Programming Assignment #1: IrmaMoves

Spring 2018

Due: Sunday, February 11, *before* 11:59 PM

Table of Contents

Programming Assignment #1: IrmaMoves 1
1. Overview 3
2. Important Note: Test Case Files Look Wonky in Notepad 3
3. IrmaMoves.h 4
The Move Struct 4
The Location Struct 5
The Irma Struct 5
4. Function Requirements 5
5. Compilation and Testing (CodeBlocks) 8
6. Compilation and Testing (Linux/Mac Command Line) 9
7. Getting Started: A Guide for the Overwhelmed 10
9. Deliverables 12
10. Grading 12
Appendix A: Irma Movement 13
Board Configuration 13
Denoting Positions on the Board 13
Translating Board Positions to 2D Array Coordinates 13
Denoting Movement 14
Examples 15

Abstract

In this programming assignment, you will implement an interpreter for algebraic notation for recording and describing the hurricane Irma moves. In order to finish this program, you will need to learn a bit about how hurricane Irma can move on the map board, as well as how those moves are denoted with algebraic notation.

By completing this assignment, you will gain experience manipulating strings, 2D arrays, structs, and struct pointers in C. You will also gain experience working with dynamic memory allocation and squashing memory leaks. On the software engineering side of things, you will learn to construct programs that use multiple source files and custom header (.h) files. This assignment will also help you hone your ability to read technical specifications and implement a reasonably lengthy, non-trivial project with interconnected pieces.

Attachments

IrmaMoves.h
testcase{01-05}.c
testcase{01-05}.txt

Deliverables

IrmaMoves.c

(***Note!*** Capitalization and spelling of your filename matters!)
(This project has been inspired and adopted from Sean Szumlanski's projects)

1. Overview

The algebraic notation has been used in a few different notation systems to denote the moves made in a game. In our game, we are going to predict the wind speed and wind gusts of hurricane Irma when it starts from a point and moves toward its destination. The Irma's moves with its initial wind speed and wind gusts, and its destination is recorded as a series of strings that look something like this:

h6 150 100 f4

"h6" and "f4" are the coordination of start and destination points in a board of 8x8 grid of squares, respectively, while the first integer number denotes the wind speed and the second integer number denotes the wind gusts of Irma when it starts its movement. For this assignment, you will write a program that parses through a series of algebraic notation strings and prints out what the map board looks like at the end. To do this, you will first have to learn (or brush up on) how Irma moves (see ***Appendix A*** in this document).

A complete list of the functions you must implement, including their functional prototypes, is given below in Section 4, "Function Requirements". You will submit a single source file, named `IrmaMoves.c`, that contains all required function definitions, as well as any auxiliary functions you deem necessary. In `IrmaMoves.c`, you will have to `#include` any header files necessary for your functions to work, including the custom `IrmaMoves.h` file we have distributed with this assignment (see Section 3, "IrmaMoves.h").

Note: You will *not* write a main() function in the source file you submit! Rather, we will compile your source file with our own `main()` function(s) in order to test your code. We have attached example source files that have `main()` functions, which you can use to test your code. You can write your own `main()` functions for testing purposes, but the code you submit must not have a `main()` function. We realize this is completely new territory for most of you, so don't panic. We've included instructions for compiling multiple source files into a single executable (e.g., mixing your `IrmaMoves.c` with our `IrmaMoves.h` and `testcaseXX.c` files) in Sections 5 and 6 of this PDF.

Although we have included test cases with sample `main()` functions to get you started with testing the functionality of your code, we encourage you to develop your own test cases, as well. Ours are by no means comprehensive. We will use much more elaborate test cases when grading your submission.

2. Important Note: Test Case Files Look Wonky in Notepad

Included with this assignment are several test cases, along with output files showing exactly what your output should look like when you run those test cases. You will have to refer to those as the gold standard for how your output should be formatted. Please note that if you open those files in Notepad, they will appear to be one long line of text. That's because Notepad handles end-of-line characters differently from Linux and Unix-based systems. One solution is to view those files in an IDE (like CodeBlocks), or you could use a different text editor (like Atom or Sublime).

