Author: T. Pikies, 11/23/21

Ships

The aim of the project is to write a simple game Battleship. Then extend it, so it will become interesting. Consult https://en.wikipedia.org/wiki/Battleship_(game) for an overview. The rules are modified a bit so the progress made on basic version is not lost when the game is extended.

The goals to achieve are as follows:

1. Implementing the logic of base game:

All of the following functionalities should be implemented to obtain any points for the project.

a) [1pt] The board is of fixed size 21 x 10 - numbered from 0 to 20 in Y dimension and from 0 to 9 in X dimension. There are 2 players: A and B. First starts player A by default.
By default each of the player has exactly 1 ship of size 5, 2 ships of size 4, 3 ships of size 3, 4 ships of size 2; unless otherwise specified. The classes of ships are denoted by [CAR]RIER, [BAT]TLESHIP, [CRU]ISER, [DES]TROYER. The numbers of ships can be either default or specified. For simplicity assume that the number of ships of given type is bounded by 10.

b) [1pt] There is a possibility of placing the ships by the players. A can put ships in rows 0...9; B can put ships in rows 11...20. More complicated rules (i.e. about siding ships) are not required yet.

c) [1pt] The players can make moves, one after another, the only possible move is to either place the ships or shot one shot at a given position. The number of shoots does not depend on the number of ships present, each player can shoot once, wherever she want. The moves of the players in the consecutive turns are given by [playerX] command.

d) [1pt] The game can detect which players wins. The checking of victory condition (the opposing player has no healthy ship fragments left) should be done at the end of the turn.

e) [1pt] The game can print the state of the game. This is done by PRINT x command with x = 0. The printing should simply print a rectangle 21 x 10 with + marking a present ship and x marking a destroyed part of a ship. Assume that Y axis "grows down" and X axis "grows right".

The required validation is given in the last chapter.

2. Implementing the extended logic.

To implement the extended logic all the basic functionalities should be present. Observe, some of the functionalities are dependent on others. E.g. it will be hard to execute and verify the commands produced by A.I if printing the state of the game does not work. Or it will be hard to save the state of the game if SHIP is not implemented. Some, can be implemented only partially if others are not present e.g. extended printing.

a) [1.5pt] The board can be of arbitrary size. The player starting positions can be bounded into arbitrary rectangle. There can be specified reefs, i.e. forbidden fields on the map and no ship

under any circumstances can be present on such a field. The number of reefs specified can be quite large, but bounded by the size of board. Also, some of the ships of the players can be preallocated. Please make the size and allocation of the board truly dynamic.

b) [1.5pt] A player can make an order to move a ship. In each round each of the players can make any ship rotate, or move one field in its direction. Each ship except the carrier can move up to 3 times in a round. Carrier can move only up to 2 times. However, due to the fact that the ships are ships, they cannot rotate in place, and, for simplicity, are always going forward 1 field. Hence, in essence, after rotation the back of the ship is exactly one field further than the head was in the direction pointed before the rotation. Assume that the ships during a move cannot collide with reefs etc, despite the fact that the move is physically unfeasible.
E.g. for a destroyer changing course to the right twice.

```
#          #+++       #   +
#+#  ->    # #  ->    # # +
#+         #          #   +
#+         #          #
```

c) [1.5pt] Each of the ships contain 2 sections: cannon as a field after head, and engine at it's back. The size of ship determines the range of the cannon. For carrier it is infinite, for others it is the size of the ship calculated for the field containing the cannon, the field is not included.
The formula to calculate distance between points $p_1 = (y,x)$ and $p_2 = (y',x')$ is as follows:

$$dist(p_1, p_2) = sqrt(|y-y'|^2 + |x-x'|^2)$$

In essence, if there is a ship, where cannon is at position p_1 then a field p_2 is in the range if dist (p_1, p_2) is less than or equal to length of the ship.

