CS162 Winter 2018
Final Project

David LaMartina

March 20, 2018

Problem and Parameters

Goals

- Design a game with pointer-linked spaces
- Use object-oriented programming concepts

Requirements

Implement a one-player, text-based game in which the player can move through spaces to get items and accomplish goals. You (the programmer) have the freedom to decide what kind of theme the game has, so long as the requirements are met.

Space class

The game requires a Space class, which represents the space the player can occupy. The Space class must be abstract (and thus have pure virtual functions). It must also include **4 space pointers**: top, right, left and bottom. **This class will be used to create a game with a linked space structure.** Any unused Space pointers should be set to NULL.
The game should also include at least 3 classes derived from Space, to be instantiated to represent specific types of spaces on the board.
The game must include at least 6 total spaces.

Gameplay

1. In addition to a theme, the game must have a **goal** for the player to reach to achieve victory.
2. The game must keep track of which space the player is in. This tracking can be in the form a visualized space, printing the game out and indicating where the player is, or printing text describing where the player is and what characteristics the adjacent spaces have.
3. You must create a container for the player to carry items, and at least one of those items must be required as a part of the solution to accomplish the game's goal (e.g. a key to open a locked door).
4. The game must include a time limit, which limits the amount of time / steps / turns the user can take before losing the game.
5. The user must be able to **interact** with parts of the space structure, not just collect items to complete the game.

Interface

1. At the beginning of the game, the goal of the game must be declared and printed to the user.
2. The game cannot contain free-form input.
3. The game must provide the user a menu option for each scenario that requires input.
4. A printed map is preferred but not required.

Design

Theme and Goal

The game will be called "Ride the Worm," and it will be based upon Frank Herbert's 1965 science fiction novel *Dune*. The player will be a Fremen, one of the native inhabitants of the planet Arrakis (Dune). The goal will be to travel from one sietch (a town or safe spot of sorts) to another.

The player will begin at a sietch located in one spot on the map / board, and their goal will be to travel to another sietch located on another part of the board. The player will have a thirst counter, however, and that thirst counter will likely run out before they can reach the next sietch. To make it there alive, they'll have to ride the worm!

Sandworms will exist throughout the map, but they will be invisible until the player causes them to surface by using a thumper (an item to be stored in the player's inventory). Once the sandworm appears, the player will also have to use maker hooks to mount and steer to the worm towards the next sietch. Sandworms can travel much faster than Fremen on foot, and riding one will allow the player to complete their journey before they die of thirst.

If the player doesn't make it to the next sietch before their thirst counter runs out, they will die, and the game will end. If the player completes their journey, they will be prompted to either continue to another sietch (redraw the board) or end the game. The game will keep a tally of the number of successful journeys, so that (in theory), successive players might shoot for higher scores.

Implementation Overview

The game board will have a linked space structure, in which each "space" is a derived Space class object with four pointers (North, East, South and West) to base Space class objects. Thus, some of the perimeter spaces' pointers will point to NULL.

When the game begins, the board will be drawn with a random placement of 2 sietches, which will be derived Space class objects. Each board will also contain a number of worms, which will be represented by a sequence of Worm segments (also derived Space class objects). Finally, all of the spaces that are not sietches or worms will be Sand, the third type of derived Space class object.

At the beginning of each turn, the worms will "move," their movement governed by a Board class function. Their movement will involve the swapping / deletion of Sand spaces for Worm segments (and vice versa), with the caveat that a worm can never take over a Sietch space or occupy any of the 8 Sand spaces surrounding it.

Following the worms' movement, the player will be able to do the following (in this order):

- Consume spice: If a player has 1 or more Spice objects in their inventory, they will be able to consume it, granting them prescience - the ability to see underground and find out where the worms are located on the board. Consuming spice at the beginning of a turn will cause the worms to remain visible for the rest of the turn AND the duration of the following turn.

- Move: On foot, the player can move up to 2 spaces per turn. Each move will begin with a directional input (North, East, South or West). Each move has a probability of uncovering a patch of spice, which will be placed into the player's inventory if there is room. The player can move freely across sand and submerged worms.
- Thump: After each movement, the player will have the ability to use one of the thumper items in their inventory, which creates a rhythmic vibration that draws nearby worms to the surface, making them visible. If the player is immediately adjacent to a worm segment, and if they have a maker hooks item in their inventory, they can choose to ride the worm. If they cause a worm to surface when they're right on top of it, they'll die, and the game will be over.
- Ride: If the player is able to mount and ride a worm during their turn, they will be able to ride that worm up to 10 total spaces in up to 2 different directions. "Riding" will involve the swapping of Worm spaces for Sand spaces (and vice versa). After the ride, the worm will submerge, and the player will continue their turn.