3. IrmaMoves.h

Included with this assignment is a customer header file that includes the struct definitions and functional prototypes for the functions you will be implementing. You should #include this file from your `IrmaMoves.c` file, like so:

```
#include "IrmaMoves.h"
```

The "quotes" (as opposed to <brackets>) indicate to the compiler that this header file is found in the same directory as your source, not a system directory.

When working on this program, do not modify `IrmaMoves.h` in any way. Do not copy its struct definitions or functional prototypes into your `IrmaMoves.c` file. Do not send `IrmaMoves.h` to us when you submit your assignment. We will use our own unmodified copy of `IrmaMoves.h` when compiling your program.

If you write auxiliary functions ("helper functions") in your `IrmaMoves.c` file (which is strongly encouraged!), you should ***not*** add those functional prototypes to `IrmaMoves.h`. Our test case programs will not call your helper functions directly, so they do not need any awareness of the fact that your helper functions even exist. (We only list functional prototypes in a header file if we want multiple source files to be able to call those functions.) So, just put the functional prototypes for any helper functions you write at the top of your `IrmaMoves.c` file.

Think of `IrmaMoves.h` as a bridge between source files. It contains struct definitions that might be used by multiple source files, as well as functional prototypes, but only for functions that might be defined in one source file (such as your `IrmaMoves.c` file) and called from a different source file (such as the `testcaseXX.c` files we have provided with this assignment).

There are three structs defined in `IrmaMoves.h`, which you will use to implement the required functions for this assignment. They are as follows:

The Move Struct

```
typedef struct Move
{
   Irma irma;             // an instance of Irma
   L Location from_loc;   // location where Irma is moving from
   Location current_loc;  // location where Irma is passing over
   Location to_loc;       // location where Irma is moving to
} Move;
```

This struct will hold information about Irma moves. Your program will process a string of algebraic notation, such as "h6 150 100 f4", and when it does so, certain pieces of that string will be extracted, interpreted, and used to fill out certain fields in this struct.

The `irma` struct will hold the information of wind speed/gusts of start point. The `from_loc` field will hold the location of Irma whose movement is being described (a column and row on the map board; see struct definition below). The `current_loc` field will hold the current location of Irma while moving from its start point to its destination. The `to_loc` field will hold the column and row of the location to which Irma is moving.

Some of these fields will be set by your `parseNotationString()` function. Others cannot be set until you actually go to apply the move to the map board in your `predictIrmaChange()` function, which will call a function to help determine which direction Irma should move.

The Location Struct

```
typedef struct Location
{
   char col;   // the square's column ('a' through 'h')
   int row;    // the square's row (0 through 7)
} Location;
```

This struct holds locations on the map board, and is used by the `Move` struct defined above. For information on how rows and columns are labeled on a map board, see "Appendix A: Irma Movement".

The Irma Struct

```
typedef struct Irma
{
   int ws;     //  wind speed (MPH)
   int wg;     //  wind gusts (MPH)
} Irma;
```

This struct holds the information of wind speed/gusts of start point.

4. Function Requirements

In the source file you submit, `IrmaMoves.c`, you must implement the following functions. You may implement auxiliary functions (helper functions) to make these work, as well. Please be sure the spelling, capitalization, and return types of your functions match these prototypes exactly.

In this section, I sometimes refer to `malloc()` when talking about dynamic memory allocation, but you're welcome to use `calloc()` or `realloc()` instead, if you're familiar with those functions.

```
char **createMapBoard(void);
```

Description: Dynamically allocate space for a 2D char array with dimensions 8x8, and populate the array with the initial configuration of a map board as described in "Appendix A: Irma Movement" (and as shown in the test case output files included with this assignment). Essentially, you have to hard code the initial map configuration based on the information given in Appendix A. As an example, after dynamically allocating space for map board, you can use the following line of code to store proper letter in each square of map board:

```
strncpy(board[0], "FF      ", 8);
```

Output: This function should not print anything to the screen.