If the engine is damaged the ship cannot move. If the cannon is damaged the ship cannot shoot. For each of the ships the player can specify in each round the number of shoots equal to the size of the ship.

d) [1.5pt] Each of the ships contains additional section, radar at it's head. The radar provides an information about positions of other ships, i.e. if there is a opposing ship in the range of the radar. If the radar is damaged, then the range of the radar is 1; otherwise it is the length of the ship. A field is in range if the distance (the same formula as for the cannon) between the field and the radar is at most the range of the radar. Of course, a ship knows it's position always. Also, carriers can send spy planes (in number up to equal to the number of shoots - 5), the planes revel presence of enemy ships in 3x3 square each. Sending a planes count as shooting. Printing from player's perspective should be implemented.

e) [1pt] Extended printing, in addition to the previous point the following things should be printed:
   a) engine as %
   b) cannon as !
   c) radar as @

   If the cannon and engine are taking the same field, then engine should be printed. Also, the numbers of rows and columns should be printed, e.g. as:

```
  00000000001
  01234567890
00
01
00 # @!++%
01
```

```
02   #%
03    @
```
Extended printing is also called by PRINT x command, with x = 1.

f) [1pt] Printing (Saving) the state of the game. The program should be able to print commands which allows to reconstruct the state of the game. Hence, there should be an option that program prints commands that fully specify the state of the game after the execution. The program should use no players command - it can only use [state] commands.
To give an example of desire output:

```
[state]
      BOARD_SIZE 21 10
      NEXT_PLAYER A
      INIT_POSITION A 0 0 9 9
      SET_FLEET A 1 1 0 1
      SHIP A 5 9 N 0 CAR 11111
      SHIP A 9 4 W 0 BAT 1011
      SHIP A 9 0 W 0 DES 11
      INIT_POSITION B 11 0 20 9
      SET_FLEET B 0 1 1 0
      SHIP B 11 4 E 0 BAT 1111
      SHIP B 11 7 W 0 CRU 011
[state]
```

Observe: no [player] command is used. The state of the game – what is the size of the board, where the players can place their ships, which player should go next, what are the positions and states of player's ships, what fleets they have – is preserved.

g) [2pts] Introducing the basic "A.I." (Artificial Intelligence):
   a) A.I. can position the ships on the board in accordance to the rules, it should do this randomly.
   b) A.I. can shoot at the fields randomly, but it avoids shooting at it's own units / reefs; moreover if an enemy ship can be shoot at it does so.
   c) A.I. can randomly move the ships.

In essence, the program should execute all commands given in the input. Then if the next expected player is A.I., it should dump the state of game and it should pass the execution to A.I. However, A.I. do not have to make any changes on the map, shoot at the ships, etc. It should only generate a sequence of commands, starting and ending with [playerX]. Then the execution should end. It should be possible to execute the generated commands without any modifications of them, and they should be correct for the state given to A.I.
Summing up:

1. The state of the game given to the player A.I. should be printed, as with SAVE.
2. Also, to test the functionality, generate in addition:
   ```
   [state]
   PRINT 0
   [state]
   ```
3. Then the actions of A.I. should be printed as [playerX] commands.
4. Again, to test the functionality, generate in addition:
   [state]
   PRINT 0
   [state]
   Hence after the execution a sequence of commands like:
   [state]

…
[state]
[state]
PRINT 0
[state]
[playerX]
…
[playerX]
[state]
PRINT 0
[state]
should be generated in fact. Also, to make the executions consistent please use the seed given by SRAND command.

3. Implementing advanced logic.

All the functionalities from extended logic should be implemented.

a) [1pt] Introduce back the basic shooting command. Observer that if you have a collection of ships and collection of targets, then you can try to match the ships with the targets. Check if the specified shoots are feasible with respect to ships and execute them. Also, due to the fact that the ships may move (and occupy exactly 3 different positions) and it is not specified at which time the ship shoots, you can assume that the ship would be shooting exactly when necessary. Matching (bipartite) seems to be a keyword here...

b) [1pt] A.I. should be able to execute some reasonable strategy. For example, to always move carrier out of the range of other ships, if possible; or to scout the map initially.

c) [1pt] Two A.I. should be able to play with each other. Prepare an environment to show that it is feasible.

Most likely, you will need an additional information to preserve execution of the strategy of A.I. However, still the program should run turn by turn. Hence, the A.I. should be able to generate and post an information which allows it to continue the strategy in the next turn.

4. Details of the interface:

The commands are grouped into two groups: the description of the state of the game, the description of a players move in a given turn. E.g. a group [playerA] [playerA] specifies the commands given by A in a single turn. The players are doing the turns alternately. The commands inside a group should be executed sequentially, e.g. if a ship moves, shoots, and moves, then the shoot should be done with respect to the first move.

