `make_taller(…)` you would have this:

```
XXXXX XXXXX XXXXX
X       X       X
XXXXX X X X XXXXX
X     XXXX      X
X XXXXX X XXXXX X
X       X       X
XXXXX XXXXX XXXXX
```

Notice how you can't move between the paths in the left and right halves of this maze.

Adding a `PATH` to the middle row and middle column isn't quite enough because you would need diagonal movement to go between the two hemispheres.

```
XXXXX XXXXX XXXXX
X       X       X
XXXXX X X X XXXXX
X     XX XX     X
X XXXXX X XXXXX X
X       X       X
XXXXX XXXXX XXXXX
```

The solution is to start at that center cell, but then keep expanding to the left and right until there is a `PATH` immediately above or below, or to the left *and* right.

This is the correct result for the example above:

```
XXXXX XXXXX XXXXX
X       X       X
XXXXX X X X XXXXX
X     X   X     X
X XXXXX X XXXXX X
X       X       X
XXXXX XXXXX XXXXX
```

~~Note: Someone pointed out the "correct" result above is actually from the wrong example. I will fix later today. The principle is correct, but it's the wrong maze. [10/20/2017 11:15 AM]~~

3. **Is it okay if the result of `make_wider(…)` contains four openings (two on top, two on bottom)?**
   Yes.

4. **Why isn't the amaze utiity included in the starter code?**
   You can copy it from HW09.

5. **Why isn't there a starter maze.c?**
   By now, you should be comfortable creating a new C file from scratch. Be careful when writing the function signatures.

6. **Why do some parameters include const?**
   The `const` qualifier tells the compiler that you don't intend to modify those parameters, so that it can give you an error in case you accidentally do so. This page has a good explanation about `const`.

7. **How do I know if my tests in `main(…)` are comprehensive?**
   Make sure you have at least called each of the six functions in meaningful way. You can also

use the gcov tool (available on ecegrid) if you wish. gcov is optional.

8. **Are you going to test if my tests in `main(…)` are comprehensive?**
   We might. We are looking into this. If we do, we'll probably use gcov. Either way, if you don't test your code, it is virtually impossible to get it working. We added this requirement mainly as an impetus to get you focused on your testing. Even if we don't test the comprehensiveness (aka "test coverage") we will definitely test your `main(…)` in the way that was done for HW06.

9. **What is gcov?**
   Google it. The first result explains it as well as we can.

10. **What should my `main(…)` output?**
    It's up to you. Printing the contents of a maze before and after each operation is a possibility.

11. **Is there a utility function I can use to print mazes?**
    No, but you can easily make one using `fgetc(…)` and `fputc(…)`. This example from Prof. Quinn's 10/13 lecture might get you started. You may copy and adapt a few lines from that, but you wouldn't want to copy the whole thing.

12. **Is it okay if `main(…)` prints to the console?**
    Yes.

13. **What should be in my expected.txt?**
    This is just like HW02. An illustration of how to approach your `main(…)` and expected.txt can be found in this Piazza post.

14. **For `make_wider(…)` and `make_taller(…)`, does the middle cell need to lead to a path all the way to the maze opening?**
    No. You just need to connect some path cell on one side to a path cell on the other side. If you had some reasonable interpretation of this that was different from ours, feel free to let us know after scores are out. We will try to accomodate any reasonable interpretation that does not directly violate the specification.

15. **What arguments will you pass to my test code when uou call it??**
    None.

Don't see your question? Check the Q&A HW09 or try Piazza.

# Updates

10/22/2017 Removed the contradictory statement in write_maze function.
10/22/2017 Fixed the formatting of mazes in the description. Also updated the correct maze in Q&A 2.
10/22/2017 Added the fact that both width and height of a given maze are odd.
10/25/2017 Remove the hint on using malloc twice to allocate memory for maze.
10/27/2017 Added to QA. Clarified that test code may print to terminal.
11/26/2017 Added sloc from cloc for screenshot.