Return Value: A pointer to the dynamically allocated 2D array (i.e., the base address of the 2D array), or `NULL` if any calls to `malloc()` fail.

```
char **destroyMapBoard(char **board);
```

Description: Free all dynamically allocated memory associated with board. You may assume this function receives a dynamically allocated 2D char array with dimensions 8x8.

Output: This function should not print anything to the screen.

Return Value: This function must return `NULL`.

```
void printMapBoard(char **board);
```

Description (and Output): This function takes a pointer to an 8x8 char array and prints the map board represented by that array using the format described below. This format is also shown explicitly in several of the test case output files included with this assignment.

The printout of the map board should be preceded by a line of eight equal symbols ('='), and it should also be followed by a line of eight '=' symbols, followed by a blank line. For example:

```
========
FF
 F
 FF
  F
  K
C  B
 CC D
  C  DD
========
          ← Note: This is a completely blank line. (No spaces.)
```

Return Value: There is no return value. This is a `void` function.

```
char **predictIrmaChange (char* str, Move *irmaMove);
```

Description: This function should start by *printing the map board with Irma in its starting position*, using the arrangement described in "Appendix A: Irma Movement" and the specific format shown in the test case output files included with this assignment. (You will want to call `createMapBoard()` from within this function to create a 2D `char` array to represent the map board, and then call the `printMapBoard()` function to print it to the screen. See above for the descriptions of both of those functions.)

After printing the initial map board, this function should parse all of the algebraic notation strings passed in through making calls to the `parseNotationString()` function (described below). The `printMapBoard()` function is called once after the Irma's location has been identified, and another time after the board's final configuration has been processed. Note that this function should print the resulting board (again, with a line of eight equal signs above and below the board each time, always followed by a blank line).

In addition, updating Irma's wind speed/gusts and checking Irma directions should be

implemented in this function as described in "Appendix A: Irma Movement".

Output: This should print out the map board in the manner described above, with precisely the same format shown in the test case output files included with this assignment.

Return Value: A pointer to the dynamically allocated 2D array (i.e., the base address of the 2D array), or `NULL` if any calls to `malloc()` fail.

```
void parseNotationString(char* str, Move* irmaMove);
```

Description: This function receives an algebraic notation string, `str`, and one `Move` struct pointer. The function must parse str and extract information about Irma moves encoded there, and populate all corresponding fields in the struct pointed to by `irmaMove`.

At the very least, it will always be possible to set the `from_loc, current_loc, to_loc`, and `irma.ws, irma.wg` fields in `Irma` struct. It is necessary to denote which column and/or row the Irma's move is coming from, and also the column and/or row in a move's `to_loc` field which determine where Irma ends. To initialize `from_loc, current_loc, to_loc` fields, you must initialize that move's `from/current/to_loc.col` fields to 'x' to indicate that the column is currently unknown. Similarly, the move's `from/current/to_loc.row` coordinate must be initialized to -1.

After initialization, update `from_loc, current_loc, to_loc`, and `irma.ws, irma.wg` fields through extracting information from given algebraic notation string.

Output: This function should not print anything to the screen.

Return Value: There is no return value. This is a `void` function.

```
double difficultyRating(void);
```

Output: This function should not print anything to the screen.

Return Value: A double indicating how difficult you found this assignment on a scale of 1.0 (ridiculously easy) through 5.0 (insanely difficult).

```
double hoursSpent(void);
```

Output: This function should not print anything to the screen.

Return Value: An estimate (greater than zero) of the number of hours you spent on this assignment.

5. Compilation and Testing (CodeBlocks)

The key to getting multiple files to compile into a single program in CodeBlocks (or any IDE) is to create a project. Here are the step-by-step instructions for creating a project in CodeBlocks, which involves importing `IrmaMoves.h`, `testcase01.c`, and the `IrmaMoves.c` file you've created (even if it's just an empty file so far).

