EAS 240: Introduction to Programming for Engineers

Individual Project: Battleship - Fall 2017

Assigned: Wednesday, March 22
Due: Friday, April 7 at 11:59 PM

1 Introduction

1.1 Project Overview

Battleship is a two-player guessing game.[1] Each player arranges a fleet of ships on a two-dimensional grid, while keeping their locations concealed from the opposing player. The players take turns calling attacks on each other's fleets and the first player to destroy the opponent's fleet wins the game.

In this project, you will create a simple one-player version of this game, where the player calls attacks on a fleet of ships at unknown locations on a one-dimensional grid. Although the game is conceptually simple, you will find that implementing it requires using almost every C programming concept that we have covered in lecture, including:

1. User and file input/output (I/O).
2. Control flow (`if-else` statements, `while` loops, `for` loops, etc.).
3. Functions, header files, and source files.
4. Arrays, strings, pointers, and dynamic memory allocation.
5. Structures.

1.2 C Learning Outcomes

While completing this project, you will learn the following about C programming:

1. How to use command-line arguments in your programs (Section 5.10 in K&R).
2. How to work with structures, structure pointers, and arrays of structures (Lecture 9 and Sections 5-5.4 in K&R).
3. How to create dynamically sized arrays of built-in and user-defined data types using dynamic memory allocation (Lecture 7).
4. How to divide a large program into small manageable parts.
5. How to test each part of a program individually.

[1] `https://en.wikipedia.org/wiki/Battleship_(game)`

1.3 Time Commitment

This project requires you to implement 7 functions of varying difficulty (including `main()`). Depending on your current proficiency with C, and how much time you need to review the lecture slides and textbook, this may take you anywhere from 4 to 40 hours. I recommend that you attempt to implement and test at least one function per day so that you have sufficient time to overcome any roadblocks that you encounter.

2 Preliminaries

2.1 Game board

Consider a one-dimensional game board of integer size `boardsize`. All ships in the fleet must sit entirely on the game board. You will number grid points on the game board starting at 0 and ending at `boardsize - 1`. You will assume that there are `numships` battleships on the board at the start of the game. The `boardsize` and `numships` variables will be initialized from a configuration file as described later in Section 2.4.

Note: There is no need to represent the board as an array. Just knowing the `boardsize` will be sufficient.

2.2 Battleship structure

In your program, you will represent a battleship as a structure with three member variables. Here is the structure's declaration

```
struct battleship {
        int *body;      /* pointer to an array */
        int size;       /* battleship size */
        int pos;        /* battleship position */
};
```

where:

1. `battleship` is the **structure tag**. The tag names the structure, and can be used to define battleship structure variables. For example, after the battleship structure is declared as above, then

   ```
   struct battleship ship;
   ```

   defines the variable `ship` as type `struct battleship`.

2. The **member variable** `size` defines the battleship's size. This represents the number of grid points that the ship occupies on the one-dimensional game board and the number of hits that it will take to be destroyed. If the variable `ship` is type `struct battleship`, then we can access its size as `ship.size`. The size of each ship will be determined at run-time (as opposed to compile-time) from a configuration file as described later in Section 2.4.

3. The **member variable** `pos` defines the battleship's position on the one-dimensional game board. If the variable `ship` is type `struct battleship`, then we can access its position as

`ship.pos`. The ship will occupy grid positions `ship.pos`, `ship.pos + 1`, ..., `ship.pos + ship.size - 1`. Therefore, if the game board has size `boardsize`, then valid ship positions satisfy `ship.pos` ≥ 0 and `ship.pos + ship.size` $\leq$ `boardsize`. The position of each ship will be determined at run-time (as opposed to compile-time) from a configuration file as described later in Section 2.4.

