CSCI 104 - Fall 2018

Data Structures and Object Oriented Design

Github
Ask on Piazza

Home
People/Office Hours
Sections
Syllabus
Lecture Schedule
Exercises
Assignments
Labs
Tools and Links

Homework 4 (Part 1 of Project)

- Due Date: Tuesday, October 9, 11:59pm
- Directory name in your github repository for this homework (case sensitive): `hw4`.
- Skeleton headers are provided for you under `hw4-skeleton` in the `homework-resources` repo.
- Provide a `README.md` file.
- You should provide a `Makefile` to compile and run your code.

Overview

The big class project this semester will be to write a Scrabble Game. Scrabble (and its clones like "Words with Friends") is a game for typically 2-4 players, in which the players take turns placing tiles on a board, forming correct words, and gaining points for their words. The rules can be found at the Scrabble Pages, but a pretty much complete version is also given below. Notice that in a few cases, we deviate from the official Scrabble rules, mostly to make your programming task slightly easier.

This project will consist of two parts. Here is how the parts will build on each other:

- In Homework 4, you will write a Scrabble Game to be played between 1-8 human players. It will all be in text mode with `cin`, `cout` for interaction.
- In Homework 8, you will add computer players with different strategies, thereby allowing a single human player (or more) to play against the computer.

None of this is terribly important in Homework 4 just yet, but you may want to keep it in mind while developing the class

architecture and code layout for this project. By the ti
are done, you will likely have well above 1000 lines o
and in adding/changing functionality, your life will be
you design things carefully.

Rules of Scrabble

As mentioned above, the rules are mostly described on the Scrabble Pages. However, here is a summary, including some generalizations that we apply here, some clarifications, and a few minor changes and discussions.

Letters, Hands, Board

- There is some alphabet (in English: a-z, but we want to keep it generic so your game easily ports to other languages). For each letter, there is a given number of copies of that letter, and a given score for that letter.
- There is also a given number of blanks, which by default have a score of 0. (We will allow other scores here.) Blanks can be used as any one letter, but once played, the letter cannot change.
- The letters are kept in a "bag" (in real life): players will draw tiles from the bag, which means that they get a uniformly random set from the remaining tiles.
- Each player maintains a hand of `k` tiles which they use to form words. In English, typically we have `k=7`, but we allow flexibility; for instance, German Scrabble usually has `k=8`.
- The board is a rectangle of squares. One of these squares is the start square. A subset of squares may be bonus squares, meaning that they multiply the value of a letter or of a word (see "Scoring" below). The standard Scrabble board is a 15x15 square, but again, we will want to keep it generic here.

Moves

Moves are of one of three types

1. Place a word.
    - When you place a word, you use anywhere from 1-k of your tiles and place them either horizontally or vertically in a line on the board. You then draw tiles from the bag equal to the number of tiles you played.

    - Your tiles, together with tiles that may al on the board, must form a single contigu horizontal or vertical line - no gaps allow
    - You cannot place a tile on top of another previously placed tile.
    - Except for the first move of the game, at least one of the placed tiles must be adjacent (horizontally or vertically) to a previously placed tile.
    - For the first move of the game, one of your tiles must be on the start squre.
    - Each maximal sequence (meaning that on both ends, it is bordered by an empty square or the end of the board) of two or more letters that is formed by your placement of tiles (notice that there could be multiple) must be a legal word according to the dictionary (see below). Sequences are read left to right, and top to bottom. Single letters are not considered words.
    - The moment you place a word including one or more blank tiles, those blank tiles are assigned letters (by you), which they keep for the rest of the game.
2. Return tiles.
    - You can return any number of tiles (from 1 to `k`). When you do this, you first return the tiles, and then you draw the same number of tiles from the bag to replace them (possibly getting some of the same tiles back), but that is the end of your move.
3. Pass.
    - You can pass your turn and not do anything.

Dictionary and Legal Words

- We provide you with a dictionary file containing all words that are deemed legal. This will be a standard English dictionary, but again, your program should be generic.
- Words are legal if and only if they are in the dictionary.
- The program should not let a user play an illegal word (this is slightly different than tournament Scrabble rules).