1. Start CodeBlocks.
2. Create a New Project (*File -> New -> Project*).
3. Choose "Empty Project" and click "Go."
4. In the Project Wizard that opens, click "Next."
5. Input a title for your project (e.g., "IrmaMoves").
6. Choose a folder (e.g., Desktop) where CodeBlocks can create a subdirectory for the project.
7. Click "Finish."

Now you need to import your files. You have two options:

1. Drag your source and header files into CodeBlocks. Then right click the tab for **each** file and choose "Add file to active project."

   – or –

2. Go to *Project -> Add Files...*. Browse to the directory with the source and header files you want to import. Select the files from the list (using CTRL-click to select multiple files). Click "Open." In the dialog box that pops up, click "OK."

You should now be good to go. Try to build and run the project (F9).

Note that if you import both `testcase01.c` *and* `testcase02.c`, the compiler will complain that you have multiple definitions for `main()`. You can only have one of those in there at a time. You'll have to swap them out as you test your code.

Yes, constantly swapping out the test cases in your project will be a bit annoying. You can avoid this if you're willing to migrate away from an IDE and start compiling at the command line instead. If you're interested in doing that in Windows, please look around online for instructions on how to make that happen.

6. Compilation and Testing (Linux/Mac Command Line)

To compile multiple source files (`.c` files) at the command line:

```
gcc IrmaMoves.c testcase01.c
```

By default, this will produce an executable file called `a.out`, which you can run by typing:

```
./a.out
```

If you want to name the executable file something else, use:

```
gcc IrmaMoves.c testcase01.c -o IrmaMoves.exe
```

...and then run the program using:

```
./IrmaMoves.exe
```

Running the program could potentially dump a lot of output to the screen. If you want to redirect your output to a text file in Linux, it's easy. Just run the program using the following command, which will create a file called `whatever.txt` that contains the output from your program:

```
./IrmaMoves.exe > whatever.txt
```

Linux has a helpful command called `diff` for comparing the contents of two files, which is really helpful here since we've provided several sample output files. You can see whether your output matches ours exactly by typing, e.g.:

```
diff whatever.txt output01.txt
```

If the contents of `whatever.txt` and `output01.txt` are exactly the same, `diff` won't have any output. It will just look like this:

```
navid@ubuntu:~$ diff whatever.txt output01.txt
navid@ubuntu:~$ _
```

If the files differ, it will spit out some information about the lines that aren't the same. For example:

```
navid@ubuntu:~$ diff whatever.txt output01.txt
1c1
< fail whale :(
---
> Hooray!
navid@ubuntu:~$ _
```

7. Getting Started: A Guide for the Overwhelmed[1]

Okay, so, this might all be overwhelming, and you might be thinking, "Where do I even start with this assignment?! I'm in way over my head!"

Don't panic! Here's my general advice on starting the assignment:

1. First and foremost, start working on this assignment early. Nothing will be more frustrating than running into unexpected errors or not being able to figure out what the assignment is asking you to do on the day that it is due.

2. Secondly, glance through this entire PDF to get a general overview of what the assignment is asking you to do, even if you don't fully understand each section right away.

3. Thirdly, before you even start programming, read through Appendices A to familiarize yourself with the game of Irma prediction and the algebraic notation you'll be parsing in this assignment.

4. Once you have an idea of how algebraic notation works, open up the test cases and sample output files included with this assignment and trace through a few of them to be sure you have an accurate understanding of how notation works.

5. Once you're ready to begin coding, start by creating a skeleton `IrmaMoves.c` file. Add a header comment, add some standard `#include` directives, and be sure to `#include "IrmaMoves.h"` from your source file. Then copy and paste each functional prototype from `IrmaMoves.h` into `IrmaMoves.c`, and set up all those functions return dummy values (either `NULL` or nothing at all, as appropriate).

6. Test that your `IrmaMoves.c` source file compiles. If you're at the command line on a Mac or in Linux, your source file will need to be in the same directory as `IrmaMoves.h`, and you can test compilation like so:

   ```
   gcc -c IrmaMoves.c
   ```

   Alternatively, you can try compiling it with one of the test case source files, like so:

   ```
   gcc IrmaMoves.c testcase01.c
   ```

   For more details, see Section 6, "Compilation and Testing (Linux/Mac Command Line)."