4. The **member variable** `body` is an integer pointer. If the variable `ship` is type `struct battleship`, then we can access this pointer as `ship.body`. We want `ship.body` to point to the first element of an array with `ship.size` elements, each representing a segment of the ship. However, since `ship.size` is not known at compile-time, we must dynamically allocate memory to the array. The statement

```
ship.body = (int *) malloc(ship.size * sizeof(int));
```

allocates a contiguous block of memory that is large enough to hold `ship.size` integers and sets `ship.body` to point to the start of the block. We can then treat `ship.body` as an array with `ship.size` elements, i.e., we may access `ship.body[0]`, `ship.body[1]`, ..., `ship.body[ship.size - 1]`. If no attack has landed on the `i`th segment of the ship, then `ship.body[i]` will have value 1; otherwise, it will have value 0, indicating that the segment is **damaged**. If all elements of `ship.body` are 0, then the ship is **sunk** (i.e., destroyed).

2.3 Fleet array

You will represent the fleet of battleships as an array of battleship structures. The number of ships will be determined at run-time (as opposed to compile-time) from a configuration file as described later in Section 2.4. Since the number of ships is not known at compile-time, we must dynamically allocate memory to the array. To do this, we first need to define a battleship structure pointer:

```
struct battleship *fleet;
```

Then, the statement

```
fleet = (struct battleship *) malloc(numships * sizeof(struct battleship));
```

allocates a contiguous block of memory that is large enough to hold `numships` battleship structures and sets `fleet` to point to the start of the block. We can then treat `fleet` as an array with `numships` elements, i.e., we may access `fleet[0]`, `fleet[1]`, ..., `fleet[numships - 1]`. Note that `fleet[i]` is now the `i`th battleship and we can access its member variables just like any other battleship structure variable, i.e., `fleet[i].size`, `fleet[i].pos`, and `fleet[i].body[j]` (this accesses the `j`th segment of the `i`th ship's body).

2.4 Game configuration file

As mentioned above, the board size, number of ships, ship sizes, and ship positions will be determined at run-time from a configuration file. An example configuration file is listed as follows:

```
boardsize: 20
numships: 2
ship0: 4 4
ship1: 2 13
```

This configuration file defines the board size to be 20, the number of ships to be 2, and then provides the size and position (in that order) of each of the two ships. The game board created by this configuration file is illustrated in Figure 1 and a detailed view of the body of each ship in the fleet is illustrated in Figure 2.

Another example configuration file is listed as follows:

```
boardsize: 50
numships: 4
ship0: 5 2
ship1: 4 10
ship2: 3 20
ship3: 2 25
```

This configuration file defines the board size to be 50, the number of ships to be 4, and then provides the size and position (in that order) of each of the four ships.

Note: When creating the configuration file, make sure that (i) the number of ships listed matches the number specified next to the numships label; (ii) all ships sit entirely on the board given the board size, their positions, and their sizes; and (iii) no ships overlap given their positions and sizes. To keep the program as simple as possible, you are not expected to provide any formal error checking for the configuration file; however, to avoid trouble when testing your code, be sure to follow the aforementioned guidelines.

2.5 Command-line arguments

You will use a command-line argument to specify the input file name. In particular, if your executable program is called `battleship.out` and your configuration file is called `config.txt`, then you will execute the program from the terminal as:

```
./battleship.out config.txt
```

Read Section 5.10 in K&R to learn how to use command-line arguments. The first and third example programs will be especially helpful. You should test both of these yourself to understand how they work before you attempt to add command line arguments to your own program.

2.6 Game play example

Below is an example of the game being played based on the first configuration file listed in Section 2.4, with its game board illustrated in Figure 1. A full description of what you need to do to create this program is provided in Section 3.

```
Loading config-example-1.txt...
Enter attack coordinate (0 - 19):
-1
Invalid attack coordinate.
Enter attack coordinate (0 - 19):
25
Invalid attack coordinate.
Enter attack coordinate (0 - 19):
10
```

```
Miss!
Enter attack coordinate (0 - 19):
13
Hit!
Enter attack coordinate (0 - 19):
12
Miss!
Enter attack coordinate (0 - 19):
14
Hit!
You sunk my battleship!
Enter attack coordinate (0 - 19):
4
Hit!
Enter attack coordinate (0 - 19):
5
Hit!
Enter attack coordinate (0 - 19):
6
Hit!
Enter attack coordinate (0 - 19):
7
Hit!
You sunk my battleship!
You sank 2 battleships in 8 turns!
```

