ENGGEN 131 – Semester Two – 2023

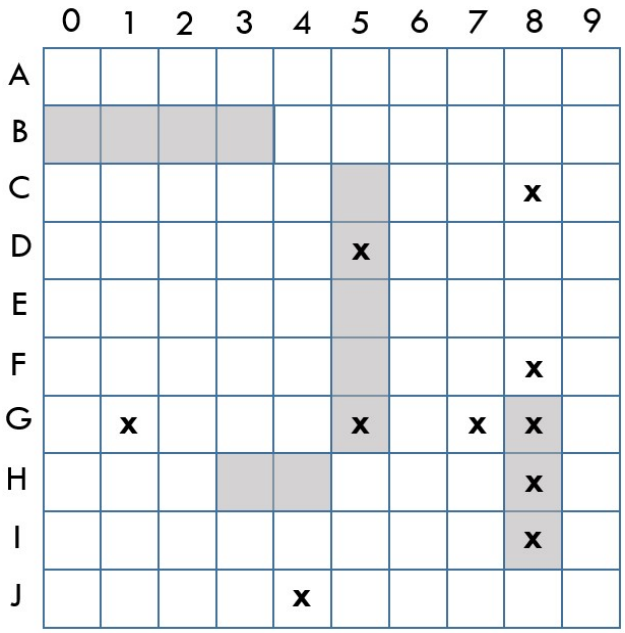
C Programming Project

BTLSHPS

Deadline: 11:59pm, Saturday 21st October
Worth: 10% of your final grade

A little background

Battleships is a classic strategy guessing-game for two players. The game is played using 2-dimensional grids, like the example shown on the right. Each player has their own fleet of "ships", represented in the example diagram as grey rectangles, which are placed on a map either horizontally or vertically. In the example, a fleet of ships for one of the players is shown, and the fleet consists of four ships of lengths: 2, 3, 4, and 5.



Players then take turns trying to guess the location of their opponent's ships. In the example, these guesses are indicated by the small '`x`'. A ship is said to be "hit" when the location of at least one element of the ship has been guessed correctly. When all elements of a ship have been guessed, then the ship is said to have been "destroyed". In the example, the ship of size 5 has been hit (twice) and the ship of size 3 has been destroyed. The goal of the game is for each player to destroy their opponent's ships before their own fleet is destroyed.

The earliest versions of the Battleships game were produced almost 100 years ago, and were played on paper using pen. In 1967, Milton Bradley (a game company) produced a version of the game that used plastic boards and pegs, and computer versions of the game have been around since 1979. As an interesting piece of trivia, the 2012 movie "Battleship" (based on the game, and featuring Rihanna in her acting debut) became the eighth biggest box-office flop in movie history. Not as bad as the Lone Ranger, but still bad.

In this project, you will be implementing – step by step – the required functionality for our own version of Battleships. There will be three play modes: human vs human, human vs bot, and bot vs bot. Once you have implemented the core functionality, you can design your own AI algorithms that determine how the bots play.

You can then have fun with all your friends: real and artificial!

An important note before we begin…

Welcome to the C Programming project for ENGGEN131 2023!

This project is organized around an interactive strategy guessing-game (Battleships) that can be implemented as a series of related tasks. When combined, the tasks will complete the implementation for the game. For each task there is a problem description, and you must implement *one required function* to solve that problem. In addition to the required function, you may define *other optional functions* which the required function calls upon (i.e. so-called *helper* functions). Using helper functions is often a useful way to organize your code, as several simpler functions can be easier to read and understand than one complex function.

Do your very best, but don't worry if you cannot complete every task. You will get credit for each task that you do solve.

IMPORTANT - Read carefully

This project is an assessment for the ENGGEN131 course. It is an **individual** project. You do not need to complete all of the tasks, but the tasks you do complete should be an accurate reflection of your capability and effort. You may discuss ideas in general with other students, but writing code must be done by yourself. *No exceptions.* You must not give any other student a copy of your code in any form – and you must not receive code from any other source (friends, students, AI language models, etc.) in any form. There are absolutely NO EXCEPTIONS to this rule.

Please follow this advice while working on the project – the penalties for academic misconduct (which include your name being recorded on the misconduct register for the duration of your degree, and/or a period of suspension from Engineering or expulsion from the University) are simply not worth the risk.

Acceptable	*Unacceptable*
• Describing problems you are having to someone else, either in person or on the Ed forum, without revealing *any* code you have written • Asking for advice on how to solve a problem, where the advice received is general in nature and does not include *any* code • Discussing with a friend, away from a computer, ideas or general approaches for the algorithms that you plan to implement (but *not* working on the code together) • Drawing diagrams that are illustrative of the approach you are planning to take to solve a particular problem (but *not* writing source code with someone else)	• Working at a computer with another student • Writing code on paper or at a computer, and sharing that code in any way with anyone else • Giving or receiving any amount of code from anyone else in any form • In short, there are no exceptions to the rule: you cannot share code with others.