The game will begin with the player immediately adjacent to the starting sietch. Their thirst counter and inventory (maker hooks, thumpers and spice) will be constantly displayed. Once a move takes them into the ending sietch, their inventory items (hooks and thumpers) will be partially replenished, giving them a shot at getting to the next sietch, should they choose to continue.

Space Class

The Space class is the basic unit of the board. An abstract class with pure virtual functions, it cannot be instantiated. It is only programmed to allow for the creation and instantiation of the Sand, Sietch, Bush and Worm derived Space classes.

- Data members
  - 4 Pointers to Space objects (north, east, south, west)
  - nextSegment and prevSegment Space pointers (For connecting worm segments)
  - bool spice: Represents presence (or lack) of a spice object that may be picked up. Set to true or false probabilistically during board setup. Set to false when player picks up spice.
  - bool wormSwap: If true, a Worm segment / object (the only "moving" part of the board) can be swapped with this space.
  - bool submerged: Determines whether space is visible to player - only changed back and forth for Worm class
- Functions
  - Constructor: sets Space pointers (usually to null, but not always) and sets spice, wormSwap, submerged to false by default
  - Destructor: Deletes all space objects
  - void move (pure virtual): Represents player moving into that space; specific action determined by derived class
  - bool isWorm (pure virtual): Returns true if worm segment; used by Board class to determine whether a space is a worm segment when it shifts the worms
  - void print (pure virtual): Used to print the board. Each overridden print function will print a different character. Takes a pointer to a Player object, so it can assess whether that player has activated prescience
  - void emerge (pure virtual): makes a submerged space emerge
  - void submerge (pure virtual): makes an emerged space submerge
- Extra: Space will have friend classes Board and Worm. Friend class Board allows the Board to access critical member variables of nodes, and friend class Worm allows the Worm class to recursively call emerge, so that all worm segments for a given worm can be emerged when any one of them is emerged by a thumper.

Sand Class

The Sand class is the most prevalent type of space on the board for Ride the Worm. Each time a player moves over a Sand space, there is a chance they will be able to pick up a Spice object (assuming their inventory isn't full). In order to ride a worm, a player must be occupying a Sand space immediately adjacent to a Worm segment.

- Functions
  - constructor: sets spice to true or false based on random number generation, sets wormSwap to true (All sand spaces other than the 8 spaces surrounding a sietch can be swapped for a worm segment).
  - void move (overridden): Allows player to move, has a certain probability of adding spice to player's inventory. Decrements thirst counter
  - bool isWorm (overridden): returns false
  - print (overridden): Prints an empty space
  - emerge and submerge: Do nothing; defined only to allow these functions to be called on every board space
  - print: Prints a '+' character, unless wormSwap is false (surrounding Sietches) in which case it prints an empty space
  - Friend class Board

Sietch Class

There are two Sietch spaces / objects per board - the starting Sietch and the ending Sietch. The Sand spaces immediately surrounding a Sietch cannot be swapped for Worm segments, so a player is "safe" from being killed by a worm as they leave their starting Sietch. If the player arrives at (moves into) the destination Sietch, their inventory will be at least partially replenished, and they will have the option to continue the game via another journey to another destination Sietch.

- Functions
  - void move (overridden): Replenishes maker hooks, thumpers and thirst counter
  - bool isWorm: returns false
  - emerge and submerge: Do nothing; defined only to allow these functions to be called on every board space
  - void print (overridden): Prints an 'S' character
  - Friend class board

Bush Class

This derived Space class represents a Creosote Bush, which has hyrdophobic leaves that prevent the loss of water. These spaces serve as a way for the player to avoid depletion of their thirst counter. If a player moves onto a Creosote Bush, their thirst counter will not be decremented at the end of that turn.

- Functions
  - void move: Increments, rather than decrements thirst counter
  - bool isWorm: returns false
  - print: Prints a 'B'
  - emerge and submerge: Do nothing; defined only to allow these functions to be called on every board space
  - Friend class Board

Worm Class

The Worm class is a derived Space class representing a segment of a sandworm - the create the player will most likely need to "ride" to complete their journey to the destination sietch without dying of thirst.