Scoring

- You will get the sum of scores of all the maxim that are formed by your new tiles. (By "maxima mean that they are bordered on both sides by squares or the end of the board; if you put down "Hello", you don't also get the scores for "Hell", "He", "lo", and "ell".)
- The score of a word is computed as the sum of all letter scores in the word (see next bullet), multiplied by the product of all applicable word bonuses (see two bullet points down).
- The score of a letter that was already on the board is exactly the score as printed on the letter. The score of a letter that you placed **this turn** is the score printed on the letter times the bonus letter multiplier of the spot you placed it on. (If there's no bonus letter multiplier given, this factor is 1.)
- On your turn, you may have placed tiles on one (or in rare cases more) tiles that have word multiplier bonuses. In scoring a word, the multipliers under all tiles you placed **as part of this word** are multiplied up and applied to the letter score of the word, after you've already applied letter bonuses.
- If you use `k` (i.e., the maximum possible) letters in one turn, you get a bonus of 50 points in addition to all the points that your word(s) earned.
- For some examples illustrating this scoring, scroll to the bottom of this page or see the Hasbro Scoring Examples. (Click on "scoring" and scroll down.)

Start, Process, and End of Game

- Play proceeds by player order, round robin.
- Play finishes once a player runs out of letters when all tiles have already been taken from the bag.
- At that time, each other player subtracts from their score the sum of points of all tiles that they had left. If the game ended because a player ran out of tiles, that player gets to add that sum (over all other players) to his/her score.
- Play can also finish if every single player passes their turn without any other moves occuring in that span.
- Many Scrabble players apply a maximum time per move, but we will ignore that, at least for now.

Appearance of your Game and Interaction

Your game should live in a main file of `scrabble.cpp` (though you'll likely have a number of other files). At the command line, you should just start it with `./scrabble <configuration-file-name>`

(See below about configuration files.)

Here, we will describe how the interaction between the user and the game should go. You have liberty in choosing what the game **outputs**, so long as the output is reasonably readable and understandable. However, you must exactly follow our prescribed **input** format, as we will be partly evaluating your program by feeding it automated input. If you were expecting different input formats or orders, this will not work, and you will automatically lose some points.

When the game is started, the user should be queried about the number of players. In response, the user enters a number between 1 and 8. After the number of players is entered, the game should query for the names of all players. Each name will be entered as a string, and may contain white space.

After all names have been entered, players will be queried round robin for their moves. The game start with player 1, then player 2, and so on.

When it's a player's turn, the game should show the player in a convenient form the current state of the board, the tiles the player has, and the current score of the game. It should then query the player for his/her turn. The player can enter one of the three following moves (everything could be in upper or lower case):

- `PASS`: the player simply passes his/her turn.
- `EXCHANGE <string of tiles>`: the player wishes to discard these tiles back into the bag, drawing the same number of new tiles to replace them. The tiles will all be letters (upper or lower case) or the character `?` for a blank tile. The string will contain at least one letter.
- `PLACE <dir> <row> <column> <string of tiles>`: the player wishes to place a word on the board. The first tile will be placed in the given row and column (where counting starts at 1, not 0). Row 1 is the top row of the

board, and column 1 is the leftmost column of board. If `<dir>` is `-`, then the tiles will be plac horizontally, and if `<dir>` is `|`, then the tiles v placed vertically. The string of tiles should be tiles from the player's hand (no spaces), in the order in which they are to be placed. If the player wishes to use a blank tile, he/she uses `?`, immediately followed by the character to be used for the blank. Note that if the player wants to start a word using an already placed letter or letters, then the user chooses `<row>` and `<column>` according to the first letter they place, not the first letter of the word.

Below are some examples for this command format. When the player gives a legal command, this command should be immediately executed. **Do not ask the player for a confirmation, as this will screw up our testing.** Of course, some commands may be illegal. Below, we will tell you which errors you must handle, and which you can ignore if you want. When a user makes an error, you should tell them what went wrong, and ask them for another move.

