Jesse McKenna
CS162
12/5/2017
Project 5: Text Game

I. Design

1. *Problem to be solved*

Create a text-based game that incorporates spaces that the player can interact with, an inventory to store collected items, and a time limit.

Decided to make a game with a cave theme, in which the goal is the exit, but it is blocked by holes that have to be filled in with rocks, which require a pickaxe to mine. An inventory size of two (including pickaxe) means the player can only hold one rock at a time, and has to plan ahead to reach the exit before running out of steps.

2. Pseudocode solution

```
introduction
menu:  play or exit?
        if exit: quit

play:   create board and place player at origin
                read "map.txt" file line by line
                board cols = length of first line
                for each line, create next board node based on the map char
                board rows = number of lines
        initialize steps
        print goal of game
        until user chooses to quit:
                print board
                show commands: WASD to move, E to interact, I to show inventory, or Q to quit
                if Q: quit
                if E: interact
                        if current space is pickaxe, menu: pick up pickaxe?
                                if choose to pick up, add pickaxe to inventory
                        if current space is rocks:
                                if have pickaxe and inventory not full, menu: mine rocks?
                                        if choose to mine rocks, add rocks to inventory
                                if have pickaxe and inventory full, print a message
                                if don't have pickaxe, print a message
                if I: print inventory
                if WASD: move
                        arrive
                                if new space is hole:
                                        if don't have rocks, game over
                                        if have rocks, 'fill' hole (change symbol to empty space)
                                if new space is exit:
                                        win
                if quit: quit game
                if steps < 0, lose

menu:  play again or quit?
        if play again: repeat
        if quit: quit
```

3. Implementation details

Space class:

- Member variables:
  - const char PLAYER_SYMBOL
  - Space* top
  - Space* bottom
  - Space* left
  - Space* right
  - enum Direction {UP, DOWN, LEFT, RIGHT}
  - bool player
  - bool walkable
  - char symbol
  - char defaultSymbol
- Member functions:
  - Constructor
  - virtual Destructor
  - bool isWalkable()
  - bool hasPlayer()
  - void setPlayer(bool status)
  - char getSymbol()
  - virtual void setSymbol(char symbol)
  - void setAdjacent(Direction direction, Space* target)
  - Space* getAdjacent(Direction direction)
  - pure virtual void arrive(Player* player)
  - pure virtual void inspect(Player* player)

emptySpace subclass: can hold items

- Member functions:
  - Constructor
    - walkable = true
    - defaultSymbol = ‘ ’
  - Destructor
  - virtual void arrive(Player* player): does nothing
  - virtual void inspect(Player* player):

    if symbol is ‘P’ (pick), player can pick up pickaxe

    if symbol is ‘^’ (rock), player can pick up rock if holding pickaxe

    if symbol is ‘ ’ (empty), does nothing

    - virtual void setSymbol(char symbol): used during board creation to place items

exitSpace subclass: wins the game if reached
- Member functions:
    - Constructor
        - walkable = true
        - defaultSymbol = ‘E’
    - Destructor
    - virtual void arrive(Player* player): set player state to WIN
    - virtual void inspect(Player* player)

holeSpace subclass: loses the game if reached without holding rocks
- Member functions:
    - Constructor
        - walkable = true
        - defaultSymbol = ‘@’
    - Destructor
    - virtual void arrive(Player* player):
      if player has rocks, set symbol to ‘ ’
      if player does not have rocks, set player state to LOSE
    - virtual void inspect(Player* player): does nothing

wallSpace subclass: impassable border space
- Member functions:
    - Constructor
        - walkable = false
        - defaultSymbol = ‘#’
    - Destructor
    - virtual void arrive(Player* player): does nothing
    - virtual void inspect(Player* player): does nothing

Game class:
- Member variables:
    - const int START_STEPS
    - Player* player
- Member functions:
    - Constructor
    - Destructor
    - play()

Board class:

- Member variables:
    - int boardRows
    - int boardCols
    - Space* origin
    - Space* playerSpace
    - string mapFilename
    - string map: to hold all chars read from map.txt file
- Member functions:
    - Constructor
    - Destructor
    - Space* getPlayerSpace()
    - void print()
    - bool playerMove(char direction): to accept WASD input
    - void readMap()
    - Space* createSpace(char type)

Player class:

- Member variables:
    - enum State {NONE, PLAYING, WIN, LOSE}
    - State state
    - const int INVENTORY_MAX: max size for inv
    - vector<char> inv
- Member functions:
    - Constructor
    - Destructor
    - void setState(State newState)
    - State getState()
    - void addItem(char item)
    - bool hasItem(char item)
    - bool useItem(char item)
    - void printInventory()
    - bool inventoryFull()

II. Test Table

Test Case	Input	Driver Functions	Expected	Observed
Main menu: Play	1	Menu class, switch statement	Continue to next prompt	As expected
Main menu: Exit	2	Menu class, switch statement	Exit program	As expected
Pick up pickaxe	1	emptySpace::interact(), player::addItem()	Pickaxe added to inventory	Inventory had multiple pickaxes – realized inventory from previous plays was not cleared. Fixed by creating and deleting player within each play, not in constructor/destructor
Walk to hole without rocks	WASD	holeSpace::arrive()	Game over	As expected
Walk to hole with rocks	WASD	holeSpace::arrive(), player::hasItem()	Hole changes to empty symbol, rocks removed from inventory, player continues onto space	As expected
Inspect rocks without pickaxe	E	emptySpace::inspect(), player::hasItem()	Display message	As expected
Inspect rocks with pickaxe and inventory full	E	emptySpace::inspect(), player::hasItem(), player::inventoryFull()	Display message	As expected
Inspect rocks with pickaxe and inventory not full	E	emptySpace::inspect(), player::hasItem() player::inventoryFull()	Display menu with option to mine rocks	As expected
Choose to take rocks	1	player::addItem()	Rocks added to inventory	As expected
End menu: Play again	1	Menu class, switch statement	Restart game	As expected
End menu: Exit	2	Menu class, switch statement	Exit program	As expected
valgrind	valgrind ./main	-	No leaks found	

III. Reflection

Included constants in symbols.hpp for board symbols

I knew from the start that I should make the symbols for my board (ex. ‘X’ for player, ‘ ’ for empty space, etc.) constants, but I initially had them scattered around throughout the program, closest to where I thought I would need them. This created issues as I added more components, however, because they all seemed to need the same constants. Passing the constants up through the chain of inheritance seemed cumbersome, but including the same constants in multiple places was obviously even worse, because it would introduce the possibility of conflicting sources of information.

I eventually decided to simply make a helper header file containing nothing but the char constants used throughout the game. I then simply included this header file anywhere that those constants were needed. It made a huge difference to the simplicity of my program’s implementation, and ensures that one unique char represents a given space or item throughout the program, from board creation to inventory management and player interaction with spaces.

Created board from map.txt file instead of manually

My first rough draft of the program simply created an nxn board of empty spaces that a player ‘X’ could move around in. Once I had designed the level that I wanted to implement, however, I realized how difficult it would be to specify where each type of space should go.

The solution that I settled on was to simply draw the map in a .txt file, then read the file line by line, creating the in-game ‘board’ accordingly. Although this took some work to set up, the benefits were immediately obvious once it was implemented. I am able to easily change the map if I wanted to, the board size is automatically measured based on the .txt file, and the code for board creation is much more readable than it would have been had it included a bunch of special-case locations for various space types.

Difficulty deciding where player.hpp should fit in

The biggest struggle I had during this project was determining where the player should fit in code-wise, and what kind of entity the ‘player’ should be represented as. Because the player has to be displayed on screen, I initially thought it would make sense to player to be a type of Space, which has a member variable for its inventory, knows its own coordinates, and can move itself.

However, because Board manages all the spaces in the game, it became clear that it makes more sense for Board to keep track of the player’s location, and to move the player around. Also, making the Player a space would have meant that every time the Player was moved, it would be replacing the existing space in that position, which would then have to be reinstated when the player left. This seemed like a lot of extra work, and I ultimately decided instead to simply keep track of a pointer to the space where the player currently is, and temporarily change that space’s symbol to ‘X’, while preserving all its other attributes like normal.

The final challenge was how to allow the spaces access to the player's inventory (such as when a hole space uses up rocks that a player is holding) while also allowing the Game class to know when a player has reached a game over condition from falling in a hole or from reaching the exit. The solution to this was to contain player in Game, which is higher in the inheritance hierarchy, and to pass a pointer to Player to the spaces when interacting with them. I am not sure if this is the optimal method, and I would be interested to find out what others would have done to solve this issue, but it seems to work well enough for the program.