School of Information Technology and Electrical Engineering
CSSE2310— Semester 2, 2020
Assignment 1 (v1.1)
Due: 04 September 2020 23:59
Marks: 60
Weighting: 33% of your overall assignment mark (CSSE2310)

Introduction

Your task is to write a program (called `naval`) which simulates a naval battle between the player and an automated opposing side (called CPU in this document).

Both sides, the player and the CPU, have a grid board where their ships are placed. The player cannot see the positions of the automated player's ships and must make guesses to determine their position. Likewise, the automated player will try to guess the player's ship locations. To make the assignment simpler, the CPU player's 'guesses' will be read from a file specified on the command line. You do not need to create an AI algorithm for this assignment!

The ship positions for each player will be read from files specified on the command line.

Student conduct

This is an individual assignment. You should feel free to discuss **general** aspects of C programming and the assignment specification with fellow students, including on Piazza. In general, questions like "How should the program behave if ⟨this happens⟩?" would be safe, if they are seeking clarification on the specification.

You must not actively help (or seek help from) other students with the actual design and coding of your assignment solution. It is **cheating to look at another student's assignment code** and it is **cheating to allow your code to be seen or shared in printed or electronic form by others**. All submitted code will be subject to automated checks for plagiarism and collusion. If we detect plagiarism or collusion, formal misconduct proceedings will be initiated against you, and those you cheated with. That's right, if you share your code with a friend, **both of you are in trouble**.

Uploading or otherwise providing the assignment specification to a third party including online tutorial and contract cheating websites is considered misconduct. The university is aware of these sites and they do cooperate with us in misconduct investigations.

The course coordinator reserves the right to conduct interviews with students about their submissions, for the purposes of establishing genuine authorship. If you write your own code, you have nothing to fear from this process.

In short - **Don't risk it!** If you're having trouble, seek help early from a member of the teaching staff. Don't be tempted to copy another student's code or to use an online cheating service. You should read and understand the statements on student misconduct in the course profile and on the school web-site:

`http://www.itee.uq.edu.au/about_ITEE/policies/student-misconduct.html`

Specification

The program (`naval`) will take four filenames as parameters. The first file will describe the board and rules for the game. The second and third file describe the locations of the players and CPU ships respectively. The fourth file contain the positions that the CPU player will guess during their turn. Example usage is below.

```
./naval standard.rules player.map cpu.map cpu.turns
```

Spec clarification: These are *example file names* - your program should parse the commandline to get these names, and make no assumptions about them. If files `foo`,

`bar`, `baz` and `beep` exist in the current working directory, and have vaild contents as a rule, map and turns files respectively, then the following commandline would also be expected to work.

```
./naval foo bar baz beep
```

Rules file

To make things simpler to begin with, specifying `standard.rules` as the rules file will use the following rules (even if there is no file called `standard.rules`):

Spec clarification: If the first argument to your program is the string "standard.rules", then you may use hard-coded rules that are equivalent to the rules file listed below. This is purely to make it easier to get started creating your program. If you fully implement rules file parsing, then you don't need to worry about this because you can treat "`standard.rules`" like any other filename, just read it and work with it. Note there are additional functionality marks allocated to handling non-stnadard rules, so you should be aiming for this anyway.

- The grid will be 8 by 8 cells.
- There will be five ships in the following order:

Ship 1: 5 cells long.

Ship 2: 4 cells long

Ship 3: 3 cells long

Ship 4: 2 cells long

Ship 5: 1 cells long

The format for the rules file is as follows (all lines end in `\n`):

- Lines beginning with the '#' character are comments, and must be ignored
- The first line consists of two positive integers (`>0`) giving the width and height of the board.
- The next line consists of a single integer giving the number of ships for each player. You may assume there will not be more than 15 ships per player.
- The rest of the file consists of lines with a single integer describing the length of each ship.

So `standard.rules` could be written:

```
# 8x8 board
8 8
# 5 ships per player
5
# Here follow the ship lengths
5
4
3
2
1
```

Listing 1: Format of rules file

Please **do not commit a `standard.rules` file** to your subversion repository.

