P2 - Battleship

Due Apr 1 by 11:59pm **Points** 50 **Submitting** a file upload **File Types** c
Available Mar 21 at 12am - Apr 30 at 11:59pm about 1 month

This assignment was locked Apr 30 at 11:59pm.

Learning Objectives

This assignment aims to practice dynamic memory allocation, traversing 2D arrays using address arithmetic, and file I/O in C.

Overview

The board game **Battleship** (https://en.wikipedia.org/wiki/Battleship_(game)) inspires this assignment. We will be implementing a small part of this popular game. We will be loading a save game file for one player. Verifying that the ships are valid as they are added to the board and then adding attacks made by the opponent to that player's board. Finally, we will determine if the player has lost or if the game is still in progress.

Files

- **battleship_template.c** (https://canvas.wisc.edu/courses/230411/files/19028873/download?download_frd=1) (rename this file to battleship.c after downloading)
- **battle1.txt** (https://canvas.wisc.edu/courses/230411/files/19028877/download?download_frd=1) **output1.txt** (https://canvas.wisc.edu/courses/230411/files/19028874/download?download_frd=1) (save game files and output after running the program)
- **battle2.txt** (https://canvas.wisc.edu/courses/230411/files/19028872/download?download_frd=1) **output2.txt** (https://canvas.wisc.edu/courses/230411/files/19028875/download?download_frd=1)
- **P2_Extra_Tests** (To use runtests.sh, you will need to edit the first line and replace it with the name of you battleship.c file. Note: in1 is battle1.txt and in2 is battle2.txt)

Results-Based Grading

The grading for this assignment is results-based. We provide a template file and strategy advice for completing the project; however, you may change anything in the template.

A key objective of this assignment is for you to practice using pointers. To achieve this, you are **not allowed** to use indexing to access the array that represents the battleship board. Instead, you are required to use address arithmetic and dereferencing to access it. You may not use brackets, e.g., writing "board[1][2]" to access square (1,2) of the game board. Submitting a solution using indexing to access the battleship board will result in a **50% reduction of your score**. You may use indexing to access any other arrays that your program uses.

Specifications

Board: We will be using a nonstandard game board for our project. Our game board is a rectangular board with R rows and C columns of characters. Please find the row and column sizes for each game in the battle.txt files. Note the upper left corner of the board is at position (0,0), and the lower right corner is at (R-1, C-1). For example, an empty board with dimensions 3 4 would look like:

```
….
….
….
```

Use malloc to reserve dynamically allocated memory on the heap after reading the size of the game board. You may either reserve a contiguous block of 1D memory then use math to map the 2D indexes to this block of memory. This first option is easier. Or you may reserve space for an array of pointers that each point to one row of the game board. We recommend you choose the option that makes more sense to you.

Remember to verify that the pointer returned by malloc is valid and to free the memory when you are finished.

The minimum board size is (1, 1). We are not providing the maximum board size that we will use for testing.

Ships: The ships are described in the save game file by four pieces of data. The first two numbers are the upper left coordinates of the ship. The third number is the length of the ship. And the fourth datum character is either 'V' or 'H', indicating the ship's orientation – either vertical or horizontal, respectively. Adding ship "0 1 3 V" to the board above would look like this:

```
.S..

.S..

.S..
```

The number of ships is provided in the battle.txt file for each saved game.

The template file provides a struct definition to hold the data for each ship. Using this struct is optional. We recommend using malloc to reserve space for an array of SHIP structs to store this data. Remember to free this memory when finished. If you prefer to approach storing this data in another way – you are welcome to ignore this struct definition

At most, we will test your code with 26 ships.

Attacks: The number of attacks is stored in the battle.txt file. Each attack has two pieces of data the row and column coordinates. For each attack, we will change the game board. If the attack hits a ship, we will replace the ‘S’ with the letter ‘H’. If the attack misses a ship, we will replace the ‘.’ with the letter ‘M’.

The template file provides a struct definition to hold the data for each attack. Using this struct is optional. We recommend using malloc to reserve space for an array of ATTACK structs to store this data. Remember to free this memory when finished. At most, we will test your code with R*C attacks.