Once a player makes a legal move, the word is permanently placed on the board. You should show the player the results of their move (words formed, points earned, new letters picked up, etc.), and wait until they press return (hint: use `std::cin.ignore()`) to continue. Then, play moves on to the next player.

At the end of the game, the final scores of all players should be shown, including the winner (or winners, if there was a tie). Don't wait for any kind of input to terminate the program after the last move. For our automated testing to work, you should print a string of the form `Winner: <player>` or `Winners: <player1> <player2>` (if there are multiple winners) at some point in your score report.

Examples of commands

Here are a few examples of what you would enter, and what the effects would be.

- `PASS`: This is a command to just pass. (Duh!)
- `EXCHANGE aa?`: The player wishes to throw two copies of the letter `a` and a blank tile back into the bag and draw three new tiles.

- `PLACE | 3 3 CAKE`: The player wishes to put th
  `C`, `A`, `K`, `E` down in order, starting at row 3,
  3, going down.
- `PLACE - 5 1 POE?MON`: Imagine that the previous move (CAKE) has been executed, and no other relevant tiles are around. Then, the player is indicating that they want to build a horizontal word starting in row 5, column 1 (left border of the board), placing the tile `P` there, `O` in row 5, column 2, `E` in row 5, column 4, placing a blank tile that is interpreted as an `M` in row 5, column 5, and so on. Thus forming the word POKEMON with the `K` that was already there at row 5, column 3.

Notice that when placing tiles, you do *not* specify the tiles that your word will also be using which are already on the board, only the ones you are adding (in order).

Errors in commands

Obviously, there are a lot of errors that a user could make. Here is a list that is hopefully close to complete. There are some that you are required to handle. The other ones will not lose you points if you don't handle them, but you are of course welcome to handle them anyway, so you enjoy your own game more.

Errors you must handle:

- User is proposing to place/discard tiles he/she does not have, or does not have enough copies of (e.g., trying to use two `A` tiles when having only one).
- User is trying to place a word starting at a position that is already occupied by a tile.
- One or more of the tiles placed would be out of bounds of the board.
- One or more of the words formed by placing the tiles is not in the dictionary.
- A user is trying to make a first move that does not use the start square.
- A user is trying to make a move (other than the first move) that does not have at least one tile adjacent to a previously placed tile.

Errors you don't have to handle:

- User types a non-existing command (like `PAS`
`DISCARD`).
- User enters the wrong number of arguments fo
command.
- User enters a character other than `-` or `|` for the direction.
- User enters a non-numerical (or floating point) value for the row or column.
- User enters 0 tiles to discard or place.
- User enters tiles that are not letters or `?`.
- User ends a `PLACE` command with a `?` (not indicating what letter the `?` should be interpreted as).

When an error occurs, you should tell the user what went wrong, and prompt them for another move. So that we can test some of these errors automatically, your program should use the string "error" (case doesn't matter) at some point in every error message it prints.

Files and Formats

Since there are quite a few parts here that should be "configurable" by the user (such as language, board layout, etc.), you will be interacting with several files. There may also be some configuration options later on. In order to avoid having to give a huge number of command line arguments, we will have one master configuration file which lists all the other files.

As a general rule for this problem, we will not try to give you "stupid" inputs, for instance, you don't need to worry about us testing your game with tile sets that have more tiles than there are spaces on the board, or providing file names to files that don't exist, etc.

Configuration File

The configuration file will tell you the names of the tile file, the board file, and the dictionary file, as well as the number of tiles in each player's hand. We may later add other options to this. Here is an example of what the file might look like

```
HANDSIZE:   7
TILES:     english-tileset.txt
DICTIONARY: ./dictionaries/english-dictionary.txt
```

```
BOARD:  ./boards/standard-scrabble.txt
SEED: 99972
```

We promise you that (for now) each line will begin with one of the five keywords "HANDSIZE:", "DICTIONARY:", "BOARD:", "SEED:" or "TILES:". There could be extra spaces between the colon and the value. The items could occur in arbitrary order, i.e., "HANDSIZE:" doesn't have to be on the first line. There could also be empty lines. There won't be multiple different occurences of the same item, e.g., we won't give you two different "BOARD:" lines.

