Programming Assignment: Battleship

Learning Outcomes

- Do things the C++ way in terms of structures and dynamic memory allocation/deallocation
- Namespaces
- References
- Two-dimensional arrays
- Reading, understanding, and interpreting C++ code written by others

Task

The task is to implement a very simple version of the popular board game Battleship. The client will place boats in an ocean that you create and will attempt to sink each one by aiming shots into the ocean. The picture on the left illustrates illustrates an ocean organized as a board consisting of 8×8 grid with 3 boats. Each boat occupies 4 squares on board and is identified by IDs 1, 2, and 3. The picture on the right illustrates the values in the board after the 3 boats have been sunk.

0	0	0	0	0	0	0	0
0	0	0	0	0	2	0	0
0	0	0	0	0	2	0	0
0	1	1	1	1	2	0	0
0	0	0	0	0	2	0	0
3	3	3	3	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

-1	-1	-1	0	-1	-1	0	-1
-1	-1	-1	-1	-1	102	-1	0
-1	0	-1	-1	-1	102	-1	-1
-1	101	101	101	101	102	-1	-1
-1	-1	0	-1	-1	102	-1	-1
103	103	103	103	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1

Instead of simply hard-coding the ocean size as 8×8 or 12×10, the ocean dimensions will be specified by the client. Your submission must handle both square and rectangular oceans of any size. The ocean size can only be limited by the amount of available computer memory. The client will also specify the number of boats to place in the ocean. Since boats will not be allowed to partially or completely be outside the ocean (contrary to the claims made by Ferdinand Magellan's crew that the world is flat and if they went too far, they would fall off the end of it), the number of boats is limited by the ocean's size.

Implementation Details

In order to keep the implementation simple (since we've just begun our C++ programming journey), all of the boats will be the same size. Also, only the client will be taking shots, meaning that the computer will not be competing to find the client's boats.

As with all programs, there are many ways to solve the problem. You are given header files to use as a starting point. These files give you the layout of the solution. The interface to the game and the *ocean* is in header file `ocean.h`. A partial `ocean.cpp` file is provided which includes the implementation of display function `dump_ocean`. We want to be sure everyone's program prints

the exact same thing and the only way to ensure that is if everyone uses the same random numbers. Files `PRNG.cpp` and `PRNG.h` generate such random numbers.

There are five functions that you need to write to complete - these are individually explained in the following sections.

Function `CreateOcean`

This function is called by the client to create an ocean and is declared as:

```
Ocean* CreateOcean(int num_boats, int hor_size, int vert_size);
```

First, dynamically allocate an object of type `Ocean`. The data members of this dynamically allocated object must be assigned appropriate values. Begin by looking at types `Ocean` and `Boat` to figure out the next steps:

```
struct Ocean {
  int *grid;        //!< The 2D ocean
  Boat *boats;      //!< The dynamic array of boats
  int num_boats;    //!< Number of boats in the ocean
  int x_size;       //!< Ocean size along horizontal x-axis
  int y_size;       //!< Ocean size along vertical y-axis
  ShotStats stats;  //!< Status of the attack
};

struct Boat {
  int hits;                 //!< Hits taken so far
  int ID;                   //!< Unique ID
  Orientation orientation;  //!< Horizontal/Vertical
  Point position;           //!< x-y coordinate (left-top)
};
```

In the `Ocean` object, dynamically allocate an array of `int`s with `hor_size * vert_size` of elements to simulate an ocean with dimensions $hor\_size \times vert\_size$. Set each position in the ocean to enumeration constant `DamageType::dtOK` so that the ocean is initially empty. Next, dynamically allocate memory for an array of `Boat`s with size `num_boats`. Clear `hits` and `ID` members of each `Boat` element to zero. Don't forget to clear data members of member `stats` to zero.

Function `DestroyOcean`

This function is called by the client to clean up after the game by deallocating all previously allocated memory and is declared as:

```
void DestroyOcean(Ocean *theOcean);
```

Use Valgrind to ensure there are no memory leaks in your program.

Function `PlaceBoat`

This function is declared as:

```
BoatPlacement PlaceBoat(Ocean& ocean, Boat const& boat);
```

The client will create a boat, specify a position to locate this boat in the ocean, and then call this function to record the boat's placement in the ocean grid. Think about all the things that would make a boat be invalid for placement: the boat's position may be outside the ocean; even if the boat's position is inside the ocean, the boat has a finite size (with length `BOAT_LENGTH` which is a constant variable defined in `ocean.cpp`) and a portion of the boat may lie outside the ocean; or, the boat's position(s) might be occupied by previously placed boat(s).