- Data members
  - bool head: Only set to true when worms are created at the beginning of a game - First segment set to head, others have head set to false
- Functions
  - void move (overridden): If submerged, has same actions as Sand class move. If above ground, movement is handled by Board class functions (player may still be able to move onto the emerged segment, provided they have maker hook)
  - isWorm: returns true
  - print: Prints various characters representing head and body, depending upon whether worm is submerged and player's prescience ability is active. It prescience is not active and the worm is submerged, this function will just print a '+', making it appear as a Sand space.
  - emerge: Sets submerged to false, changing the results of Worm's print function the next time the board is printed. Also changes the behavior of Board class functions which allow the player to mount and ride the worm. Emerge also recursively calls itself via nextSegment and prevSegment pointers, so that if one segment is emerged with a thumper, all segments will come above ground.

Item Class and derived classes

The Item class has only one function: A pure virtual getType function. Its three derived classes are:

- Spice: Represents the spice malange, which grants the player prescience (ability to see worms underground)
- Hook: Represents the maker hook, the tool necessary to mount and ride a sandworm
- Thumper: Represents the thumper, a tool necessary to shake the ground and cause nearby worms to surface.

Each of these three classes only contains the defined virtual function, returning an enum type variable SPICE, HOOK or THUMPER. These enum variables likely won't even be used; The items are intended only to fill containers to affect player behavior, as they attempt to use and wisely conserve resources in their dangerous trek across the dessert of Arrakis (Dune).

Player Class

A Space pointer called player is used to do most of the actual manipulation / interaction with the game's spaces. The Player object, however, is necessary for "holding" the player's items and possessing the prescience trait - the ability to see all worms on the board, whether they are emerged or submerged.

- Data members
  - int thirstCounter: Decrements for most player moves; if this reaches 0, player dies, and the game is over.
  - startThirst: Starting thirst value. Allows for game variable manipulation and the restoration to original thirst counter size upon successful completion of one game in a series
  - bool prescience: Flag variable for spice-induced prescience. If true, the board's print function will display all of the worms when it prints, whether they are submerged or not
  - 3 Item**: spiceContainer, hookContainer, and thumperContainer. These inventories are separated to allow for manipulation of the specific amounts of different items the player can carry. E.g. it makes sense to only allow the player to carry one spice object at a time, as such a restriction will force at least some strategy regarding WHEN to use it.

- Functions
  - Constructor: Initializes startThirst and thirstCounter and the sizes of the 3 containers according to passed-in parameters, making it easy to adjust those variables during testing. Also initializes prescience to false.
  - Destructor: Deletes the allocated memory in all 3 inventory containers.
  - setInventories: Called by constructor to set up initial inventories by allocating their memory and setting the arrays' values to either null or their respective items.
  - Getters for thirstCounter, prescience, and the numbers of spice, maker hooks, and thumpers
  - Setters for prescience, thirstCounter incrementation and decrementation, and thirstCounter restoration.
  - isEmpty and isFull functions that can be applied to any of the 3 inventory containers
  - Item-adding functions for the 3 inventory containers
  - Item-"using" (subtracting) functions for each of the 3 inventory containers

Board Class

The Board class is the primary class responsible for the implementation of the Ride the Worm game. It is essentially a 2-dimensional linked list / linked grid, with its Space "nodes" each having 4 directional pointers (north, east, south and west). Its functions are divided among 3 source files - one responsible for board construction, one responsible for worm setup and movement and one responsible for player movement.

- Data members
  - Space* upperLeft: Somewhat like a "head" pointer in a linked list, always points to the Space in the upper left corner
  - Space* player: Pointer to space player is occupying
  - Space* destination: Points to the destination sietch, so that the program can analyze when the player has arrived at their destination.
  - Space** worms: A double pointer used to point to an array of Worm object pointers. These pointers will point to the "heads" of the worms, which are responsible for "guiding" their movement during each move of the game.
  - wormDensity: Determines the size of the unit square per worm. This variable can easily be manipulated, and the smaller it is, the more worms there will be on any given instantiated game board.
  - maxWormSize: The longest possible length of a sandworm on the board
  - wormsSize: The size of the array pointed to by the worms Space**. wormsSize is a function of wormDensity, numRows and numCols.
  - ints numRows and numCols: may be fixed or variable for game implementation; define the dimensions of the board (number of Spaces to be instantiated)
- Board Construction & Print Functions
  - Constructor initializes numRows, numCols, wormDensity and maxWormSize based on passed-in integer values and instantiates the entire board. The board is first created by instantiating a grid of Sand spaces, and then placeSietches, wormSafe, placeWorms and growWorms are called to swap out those Sand spaces for sietches, bushes and worms.
  - spaceSwap: Takes a recently allocated (but not yet integrated) derived Space object (newSpace) and swaps it for an existing space on the board (oldSpace). This function is called often, when a newly allocated space must be swapped for an old one.
  - Space* sandSwap: Swaps existing old space for EXISTING new space, putting the new Sand space in the old space's place. Returns a pointer to the new Sand space, which can be used in turn to call sandSwap again. Intended only to be used in a loop when moving worms.