Task One: "Let's *see* the map!"

The Battleships game involves 2-dimensional arrays, both for storing the locations of ships and for storing the information about where guesses have been made and whether or not those guesses are successful. When you are working on the initial stages of this project, it will be very useful to be able to visualize the values that are stored in memory – in particular, the values that are stored in the 2-dimensional arrays.

The program will always include a pre-defined constant called "MAP_SIZE" which represents the number of rows and columns in the 2-dimensional arrays. You do not need to define this constant – it has been defined for you. You can simply use "MAP_SIZE" in your program. For example, MAP_SIZE may be set to a value such as 10:

```
#define MAP_SIZE 10
```

For this task, you should write a function that prints to the screen all of the values in a 2-dimensional array (where the size of the array is determined by the MAP_SIZE constant).

You must write a function called `PrintArray()`:

```
void PrintArray(int values[MAP_SIZE][MAP_SIZE])
```

which takes a 2-dimensional array as input. The function should print all of the elements of the array to the screen, such that each value is separated by a single space.

To illustrate how this function should work, consider the `TestPrintArray()` function that calls the `PrintArray()` function twice using two arrays (`map1` and `map2`):

```
void TestPrintArray(void)
{
	int map1[MAP_SIZE][MAP_SIZE] = {0};
	printf("Map 1:\n");
	PrintArray(map1);

	int map2[MAP_SIZE][MAP_SIZE] = {0};
	for (int i = 0; i < MAP_SIZE; i++) {
		map2[MAP_SIZE-i-1][i] = i;
	}
	printf("\nMap 2:\n");
	PrintArray(map2);
}
```

In this case, MAP_SIZE is set to 6. If you have defined the `PrintArray()` function correctly, the output from this code would be as follows:

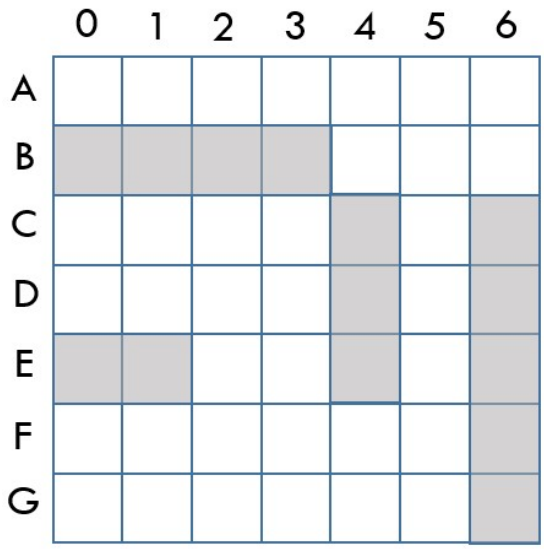
```
Map 1:
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0

Map 2:
0 0 0 0 0 5
0 0 0 0 4 0
0 0 0 3 0 0
0 0 2 0 0 0
0 1 0 0 0 0
0 0 0 0 0 0
```

Task Two: "Setting sail"

For some of the upcoming tasks, it will be useful to have a fleet of ships on the water so that you can test your functions. For this task, you will write a function that will place the four ships somewhere on the map. It does not matter where your ships are placed for this task, as long as you adhere to the following rules:

- A total of four ships must be placed within the grid, and these ships must have the sizes: 2, 3, 4 and 5.
- The ships must be placed entirely within the grid (so that no part of a ship is outside of the bounds of the grid).
- No element of a ship can be adjacent (either horizontally or vertically) to any element of another ship.
- When placing a ship, all elements of the ship's position should be set equal to the size of the ship.
- The placement of the ships should be valid for any value of MAP_SIZE between 6 and 26 (inclusive).

The diagram below illustrates one possible placement of the ships on the grid (in this example, MAP_SIZE is 7), and shows the values that should be stored in the array (note, the value 0 is used to indicate that there is no ship in the corresponding position).



	0	1	2	3	4	5	6
A	0	0	0	0	0	0	0
B	4	4	4	4	0	0	0
C	0	0	0	0	3	0	5
D	0	0	0	0	3	0	5
E	2	2	0	0	3	0	5
F	0	0	0	0	0	0	5
G	0	0	0	0	0	0	5

You must write a function called `InitialiseMap()`:

```
void InitialiseMap(int map[MAP_SIZE][MAP_SIZE])
```

which takes a 2-dimensional array as input, representing the map, and which modifies the values of the array to correspond to the placement of the four ships.