   If you're using an IDE (i.e., you're coding with something other than a plain text editor and the command line), open up your IDE and start a project using the instructions above in Section 5, "Compilation and Testing (CodeBlocks)". Import `IrmaMoves.h`, `testcase01.c`, and your new `IrmaMoves.c` source file, and get the program compiling and running before you move forward. (Note that CodeBlocks is the only IDE we officially support in this class.)

[1] Adapted from Sean Szumlanski

7. Once you have your project compiling, go back to the list of required functions (Section 4, "Function Requirements"), and try to implement one function at a time. Always stop to compile and test your code before moving on to another function!

8. You'll *probably* want to start with the `createMapBoard()` function.

   As you work on `createMapBoard()`, write your own `main()` function that calls `createMapBoard()` and then checks the results. For example, you'll want to ensure that `createMapBoard()` is returning a non-NULL pointer to begin with, and that each index in the array it returns is non-NULL as well. Then try printing out the board. Finally, look through the text cases provided with this assignment to find one that calls `createMapBoard()` explicitly. Run it and check that your output is correct. If not, go through your code (as well as the test case code) line-by-line, and see if you can find out why your output isn't matching.

9. After writing `createMapBoard()`, I would probably work on the `destroyMapBoard()` and `printMapBoard()` functions, because they're so closely related. This might require that you spend some time reviewing the course notes on 2D array allocation.

10. Next, I would work on the `parseStringNotation()` function. This one will be quite lengthy. Start by writing your own `main()` that passes simple strings to `parseStringNotation()`. Check that the results it produces are correct. Then start passing more complex strings. If you need guidance on how to call this function and determine whether it's producing the correct results, look for a test case that calls `parseStringNotation()`, and check whether your program produces the correct output with that test case. If not, trace through it by hand.

11. If you get stuck while working on this assignment, draw diagrams on paper or a whiteboard. Make boxes for all the variables in your program. If you're dynamically allocating memory, diagram them out and make up addresses for all your variables. Trace through your code carefully using these diagrams.

12. With so many pointers, you're bound to encounter errors in your code at some point. Use `printf()` statements liberally to verify that your code is producing the results you think it should be producing (rather than making assumptions that certain components are working as intended). You should get in the habit of being immensely skeptical of your own code and using `printf()` to provide yourself with evidence that your code does what you think it does.

13. When looking for a segmentation fault, you should always be able to use `printf()` to track down the *exact* line you're crashing on.

14. You'll need to examine a lot of debugging output. Just be sure to remove your debugging statements before you submit your assignment, so your code is nice and clean and easy for me to read.

15. When you find a bug, or if your program is crashing on a huge test case, don't trace through hundreds of lines of code to track down the error. Instead, try to cook up a new `main()` function with a very small test case (as few lines as possible) that directly calls the function that's crashing. The less code you have to trace through, the easier your debugging tasks will be.

9. Deliverables

Submit a single source file, named `IrmaMoves.c`, via Canvas. The source file should contain definitions for all the required functions (listed above), as well as any auxiliary functions you need to make them work.

Your source file **must not** contain a `main()` function. Do not submit additional source files, and do not submit a modified `IrmaMoves.h` header file. Your source file (without a `main()` function) should compile at the command line using the following command:

```
gcc -c irmaMoves.c
```

It must also compile at the command line if you place it in a directory with `IrmaMoves.h` and a test case file (for example, `testcase01.c`) and compile like so:

```
gcc irmaMoves.c testcase01.c
```

Be sure to include your name and ID as a comment at the top of your source file.

10. Grading

The *tentative* scoring breakdown (not set in stone) for this programming assignment is:

40%	correct output for test cases
35%	implementation details (manual inspection of your code)
5%	`difficultyRating()` is implemented correctly
5%	`hoursSpent()` is implemented correctly
5%	source file is named correctly (`IrmaMoves.c`); spelling and capitalization count
10%	adequate comments and whitespace; source includes student name and ID

Note! Your program must be submitted via Canvas, and it must compile and run to receive credit. Programs that do not compile will receive an automatic zero.