Given the ocean's dimensions, it is straightforward to determine if the boat's position is outside the ocean. For a boat of length `BOAT_LENGTH` to *fit* in the ocean, it must be completely inside the ocean and not overlap with another previously positioned boat. This can be checked using the boat's position and orientation. If the boat can fit in the ocean, determining whether these positions are already occupied by other boats is also straightforward. Straightforward because function `CreateOcean` has initialized the ocean's grid with value `DamageType::dtOK`.

The return value indicates whether or not the boat could be placed in the ocean. See `ocean.h` for valid values of type `BoatPlacement`.

Function `TakeShot`

This function is declared as:

```
ShotResult TakeShot(Ocean& ocean, Point const& coordinate);
```

The client will pick a position `coordinate` in the ocean to aim a shot at and then call this function to determine the result of the shot.

What are the values in the array of `int`s supposed to represent?

The values depend on what's in the ocean grid (that is, the array of `int`s) at a specific index.

1. When an `Ocean` is first allocated, to represent an empty ocean, every element in array `grid` must be cleared to `DamageType::dtOK`.
2. When a boat is placed in the ocean, its ID must be recorded in the array. Each boat has an ID from 1 to 99 (since 99 represents the maximum number of boats that can be placed). Every boat has length `BOAT_LENGTH` (defined in `ocean.cpp`). This means that boat $\#1$ will have four 1s in the ocean, boat $\#2$ will have four `2`s in the ocean array, and so on.
3. When a shot is taken that lands inside the ocean, the value at that array position must be updated:
    - If a boat is hit, meaning that the value at that array position is between `1` and `99`, increment the value by `HIT_OFFSET` (defined in `ocean.cpp`). This means that if boat $\#3$ was hit, the value `3` located at that index will be changed to value `3+HIT_OFFSET`. This allows you to easily see what the status of each boat is. If the value is between `1` and `99`, then that part of the boat has *not* been hit. If the value is between `1+HIT_OFFSET` and `99+HIT_OFFSET`, then that part of the boat has been hit.
    - If an open water position is hit, the value at that array position is changed from `DamageType::dtOK` to `DamageType::dtBLOWNUP`.

- if the open water position was previously hit, this is a *Duplicate* action and the value at that position (which is `DamageType::dtBLOWNUP`) is retained.
- If the same part of the boat is hit multiple times, this is a *Duplicate* action and the current value (which is `HIT_OFFSET` plus the boat's ID) is retained.

To recap, these are the valid values in the array of `int`s representing an ocean:

1. `DamageType::dtOK`: This is an un-hit portion of the ocean.
2. `DamageType::dtBLOWNUP`: This part of the Ocean has been hit.
3. A value between `1` and `99` means that this position represents a portion of a boat (with ID between `1` and `99`) that has NOT yet been hit.
4. A value between `1+HIT_OFFSET` and `99+HIT_OFFSET` means that this part of a boat has been hit. To figure out which boat has been hit, you simply subtract `HIT_OFFSET` from the value.

Algorithm `TakeShot`

There are several possible results: *Hit* (striking a boat), *Miss* (hitting an empty position), *Sunk* (a *Hit* that hits the final undamaged part of a boat causing the boat to sink), *Duplicate* (re-striking a previously hit open position or re-striking a previously hit boat position), or *Illegal* (`coordinate` specifies a position outside the ocean). See `ocean.h` for valid values of type `ShotResult`.

Algorithm TakeShot

Input: an Ocean O and a position $P(x, y)$
Output: $Hit,\ Miss,\ Sunk,\ Duplicate, Illegal$

1. **if** P is outside O **then**
2. return $Illegal$
3. **endif**

4. $V \leftarrow\ O.grid[P]$

5. [has the shot hit an open water position for the first time?]
6. **if** $(V == dtOK)$ **then**
7. increment $Miss$ count in O
8. $O.grid[P] = dtBLOWNUP$
9. return $Miss$
10. **endif**

11. [has shot re-hit either an open water position or a boat position?]
12. **if** $(V == dtBLOWNUP\ or\ V \in [1 + HIT\_OFFSET, 99 + HIT\_OFFSET])$ **then**
13. increment $Duplicate$ count in O
14. return $Duplicate$
15. **endif**

16. [there's a hit on an un-hit boat position]
17. increment Hit count in O
18. increment Hit count for boat with position $O.boats[V]$
19. $O.grid[P] += HIT\_OFFSET$

20. **if** $(O.boats[V]$ has sunk$)$ **then**
21. increment $Sunk$ count in O
22. return $Sunk$
23. **endif**

24. return Hit

Function `GetShotStats`

This function has declaration

```
ShotStats GetShotStats(Ocean const& ocean);
```

and returns the statistics from the free store object referenced by `ocean`. See `ocean.h` for definition of type `ShotStats`.