2.7 Useful resources

Some useful resources can be found at the following links:

1. **Structures:** `https://www.tutorialspoint.com/cprogramming/c_structures.htm`
2. **Dynamic memory allocation:** `https://www.tutorialspoint.com/cprogramming/c_memory_management.htm`
3. **Command-line arguments:** `https://www.tutorialspoint.com/cprogramming/c_command_line_arguments.htm`

3 Assignment

3.1 What you need to do

You will use a divide-and-conquer strategy to complete this project. We have broken the problem into nine (9) functions (including `main()`). The first two (2) functions are provided for you. You will implement and test the remaining seven (7) functions and combine them together to complete the project. We recommend that you implement and test each function in the order that they are listed below.

3.1.1 shipState

`shipState()` is provided for you. It has the following function prototype:

```
int shipState(struct battleship ship);
```

- **Input parameters:** `shipState()` takes as input a battleship structure.
- **Return type:** `shipState()` returns an integer.
- **Functionality:** `shipState()` returns 1 if the ship is operational (i.e., at least one segment is not damaged), and 0 otherwise (i.e., if all segments are damaged).

3.1.2 shipInfo

`shipInfo()` is provided for you. It has the following function prototype:

```
void shipInfo(struct battleship ship);
```

- **Input parameters:** `shipInfo()` takes as input a battleship structure.
- **Return type:** `shipInfo()` does not return anything.
- **Functionality:** `shipInfo()` prints information about the ship.
- **Hint:** `shipInfo()` does not need to be used in the program, but will be helpful for you to test and debug other functions.

3.1.3 makeShip

`makeShip()` has the following function prototype:

```
struct battleship makeShip(int size, int pos);
```

- **Input parameters:** `makeShip()` takes as input the size and position of the ship.
- **Return type:** `makeShip()` returns a battleship structure.
- **Functionality:** `makeShip()` defines and initializes a battleship structure. It sets the battleship's member variables `size` and `pos` based on its input parameters; it allocates a contiguous block of memory that is large enough to hold `size` integers and sets the member variable `body` to point to the start of the block; it assigns 1 to every element of `body`, which can be treated as an array with `size` elements after it has been allocated memory; and, finally, it returns the initialized battleship structure variable.
- **Hint:** `makeShip()` requires you to work with a structure variable. Review K&R Sections 6-6.2 and Lecture 9 to remind yourself how to work with structures. Use `malloc()` for dynamic memory allocation.
- **Testing:** Use the main source file `test_1.c` to test the functionality of `makeShip()`. Study this source file carefully since it will teach you how to (i) work with the battleship structure and (ii) write your own simple code to test functions.

3.1.4 hitShip

`hitShip()` has the following function prototype:

```
int hitShip(struct battleship *ship, int attackpos);
```

- **Input parameters:** `hitShip()` takes as input a battleship structure pointer and an attack position/coordinate.
- **Return type:** `hitShip()` returns an integer.
- **Functionality:** `hitShip()` returns 1 if the attack on position `attackpos` hits a non-damaged segment of the battleship's body, and returns 0 otherwise. If a non-damaged segment is hit (i.e., a segment with value 1), then it should be set to 0 to indicate that it is damaged. Note that `hitShip()` takes as input a battleship structure pointer so that it can access and change the structure variable in the function that called it. This is necessary to ensure that any damage to the ship is properly recorded.
- **Hint:** `hitShip()` requires you to work with a structure pointer. Review K&R Section 6.2 and Lecture 9 to remind yourself how to access member variables through a structure pointer.
- **Testing:** Use the main source files `test_2.c` and `test_3.c` to test the functionality of `hitShip()`. Study these source files carefully since they will teach you how to (i) use the `hitShip()` function and (ii) write your own simple code to test functions.