Your grade will be based largely on your program's ability to compile and produce the *exact* output expected. Even minor deviations (such as capitalization or punctuation errors) in your output will cause your program's output to be marked as incorrect, resulting in severe point deductions. The same is true of how you name your functions and their parameters. Please be sure to follow all requirements carefully and test your program thoroughly.

Additional points will be awarded for style (proper commenting and whitespace) and adherence to implementation requirements. For example, the graders might inspect your `destroyMapBoard()` function to see whether it is properly freeing up memory.

Start early. Work hard. Good luck!

Appendix A: Irma Movement

This appendix describes how each Irma moves on the map. First, however, we give a very brief overview of the game and describe the initial configuration of the board.

Board Configuration

The board is an 8x8 grid of squares, where either a letter is stored on the square or a space character. Essentially, the location of selective countries, islands, and cities have been shown on the map. In *Figure 1*, the basic board configuration has been shown.

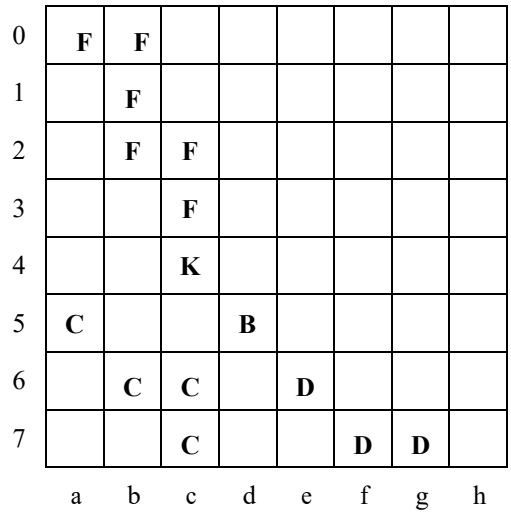


Figure 1. *The initial configuration of an 8x8 board representing Florida peninsula and neighbor islands.*

Denoting Positions on the Board

To understand algebraic notation, one must first understand how to refer to a specific square on an 8x8 board. The columns on a board are labeled 'a' through 'h', from left to right. The rows are labeled 0 through 7 from top to bottom, where the bottom rows (rows 7 and 6) are where Dominican Republic country has been represented, and the top rows (rows 0 through 3) are where the Florida state has been shown.

Coordinates for a particular square in this notation system are given as <*column*><*row*>, with no space in between. <*column*> is a single letter from 'a' through 'h', denoting which column the hurricane Irma is in, and <*row*> is a single integer on the range 0 through 7, denoting which row the hurricane Irma is in.

For example, in *Figure 2* (below), the hurricane Irma is at position *h6*.

Translating Board Positions to 2D Array Coordinates

Arrays in C are referred to as "row major," which means (among other things) that the first index given in *array*[*i*][*j*] refers to the row (that is, row *i*). The second coordinate, therefore, refers to the column (that is, column *j*). Thus, pieces at positions *h6* on a map board represented by a 2D array would be at positions array[6][7]. (See Figure 3, below.)

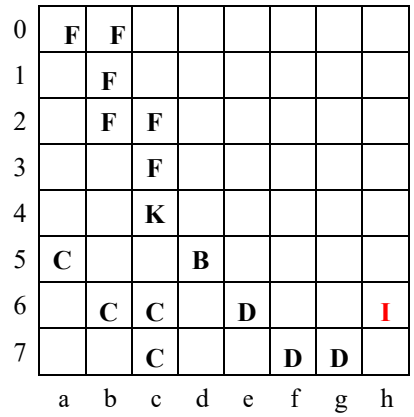


Figure 2: *A map board with Irma at position h6.*

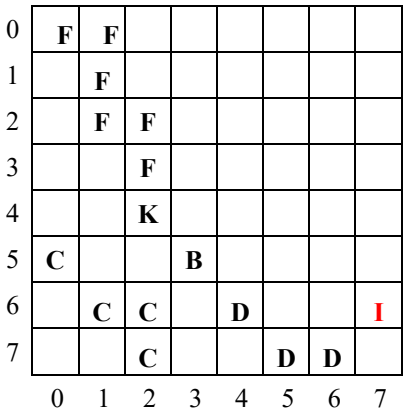


Figure 3: *A 2D array representation of a map board corresponding to Figure 2. The Irma is at array[6][7].*