Map files

The two map files describing the locations of ships (called a map file) will each consist of lines describing the position and orientation of each ship.

- Lines beginning with `'#'` are to be ignored.
- all other lines shall have the following form (ending in `\n` only):

  `xy d`

  where `x` is a capital letter (starting at 'A') and `y` is positive integer (starting at 1) which give the column and row where the ship begins. So, the top-left space is 'A1'.

  The `d` is one of `N, S, E, W` indicating the direction the rest of the ship is relative to the start point. `N` (North) is up, `S` (South) is down, `E` (East) is right, and `W` (West) is left.

For example, on an 8x8 board, Listing 2 would represent the following map (see note below for exact on-screen representation).

```
A1 S
B2 S
E3 E
F6 N
H1 W
```

Listing 2: Sample map file

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**
1	1							5
2	1	2						
3	1	2			3	3	3	
4	1	2						
5	1	2				4		
6						4		
7								
8								

Ships are not permitted to overlap or extend beyond the grid (e.g. positioning the second ship at `A3 S` would be invalid)

CPU turns file

The CPU turns contains a list of positiosn which will be guessed by the CPU, one after another.

- Lines beginning with `'#'` are to be ignored.
- all other lines shall have the following form (ending in `\n` only):

  `xy` which have the same meaning as the coordinate system described for map files above.

```
# Here are the CPU player guesses
A1
C7
H3
# This one will probably be off the board
Z26
# This one is invalid, I hope my error checking works
A0
```

Listing 3: Sample CPU turn file

Notes on input handling

Specification clarification

In general, your program should be flexible and forgiving when reading input from files and `stdin`. This means, for example, handling extraneous white space between items, leading and trailing spaces, e.g.

```
# The following would be valid lines in a map file
A3     S
  B4 W
```

Listing 4: Example of flexible input

If you are expecting to read an integer, you should only accept **valid integers**. Malformed numbers like that shown below should cause an error to be generated.

```
# The following is an invalid board file, the integers are damaged
# 8x8 board
8 8blahblah
...
```

Listing 5: Example of bad board file input

Gameplay

The following describes the gameplay sequence

1. Display the CPU and player boards:
   - Print the CPU player's board. CPU ship locations are not revealed, only the locations of the player's previous guesses.
   - Print three 'equals' (=) signs
   - Print the player board, showing the player's ships locations as numbers, and the positions of previous CPU guesses
   - Previous guesses by player or CPU are show as misses ('\') or hits ('*')

Example gameplay and output

1. Display the CPU player board with no ships revealed and the row/column headers of the board. Then display the players board with ships revealed.

```
   ABCDEFGH
 1 ........
 2 ........
 3 ........
 4 ........
 5 ........
 6 ........
 7 ........
 8 ........
===
   ABCDEFGH
 1 1......5
 2 12......
 3 12..333.
 4 12......
 5 12...4..
 6 .....4..
```

```
7 ........
8 ........
```

Listing 6: Sample board output before any turns are made

Notes on board display:

- Locations of previous guesses by the player are displayed on the CPU player board as
  - '\' - misses
  - '*' - hits
- Empty or hidden locations are printed as a full stop ('.')
- On the player board, ships are displayed as a number from '1' to 'F' (hexadecimal), representing the order they were loaded from the map file. i.e. the ship specified on the first line of the map file will be displayed with the number '1'.
- Any hits by the CPU on player ship are displayed as '*',rather than the the ship number (see example gameplay below for clarification).
- Do not display the location of CPU misses.
- The column headings are capital letters from 'A' upwards. The column header has three leading spaces so that it lines up with the rest of the board grid.
- The row headings are numbered from '1' up to the number of rows.
  - Row numbers up to 10 are to have a leading space, e.g. ' 1', ' 2', '15'.
  - All row numbers shall have a single trailing space.
- The maximum board width is 26 (column 'Z').
- The maximum board height is 26.

2. Take the player's turn

   (a) Display the prompt:

   ```
   (Your move)>
   ```

   There is no space or newline following the prompt.