- goToSpace: Returns a pointer to a specific space on the board based on coordinates (traverses the board based on passed-in integers and returns the pointed-to space it lands on)
- placeSietches: Instantiates and swaps out starting and ending sietches in the upper left 16th and lower right 16th corners of the board in a semi-random fashion. Also places the player pointer at the starting sietch and the destination pointer at the ending sietch.
- wormSafe: Called by placeSietches post-placement to create a perimeter of 8 non-worm-swappable spaces around each sietch. Creates a "safe zone" around each one
- placeBushes: Swaps out Creosote bushes semi-randomly based on a given probability throughout the board.
- Destructor: Traverses board, deleting all Space objects
- printBoard: Prints board by calling print function via every Space object. Each Space's print function takes the pointer to the current Player object, which will contain either a true or false value for prescience, informing the Worms' print functions.

- Worm-specific functions
  - placeWorms: Places worm heads pseudo-randomly (randomly within each unit square based on wormDensity)
  - growWorms: "Grows out" worms from worm heads via worms Space** pointers in a semi-random fashion
  - moveWorms: Causes every worm to attempt movement by moving their respective heads (pointed to by worms Space** pointers).
  - wormFollow: Called by moveWorms for every worm head - Causes worm segments to "follow" their head through calls to sandSwap
  - thump: Makes worms come to surface if they're submerged and within a given radius of player. Returns true if the player is standing right on top of the worm when called (This kills the player / ends the game)/
  - submergeWorms: Calls submerge via every Space pointer; only causes any real action when called on Worm spaces. Called at the end of a turn, causing all of the emerged worms to re-submerge.

- Player movement functions
  - movePlayer: Takes a direction (enum type) and a pointer to the Player object. Returns a MoveSuccess enum type object that indicates the nature and success of the move (move successful / unsuccessful or successful / unsuccessful attempt to mount and ride a worm.) If the player is able to mount a worm, the rideWorm function is called via movePlayer.
  - placePlayer: Places player pointer at a certain spot; returns false if the chosen spot is going to be an invalid move (if it's going to land the player on top of an emerged worm). Intended only for use by the rideWorm function of wormGame, since normal moves are conducted via the movePlayer function.
  - hasArrived: bool-returning function used to check whether player has arrived at the destination sietch. Called via runTurn of the WormGame class to check whether player has arrived before each turn.

WormGame class

The WormGame class is used to implement the Ride the Worm game using a pointer to a Board object and a pointer to Player object. Its functions are used primarily to call the Board class functions in a cohesive manner and output some of their results.

- Data members
  - Board* gameBoard
  - Player* gamePlayer

  - integer variables for numRows, numCols, wormDensity and maxWormSize - all used to call the Board constructor with the parameters determined at runtime
  - bool isBoardRandom - If this variable is set to true, calling the setBoard function will lead to a semi-random instantiation of the next game board.

- Functions
  - Constructor: Simple, initializes data members (including pointers to null)
  - Destructor: empty
  - newPlayer: creates new Player object via gamePlayer pointer - to be called by runSeries
  - deletePlayer: deletes Player object via gamePlayer pointer. Called via runSeries when the series is through.
  - newBoard: Instantiates board, either randomly or based on board parameters determined in runSeries.
  - setBoard: Called from runSeries if the player decides to set their own boar parameters
  - deleteBoard: Deletes current game board via gameBoard pointer, so another can be create for another game in the series.
  - useSpice: Allows player to choose to use a spice object, if one is in their inventory, showing them the locations of all worms on the game board.
  - useThumper: Allows player to choose to use a thumper object, if one is in their inventory, causing nearby worms to emerge.
  - makeMove: Allows player to choose which direction they want to move in (or no move at all). If player has maker hooks and moves onto a worm segment, makeMove will call rideWorm.
  - rideWorm: Allows the player to choose to be placed up to 10 spaces away in up to 2 directions.
  - printStats: Prints player's stats, including prescience (active / not active), thirst counter, and inventory statuses.
  - runTurn: Runs one set of 5 player moves, each move including the chance to use spice, chance to use a thumper, and a chance to move. Each move also includes the potential movement of all worms on the board. Returns a TurnResult (enum type) variable, indicating that the game should be continued, that the player died (thirst of thumped a worm), or that the player arrived at the sietch.
  - runGame: Calls runTurn in a loop until the player either dies or arrives at the destination sietch.
  - runSeries: Runs games in a loop according to the maximum number of games the player has chosen is reached. The goal of the game (at large) is to last for as many games as possible / make as many successful journeys as possible.