For example, consider the code below which calls the `InitialiseMap()` function:

```
void TestInitialiseMap(void)
{
     int map1[MAP_SIZE][MAP_SIZE] = {0};
     InitialiseMap(map1);
     printf("Map: \n");
     PrintArray(map1);
}
```

One possible output (although there are many valid outputs) is as shown below:

```
Map:
2 0 0 0 0 0
2 0 0 0 0 5
0 3 3 3 0 5
0 0 0 0 0 5
0 0 0 0 0 5
4 4 4 4 0 5
```

In this case, the `InitialiseMap()` function has been defined such that the position of ship "4" is along the bottom row and the position of ship "5" is along the rightmost column. You may choose to position your ships differently; this is just an example.

In this case, MAP_SIZE is set to 6. When explaining some of the upcoming tasks, this configuration of ships will be used as an example.

If MAP_SIZE was set to something different, for example 15, the output might look as below (although again, any valid configuration of ships on the board is acceptable for this task):

```
Map:
2 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 3 3 3 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 5
0 0 0 0 0 0 0 0 0 0 0 0 0 0 5
0 0 0 0 0 0 0 0 0 0 0 0 0 0 5
0 0 0 0 0 0 0 0 0 0 0 0 0 0 5
4 4 4 4 0 0 0 0 0 0 0 0 0 0 5
```

Task Three: "Very unpredictable"

At some point, we will want to randomly position the fleet of ships onto the map. This task takes the first step in that direction by randomly placing a *single* ship.

You must write a function called `AddRandomShip()`:

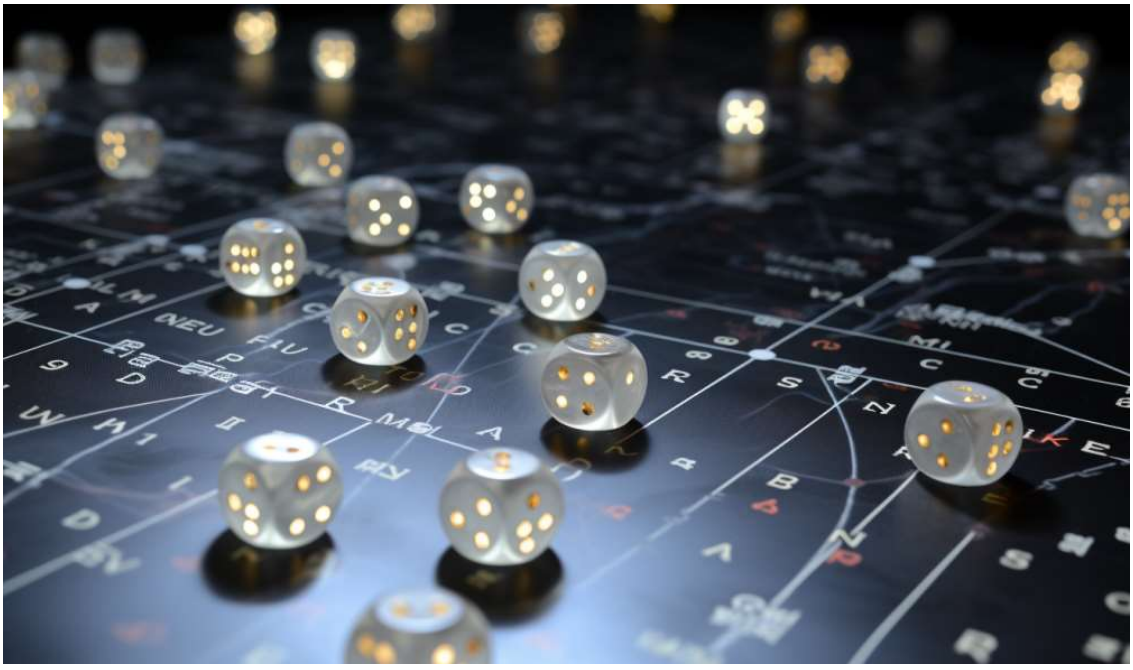
```
void AddRandomShip(int size, int map[MAP_SIZE][MAP_SIZE])
```

which takes two inputs: the size of the ship to add and a 2-dimensional array representing the grid. The function should randomly position a ship of the specified size within the grid. The direction of the ship (horizontal or vertical) should be randomly chosen.

NOTE: It should be possible for a randomly placed ship to cover any of the squares of the grid. In other words, if a very large number of randomly placed ships were generated (without checking for overlap with existing ships), every square of the grid should eventually be covered.

To generate random numbers, you should use the `rand()` function from the standard library. You should not call `srand()` to set the seed – this will be done for you.

Note, for this task, you do not need to check that the ship being added will be placed in a valid position with respect to any other ships already on the grid. You should simply generate the random direction and position for the ship, and then add it to the grid. If a sufficient number of ships were randomly placed on the grid by calling the `AddRandomShip()` function many times, then eventually every square of the grid would contain an element of a ship.