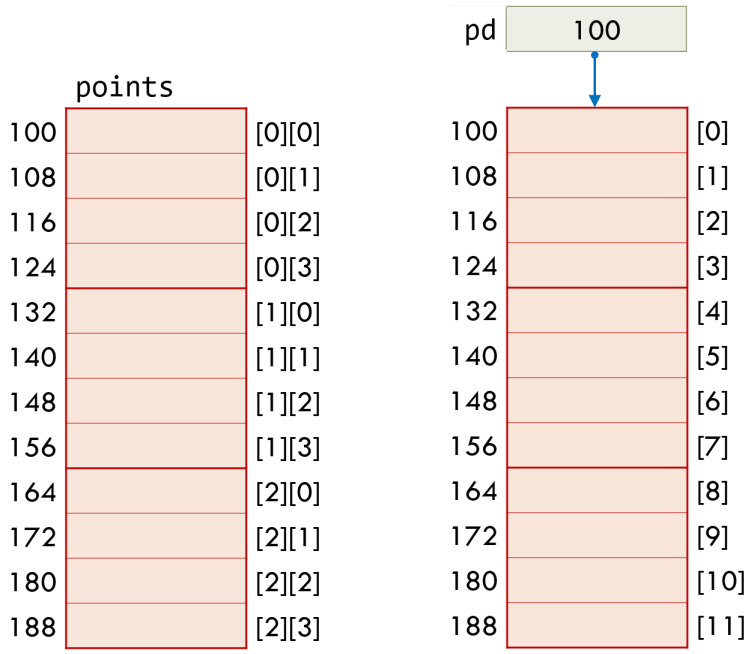
If *Ocean* is modeled as two-dimensional board, why is `Ocean::grid` a pointer to first element of one-dimensional array?

Long story short: It's easier. This is because we don't know until runtime the number of rows and columns and simulating a two-dimensional grid with a dynamically allocated one-dimensional array is trivial. Otherwise, the only other way is to dynamically allocate an array of pointers with the array size equivalent to the number of rows and then dynamically allocating an array - one per row - to represent the number of columns.

What is the math to represent a two-dimensional array with a one-dimensional array? Although we visualize two-dimensional arrays as tables and grids, that's not the way they're actually stored in memory. C and C++ store arrays in row-major order, with row 0 first, then row 1, and so forth. Compare a statically allocated two-dimensional array of `double`s with 3 rows and 4 columns called `points` to a dynamically allocated array `pd` of 12 `double`s:

```
1  double points[3][4];
2  double *pd = new double [3 * 4];
```

The following picture shows the memory layout of the two arrays with the address of each array element displayed to the left and the subscript for that element displayed to the right.



Given a row and column:

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

the static two-dimensional array can be accessed using two subscripts, but the dynamic *two-dimensional array* can only be indexed with a single subscript:

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

Programmers have to do the arithmetic to locate an element using two subscripts:

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

The general form to access an element in a two-dimensional array that is represented as one-dimensional array where `ROW` indicates the row, `COL` indicates 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, the following expression is required

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

which coincides with the picture showing both `point[2][1]` and `pd[9]` are located at address 172.

Submission Details

Please read the following details carefully and adhere to all requirements to avoid unnecessary deductions.

Header file

There is no header file to be submitted.

Source file

Your implementation must be placed in `ocean.cpp`, and this will be the only file that you will submit.

Compiling, executing, and testing

Download `ocean-driver.cpp`, `PRNG.cpp`, `PRNG.h`, (these two files are for random number generation), `makefile` (an introduction to *make* is available on the module web page), six correct output files `test?.txt`.