The group of state commands starts with [state] and the commands from this group are provided until another [state] is encountered. There are

1. **Basic:**

1. PRINT x
   Print the state of the game. For x = 0 it is the basic version. The known fields should be printed as either: ‘+’, ‘x’, ‘ ’. ‘+’ means that the field is occupied by a ship. 'x' means that there is a destroyed fragment at the position. ' ' means that the position is empty. Also, print a line containing an information how many undestroyed parts of ships remain.

2. SET_FLEET P a1 a2 a3 a4
   The number of ships of consecutive classes (CARRIER ... DESTROYER) for the player P are a1 ... a4. If the command is not specified the default numbers 1, 2, 3, 4 apply.

3. NEXT_PLAYER P
   Sets an information that P should make move next.

2. **Extended:**

   1. BOARD_SIZE y x
      Set the board size to y times x

   2. INIT_POSITION P y1 x1 y2 x2
      The player P can place ships in rectangle given by (y1, x1) (left upper) and (y2, x2) (right lower). It means that the player can place ships in [y1, y2] in Y dimension and in [x1, x2] in X dimension - the edges are included.

   3. REEF y x
      There is a reef at position (y,x). It should be printed as #.

   4. SHIP P y x D i C a1...al
      There is a ship of player P at position (y,x) in the direction D(N/W/S/E) i-th of class C (CAR/BAT/CRU/DES). The state of the segments are a1 … al, where l is the length of the ship and 1 means that the part is not destroyed, 0 that it was destroyed. This command is unaffected by starting positions requirements. Keep in mind that there are no spaces between a1 and a2, …, al-1 and al.

   5. EXTENDED_SHIPS
      Turn the extended logic of ships. I.e. the ships are composed of the 2 sections, etc. Turning this on disables the default SHOOT operations by players.

   6. SET_AI_PLAYER P
      The player P becomes an A.I.

   7. PRINT x
      The printing with x = 1 should follows the advanced rules.

   8. SRAND x
      Set the seed of chosen random number generator to x. Please use this seed in your program to make the executions consistent.

   9. SAVE
      Save the commands that allows to recreate the state of the game. Please, save even the

default information. The order of saving should be: board size; which players is next; initial positions of A, sizes of fleets of A, placements of ships of A (in order of types and ids), then the same information for B; Also, if present, reefs(in the order given in the input), extended_ships, autoshooting, if A.I. was set. Also, please save the seed of random number generator, **but increased by 1,** if given.

3. **Advanced Logic:**

    1. AUTO_SHOOTING
    This command together with EXTENDED_SHIPS toggle back the basic SHOOT command and new SPY command. However, now using this basic command you can specify all the shoots. The program should check if the shoots and sending of spy planes are feasible. Here, the moments of definitions of shoots do not matter – they could be executed earlier or later than declared. It is only important if for a given collection of targets and positions of ships shooting at the targets were feasible at all. Do not bother how this functionality affects others (e.g. PRINT) – it is important that your program is able to verify if the shoots are feasible.

    2. INFORMATION P …
    Give the information to player P. This command is for the A.I. player to preserve strategy between consecutive turn. Define your own arguments for the command and use them to construct more advanced A.I.

The group of player commands starts and ends with [playerX], where X is either A or B. Commands are give until another corresponding [playerX] is given. In the **[player]** group you have:

1. **Basic Logic:**

    1. PLACE_SHIP y x D i C
    Place the i-th ship of class C at (y,x) with direction equal to D.

    2. SHOOT y x
    Shoot at the field (y,x). Shooting can only start if all the ships were placed.

2. **Extended Logic:**

    1. MOVE i C x
    Move the i-th ship of class C x ([F]orward, [L]eft, [R]ight)

    2. SHOOT i C y x
    Let one of the shoots of i-th class ship be at (y,x).

    3. SPY i y x
    Send a spy plane from i-th carrier at the position (y,x).

    4. PRINT x
    Print the state of the game as seen by the player. That is, print the state as seen using radars and spy planes. The reefs are known every time. If a field is empty print ' '. If a field contains a visible part of ship print it. If a field is unknown print '?'. For x = 0 use basic

printing. For x = 1 advanced one.