After "HANDSIZE:" and "SEED:", there will always be a valid number. In general, we are trying to make your parsing job as easy as possible here, so if you're in doubt, you can probably assume that things are well-formed (though feel free to ask). You don't have to have separate subdirectories - this was just given as an example.

Tile Set File

The tile set file will consist of multiple lines, each of which will be of the form

```
single-character points frequency
```

Here, we will not have any extra whitespace, except possibly at the end of the line. The single character is a letter on the tile (or a `?` for the blank tile). The point number will always be a non-negative integer, and gives the number of points for playing this tile. The frequency number is always a positive integer, and tells you how many copies of this tile there are. We will never give you a tileset containing upper- and lower-case versions of the same character.

An example file might look like this:

```
a 1 6
b 3 2
? 0 2
c 3 2
...
```

So you don't have to type it up yourself, we are providing you with a file for the standard English tile set. If you want to

create tilesets for other languages, you can find frequ
on the Wikipedia page for tilesets.

Dictionary File

The dictionary file will consist of multiple lines, each a single word in all lowercase letters, consisting only of letters that also appear in the tile set. Each word will be at most 30 characters long. There may be white space after the word on a line, but not before it.

For your convenience, we are providing you with an English word list.

While our example file has the words in alphabetical order and in lowercase, we do not guarantee that this will be the case for all dictionaries we test your code on.

Board File

The board file will start with a line of two positive integers `x` and `y`, the width and height of the board. On the next line will be two positive integers `sx` and `sy`, the x and y coordinates of the starting location. We will make sure that 0 < `sx` ≤ `x` and 0 < `sy` ≤ `y`. Rows are numbered from top to bottom and columns from left to right.

After that, we will have `y` rows of exactly `x` characters each. Each character will be one of the following:

- `.` indicating a normal square with no bonus.
- A single digit 2 or 3. This indicates a letter "bonus" multiplying the value of the letter placed on this square by the given number.
- A single character `d` or `t`. This will indicate a word "bonus", multiplying the value of the word by 2 (for `d`, or double) or by 3 (for `t` or triple).

No other characters will occur. For your convenience, we are providing you with the standard Scrabble board.

Code Design Advice

This is a pretty substantial project, and the first part in particular may look a bit intimidating, since there are so many parts to it. However, it can be naturally broken down into smaller components, each of which should be quite manageable. We **strongly** recommend that before writing any code at all, you spend a few hours thinking about your code organization, what classes you need, which are the key

functions in each class, where stuff gets displayed, e
Drawing some diagrams for yourself could be incredi
helpful.

To get you thinking about the right kind of questions, here are a few thoughts/questions. You will notice that there are different "correct" answers, and we are not telling you precisely how to do this. But you should decide for yourself why you make one choice over another.

There are three classes we have already implemented for you, and are requiring you to use without any modifications.

- `Tile`: A class for an individual tile (letter or blank), which encapsulates the letter on the tile, the score for the tile, and possibly later in the game for blanks what the blank is used for. In fact, we provide you with a fully implemented `Tile` class, and require that you use it. (Not because this is the only/best way to do things, but because we want you to use our `Bag` implementation, which requires this `Tile` class.)
- `Square`: This stores any multipliers, whether it's occupied by a tile already, and if so, what letter that tile has.
- `Bag`: A class for a bag of tiles. It allows the addition of individual tiles, vectors of tiles, and the drawing of sets of a specified number of tiles. The constructor reads the tile data from a specified file, and seeds the random number generator.

Here is a list of several other natural candidates for classes. We have created some of them for you, and some of the functions you might want to include, but you can add or remove any public/protected/private functions/variables/classes that you like. You can (and probably should) create more classes than what we provide. Think carefully about your choices!

