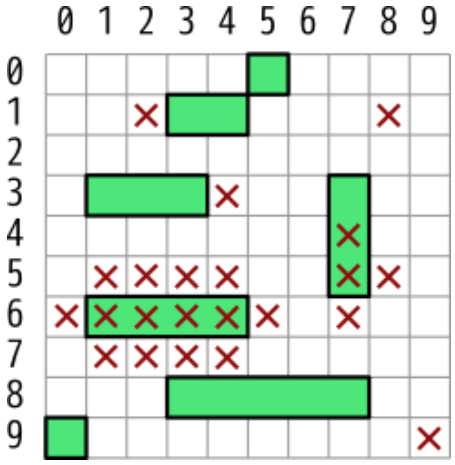
CSCI 135 - Program 2 - Game Battleship

Introduction

In this project, you will be implementing a program playing a *variant* of the game Battleship. (You can try to play a very similar game here.)



Your program is going to play a single-player version of the game, where your task is to sink all ships in the shortest time possible.

The submitted programs will compete in a multi-stage *tournament*, so that you can improve your program. The participation is *anonymous* (to classmates), but counts towards your grade (see bottom).

Submission deadlines:

For tournament: **April 7**, **April 19**, **April 22**.
Non-tournament (final deadline): **April 22**.

Rules of the game

Ships are placed on a rectangular grid. They **cannot occupy neighboring squares**, but are **allowed to touch diagonally** (see the figure above).

The locations of the ships are unknown to you. **Each turn**, you can shoot once at a square of your choice. The computer will respond, telling you whether you `MISS` or `HIT` a ship. If all squares of a ship were hit, the computer will tell you that you `HIT_AND_SUNK` the ship.

Each time you miss, a new **round** starts.

Your goal is to end the game in the **fewest number of rounds**.

Hypothetically, if your program never misses, it will win the game in a single round, however this is very unlikely. Still, try to be as efficient as possible. Reducing the number of misses will be your main goal.

How to run the program

Download and unpack the program

```
tar xvzf battleships.tar.gz
```

To build it, while being inside the program's directory type

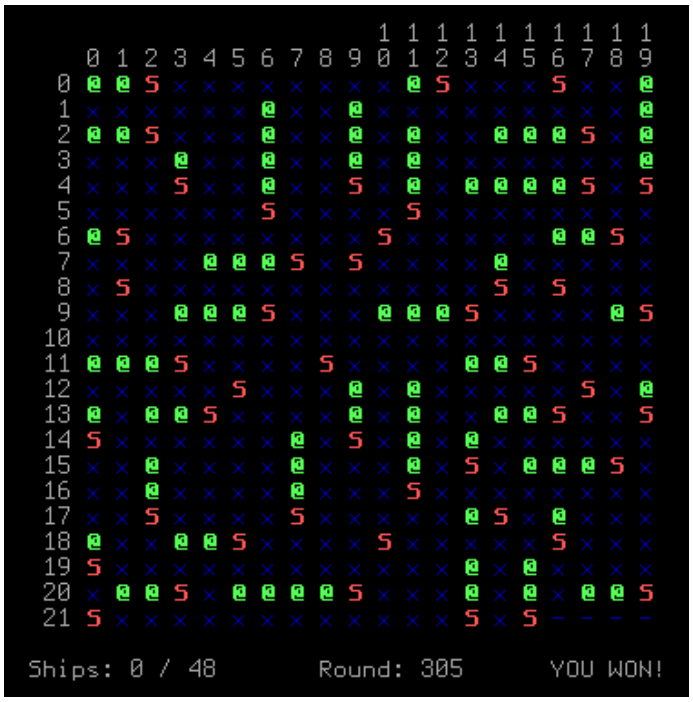
```
make
```

It creates an executable file named `battleships`.

To run, open an xterm (at least 40 lines and 85 characters wide with default board size) and type:

```
./battleships
```

It will start a new game with the board size approximately `20` by `20` (it is slightly randomized), and approximately `50` ships. The ship sizes will be from `1` to `5`.