For example, consider the following code that tests the `AddRandomShip()` function:

```
void TestAddRandomShip(void)
{
     int map1[MAP_SIZE][MAP_SIZE] = {0};
     int map2[MAP_SIZE][MAP_SIZE] = {0};

     AddRandomShip(5, map1);

     printf("Map: \n");
     PrintArray(map1);

     AddRandomShip(2, map2);
     AddRandomShip(3, map2);
     AddRandomShip(4, map2);
     AddRandomShip(5, map2);

     printf("Map: \n");
     PrintArray(map2);
}
```

The output from this code might be as follows (in this case MAP_SIZE is 8):

```
Map:
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 5 5 5 5 5 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
Map:
0 3 3 3 0 0 0 0
0 0 0 5 4 4 4 4
0 0 0 5 0 0 0 0
0 0 0 5 0 0 2 0
0 0 0 5 0 0 2 0
0 0 0 5 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

Note in the case of the second map, the ships end up in an invalid position with respect to the rules explained in Task 2. However, this is not something that should be checked by the `AddRandomShip()` function – it should simply add a randomly placed ship to the grid.

Task Four: "And counting"

At some point it will be helpful to be able to count how many elements in a 2-dimensional array are equal to a given value.

You must write a function called `CountValues()`:

```
int CountValues(int value, int map[MAP_SIZE][MAP_SIZE])
```

which takes two inputs: a value to search for and a 2-dimensional array in which to search. The function should return the number of occurrences of the value in the array.

Consider the code below which calls the `InitialiseMap()` function to generate one valid placement of four ships on the map. The behaviour of this function was explained earlier in Task 2. Then, the code calls the `CountValues()` function several times:

```
void TestCountValues(void)
{
     int map[MAP_SIZE][MAP_SIZE] = {0};
     int count, shipSize;

     InitialiseMap(map);
     PrintArray(map);

     for (shipSize = 2; shipSize <= 5; shipSize++) {
          count = CountValues(shipSize, map);
          printf("The value %d appears %d times\n", shipSize,
                                                          count);
     }
}
```

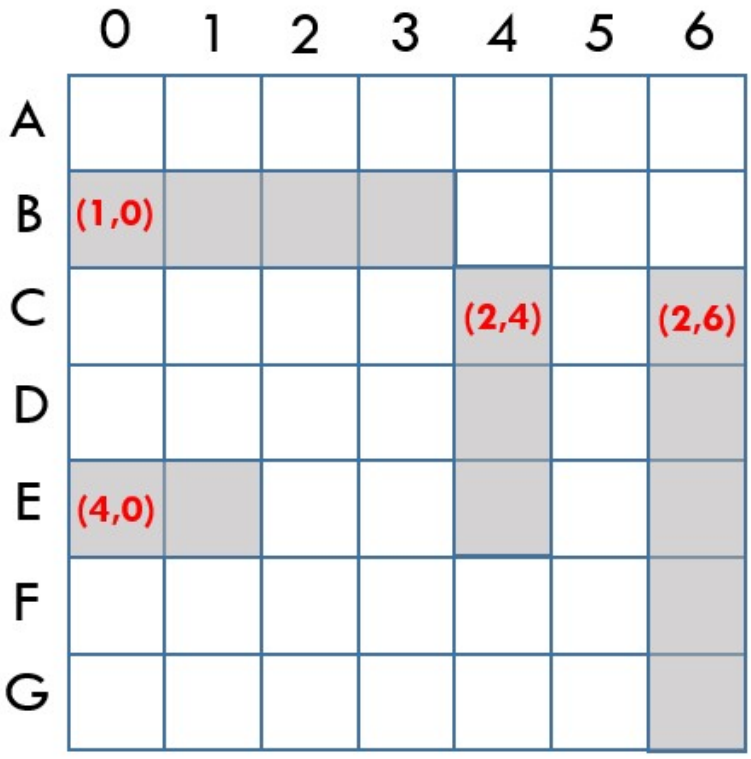
The output of this program should be as follows (in this example, MAP_SIZE is 8):

```
2 0 0 0 0 0 0 0
2 0 0 0 0 0 0 0
0 3 3 3 0 0 0 0
0 0 0 0 0 0 0 5
0 0 0 0 0 0 0 5
0 0 0 0 0 0 0 5
0 0 0 0 0 0 0 5
4 4 4 4 0 0 0 5
The value 2 appears 2 times
The value 3 appears 3 times
The value 4 appears 4 times
The value 5 appears 5 times
```

Task Five: "Top left"

It will be very useful to be able to locate a particular ship anywhere in the grid. Although ships are of a different size, one unambiguous way to describe the location of a ship is with the position of the upper most and left most element. In addition to this, if we know the size of the ship and its direction (i.e. horizontal or vertical), then we know exactly where it is.

For example, the diagram below labels the position of the upper most and left most elements of each ship:



You must write a function called `TopLeftPosition()`:

```
int TopLeftPosition(int size, int *row, int *col,
                                    int map[MAP_SIZE][MAP_SIZE])
```

which takes four inputs: the size of the ship being located, two pointers to store the row and column position of the upper most and left most element of the ship, and the 2-dimensional grid. The function should store the position of the upper most and left most element of the ship in the two pointers, and it should return direction of the ship (where 1 = horizontal; 2 = vertical)

For example, consider the code below that calls the `TopLeftPosition()` function:

```
void TestTopLeftPosition(void)
{
     int map[MAP_SIZE][MAP_SIZE] = {0};
     int row, col, direction, shipSize;

     InitialiseMap(map);
     PrintArray(map);

     for (shipSize = 2; shipSize <= 5; shipSize++) {
          direction = TopLeftPosition(shipSize, &row, &col, map);
          printf("Ship %d is at (%d, %d) facing %d\n", shipSize,
                                                row, col, direction);
     }
}
```

The output from this code is shown below:

```
2 0 0 0 0 0 0 0
2 0 0 0 0 0 0 0
0 3 3 3 0 0 0 0
0 0 0 0 0 0 0 5
0 0 0 0 0 0 0 5
0 0 0 0 0 0 0 5
0 0 0 0 0 0 0 5
4 4 4 4 0 0 0 5
Ship 2 is at (0, 0) facing 2
Ship 3 is at (2, 1) facing 1
Ship 4 is at (7, 0) facing 1
Ship 5 is at (3, 7) facing 2
```

For this task, you can assume that the ships will be in valid positions on the map (as specified by the rules that were explained earlier in Task 2).

Task Six: "Valid ships only"

At some point, we are going to want to create a randomly positioned fleet that is valid – in other words, four randomly placed ships that adhere to the rules outlined earlier in Task 2. One way to achieve this is to randomly place all four ships (using the function from Task 3) and then check to see whether each ship is in a valid position.

It will therefore be useful to be able to check whether a given ship, having been added to the grid using `AddRandomShip()`, is in a valid position with respect to the other ships on the grid.

You must write a function called `IsShipValid()`:

```
int IsShipValid(int size, int map[MAP_SIZE][MAP_SIZE])
```

which takes two inputs: the size of the ship being checked and the 2-dimensional grid. The function should return 1 (i.e. true) if the specified ship is valid with respect to all other ships, and 0 (i.e. false) otherwise.

You can assume that when this function is called, the ships will have been added to the grid using the `AddRandomShip()` function. Therefore, any configuration of ships that could be produced by making a series of calls to `AddRandomShip()` (in any order, and any number of times) are possible and should be checked.

For example, consider the code below which calls `IsShipValid()` where MAP_SIZE is 7:

```
void TestIsShipValid(void)
{
     int map[MAP_SIZE][MAP_SIZE] = {0};
     int valid, shipSize;

     InitialiseMap(map);
     PrintArray(map);

     for (shipSize = 2; shipSize <= 5; shipSize++) {
          valid = IsShipValid(shipSize, map);
          printf("Is ship %d valid? %d\n", shipSize, valid);
     }

     // Move Ship 3 to an invalid position
     map[2][3] = 0;
     map[2][0] = 3;
     // Move Ship 4 to an invalid position
     map[6][0] = 0;
     map[6][1] = 0;
     map[6][2] = 0;
     map[6][3] = 0;
```

```
    map[4][3] = 4;
    map[4][4] = 4;
    map[4][5] = 4;
    map[4][6] = 4;
    PrintArray(map);

    for (shipSize = 2; shipSize <= 5; shipSize++) {
        valid = IsShipValid(shipSize, map);
        printf("Is ship %d valid? %d\n", shipSize, valid);
    }
}
```

The output produced by this code is as follows:

```
2 0 0 0 0 0 0
2 0 0 0 0 0 0
0 3 3 3 0 0 5
0 0 0 0 0 0 5
0 0 0 0 0 0 5
0 0 0 0 0 0 5
4 4 4 4 0 0 5
Is ship 2 valid? 1
Is ship 3 valid? 1
Is ship 4 valid? 1
Is ship 5 valid? 1
2 0 0 0 0 0 0
2 0 0 0 0 0 0
3 3 3 0 0 0 5
0 0 0 0 0 0 5
0 0 0 4 4 4 4
0 0 0 0 0 0 5
0 0 0 0 0 0 5
Is ship 2 valid? 0
Is ship 3 valid? 0
Is ship 4 valid? 0
Is ship 5 valid? 0
```

In the upper map, the positions of all four ships are valid – therefore each time the `IsShipValid()` function is called, the value 1 is returned. Then, the ships are slightly modified which results in all ships being invalid:

- Ships 2 and 3 are adjacent
- Ship 4 overlaps Ship 5

At this point, every call to the `IsShipValid()` function returns 0.

Task Seven: "A random map"

Now that we can test whether a ship is in a valid location (e.g. not adjacent to another ship) using the `IsShipValid()` function, we can generate a random map using a simple algorithm:

- we continuously place four ships at random on a blank map until the map is valid.

You must write one function called `InitialiseRandomMap()`:

```
void InitialiseRandomMap(int map[MAP_SIZE][MAP_SIZE])
```

which takes one input: a 2-dimensional grid. The function should return a valid and randomly positioned fleet of ships.

For example, consider the code below that calls `InitialiseRandomMap()`:

```
void TestInitialiseRandomMap(void)
{
      int map[MAP_SIZE][MAP_SIZE] = {0};

      InitialiseRandomMap(map);
      PrintArray(map);
}
```

The code above may produce the following output (in this case, MAP_SIZE is 6):

```
4 4 4 4 0 0
0 0 0 0 0 0
2 0 0 0 0 3
2 0 0 0 0 3
0 0 0 0 0 3
5 5 5 5 5 0
```

If the code is run again, the output may change as follows:

```
0 0 3 3 3 0
0 0 0 0 0 0
5 5 5 5 5 0
0 0 0 0 0 0
2 0 4 4 4 4
2 0 0 0 0 0
```

Note that the positioning of the ships should be random – if we called this function a sufficient number of times, eventually every position on the grid would have been covered by an element of a ship at least once.

And here is one example of what the output might look like if MAP_SIZE is 15:

```
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  4  4  4  4  0  0  0  0  0  0  0  0  5  0
0  0  0  0  0  0  0  0  0  0  0  0  0  5  0
0  0  0  0  0  0  0  0  0  0  0  0  0  5  0
0  0  0  0  0  0  0  0  0  0  0  0  0  5  0
0  0  0  0  0  0  0  0  0  0  0  0  0  5  0
0  0  0  0  0  0  0  0  0  3  3  3  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  2  2  0  0  0  0  0  0  0
```

And here is an example of what the output might look like if MAP_SIZE is 24:

```
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  4  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  4  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  4  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  4  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  2  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  2  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  5  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  5  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  5  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  5  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  5  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  3  3  3  0  0  0  0  0  0  0
```

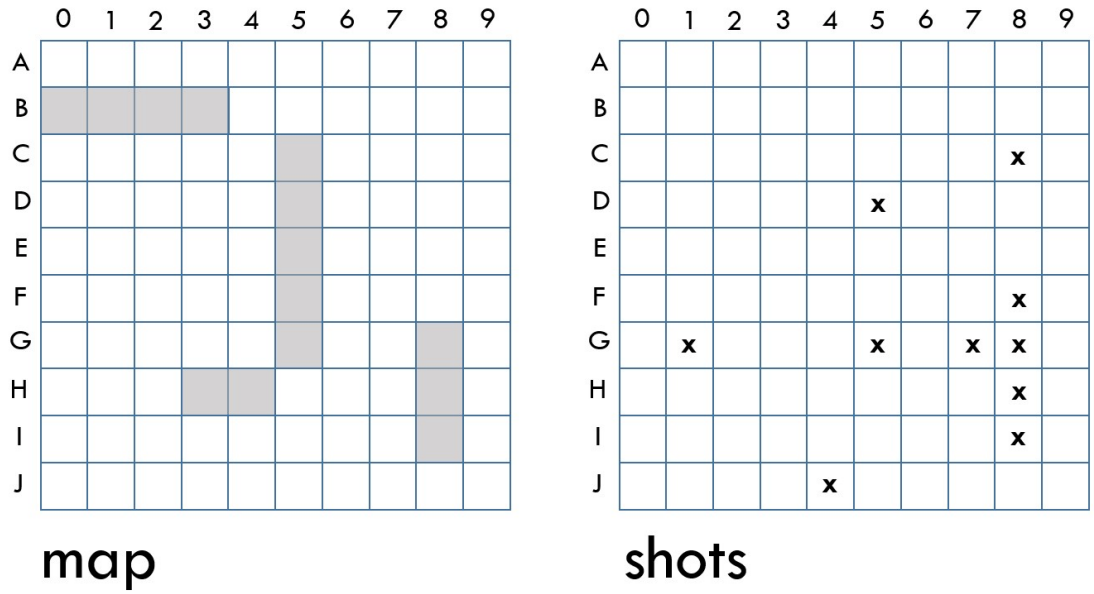
Task Eight: "Fire in the hole"

We can now implement one of the most important parts of the Battleships game: making a guess (i.e. firing a "shot") at an opponent's fleet.

When implementing this part of the game, we will be working with *two* arrays:

- one 2-dimensional array to store the locations of the ships
- one 2-dimensional array to keep track of the guesses/shots

The diagram below illustrates these two arrays – on the left is the map that records where the ships are located (we'll call this the "map" array). On the right is the grid that records the guesses/shots that have been made (we'll call this the "shots" array).



The "map" array does not change – it simply stores the locations of the ships.
The "shots" array updates every time a guess/shot is made. We would like to keep track of:

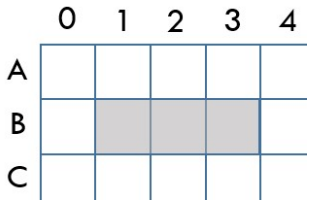
- The order in which the shots have been made
- Whether a ship has been hit (but not destroyed)
- Whether a ship has been destroyed

To do this, we will use the following encoding scheme:

- Every shot is numbered in the order that it is made, so the very first shot is "1", and subsequent shots are labeled "2", "3" and so on. *Any shot that is in the same position as a previous shot is ignored (i.e. we do not record it)*.
- When a shot misses any ship, then we simply record the number of that shot (i.e. its number in the sequence of shots)
- When a shot *hits* a ship (but the ship is not destroyed by that shot) then we record the number of the shot plus 1000 (i.e. if the number of the shot is 17, we would record 1017).

- When a shot *destroys* a ship, *all* elements of that ship are updated to record the fact the ship is destroyed. Each element of a destroyed ship is represented by the number of the shot (i.e. the order in which that element was hit) plus the size of the ship multiplied by 1000 (i.e. if a ship of size 2 is destroyed by shots 17 and 18, the corresponding elements in the "shots" array would record as 2017 and 2018).

Let's illustrate this encoding scheme with an example. Consider this board:



If shots were fired along row "B", starting from column 0, the values in the corresponding "shots" array would change as follows:

Shot: (1, 0)

	0	1	2	3	4
A					
B	1				
C					

Shot: (1, 1)

	0	1	2	3	4
A					
B	1	1002			
C					

Shot: (1, 2)

	0	1	2	3	4
A					
B	1	1002	1003		
C					

Shot: (1, 3)

	0	1	2	3	4
A					
B	1	3002	3003	3004	
C					

You must write one function called `FireShot()`:

```
void FireShot(int shots[MAP_SIZE][MAP_SIZE],
              int map[MAP_SIZE][MAP_SIZE], int row, int col)
```

which takes four inputs: a 2-dimensional grid for storing the guess (i.e. the "shots" array), a 2-dimensional grid representing the map (i.e. the "map" array), and the row and column position of the guess. The function should update the "shots" array based on the encoding scheme just described.

The example below makes this encoding scheme clear:

```
void TestFireShot(void)
{
      int map[MAP_SIZE][MAP_SIZE] = {0};
      int shots[MAP_SIZE][MAP_SIZE] = {0};

      InitialiseMap(map);
      printf("Map:\n");
      PrintArray(map);
      printf("Shots:\n");
      PrintArray(shots);

      FireShot(shots, map, 2, 0);
      printf("Shots:\n");
      PrintArray(shots);

      FireShot(shots, map, 2, 1);
      printf("Shots:\n");
      PrintArray(shots);

      FireShot(shots, map, 2, 2);
      printf("Shots:\n");
      PrintArray(shots);

      FireShot(shots, map, 2, 3);
      printf("Shots:\n");
      PrintArray(shots);

      FireShot(shots, map, 0, 0);
      FireShot(shots, map, 1, 0);
      FireShot(shots, map, 2, 0);
      FireShot(shots, map, 3, 0);
      FireShot(shots, map, 4, 0);
      FireShot(shots, map, 5, 0);
      printf("Shots:\n");
      PrintArray(shots);

}
```

The code above will produce the following output (in this case, MAP_SIZE is 6):

```
Map:
2 0 0 0 0 0
2 0 0 0 0 5
0 3 3 3 0 5
0 0 0 0 0 5
0 0 0 0 0 5
4 4 4 4 0 5
Shots:
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
Shots:
0 0 0 0 0 0
0 0 0 0 0 0
1 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
Shots:
0 0 0 0 0 0
0 0 0 0 0 0
1 1002 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
Shots:
0 0 0 0 0 0
0 0 0 0 0 0
1 1002 1003 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
Shots:
0 0 0 0 0 0
0 0 0 0 0 0
1 3002 3003 3004 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
Shots:
2005 0 0 0 0 0
2006 0 0 0 0 0
1 3002 3003 3004 0 0
7 0 0 0 0 0
8 0 0 0 0 0
1009 0 0 0 0 0
```

Task Nine: "Game over"

Finally, we can test whether or not the game is over by examining the "shots" and "map" arrays. We will write a function that checks the "shots" and "map" arrays for one player. The game will be over if all elements of all ships have been hit on the "shots" array – in that case, the other player will have won the game.

You must write one function called `CheckGameOver()`:

```
int CheckGameOver(int shots[MAP_SIZE][MAP_SIZE],
                  int map[MAP_SIZE][MAP_SIZE]);
```

which takes two inputs: the "shots" and "map" arrays. The function should return true only if all ships on that map have been destroyed.

Although in theory we could implement this function by examining only the "shots" array, this is only true if we know that there are four ships in a fleet. We should also check the "map" array in case the game is extended in the future to include a larger fleet of ships (i.e. more than four ships). When implementing this function, do not assume that the number of ships is four (you must examine the "map" array to see where the ships are actually located).

For example, consider the code below that sets up an initial map, then repeatedly takes random shots until `CheckGameOver()` returns true:

```
void TestCheckGameOver(void)
{
     int map[MAP_SIZE][MAP_SIZE] = {0};
     int shots[MAP_SIZE][MAP_SIZE] = {0};

     InitialiseMap(map);
     printf("Map:\n");
     PrintArray(map);
     printf("Shots:\n");
     PrintArray(shots);

     while (!CheckGameOver(shots, map)) {
          FireShot(shots, map, rand()%MAP_SIZE, rand()%MAP_SIZE);
     }

     PrintArray(shots);
}
```

The code above may produce the following output (in this case, MAP_SIZE is 6):

```
Map:
2 0 0 0 0 0
2 0 0 0 0 5
0 3 3 3 0 5
0 0 0 0 0 5
0 0 0 0 0 5
4 4 4 4 0 5
Shots:
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
2021 28 16 24 1 8
2025 3 5 2 19 5014
13 3018 3004 3020 0 5031
11 10 12 23 7 5026
6 17 0 29 22 5033
4030 4027 4034 4015 9 5032
```

If the code is run again, the output may change as follows (note that the shots are made in a different order this time):

```
2 0 0 0 0 0
2 0 0 0 0 5
0 3 3 3 0 5
0 0 0 0 0 5
0 0 0 0 0 5
4 4 4 4 0 5
Shots:
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
2010 29 32 24 14 6
2026 31 25 4 21 5008
20 3016 3001 3002 7 5033
0 9 23 11 18 5027
3 12 15 22 34 5035
4017 4019 4028 4030 13 5005
```

Task Ten: "The robots are coming"

Having completed the previous tasks, you can now integrate them into the `battleships.c` source file. This file contains additional code to support three game modes: human vs human, human vs bot, and bot vs bot.

To start with, two bots have been provided to you:

- A naïve bot which simply makes random guesses (and may repeat prior guesses)
- A (slightly less) naïve bot which generates one random guess, checks whether that is a repeat of a previous guess, and if it is then makes a second random guess

Here is the code for the first, and most naïve, bot:

```
void GetMoveBot1(int shots[MAP_SIZE][MAP_SIZE], int *row, int *col)
{
	int rand_row;
	int rand_col;

	rand_row = shots[0][0]; // to prevent compiler warning
	rand_row = rand() % MAP_SIZE;
	rand_col = rand() % MAP_SIZE;

	*row = rand_row;
	*col = rand_col;
}
```

It includes one statement which serves no purpose other than to avoid a compiler warning. Other than that, it chooses one row at random and one column at random and makes a guess. How much better do you think the second bot is compared to the first? You can measure this by playing the bots against each other in a tournament using the provided `battleships.c` file.

Try to think about some strategies which would make the bots more successful at playing Battleships. To test those strategies, you can implement them in the `GetMoveBot1()` and `GetMoveBot2()` functions. Once you have found your best strategy, you can use it to compete against two pre-defined bots in a tournament of 1000 games. If your bot wins a sufficient number of games in the tournament, then you win!

Resource files

You should begin by downloading the resource file from Canvas: "**Project2Resources.zip**".

Inside this archive you will find two source files:

- `project2.c`
- `battleships.c`

You should begin with "`project2.c`". This program contains a very simple `main()` function:

```
int main(void)
{
     // Initialise the seed for the random number generator
     srand((unsigned int)time(NULL));

     TestPrintArray();
     TestInitialiseMap();
     TestAddRandomShip();

     return 0;
}
```

Three minimal "Test" functions:

- `TestPrintArray()`
- `TestInitialiseMap()`
- `TestAddRandomShip()`

are provided to you to get you started. You should develop your own test functions for all of the tasks, using this structure as a model.

Skeleton function definitions for the required project functions are also provided to you. All of these function definitions are initially *incorrect*. You should correctly implement these functions.

When you have finished *all* of the tasks, you should place your function definitions into the "`battleships.c`" file. This program provides a more complete `main()` function, which implements the full game. Playing the game is a simple, robust and fun way to test your function definitions.

And you can also try to come up with some powerful bots!

And that's all there is to it - good luck and have fun coding!

ENGGEN131 C Project
Paul Denny
October 2023