- `ConsolePrinter`: Outputs the board to console. This is fully implemented, but you may make modifications if you desire a different style.
- `Player`: Captures all of the relevant information about a player (name, score, currently held tiles), and may have a function to return a move the player wishes to make. In implementing this class, keep in mind that in

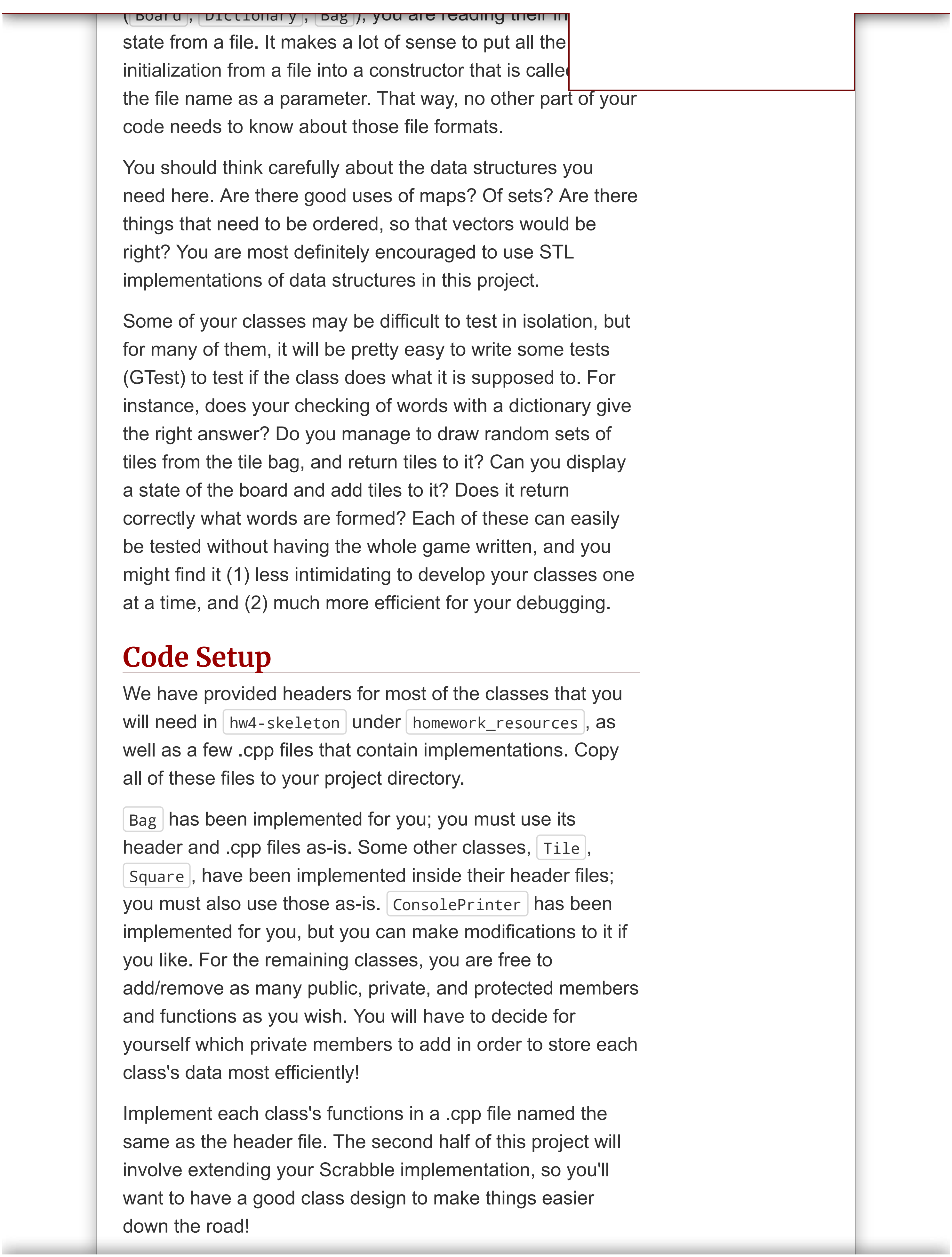
a few weeks, you will be adding computer play
the game. Does that affect your design?