Tests

The "hard part" of this program, by a long shot, was coding the board and the instantiation and movement of worms. Getting their movement to work properly without segmentation faults and memory leaks was half the battle. So, I've split the testing of this game into two parts: the worms and their movement, and the game itself. To test the worms, I've included a worm test option in the final program.

Worm Tests

To test the worms, I coded the same parameter restrictions for the worm test part of the program as for the player-determined parameters of the game. These include the board parameters of:

- number of rows between 15 and 50
- number of columns between 15 and 50
- Worm Density between 5 and 15

- Max Worm Size between 5 and 15

I also set the number of tested turns between 100 and 10,000.

Normally, given the way the game is programmed, the worms wouldn't actually be visible, since the test involves no spice and no thumpers - thus no prescience and no emergence of submerged worms. To make sure the worms would be visible, I just instantiated a board and Player within the main file, set the Player's prescience to true, and called the print function with that always-prescient Player for every iteration of the test.

For the specific tests, I used combinations of 15, 30 and 50 for the board dimensions and various worm density and max worm size combinations. I ran each of the following tests through valgrind for 1,000 iterations. Each returned no memory leaks, no errors and no memory still accessible. Worms did get "stuck" sometimes, but that is acceptable given the programmed conditions for them NOT being able to move in certain directions (can't go out of bounds, can't go through bushes, can't swap one worm segment for another). In fact, a high frequency of worms getting stuck likely means a less-than-viable set of board parameters (too many worms of too great a length for a given board size).

Board dimensions	**Worm Density**	**Max Worm Size**
15 x 15	5	5
15 x 15	5	10
15 x 15	5	15
30 x 30	5	5
30 x 30	5	10
30 x 30	10	15
50 x 50	5	5
50 x 50	5	10
50 x 50	10	15
15 x 30	5	10
15 x 50	5	10
30 x 50	5	10
30 x 15	5	10
50 x 15	5	10
50 x 30	5	10

Game Tests

Since each game involves a wide variety of inputs and potential happenings, I decided not to use the same type of test table I've used for simpler programs in this course. Instead, I focused on various functionalities and results that should be seen, given a certain set of inputs and conditions. These piecemeal test were conducted over several game runs, most of which used the pre-programmed set of board parameters: 30 x 30 board, worm density of 6, max worm size of 10. The results and observations of these tests are as follows:

- Thumpers
  - The use of a thumper when prescience is not activated causes nearby worms to surface, as verified by the later use of spice to see where the emerged and non-emerged worms "really" are / were.
  - The use of a thumper when prescience IS activated causes the worms within the 5-space perimater square of the player to emerge.
  - Using a thumper when directly on top of a submerged worm causes the player to die, an output message to be printed saying as much, and the game AND SERIES to end.
- Spice
  - The use of spice at any point in the turn (during any of the 5 moves) results in all worms being displayed with the appropriate characters being printed ('U' for the head and 'O' for the body segments, instead of 'H' and 'W', respectively).