   (b) Read from standard input to get the player's guess.
   If the coordinates given match the location of a CPU player ship, then the program should print "Hit".
   If the coordinates do not match then print "Miss".
   If a user repeats a guess, output the message "Repeated guess", and reprint the prompt and read another guess.
   If the input is too big, too small or invalid in some other way, then the program should print "Bad guess", reprint the prompt and read another guess.

   (c) When the last cell of a ship has been hit, the program should print "Ship sunk".

3. If the CPU's last ship has been sunk the program should print "Game over - you win" and then exit with return code 0.

4. Take the CPU player's turn. The sequence for the CPU player is identical for the human player with the following exceptions:

   - The CPU's guess is read from the provided CPU turns file, one line at a time ignoring comments (lines starting with '#').
   - The CPU's guess is printed to the screen immediately following the prompt, e.g.

   ```
   (CPU move)>G5
   ```

- If the end of the CPU turn file is reached, the program should print “CPU player gives up” and then exit. See ”Error Codes” below for details of the required exit code.

Note that the CPU’s guess is error checked and reported exactly the same as the human player - ”Hit”, ”Miss”, ”Repeated guess” and ”Bad guess”. Repeated or bad guesses cause CPU’s turn to be repeated until a valid guess is made.

5. If the player’s last ship has been sunk the program should print “Game over - you lose” and then exit with return code 0.

6. If the game is not over go to Step 1

The following shows an example session. Note that in this example, the player and the CPU have the same board layout, e.g.

`./naval standard.rules test.map test.map cpu.turns`
This will not normally be the case!

```
   ABCDEFGH
 1 ........
 2 ........
 3 ........
 4 ........
 5 ........
 6 ........
 7 ........
 8 ........
===
   ABCDEFGH
 1 1......5
 2 12......
 3 12..333.
 4 12......
 5 12...4..
 6 .....4..
 7 ........
 8 ........
(Your move)>C1
Miss
(CPU move)>A1
Hit
   ABCDEFGH
 1 ../.....
 2 ........
 3 ........
 4 ........
 5 ........
 6 ........
 7 ........
 8 ........
===
   ABCDEFGH
 1 *......5
 2 12......
 3 12..333.
 4 12......
 5 12...4..
 6 .....4..
 7 ........
```

```
 8 ........
(Your move)>B3
Hit
(CPU move)>T12
Bad guess
(CPU move)>A1
Repeated guess
(CPU move)>A2
Hit
   ABCDEFGH
 1 ../.....
 2 ........
 3 .*......
 4 ........
 5 ........
 6 ........
 7 ........
 8 ........
 ===
   ABCDEFGH
 1 *......5
 2 *2......
 3 12..333.
 4 12......
 5 12...4..
 6 .....4..
 7 ........
 8 ........
(Your move)>H1
Hit
Ship sunk
(CPU move)>D4
Miss
   ABCDEFGH
 1 ../....*
 2 ........
 3 .*......
 4 ........
 5 ........
 6 ........
 7 ........
 8 ........
  ===
   ABCDEFGH
 1 *......5
 2 *2......
 3 12..333.
 4 12......
 5 12...4..
 6 .....4..
 7 ........
 8 ........


(Some time later...)
```

```
  ABCDEFGH
 1 ../....*
 2 **......
 3 **..***.
 4 **......
 5 **../*/.
 6 /.../*/.
 7 ........
 8 ........
  ===
  ABCDEFGH
 1 *......5
 2 *2......
 3 12..333.
 4 12......
 5 12...4..
 6 .....4..
 7 ........
 8 ........
(Your move)>A1
Hit
Ship sunk
Game over - you win
```

Listing 7: Sample gameplay

Upon game completion the program should exit with status 0. Please note that your program output must match this specification **exactly**. The functionality part of your assignment will be evaluated programmatically so deviation (even in matters like whitespace) is not acceptable. If your assignment cannot perform simple tasks it may not be tested for more complex functionality. For example if your program can read a map file but cannot display it we cannot determine if it has read the file correctly.

Error Handling

The table below describes how your program should respond to error conditions. The message should be printed to **standard error** and then the program should exit with the appropriate return code.

Under normal operation, your program should exit with status zero.

There should not be any combination of input which causes your program to crash or loop indefinitely (except of course if the user repeatedly guesses the same cells).