In one of the functions you write for this assignment, you will have to take a string containing coordinates like "h6" and translate it to its corresponding 2D array coordinates (in this case, [6][7]).

Denoting Movement

The moves in an Irma predication game are denoted by a sequence of strings. Each string begins with a sequence of characters (without spaces) denoting the start point of hurricane Irma, followed by a single space, followed by an integer number denoting the wind speed of Irma, followed by a single space, followed by an integer number denoting the wind gusts of Irma, followed by a single space, and followed by a sequence of characters (without spaces) denoting the end point of Irma. For example:

h6 150 100 f4

"h6" denotes the movement of Irma starts from this location or array[6][7], the wind speed and wind gusts of Irma are *150 mph* and *100 mph*, respectively, end it ends in "f4" location of the board or array[4][5]. The notation for a standard move is structured like:

{*fromCol*}{*fromRow*} {*wind speed*} {*wind gusts*} {*toCol*}{*toRow*}

The denoting movements always follow above structure, and there is no cases that components in {curly braces} may be left out.

The direction of Irma movement follows the below rules:

- **First,** Irma starts its movement from the given start point toward ***either east (left) or west (right)*** depending on the destination coordination. While Irma ***horizontally*** passes over either ocean or land (which have been identified with letters in our map board), its wind speed and wind gusts changes as follows:

    - If it horizontally passes over ocean, its wind speed and wind gusts will go up as follows:

```
Irma.WindSpeed += 10;
Irma.WindGusts += 5;
```

- If it horizontally passes over land, its wind speed and wind gusts will go down as follows:

  Irma.WindSpeed -= 15;
  Irma.WindGusts -= 10;

- **After Irma reaches its destination column,** Irma starts its movement toward ***either north (up) or south (down)*** depending on the destination coordination. While Irma ***Vertically*** passes over either ocean or land (which have been identified with letters in our map board), its wind speed and wind gusts changes as follows:
  - If it vertically passes over ocean, its wind speed and wind gusts will go up as follows:

    Irma.WindSpeed += 6;
    Irma.WindGusts += 3;
  - If it vertically passes over land, its wind speed and wind gusts will go down as follows:

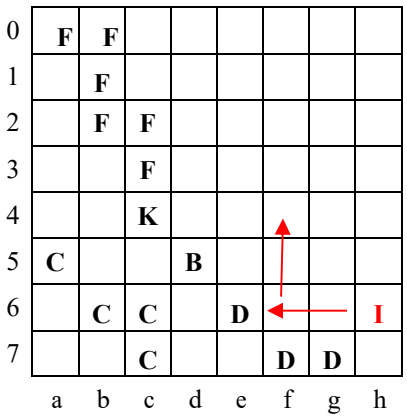
    Irma.WindSpeed -= 2;
    Irma.WindGusts -= 1;

- **Finally,** when Irma reaches its destination, its wind speed and wind gusts must be updated.

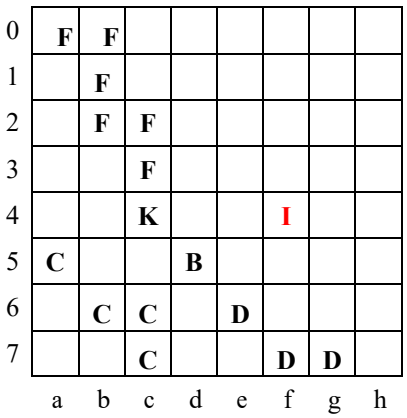
Examples

This section provides several examples of how Irma moves in our map board, and what would be its wind speed and wind gusts over the course of this movement.