Battle.txt: Below is a sample save game file decorated with descriptive comments that describe each line. Actual game files do not have these comments. Please see the provided battle.txt files

```
Board Size (Rows, Columns)  // ignore this line
4 6 // the number of rows and the number of columns separated by a single space
Number of Ships // ignore this line
1  // the number of ships
0 1 3 V  // ship 0: row column length orientation, separated by a single space
Number of Attacks  // ignore this line
2 // the number of attacks
0 1 // attack 0: row column, separated by a single space
0 2 // attack 1: row column, separated by a single space
```

After the series of attacks described in this file, the board will look like this:

.HM.

.S..

.S..

Assume all battle.txt files are correctly formatted.

Output: Your program will provide status updates on the game loading process in five stages.

Stage 1: **Reading Save Game File**. This stage logs the data read from the save game file as the file is read in. Please see the sample output file for the formatting of this data.

Stage 2: **Initializing Board**. This stage prints the empty board using ‘.’ to represent each space.

Stage 3: **Adding Ships**. This stage first verifies that the ships are valid in two steps. First, it confirms that each ship fits on the board and does not go out of bounds. If it does go out of bounds, it prints an error message and skips that ship. Then it verifies that the ships do not overlap with any already placed ships. If the ship tries to occupy a space that already has a ship, it prints an error message and skips the ship. Please see output2.txt for the exact text of these messages. If the ship is valid, it adds the ship to the board by replacing the ‘.’ with the letter ‘S’ for the ship's squares.

Stage 4: Processing Attacks. At this stage, the program goes through each of the attacks. If the attack is not valid, for example, attacking square (7,7) on a board that is only 3x5, the program prints an error message and skips this attack. For attacks that are in bounds, if the attack hits a ship, then the square is updated from ‘S’ to ‘H’ to represent the hit. Otherwise, the attack misses, and the square is updated from ‘.’ to ‘M’ for miss. Duplicate attacks are ignored with no error message.

Stage 5: Game_Over. Finally, the program analyzes the state of the game. If all of the ships have been sunk the player has lost and the message “All ships have been sunk - game over” is printed. Otherwise the message “Ships remain afloat - game on” is printed. If any square of the board contains an ‘S’ then the battle has not yet been lost.

Strategy

You are strongly encouraged to **develop your solution in phases**. For example, first code a solution that uses indexing. Once you have that solution working, you can replace indexing with pointer arithmetic before final testing and submission. If you use this approach, make sure to replace all accesses to your board use address arithmetic and dereferencing to avoid a penalty.

You are strongly encouraged to use incremental development to code your solution rather than coding the entire solution, followed by debugging that entire code all at once. Incremental development adds code in small increments. After each increment is added, test it to verify it works as desired before adding the next increment of code. Bugs in your code are easier to find since they are more likely to be in the new code increment rather than the code you have already tested.

Rename the template to battleship.c. Do your work in this file. Compile and run your program **battleship.c** using the save game file, battle1.txt and output redirection (> output1.txt – will send your printf statements to this file instead of printing them to the screen). You can then use diff to verify your output matches the sample output.

```
gcc -o battleship battleship.c -m32 -Wall
```

```
./battleship battle1.txt > myoutput1.txt
```

```
diff output1.txt myoutput1.txt
```

You may find it easier to write the code that reads the input file directly in main rather than writing a function to perform this task. We are providing function prototypes used when we developed our solution. You are welcome to use these or modify them.

Using library functions is something you will do a lot when writing programs in C. Each library function is fully specified in a manual page. The **man** command is very useful for learning the parameters a library function takes, its return value, detailed description, etc. For example, to view the manual page for **fopen**, you would issue the command **man fopen**. If you are having trouble using man, the same manual pages are also available online. You will need some of these library functions to write this program and will see that some of them are already used in our code. You do not need to use all of these functions since a couple of them are just different ways to do the same thing.

- **fopen()**to open the file.