All files should be opened before either of them are processed (rules first, then map). If the standard rules are in use (and you are not reading a file) pretend that it opened successfully.

Extra parameters on the command line should be silently ignored.

Errors marked with '*' below indicate updates from a previous version of the spec

Error	Message	Exit status
Not enough parameters	Usage: naval rules playermap cpumap turns	10
Rules file is missing or unreadable	Missing rules file	20
Player map file is missing or unreadable	Missing player map file	30
*CPU map file is missing or unreadable	Missing CPU map file	31
*CPU turns file is missing or unreadable	Missing CPU turns file	40
Error processing rules file	Error in rules file	50
Ships overlap in player map	Overlap in player map file	60
Ships overlap in CPU map	Overlap in CPU map file	70
Ship out of bounds in player map	Out of bounds in player map file	80
Ship out of bounds in CPU map	Out of bounds in CPU map file	90
Other error processing player map file	Error in player map file	100
Other error processing CPU map file	Error in CPU map file	110
Error processing turns file	Error in turns file	120
Player input runs out before game is over	Bad guess	130
CPU input (moves) run out before game is over	CPU player gives up	140

Style

You must follow version 2.0.4 of the CSSE2310/COMP7306 C programming style guide available on the course BlackBoard site.

Submission

Your submission must include all source and any other required files (in particular you must submit a `Makefile`). Do not submit compiled files (eg `.o`, compiled programs).

Do not submit rules, board or CPU moves files (including standard.rules).

Your program must compile with:
`make`

Your program must be compiled under gcc with at least the following switches:
`-pedantic -Wall --std=gnu99`

You are not permitted to disable warnings or use pragmas to hide them.

If any errors result from the `make` command (i.e. the `naval` executable can not be created) then you will receive 0 marks for functionality (see below). Any code without academic merit will be removed from your program before compilation is attempted (and if compilation fails, you will receive 0 marks for functionality).

Your program must not invoke other programs or use non-standard headers/libraries.

Your assignment submission must be committed to your subversion repository under

`https://source.eait.uq.edu.au/svn/csse2310-sXXXXXXX/trunk/ass1`

where sXXXXXXX is your moss/UQ login ID.

Your submission will be assessed by the contents of your SVN repository at the due date/time. Any subversion commits after the due date will be ignored.

Spec update: Sorry I got this wrong first time. We will initialise your repositories with the correct structure. If you have already created the structure documented in an earlier version of this spec, don't panic. Please email `csse2310@helpdesk.eait.uq.edu.au` and we will help you sort it out.

You must ensure that all files needed to compile and use your assignment (including a Makefile) are committed and not just sitting in your working directory. Do not commit compiled files or binaries. You are strongly suggest encouraged to check out a clean copy for testing purposes.

The late submission policy in the CSSE2310 Electronic Course Profile applies. Be familiar with it.

Marks

Marks will be awarded for functionality and style and documentation.

Functionality (42 marks)

Provided your code compiles (see above), you will earn functionality marks based on the number of features your program correctly implements, as outlined below. Partial marks will be awarded for partially meeting the functionality requirements. Not all features are of equal difficulty. **If your program does not allow a feature to be tested then you will receive 0 marks for that feature**, even if you claim to have implemented it. For example, if your program can never open a file, we can not determine if your program can detect hits correctly. The markers will make no alterations to your code (other than to remove code without academic merit).

Marks will be assigned in the following categories.

1. Program correctly handles invalid command lines (2 marks)
2. With standard rules, ships horizontal (east) [16 total]
   - Correctly handle "hit" (2 marks)
   - Correctly handle "miss" (2 marks)
   - Correctly detect "sunk" (2 marks)
   - Correctly handle bad guess (1 mark)
   - Correctly play whole game (7 marks)
   - Correctly detect invalid maps (2 marks)
3. With standard rules, ships in any direction same breakdown as #2 [16 total]
4. Non-standard rules [8 total]
   - detect invalid map (1 mark)
   - detect invalid rules (1 mark)
   - Correctly play whole game (6 marks)

Style (8 marks)