Note that it is possible to choose other initial condition, specifying

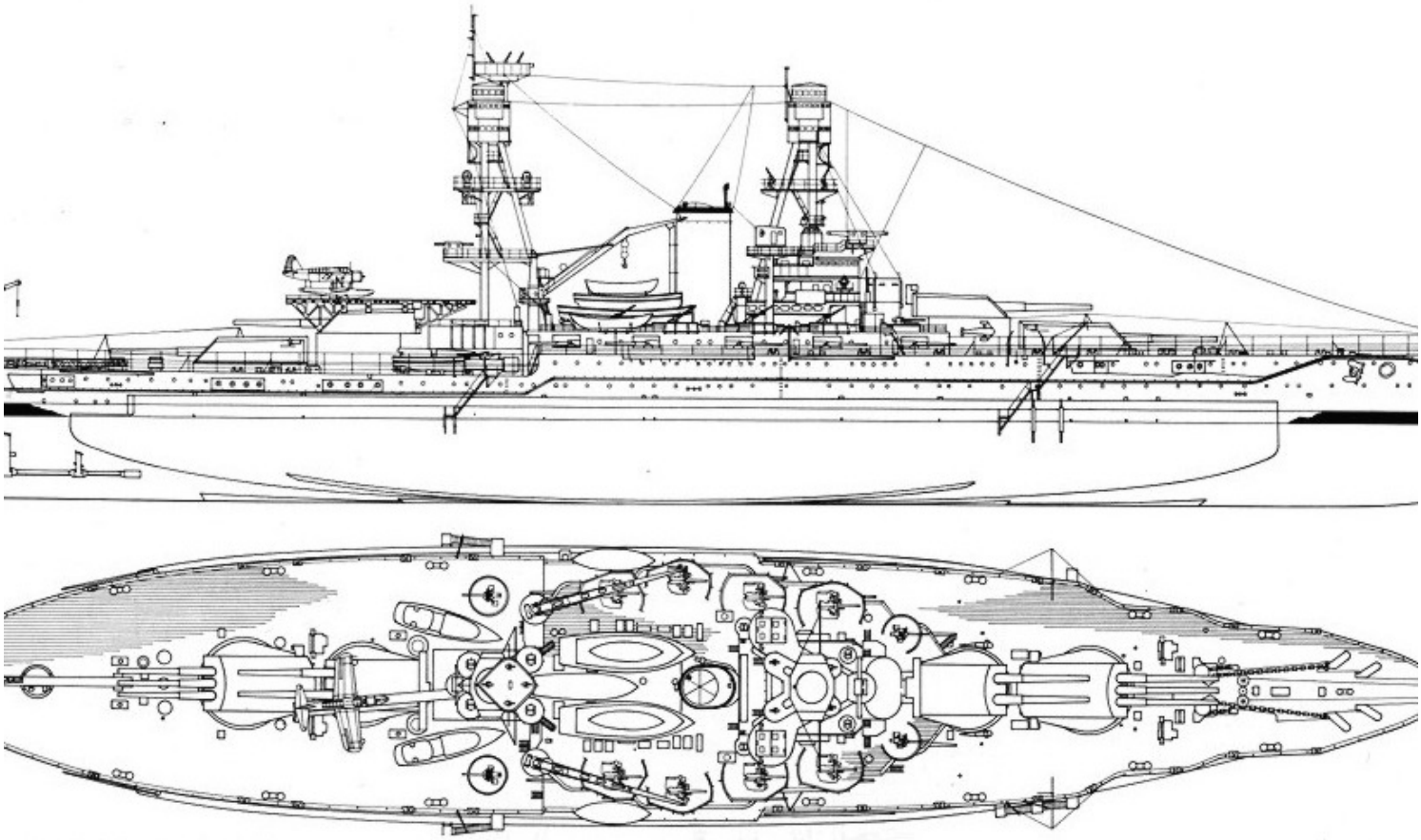
```
./battleships <rows> <cols> <sml> <lrg> <num> <seed>
```

the number of **row**s and **col**umns, the **small**est and the **large**st ship sizes, the **num**ber of ships, and the random **seed**. The board dimensions cannot be smaller than `5` or larger than `35`. If the requested number of ships does not fit on the board, there will be fewer ships.

When running the program you can **press the key** `[P]` to pause, `[S]` to step one turn, `[F]` to fast-forward, and `[Q]` to quit.

The **game log** is there primarily for testing and debugging purposes, a replacement for `cout`, since the normal `cout` would not work well (and so should not be used). Note that you may output anything you want there to help you debug your program.