- **malloc()**to allocate memory on the heap
- **free()**to free up any dynamically allocated memory
- **fgets()**to read each input from a file. fgets can be used to read input from the console as well, in which case the file is stdin, which does not need to be opened or closed. An issue you need to consider is the size of the buffer. Choose a buffer that is reasonably large enough for the input.
- **fscanf()/scanf():**Instead of fgets() you can also use the fscanf()/scanf() to read input from a file or stdin. Since this allows you to read formatted input you might not need to use strtok() to parse the input.
- **fclose()**to close the file when done.
- **printf()**to display results to the screen.
- **fprintf()**to write output to a file.
- **atoi()** to convert the input which is read in as a C string into an integer

Submission

- You will only be submitting battleship.c
- Verify that you are not using indexing, but instead using arithmetic addressing to access your 2D array
- Verify you are dynamically allocating the board on the heap with malloc
- You must format your print messages and your board output exactly as specified above and demonstrated in the sample output files.
- We will compile your programs with **gcc -Wall -m32** on the Linux lab machines. So, your program must compile there, and without errors.
- Please follow the same style guide as given in P1

Style (5 points)

1. Use meaningful variable names. Either use underscores for multi-word variables or CamelCase. Be consistent.
2. Be consistent with capitalization. For example, capitalize function names, use all caps for structs, and #defines, and use lower case for variables.
3. Indent to match scope. Use spaces instead of tabs.
4. Organize your code as follows:
    1. #include <>
    2. #include ""
    3. #defines
    4. Data Types (e.g., structures and typedef)
    5. Global variables – (use of global variables only when necessary)
    6. Function Prototypes
    7. Code - main() should be either first or last.
5. Comments
    1. Describe what each block of code is trying to do.

2. Good variable names may make this description easier.
3. For functions describe the purpose of the function and indicate the purpose of input parameters, output parameters, and the return value.

Project 2

Criteria	Ratings		Pts
0. Compilation	**6 pts** **Code compiles**	**0 pts** **Code does not compile**	6 pts
1. Execution test: battle1.txt (provided)	**8 pts** **Correct answer**	**0 pts** **Incorrect answer**	8 pts
2. Execution test: battle2.txt (provided)	**8 pts** **Correct answer**	**0 pts** **Incorrect answer**	8 pts
3. Execution test: 1x1 board no attack	**2 pts** **Correct answer**	**0 pts** **Incorrect answer**	2 pts
4. Execution test: 1x1 board with attack	**2 pts** **Correct answer**	**0 pts** **Incorrect answer**	2 pts
5. Execution test: no ships, with attacks	**2 pts** **Correct answer**	**0 pts** **Incorrect answer**	2 pts
6. Execution test: overlapping ships	**2 pts** **Correct answer**	**0 pts** **Incorrect answer**	2 pts
7. Execution test: regular ships and attacks	**2 pts** **Correct answer**	**0 pts** **Incorrect answer**	2 pts
8. Execution test: duplicate attacks	**2 pts** **Correct answer**	**0 pts** **Incorrect answer**	2 pts
9. Execution test: attacks everywhere	**2 pts** **Correct answer**	**0 pts** **Incorrect answer**	2 pts
10. Execution test: all ships sunk	**2 pts** **Correct answer**	**0 pts** **Incorrect answer**	2 pts

Criteria	Ratings		Pts
11. Execution test: all horizontal ships	**2 pts** **Correct answer**	**0 pts** **Incorrect answer**	2 pts
12. Execution test: horizontal out of bound	**2 pts** **Correct answer**	**0 pts** **Incorrect answer**	2 pts
13. Execution test: all vertical ships	**2 pts** **Correct answer**	**0 pts** **Incorrect answer**	2 pts
14. Execution test: vertical out of bound	**2 pts** **Correct answer**	**0 pts** **Incorrect answer**	2 pts
15. Execution test: attacks everywhere + a duplicate attack	**2 pts** **Correct answer**	**0 pts** **Incorrect answer**	2 pts
16. Execution test: large board	**2 pts** **Correct answer**	**0 pts** **Incorrect answer**	2 pts
			Total Points: 50