3.1.5 fleetState

`fleetState()` has the following function prototype:

```
int fleetState(struct battleship *fleet, int numships);
```

- **Input parameters:** `fleetState()` takes as input a battleship structure pointer and the number of ships in the fleet.
- **Return type:** `fleetState()` returns an integer.
- **Functionality:** `fleetState()` returns 1 if at least one ship in the fleet is operational, and returns 0 otherwise (i.e., if all ships in the fleet are destroyed).
- **Hint:** Within `fleetState()`, you can access the `i`th battleship in the fleet as `fleet[i]` and determine if it is operational or destroyed with the function `shipState()`.
- **Testing:** Create your own source file to verify that `fleetState()` works correctly.

3.1.6 attack

`attack()` has the following function prototype:

```
void attack(struct battleship *fleet, int numships, int attackpos);
```

- **Input parameters:** `attack()` takes as input a battleship structure pointer, the number of ships in the fleet, and the attack position/coordinate.
- **Return type:** `attack()` does not return anything.
- **Functionality:** `attack()` checks to see if any ship in the fleet is hit by the attack at `attackpos`. If a ship is hit, then it prints `"Hit!\n"`; if the ship is destroyed by the hit, then it prints `"You sunk my battleship!\n"`; if no ship is hit, then it prints `"Miss!\n"`;
- **Hint:** Within `attack()`, you can access the `i`th battleship in the fleet as `fleet[i]` thanks to the close relationship between arrays and pointers. Use `hitShip()` to determine if a ship has been hit and use `shipState()` to see if a hit ship has been sunk/destroyed.
- **Testing:** Create your own source file to verify that `attack()` works correctly.

3.1.7 setupGame

`setupGame()` has the following function prototype:

```
struct battleship *setupGame(char *filename, int *boardsize, int *numships);
```

- **Input parameters:** `setupGame()` takes as input a character pointer and two integer pointers.
- **Return type:** `setupGame()` returns a pointer to an array of battleship structures.
- **Functionality:** `setupGame()` defines a battleship structure pointer named `fleet`; opens and reads the configuration file specified by the `filename` string; sets the caller's variables `boardsize` and `numships` using their respective pointers; dynamically allocates memory to the struct battleship pointer; initializes the array of battleship structures; and returns the battleship structure pointer.
- **Hint:** First, read the `boardsize` and `numships` values from the configuration file using `fscanf()`. Then, use `malloc` to allocate a contiguous block of memory that is large enough to hold `numships` battleship structures. Next, read the size and position of each battleship from the configuration file using `fscanf()` and use `makeShip()` to initialize `fleet[0]` through `fleet[numships - 1]`. Finally, return `fleet`.
- **Testing:** Create your own source file to verify that `setupGame()` works correctly.

3.1.8 playGame

`playGame()` has the following function prototype:

```
int playGame(struct battleship *fleet, int numships, int boardsize);
```

- **Input parameters:** `playGame()` takes as input a battleship structure pointer, the number of ships in the fleet, and the board size.
- **Return type:** `playGame()` returns an integer.

- **Functionality:** While the fleet is not sunk, `playGame()` prompts `"Enter attack coordinate (0 - %d):\n"`, where %d is replaced by `boardsize - 1`. It then reads the attack position from standard input using `scanf()`. If the attack position does not land on the board, it prints `"Invalid attack coordinate.\n"` and repeats the prompt. If the attack position is legal, then it calls the `attack()` function. Finally, after the fleet is sunk, `playGame()` returns the number of turns that the user took to sink the fleet (only count turns with attacks on legal board positions).
- **Hint:** Use `fleetState()` to determine if the fleet has been sunk/destroyed.
- **Testing:** Create your own source file to verify that `playGame()` works correctly.

3.1.9 main

`main()` begins by parsing the program's input arguments. If the program is not called correctly (either too many or two few input arguments), then print `"Usage: %s config-file\n"`, where %s is replaced by the program's executable file name. If the program is called correctly, then print `"Loading %s...\n"`, where %s is replaced by the configuration file's name. Subsequently, use the `setupGame()` function to initialize the fleet and the `playGame()` function to start the game. Finally, use `free()` to cleanup the dynamically allocated memory (free `fleet[i].body` for each ship in the fleet and then free `fleet`).