3. **Advanced Logic:**

    1. SHOOT y x
       Shoot at the field (y,x).

    2. SPY y x
       Send a spy plane at (y,x).

5. Validation:

Every time a problem is detected a communicate "INVALID OPERATION O SPECIFIED: X", where O is the TAG of operation and arguments, and where X is reason why it is invalid. After printing the execution should end. The texts in brackets are validation commands that should be printed.

1. Concerning the basic logic:
    1. PLACE_SHIP should verify that:
        1. the ships are in starting positions of players (NOT IN STARTING POSITION),
        2. the same ship is not added twice (SHIP ALREADY PRESENT),
        3. that it is not adding too many ships to a given class (ALL SHIPS OF THE CLASS ALREADY SET).
    2. SHOOT should verify that:
        1. the shoot is at a position in the board (FIELD DOES NOT EXIST),
        2. and that all ships that should be placed were already placed (NOT ALL SHIPS PLACED).
    3. Parsing of [playerA] ([playerB]) should check if the expected player is A (B) (THE OTHER PLAYER EXPECTED).
2. Concerning the extended logic and state commands:
    1. SHIP should verify that:
        1. the ship is not placed on reefs (PLACING SHIP ON REEF),
        2. the ship is not placed too close to other ship(PLACING SHIP TOO CLOSE TO OTHER SHIP),
        3. the same ship is not added twice (SHIP ALREADY PRESENT),
        4. that it is not adding too many ships to a given class (ALL SHIPS OF THE CLASS ALREADY SET).
    2. REEF should verify that is placed on board (REEF IS NOT PLACED ON BOARD).
3. Concerning the extended logic from layer’s perspective:
    1. PLACE_SHIP should verify also that:
        1. the ship is not placed on reefs (PLACING SHIP ON REEF),
        2. the ship is placed at least on tile away from other ship(PLACING SHIP TOO CLOSE TO OTHER SHIP).
    2. SHOOT should also verify that:
        1. the ship has not destroyed cannon (SHIP CANNOT SHOOT),
        2. the ship is not shooting too many shoots (TOO MANY SHOOTS),
        3. the ship is shooting in the cannons range(SHOOTING TOO FAR).
    3. MOVE should verify that:
        1. the ship has not destroyed engine (SHIP CANNOT MOVE),
        2. the ship is not moving too many times(SHIP MOVED ALREADY),
        3. the ship is not placed on reef(PLACING SHIP ON REEF),

        4. the ship not moves out of board (SHIP WENT FROM BOARD)
        5. the ship is not placed too close to other ships (PLACING SHIP TOO CLOSE TO OTHER SHIP).
    4. SPY should verify that:
        1. the ship has not destroyed cannon (CANNOT SEND PLANE),
        2. the ship is not sending too many planes (ALL PLANES SENT).
4. Concerning the advanced logic:
    1. SHOOT should verify that the shoots specified are feasible (SHOOTS ARE UNFEASIBLE).

6 General requirements:

- Use the language C. You can use the standard C libraries: stdlib.h (e.g. functions: atoi, srand, qsort etc.), stdio.h (e.g. functions: fscanf, fopen, gentline etc.), string.h (e.g. functions: strcmp, strcpy etc.). You can use objects cin, cout, cerr, clog and their counterparts from the standard library <iostream>.
- The project can be written with the use of object oriented programming paradigm, and C++ can be used for compilation.
- Configuration of the program should make it possible to easily change all parameters. Easy change is understood as modification of a constant in the program.
- The standard template library (STL), and libraries: <ifstream> , <string>, <sstream> etc cannot be used. In particular string is forbidden.
- All requirements from enauczanie Project instructions / General requirements (pdf) apply.

7. Final remarks

General tips:

- Do not assume silly things, e.g. that a ship will be always on field 0,0; or that there are 5 reefs, or that the size of a custom board is always 123x321. Or that the collision of ships will be at field (7,7). Please also write a general logic – so the program should work for arbitrary reasonable input.
- Do not mix malloc/free with new/delete.
- For arrays use delete[] operator if they are allocated by new[].
- If there is some error in the instruction or the instruction differs from the tests, then first clarify everything and only then start to code.
- Write the program defensively. Use checks / assertions to be sure that the data makes sense and that the parsing is working as required. Use the validation to your advantage.
- All output should be directed to stdout.

