Programming Evaluation

Module 2 – High-level Programming II

Purpose

This programming evaluation will give you some practice with structures, namespaces, references, dynamic memory allocation, and 2-dimensional arrays. It will also give you a chance to learn about interfaces and how to read source code.

Information

The task is to implement a very simple version of the popular board game *Battleship*. The player will place boats in an "ocean" that you create and will attempt to sink each one by taking shots into the ocean.

```
0   0   0   0   0   0   0   0        -1  -1  -1   0  -1  -1   0  -1

0   0   0   0   0   2   0   0        -1  -1  -1  -1  -1 102  -1   0

0   0   0   0   0   2   0   0        -1   0  -1  -1  -1 102  -1  -1

0   1   1   1   1   2   0   0        -1 101 101 101 101 102  -1  -1

0   0   0   0   0   2   0   0        -1  -1   0  -1  -1 102  -1  -1

3   3   3   3   0   0   0   0       103 103 103 103  -1  -1  -1  -1

0   0   0   0   0   0   0   0        -1  -1  -1  -1  -1  -1  -1  -1

0   0   0   0   0   0   0   0        -1  -1  -1  -1  -1  -1  -1  -1
```

An 8x8 board with 3 boats placed. **The board after sinking all 3 boats.**

Details

Instead of simply hard-coding the size of the board at 10x10, we are going to allow the player to specify the dimensions of the board. (The board is the ocean.) You should be able to handle boards of any rectangular size (square and non-square). You are also going to allow the player to specify the number of boats to place in the ocean. The only limit to the number of boats is the size of the ocean, since boats will not be allowed to overlap or leave the ocean. (Contrary to what Ferdinand Magellan's crew claimed, the world is flat, and if you go too far, you will fall off the end of it.)

In order to keep the implementation simple (we are studying new C++ techniques, after all), all of the boats will be the same size. Also, only the player will be taking shots, meaning that the computer will not be trying to find the player's boats. Again, we are studying C++ in this course, not artificial intelligence. (However, after completing this assignment, you will have most everything you need to create a more sophisticated game.)

Like all programs, there are many ways to implement this. For this assignment, you are given some header files to use as a starting point. These files give you the layout of the solution. The interface to the game is included in a header file named ***WarBoats.h***. This file contains all of the information that the client (the player) needs. You are also given a file named ***Ocean.h***, which defines what an Ocean is. A partial ***Ocean.cpp*** file is provided which includes the implementation of the display function called DumpOcean. (I want to be sure everyone's program prints the exact same thing and the only way to ensure that is if everyone has the same code.)

All of your implementation will be placed in ***Ocean.cpp***, and this will be the only source file that you will submit. You are not allowed to include any other files in ***Ocean.cpp***. There are a couple of files included already, but you will not need any others.

Provided Files

File	Description
Warboat.h	This is the interface file. It contains all enums, structs and function prototype that you will use or implement. This file will be included in all files that are using the enums, structs or prototyped functions.
Ocean.h	This file contains the Ocean structure which defines what an Ocean is. This file is included anywhere an Ocean instance is created and/or used.
Ocean.cpp	This is where you will be implementing all the required functions for the game to be played. This is the only file that you will be changing and submitting. This file is partially implemented, you just need to implement all the remaining functions that are prototyped in ***Warboat.h***.
PRNG.h PRNG.cpp	The PRNG files are used to generate random numbers. These files will be used by the drivers. You don't need to change anything in them. They are already included and used in the driver files. You just need to add them as part of the project.
driver_sample.cpp	The ***driver_sample*** file tests all of your implemented functions (the ones you are implementing in ***Ocean.cpp***). You need to pass all tests found in here first. Once these tests are passed, you will use the ***driver_big*** file to run some more tests.
driver_big.cpp	This file contains more rigorous tests that you need to run/pass after you are done with the initial tests found in ***driver_smaple.cpp***.

Code Details

There are only 5 functions that you need to write for this assignment, and three of them are fairly simple.

Functions listed in the interface
`Ocean *CreateOcean(int num_boats, int x_quadrants, int y_quadrants);`
The client calls this to create an ocean (game board). Client specifies the dimensions of the ocean and the number of boats that will be placed in it. An ocean will be dynamically created and a pointer to it will be returned to the client.
`void DestroyOcean(Ocean *theOcean);`
Client calls this to clean-up after the game. The function destroys an ocean that was created above by simply making sure that all memory that was allocated is de-allocated (deleted) properly.
`BoatPlacement PlaceBoat(Ocean &ocean, const Boat& boat);`
The client calls this to place boats in the ocean. The client will create a boat and pass it to the function. The function must ensure that the boat will "fit" in the ocean. Do this by checking the location where it is to be placed, the orientation (horizontal/vertical), and whether or not it will overlap with another boat or stick outside of the ocean. The return value indicates whether or not the boat could be placed in the ocean.
`ShotResult TakeShot(Ocean &ocean, const Point &coordinate);`
Client calls this in an attempt to hit one of the boats. The coordinate parameter indicates where the client is attempting to strike. There are several possible results: Hit, Miss, Sunk, Duplicate, or Illegal. Hit, Miss, and Duplicate are obvious. Sunk is returned when a shot hits the last undamaged part of a boat. Sunk implies Hit. Illegal is any coordinate that is outside of the ocean (e.g. x/y less than 0 or outside the range).
`ShotStats GetShotStats(const Ocean &ocean);`
Client calls this to get the current status of the game.