Before Move:

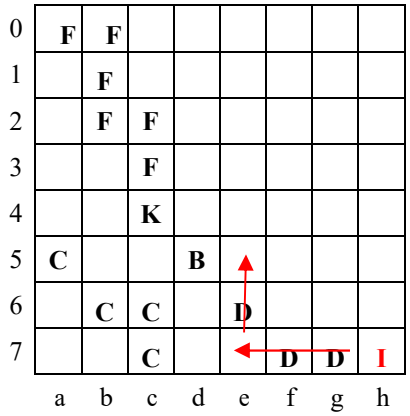


After Move:

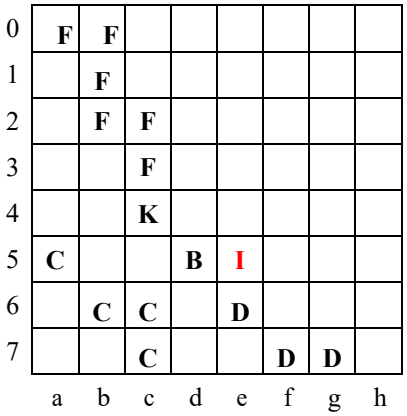


Example 1: *"h6 150 100 f4" denotes that Irma moves from h6 to f4. Irma starts its movement with wind speed and wind gusts of 150 mph and 100 mph, respectively at h6.* ***Note*** *that Irma should move to west to get to f6 first, then it can move to north to get to its destination. When Irma moves to g6, our game checks that whether Irma hits the land or not. Since there is no land on g6, Irma's wind speed goes up to 160mph (150 + 10), and its wind gusts goes up to 105mph (100 + 5).* ***Next,*** *Irma moves to west one more time to get to f6. Since there is no land on f6, Irma's wind speed goes up to 170mph (160 + 10), and its wind gusts goes up to 110mph (105 + 5).* ***Next,*** *Irma moves to north to get to f4. The first square that Irma passes over in our example is f5. Again, when Irma moves to f5, our game checks that whether Irma hits the land or not. Since there is no land on f5 and Irma's movement is vertically now, Irma's wind speed goes up to 176mph (170 + 6), and its wind gusts goes up to 113mph (110 + 3).* ***Finally,*** *Irma moves to north one more time to get to f4. Since there is no land on f5 and Irma's movement is vertically, Irma's wind speed goes up to 182mph (176 + 6), and its wind gusts goes up to 116mph (113 + 3).*

Before Move:



After Move:



Example 2: *"h7 110 100 e5" denotes that Irma moves from h7 to e5. Irma starts its movement with wind speed and wind gusts of 110 mph and 100 mph, respectively at h7.* ***Note*** *that Irma should move to west to get to e7 first, then it can move to north to get to its destination. When Irma moves to g7, our game checks that whether Irma hits the land or not. Since there it hits land on g7, Irma's wind speed goes down to 95mph (110 - 15), and its wind gusts goes down to 90mph (100 - 10).* ***Next,*** *Irma moves to west one more time to get to e7. Since there it hits land on f7, Irma's wind speed goes down to 80mph (95 -15), and its wind gusts goes down to 80mph (90 - 10).* ***Next,*** *Irma moves to west one more time to get to e7. Since there is no land on e7, Irma's wind speed goes up to 90mph (80 + 10), and its wind gusts goes up to 85mph (80 + 5).* ***Next,*** *Irma moves to north to get to e5. The first square that Irma passes over in our example is e6.* ***Again,*** *when Irma moves to e6, our game checks that whether Irma hits the land or not. Since it hits land on e6 and Irma's movement is vertically now, Irma's wind speed goes down to 88mph (90 - 2), and its wind gusts goes down to 84mph (85 - 1).* ***Finally,*** *Irma moves to north one more time to get to e5. Since there is no land on e5 and Irma's movement is vertically, Irma's wind speed goes up to 94mph (88 + 6), and its wind gusts goes up to 87mph (84 + 3).*