The Programming Interface



As you can see, **the game already implements a certain tactic**, which is definitely not optimal. Improving the tactic will be your main task in this project.

Let us explain what you can do.

The `bot.cpp` file

Open the file **`bot.cpp`** . This is the game logic file you will be working on, and **this is the only file you are going to edit and submit**. Keep all the other files unchanged.

There are two function you have to implement. The first of them, called **`init`** , is executed when the game starts:

```
void init(rows, cols, num, &screen, &log)
{
 ...
}
```

- `rows` , `cols` , and `num` are the dimensions of the board, and the number of ships.
- `screen` is the game board representation, which you can mark. Here, you can label the locations of the ships, or in principle do anything you want with it. It can help you see what's going on.
- `log` is an analog of `cout` , which lets you print in the corresponding window.

If your program requires any global variables, they can be initialized in the **`init`** function as well.

The second function, **`next_turn`**, is called on each turn, here you can actually shoot at the ships:

```
void next_turn(sml, lrg, num, &gun, &screen, &log) {
 ...
}
```

- `sml` and `lrg` are the sizes of the smallest and the largest ship currently on the board.
- `num` is the number of ships.
- `gun` is an object that lets you shoot **once** at the location of your choice if you call `gun.shoot(row, column)`. The possible returned values are:
  - `MISS`
  - `HIT`
  - `HIT_N_SUNK`
  - `ALREADY_HIT`
  - `ALREADY_SHOT`

  The first three are self-explanatory. `ALREADY_HIT` will be returned if you try to shoot at the location where you had already shot in past and hit the ship. `ALREADY_SHOT` will be returned if you try to shoot multiple times per turn.
- `screen` and `log` are the same as in the previous function.

Notice how the `screen` is used. It **is only a display** that lets you **visualize** what's going on:

`screen.mark(row, col, ch, color);` marks the square (row, col) with the character `ch`. Possible colors are `RED`, `GREEN`, `BLUE`, and `GRAY`.

What strategy can you use?

Implement the best possible playing strategy you can come up with. It should win the games in the fewest number of rounds possible.

The only file you can modify and submit is `bot.cpp`.

We are going to test the performance of your programs on two sets of maps:

- Maps packed with ships of various sizes from small to large.
- Maps with only large ships (Use the command line options we mentioned earlier to generate such large-ship maps).

You are free to choose any strategy. But we can give a few hints. Your solution can gradually evolve from a simple one to a much more complex:

1. You can start by shooting at random unexplored spots.
2. Then you can make a global 2D array to store where you hit and where you miss.
3. When you `HIT` a ship, remember that (or search in the 2D array for such a location) and try shooting around that square on the next turn.
4. When `HIT_N_SUNK` an entire ship, you can mark the surrounding squares as `MISS` in your array:

5. The previous change was a big improvement, but you can do even better. Keep track of where the ships can be and rule out impossible regions. If you know what the smallest and the largest ships are you can make use of this knowledge here. (*Test it on maps generated with all large ships.*)

6. You can not only rule out impossible squares, but also try to estimate where ships are most likely to be, can't you?

7. Come up with even better ideas and optimizations, they are possible.

How to submit

You are to submit a single file `bot.cpp` .

Since we are running the tournament, there will be **three submission days**, each contributing to the final tournament score with the increasing weights: **10%**, **30%**, and **60%**.

- 1st *tournament* submission: **by Friday, April 7, end of the day**.

[– Spring break interrupts –]

- 2nd *tournament* submission: **by Wednesday, April 19, end of the day**.
- 3rd *tournament* submission: **by Saturday, April 22, end of the day**.

And on the third (and final) submission day, you should **also submit the code for grading** in your CS-136 (Lab) Blackboard link.

- **Final *non-tournament* submission**: same day, Saturday, April 22, end of the day, but submitted at your CS-136 (Lab) Blackboard link for grading.

GRADING

All 135 and 136 programs this semester will be graded on:

- *Correctness*: Does your program work?
- *Testing*: Have you generated sufficient and good test data to give reasonable confidence that your program works?
- *Structure*: Have you structured your code to follow proper software engineering guidelines? This includes readability and maintainability. Note that

- *Documentation*: How well documented is your code? Good documentation does not repeat the code in English, but explains the point of each code block, highlighting any design decisions and/or tricky implementation details.

For this program, a major factor in the correctness component is your performance in the tournaments.