To get a feel for how these functions are supposed to work, you simply need to look at the code that is provided in the sample driver. This is the best way to see how they are used. All of these functions are called in the sample driver, some of them many times. The client (driver) will typically call ***CreateOcean*** and ***DestroyOcean*** only once, at the beginning and end of the game, respectively. The driver will call ***PlaceBoat*** once for each boat to place. However, if the driver tries to place a boat in an illegal location, you will return a value indicating it's an illegal location and the driver will try again with another location. Once the boats are placed, the player (driver) makes repeated calls to ***TakeShot***, which is the primary action during the game. The player can stop taking shots when all of the boats have been sunk.

The only functions that require non-trivial code are ***PlaceBoat*** and ***TakeShot***. The ***PlaceBoat*** function requires you to make sure that a boat can be legally placed at the specified position. You'll have to have some logic that checks to make sure the boat placement is legal. The ***TakeShot*** function requires you to validate the coordinates and then to determine the type of shot (e.g. hit, miss, duplicate, etc.)

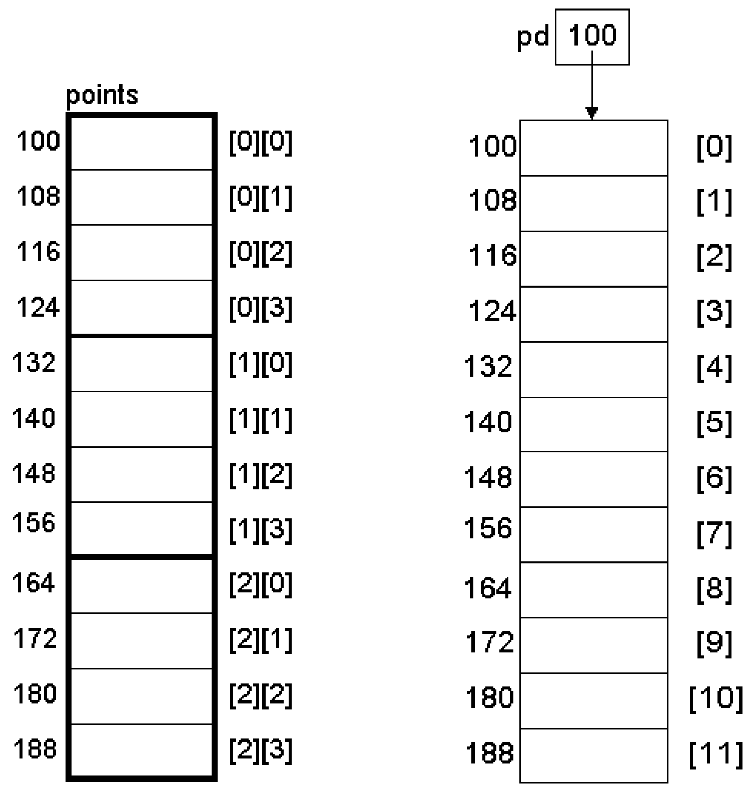
TakeShot function pseudocode

```
BEGIN FUNCTION

  IF the shot is out of the ocean (grid) THEN
    Return Illegal shot result
  END

  Read value at position of shot from grid

  IF value is OK THEN
    Increment misses
    Set position to Blownup
    Return Miss shot result
  END

  IF value is a Duplicate THEN
    Increment duplicates
    Return Duplicate shot result
  END

  Increment hits in the ocean
  Increment hits for this boat
  Update position to boat hit

  IF number of hits for this boat is >= BOAT_LENGTH THEN
    Increment sunk
    return Sunk shot result
  END

  Return Hit shot result

END FUNCTION
```