- Movement
  - The player can move in any direction, provided they don't attempt to go out of bounds.
  - Moving onto a Sand space results in a 1 in 4 chance of receiving a Spice item in the inventory.
  - Moving onto a Bush space results in the thirst counter being incremented, rather than decremented.
  - Moving onto a prescience-revealed submerged worm segment results in similar functionality to moving onto a Sand space (there is a chance of getting spice).
  - Attempting to move onto an emerged Worm segment is an invalid move when player has no maker hooks
  - When the player arrives at the destination sietch, their thirst counter, maker hooks and thumpers are replenished.
- Worm Riding
  - When the player moves onto a worm segment, a maker hook is subtracted from their inventory, and they are prompted to move up to 10 spaces in up to 2 directions.
  - If the player chooses to move all 10 spaces in the first direction, they are not prompted for the second move.
- Thirst Counter
  - The thirst counter decrements for every move, except for a move onto a Creosote bush or a sietch.
  - When the thirst counter reaches 0, the player dies (if they have not reached a sietch), and the game and series are over.
  - If the player reaches the destination sietch just as their thirst counter hits 0, they still win that game and progress to the next.
- Player stats
  - The thirst counter is decremented for every move onto a Sand or Worm space.
  - When prescience is active, the stats reflect that activation.
  - The spice container reads the correct amount of spice (either 0 or 1, since that container only allows for 1 item).
  - The maker hooks container starts at 10 and decrements for every worm ride.
  - The thumpers container starts at 10 and decrements for every thump.
- Game organization
  - The game begins with a prompt to either play the game, test the worms, or exit.
  - Choosing to play the game results in prompts for:
    - The maximum number of games in the series
    - Random / user-selected / pre-programmed parameters - Pick one
    - User-selected parameters: Prompts for rows, columns, worm density and maximum worm size
  - Selecting random board parameters (effectively always) results in a board of different dimensions for every game in the series.
  - Even when fixed (pre-programmed or user-selected) parameters are selected, there are slight variations in the board from one game in the series to the next (as there should be).

Challenges, Changes and Solutions

This program was larger than anything I've created before. It was relatively straightforward, but there were plenty of variables whose interactions I had to consider. Given the pointer-linked space / board structure, there were also plenty of opportunities for segmentation faults and memory leaks that I needed to account for. Here are the biggest things I had to tackle, consider and in some cases change as I went about designing and coding the program.

- Player pointer parameter for print function: I knew I wanted to be able to call print via every space on the board, so that would need to be a pure virtual function in the Space class and overridden in the derived classes. However, I realized that if I was going to print Worms differently based on whether they were submerged or surfaced - and based on whether or not the player had activated prescience - I needed a way to get extra information to the print function. So, I set it up to receive a pointer to a Player object as a parameter, so it could analyze whether or not that Player object had activated prescience.

- Additional Space pointers: nextSegment and prevSegment: These pointers weren't in my very first design, but I soon realized they would be necessary. In order to cause all of the segments of a worm to 1. move in coordination and 2. surface in coordination in response to a thumper, I would have to link them via another set of pointers. Creating the nextSegment and prevSegment pointers for the base Space class allowed this, and using these pointers instead of the directional pointers made it possible to use sandSwap to swap spaces piecemeal as the worms move segment by segment.

- Updating upperLeft and player pointers: I spent a couple of hours in frustration, trying to figure out why I was routinely getting segmentation faults whenever worms moved into the upper left corner of the board. I then realized it was because I hadn't accounted for the swap-out of the upperLeft pointer in the spaceSwap function. To make sure this didn't happen with other pointers as I kept coding, I modified the code to ensure that player and upperLeft would always be pointed at the appropriate spaces if the spaces they are "on" are swapped out.

- Variability in constructors: This was an evolving set of considerations, but overall, I wanted to make the game's code easy to manipulate, so that, in theory, I could come back in and change up a host of parameters to affect the strategic requirements of the game. To do this, I generally created constructors with several parameters, rather than rigidly initializing variables such as numRows, numCols, wormDensity, etc. Some of these are even modifiable to a degree at runtime, but the main point here is that I wanted them to be easily changeable if I or someone else were to go back into the code.

- Emerge and submerge functions: My earliest design didn't include these Space class member functions, but I realized it would be much easier to include them than to try to modify Space class member variables via the Board class (even though the Board class is a friend of Space and all derived Space classes). In particular, creating an emerge virtual function allowed it to be called recursively. This proved invaluable in creating a simple solution to the problem of making all segments of a worm emerge when any one of them was revealed by a thumper (making the entire worm surface at once, rather than just one segment). A similar functionality could have been applied to the submerge function, but that wasn't necessary since it's called for the entire board at once, at the end of every turn, via submergeWorms.

- Generalizations for manipulating large swaths of spaces: This is one area that still stumps me. My thump function (and to a lesser degree, wormSafe) are large and clunky, but I ran out of time to figure out a more elegant, generalized way to call emerge on every space within a given surrounding area - WHILE ALSO accounting for going out of bounds (null pointers). This is one thing I'd change, if possible and if given more time.