General warnings:

- Be warned that during the presentation of the code you have to present the code submitted to the STOS. Presenting a modified (with respect to the version submitted to STOS) code will result in obtaining up to 0 points for the assignment. A modification of the code is anything, like: adding (removing) [changing] instructions, comments, formatting of the code. Keep in mind that this is not an exhaustive lists. The code that you present has to be completely identical

to the presented on STOS and it is your task to check this. To give an example: for an extra comment or better formatting then in the version submitted to STOS you will be given 0pts.

- Also, if the code does not compile on STOS or it works on your computer but not on STOS, it is up to you to prove that it is correct. I.e. you should analyse why it does not work, check the differences of the compilation/execution and prove that the behavior of the compiler is not standard. Unless you do so, you are given 0pts, or the functionalities are not accepted, respectively. Of course, if the behavior is hard to analyse you can ask an instructor for a help -- but the rules of grading still apply. Also, you should write the code that conforms to a chosen C/C++ standard and does not demand any optional functionalities from some version of C++. To give an example of such functionality: variable length array.
- The tests on STOS are not directly related to the grade. I.e. if they fail, then it means that some functionality is not working; but if they pass, this does not guarantee that the functionality is working. You should be sure that your code is reasonable and works as intended.
- If you have implemented basic A.I., then you should prepare short data that will demonstrate that the program works. Be also well prepared to demonstrate it (so you will not take time of others for setting the environment).

Tests:

For the examples how input looks like and how output should looks like consider tests.

Basic ones:

- T1-T7 are testing basic placing of ships.
- T8 are testing placing of custom numbers of ships.
- T9-T12 are testing basic shooting.
- T13 is testing victory checking.

Advanced ones:

- T14-T20 are testing preassignment of ships, custom initial positions, reefs, more rules about placing the ships.
- T21-T25 are testing movements of ships.
- T26-T29 are testing extended shooting.
- T30, T31 are testing radar.
- T32 is testing extended printing.
- T33 is testing saving.
- T34-T36 are testing A.I. They collect only output from your program, they should always fail.

A sample input and corresponding output for basic mode:

```
[state]
SET_FLEET A 0 1 0 0
SET_FLEET B 0 0 1 1
[state]
[playerA]
PLACE_SHIP 6 0 N 0 BAT
[playerA]
[playerB]
PLACE_SHIP 16 0 N 0 CRU
PLACE_SHIP 16 2 N 0 DES
[playerB]

[state]
PRINT 0
[state]

[playerA]
SHOOT 17 0
[playerA]
[playerB]
SHOOT 7 0
[playerB]

[state]
PRINT 0
```

[state]

+
+
+
+

+ +
+ +
+

PARTS REMAINING:: A : 4 B : 5

+
X
+
+

+ +
X +
+

PARTS REMAINING:: A : 3 B : 4

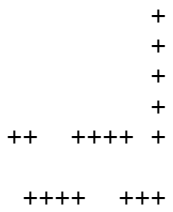
A sample input and corresponding output for advanced mode:

[state]
SET_FLEET A 1 1 0 1
SET_FLEET B 0 1 1 0
[state]

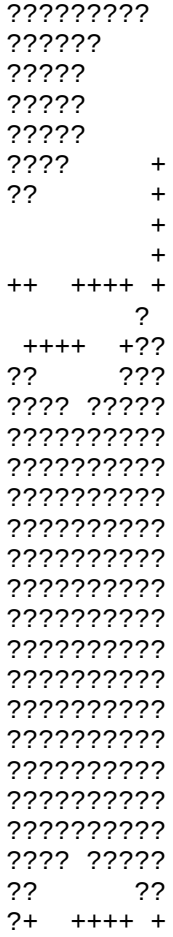
[playerA]
PLACE_SHIP 9 4 W 0 BAT
PLACE_SHIP 9 0 W 0 DES
PLACE_SHIP 5 9 N 0 CAR
[playerA]
[playerB]
PLACE_SHIP 11 7 W 0 CRU
PLACE_SHIP 11 4 E 0 BAT
[playerB]

[state]
PRINT 0

[state]
[playerA]
PRINT 0
[playerA]
[playerB]
PRINT 0
[playerB]



PARTS REMAINING:: A : 11 B : 7



?
 ++++ +++
?
?
??   ??
???? ?????
??????????
??????????
??????????
??????????
??????????