Frequently Asked Questions

Question	Can I modify the ***Ocean.h*** file and add more members?
Answer	No. Everything you need for this assignment is already there. Of course, there are many ways that you could implement this assignment, but I want everyone to work with the same structures. The only file that you are to modify is ***Ocean.cpp*** that I gave you. There's a comment in that file that shows you exactly where all of your code belongs. You should probably only need to write five functions in the file (one for each of the functions listed in ***WarBoats.h***).
Question	What do we do if the client asks us to create an ocean with five boats, and then calls ***PlaceBoat*** more than five times?
Answer	You don't have to worry about things like that. At this stage of the program, there are many things that the client can do that will cause bad things to happen. We will see later on how we can prevent many of those behaviors. For now, don't try to account for everything that could go wrong. If the client says there will be five boats, you can assume that's how many you'll get. Also, you will get the boat IDs from 1 to 5, which makes them easy to use as indices into the Boats array.
Question	The function ***CreateBoat*** is supposed to return a pointer. Isn't that going to be unsafe? Returning a pointer from a function means that the pointer is no longer defined.

Answer	It's safe to return a pointer, as long as the pointer is pointing at something that will still exist when the function returns. The handout says that an ocean will be created *dynamically* and a pointer to it will be returned. This means that you will use new to dynamically allocate the memory for the ocean. Your code will look something like this:
	
	`Ocean *CreateOcean(int num_boats, int x_quadrants, int y_quadrants)`
	`{`
	`  // Create an ocean dynamically`
	`  Ocean *ocean = new Ocean;`
	
	`  // Create the array for the boats dynamically (use []`
	`  // with new for arrays)`
	`  ocean->boats = new Boat[num_boats];`
	
	`  // Create the 2D array for the grid`
	`  ocean->grid = new int[x_quadrants * y_quadrants];`
	
	`  /*`
	`     Here, you must initialize all of the fields of the`
	`     structs being used (Ocean, Stats, Boats, grid, etc.)`
	`  */`
	
	`  // Return a pointer to the ocean`
	`  return ocean;`
	`}`

	Don't forget that everything that you allocated with new must be de-allocated with delete. This will be done in ***DestroyOcean***. (Don't forget the difference between delete and delete[].)
Question	If the **Ocean** is supposed to represent a 2-dimensional array, why is **Ocean.grid** a pointer to the first element of a single-dimensional array?
Answer	Long story short: It's easier. (This is because it's a *dynamic* array, not a static one, so you have no way of determining how many rows/columns there will be when you declare the ***Ocean.grid*** variable. Treating a 2D array as a 1D array is trivial. You just have to do the math (read: pointer arithmetic) yourself. <u>Example:</u> Compare a static 2D array called points with a dynamic 1D array. There are 3 rows and 4 columns. The address of each cell is displayed to the left and the subscript for that cell is displayed to the right. `double points[3][4];` `double *pd = new double[3 * 4];`  points 100 [0][0] 108 [0][1] 116 [0][2] 124 [0][3] 132 [1][0] 140 [1][1] 148 [1][2] 156 [1][3] 164 [2][0] 172 [2][1] 180 [2][2] 188 [2][3] pd 100 100 [0] 108 [1] 116 [2] 124 [3] 132 [4] 140 [5] 148 [6] 156 [7] 164 [8] 172 [9] 180 [10] 188 [11]

Given a row and column:

```
int row = 1, column = 2;
double value;
```

- The static 2D array can be accessed using two subscripts, but the dynamic "2D array" can only be indexed with a single subscript.

```
value = points[row][column]; /* OK      */
value = pd[row][column];     /* ILLEGAL */
```

- We (the programmers) have to do all of the arithmetic to locate an element using two subscripts:

```
value = pd[row * 4 + column];
```

- The compiler is still doing some of the work for us:

```
value = *(address-of-pd + (row * 4 + column) *
sizeof(double));
```

- The general form to access a cell in a 2D array (represented as a 1D array) where ROW is the row, COL is the column, WIDTH is the number of columns, and ARRAY is the name of the array is:

```
ARRAY[ ROW * WIDTH + COL ]
```

- So, to access row 2, column 1 in the example above:

```
pd[ 2 * 4 + 1 ] = pd[ 9 ]
```

which coincides with the diagram that shows point[2][1] (address 172) is the same as pd[9] (address 172). Sorry, but there's no magic here.

Testing your code

A "Grading Scripts" folder is provided in order for you to check if your implementation is correct. In order to run the tests follow these steps:

- Grab your **Ocean.cpp** file (which has all of your implementation) and place it in the "Grading Scripts" folder.
- Double click on the **"RunAll.cmd"** file.
    - All tests will run automatically. If a test fails, the script will stop at that test and you will get a message on the console that explains which test failed and why.

      ***NOTE:* If you are working on a DigiPen lab/classroom machine you need to run the "DigiPen-RunAll.cmd" file instead of "RunAll.cmd"**

What to submit

You must submit the CPP file (**Ocean.cpp**) in a single .zip file (go to the class Canvas page and you will find the submission link).

Do not submit any other files than the ones listed.