- `Move`: Captures an individual proposed move, the sequence of letters, location, and direction. Alternatively, the letters to return to the bag, or just the fact that it's a pass move. This is a natural place for inheritance from a superclass `Move`.
- `Dictionary`: Should be able to check *efficiently* whether a word is allowed. Keep in mind that word lists can be quite long.
- `Board`: presumably stores what tiles have been placed where, along with the (static) information about the bonus squares. You need a function that figures out what words would be formed by a proposed move, and return those words for checking. (It's quite likely you will find this the most cumbersome function in the whole project.)
- `Scoreboard`: Is there enough functionality that's specific to keeping track of the score of all players that you want a separate class for it, or should it just be mixed in with the others?
- `Game`: Do you want a `Game` class that binds all of this together, or would you just implement that in `main`?

A few further questions to think about:

- Whose job is it to check whether a proposed move is legal? Should the `Player` class get access to the board and dictionary and call the functions for those and only ever return legal moves? Or maybe, the `Board` itself should check legality? Or maybe, the checking is split: the `Player` class checks if the player has all the required tiles, and the `Board` class checks whether the words are legal?
- How should all those errors be handled and communicated between parts of the program? Do you want to throw exceptions? Pass around results or indicator variables? What's a clean way to not overload code with too many `if` statements everywhere?

For each class that you design, we recommend that you have a separate .h and .cpp file to implement it (except possibly for classes that are just structs with extremely limited additional functionality). Also, for many of the parts



(`Board`, `Dictionary`, `Bag`), you are reading their in
state from a file. It makes a lot of sense to put all the
initialization from a file into a constructor that is callec
the file name as a parameter. That way, no other part of your code needs to know about those file formats.

You should think carefully about the data structures you need here. Are there good uses of maps? Of sets? Are there things that need to be ordered, so that vectors would be right? You are most definitely encouraged to use STL implementations of data structures in this project.

Some of your classes may be difficult to test in isolation, but for many of them, it will be pretty easy to write some tests (GTest) to test if the class does what it is supposed to. For instance, does your checking of words with a dictionary give the right answer? Do you manage to draw random sets of tiles from the tile bag, and return tiles to it? Can you display a state of the board and add tiles to it? Does it return correctly what words are formed? Each of these can easily be tested without having the whole game written, and you might find it (1) less intimidating to develop your classes one at a time, and (2) much more efficient for your debugging.

Code Setup

We have provided headers for most of the classes that you will need in `hw4-skeleton` under `homework_resources`, as well as a few .cpp files that contain implementations. Copy all of these files to your project directory.

`Bag` has been implemented for you; you must use its header and .cpp files as-is. Some other classes, `Tile`, `Square`, have been implemented inside their header files; you must also use those as-is. `ConsolePrinter` has been implemented for you, but you can make modifications to it if you like. For the remaining classes, you are free to add/remove as many public, private, and protected members and functions as you wish. You will have to decide for yourself which private members to add in order to store each class's data most efficiently!

Implement each class's functions in a .cpp file named the same as the header file. The second half of this project will involve extending your Scrabble implementation, so you'll want to have a good class design to make things easier down the road!

Besides .cpp files corresponding to the headers in th skeleton, you should write a `main.cpp` file containing main() function and any additional classes for your g You can add as many extra files/classes as you want.

You may use all containers from the C++ standard library on this assignment.

Submission Link

You can submit your homework here. Please make sure you have read and understood the submission instructions.

WAIT! You aren't done yet. Complete the last section below to ensure you've committed all your code.

Commit then Re-clone your Repository

Be sure to add, commit, and push your code in your hw4 directory to your `hw-usc-username` repository. Now double-check what you've committed, by following the directions below (failure to do so may result in point deductions):

1. Go to your home directory: `$ cd ~`
2. Create a `verify` directory: `$ mkdir verify`
3. Go into that directory: `$ cd verify`
4. Clone your hw-username repo: `$ git clone git@github.com:usc-csci104-fall2018/hw-username.git`
5. Go into your hw4 folder `$ cd hw-username/hw4`
6. Recompile and rerun your programs and tests to ensure that what you submitted works.