Testing program through repeated commands

Create the executable program by compiling and linking directly on the command line:

```
$ g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror ocean.cpp ocean-driver.cpp PRNG.cpp -o ocean.out
```

In addition to the program's name, function `main` is authored to take two additional command-line parameters. These parameters specify a character between `1` and `6` [each integer representing a specific unit test] and the second parameter specifies the name of the output file containing the messages from the test. To execute the first test, run the program like this:

```
$ ./ocean.out 1 your-test1.txt
```

Compare your submission's output with the correct output in `test1.txt`:

```
$ diff -y --strip-trailing-cr --suppress-common-lines your-test1.txt
test1.txt
```

If the `diff` command is not silent, your output is incorrect and your code must be debugged.

You'll have to repeat this process for the other five tests. Even if your program's output matches the correct output, it is possible that you just got lucky. Read the next section for additional debugging of your program that will allow you to get more confirmation that the correction execution of your program is not based on luck.

Valgrind is required

Since your code is interacting directly with physical memory, things can go wrong at the slightest provocation. The range of problems that can arise when writing code dealing with low-level details has been covered in lectures (and a handout that goes into all the ugly details of memory errors and leaks). You should use Valgrind to detect any potential issues that may lurk under the surface. To get diagnostic messages that specify line numbers of source files, build program `debug-ocean.out` using the `-g` option for both `g++` and `clang++`. This option adds a lot of bloat to your code that is useful for debuggers and analyzers. Run Valgrind on the debug version of your program like this:

```
$ valgrind ./debug-ocean.out 1 your-test1.txt
```

The output from the above command should specify zero memory errors and leaks:

```
==13269== Memcheck, a memory error detector
==13269== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==13269== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==13269== Command: ./debug-ocean.out 1 your-test1.txt
==13269==
==13269==
==13269== HEAP SUMMARY:
==13269==     in use at exit: 0 bytes in 0 blocks
==13269==   total heap usage: 5 allocs, 5 frees, 73,764 bytes allocated
==13269==
==13269== All heap blocks were freed -- no leaks are possible
==13269==
==13269== For lists of detected and suppressed errors, rerun with: -s
==13269== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

After ensuring there are no memory errors and leaks, you must check if your program's output is exactly equivalent to the correct output in `test1.txt`.

You must repeat these tests for the remaining five tests.

Automation using `make`

Testing your output and checking for memory leaks and errors is doable by explicitly typing out commands in the Linux shell for one or two tests. However, it can become overly cumbersome when having to repeat for more than a couple of tests. This is where an automation tool like `make` shines.

Run `make` with the default rule to bring program executable `ocean.out` up to date:

```
$ make
```

There are six unit tests each specified by command-line parameters `1`, `2`, and so forth. To run unit test 1 using `make`, you'd run the command:

```
$ make test1
```

If the `diff` command is not silent, your output is incorrect and your code must be debugged.

To run all tests, run the command:

```
$ make test-all
```

To ensure there are no memory leaks or errors, you must use Valgrind to analyze the runtime behavior of your program in relation to the memory allocated from the free store. Since Valgrind requires the `-g` option, you need a different target. The *makefile* contains a target called `debug-memory`. First, you must delete the old build and then run the `make` command with target `debug-memory` like this:

```
$ make clean
$ make debug-memory
```

To run test 1 with Valgrind, you'd run the command:

```
$ make debug-test1
```

To run all tests, run the command:

```
$ make debug-test-all
```

File-level and function-level documentation

Every source and header file *must* begin with a *file-level* documentation block. This module will use Doxygen to tag source and header files for generating html-based documentation. In addition, every function that you declare and define and submit for assessment must contain *function-level documentation*. This documentation should consist of a description of the function, the inputs, and return value.

Submission and automatic evaluation

1. In the course web page, click on the appropriate submission page to submit the necessary files.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
    - F grade if your submission doesn't compile with the full suite of `g++` options.
    - F grade if your submission doesn't link to create an executable.
    - Your implementation's output doesn't match correct output of the grader (you can see the inputs and outputs of the auto grader's tests). The auto grader will assign 50% of the grade based on the input and output files given to you. The remaining 50% of the grade will be awarded based on the additional tests implemented by the auto grader.

- The auto grade will provide a proportional grade based on how many incorrect results were generated by your submission. $A+$ grade if your output matches correct output of auto grader.
- A deduction of one letter grade for each missing documentation block in your submissions. Each source file must have **one** file-level documentation block and a function-level documentation block for each defined function. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing or incomplete or copy-pasted (with irrelevant information from some previous assessment) block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an $A+$ grade and one documentation block is missing, your grade will be later reduced from $A+$ to $B+$. Another example: if the automatic grade gave your submission a C grade and the two documentation blocks are missing, your grade will be later reduced from C to F.