3.2 Files

The following files are packaged with this project description to help you complete the project:

1. `libbattleship.h`: Contains the prototypes of all the functions that you need to implement.
2. `libbattleship.c`: Contains the implementation of two functions: `shipState()` and `printInfo()`. You will implement the rest of the required functions in this file.
3. `battleship.c`: Contains an empty `main()` function. You will implement `main()` in this file.
4. `test_1.c`, `test_2.c`, `test_3.c`: Contains example code for you to test your `makeShip()` and `hitShip()` functions.

3.3 Submission guidelines

Before submitting your work to the automatic grading system, you must compress all of your files into a tar file. To do this, use the following command in the Linux terminal:

```
tar -cvf project1.tar battleship.c libbattleship.c libbattleship.h
```

This command will create the file `project1.tar` containing all of your header and source files. Submit `project1.tar` in the Project 1 area at `https://autograder.cse.buffalo.edu/`.

WARNING: If the tar file does not exactly match the format described above then the autograder system will not be able to grade your work and will assign you a 0 for every program. If this happens, then you will have to correct the file names and/or tar structure, create another tar file, and upload it again.

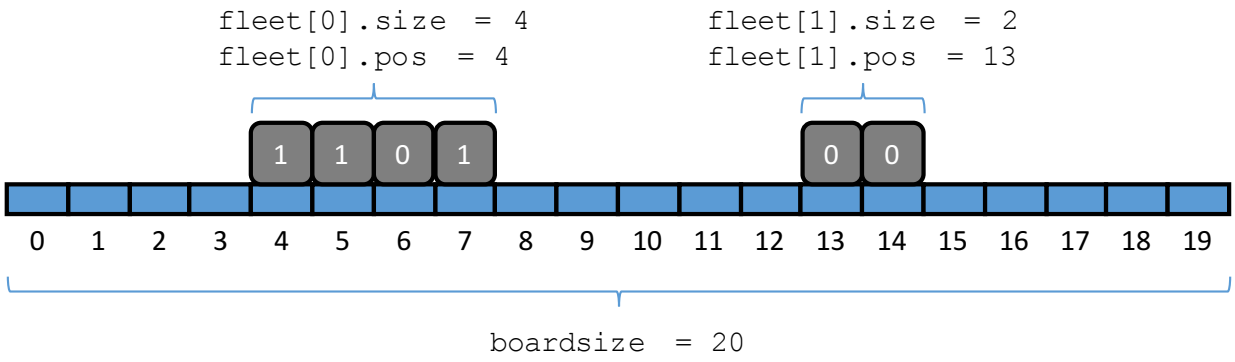

Figure 1: A fleet of 2 ships on a game board of size 20. The zeroth ship in the fleet (i.e., `fleet[0]`) has taken damage to its second body segment (i.e., `fleet[0].body[2]` is equal to 0). The first ship in the fleet (i.e., `fleet[1]`) has taken damage to both of its segments, therefore, it is sunk.

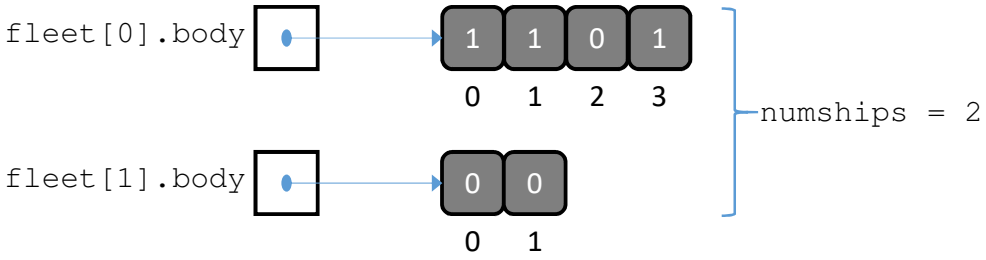

Figure 2: Detailed view of `fleet[0].body` and `fleet[1].body` from Figure 1.