Style marks will be calculated as follows: Let

- A be the number of style violations detected by `style.sh` plus the number of compilation warnings.
- H be the number of **additional** style violations detected by human markers. Violations will not be counted twice

If $A > 10$ (i.e. you have more than 10 automatic style violations), then your style mark is 0 and H will not be calculated. Otherwise, your style mark is broken into two parts:

- an automatic mark ($M_A = 4 \times 0.8^A$), and
- a human marker mark ($M_H = M_A - 0.5 \times H$) (i.e. your human mark is capped by your automatic mark and you lose 0.5 marks for each style violation detected by a human marker)

Your final style mark will then be

- $S = M_A + \max\{0, M_H\}$

The number of compilation warnings will be the total number of distinct warning lines reported during the compilation process described above. The number of style guide violations refers to the number of violations of version 2.0.4 of the CSSE2310 C Programming Style Guide.

A maximum of 5 violations will be penalised for each broad guideline area. The broad guideline areas are Naming, Comments, Braces, Whitespace, Indentation, Line Length and Overall. For naming violations, the penalty will be one violation per offending name (not per use of the name).

You should pay particular attention to commenting so that others can understand your code. The marker's decision with respect to commenting violations is final - it is the marker who has to understand your code. To satisfy layout related guidelines, you may wish to consider the `indent(1)` tool. Your style mark can never be more than your functionality mark - this prevents the submission of well styled programs which don't meet at least a minimum level of required functionality.

You are encouraged to use the '`style.sh`' tool installed on `moss` to style check your code before submission, however the marker has ultimate discretion – the tool is only a guide.

Documentation (10 marks)

PDF document containing a written overview of the architecture and design of your program (7 marks)

This document should describe, at a general level, the functional decomposition of the program, the key design decisions you made and why you made them. For example you might describe the approach you took to keeping track of user guesses, hits, and ship sinking, and the approach you took to reading map and rules files.

- Submitted via Blackboard/TurnItIn prior to the due date/time
- maximum 1 A4 page in 12 point font
- diagrams are permitted up to 25% of the page area. The diagram must be discussed in the text, it is not ok to just include a figure without explanatory discussion

Don't overthink this! The purpose is to demonstrate that you can communicate important design decisions, and write in a meaningful way about your code. To be clear, this document is not a restatement of the program specification - it is a discussion of **your** design and **your code**.

If your documentation obviously does not match your code, you will get zero for this component, and be asked to explain why.

SVN commit history assessment (3 marks)

Markers will review your SVN commit history for your assignment from the time of handout up to the due date.

This element will be graded according to the following principles

- Appropriate use and frequency of commits (e.g. a single monolithic commit of your entire assignment will yield a score of zero for this section)
- Appropriate use of log messages to capture the changes represented by each commit

Again, don't overthink this. We understand that you are just getting to know subversion, and you won't be penalised for a few "test commit" type messages. However, the markers must get a sense from your commit logs that you are practising and developing sound software engineering practices by documenting your changes as you go. In general, tiny changes deserve small comments - larger changes deserve more detailed commentary.

Total mark

Let

- F be the functionality mark for your assignment.
- S be the style mark for your assignment.
- D be the documentation mark for your assignment.

Your total mark for the assignment will be
$M = F + \min\{F, S\} + \min\{F, D\}$.

In other words, you can't get more marks for style or documentation than you do for functionality. Pretty code that doesn't work will not be rewarded!

Late Penalties

Late penalties will apply as outlined in the course profile.

Specification Updates

Any errors or omissions discovered in the assignment specification will be added here, and new versions released with adequate time for students to respond prior to due date. Potential specification errors or omissions can be discussed on Piazza or emailed to `csse2310@helpdesk.eait.uq.edu.au`.

v1.1

- Fix error in SVN repository structure instructions (Section "Submission")
- Clarify handling of `standard.rules` (Section "Rules file")
- Clarify interpretation of command line arguments and filenames (Section "Specification")
- Clarify general handling of input files and errors (Section "Notes on input handling")
- Update error message and exit code table (Section "Error Handling")
- Update style marking to include human-assessed style component (Section "Style (8 marks)")