Requirements

Game using pointer-linked spaces

Space Class (abstract)

- represents space the player can be in
- pure virtual functions and non-pure-virtual functions
- 4 Space pointers
    - top, left, right, bottom
    - can add more pointers if needed

NOTE: If structure is linear, still need 4 pointers. Unused pointers will be set to null

3 Derived Classes from Space Class

- Each representing different type of space
- Must have a special action for the player to interact with
    - Opening the door to another space, attack the monster, turn on the light switch, sing a song to please the king

6 spaces for the game at minimum

Gameplay

- Theme required
- Goal required
- Keep track of which space the player is in
    - visualized space - print out map
    - print text describing where player is at, and what adjacent spaces are around the player
- Create a container for the player
    - carries items
    - must have capacity limit
- Items for the player to obtain and place in the container
    - 1 or more items must be part of the solution to reach the goal
- Time limit
    - examples
        - limits the amount of time/steps/turns the user can take before losing
        - health system that decreases players health from space to space
        - NOTE: give enough steps to allow the game to perform testing
- User must be able to interact with parts of the Space structure, not just collecting items
    - add room interactions, attempts to escape but they don't work

Interface

- At start, goal of the game must be declared and printed
- Game cannot contain free-form input
    - typing kitchen to go to the kitchen
- Must provide user a menu option for each scenario of the game
- Not required to print a map, can be text based
    - map preferred

Design

Main

- dynamically allocate all derived rooms (game class?)
- dynamically allocate all derived item types (game class?)
- ~~pass rooms into Game functions to allocate space pointers to point to derived rooms~~

Space Class

- variables
    - string description
    - string clues
    - bool valid (empty space = false, valid = true)
    - bool item (item = true, no item = false, use to display pick up item prompt)
    - bool leak
    - bool player (player on space = true, off space = false)
- functions
    - pure virtual
        - items?
        - actions (item parameter?)
            - damage character
            - doorways
    - getters and setters
- ~~stack STL structure? probably not~~
    - ~~LIFO~~
    - ~~push all spaces on at the beginning?~~
        - ~~what happens when you pop a space off, where does it go? where are its values~~
- array of Space pointers to derived classes
- pointers - up down left right
- Derived Classes
    - Start Room Class
    - Leaking Room Class
        - can be plugged
    - Room With Item Class
        - item object

        - add to player inventory by setting item pointer to derived item object in main?
    - Escape
        - ends game, player escapes successfully
    - Empty space
        - description: “There’s nothing here.. I should keep moving.”

Item Class

- variables
    - prev pointer
    - next pointer
- functions
    - getters and setters
- Derived Item Classes
    - Plug Class
    - First Aid Class

Backpack Struct

- variables
    - item pointer stack?
    - bool full?

Player Class

- a player is passed to each space? or game?
- should contain container of items
- variables
    - backpack struct
        - contains items
    - int health = 100
        - each leak unplugged per round -20?
        - depends on leakCount
    - Space* location?
        - Row and Col
- functions
    - pickupItem(derived item &’s)
        - takes an item

Game Class

- variables
    - 5x5 array of space pointers for map
        - randomize room types? maybe later
            - rand() % 4 + 1
            - result generates specific type of room for space pointer to point to
    - player object
        - backpack struct
            - item stack
    - Player pointer

    - leak count
        - increase after round 2, 4?
        - output leak messages after these rounds
        - decrease player health per leak
- functions
    - playGame()
        - do while menu per round?
        - each round, move, use item, get room descriptions, get room clues
        - calculate health,
    - printMap()
    - initMap()
        - set all spaces to bool valid = false
            - can be set in derived class constructors
    - allocMap(room types &)
        - change spaces used to valid = true?
        - set space pointer to point to new derived space class?
        - put player on start space[2, 2]
            - setSpace[2, 2] = pStartSpace[0]
            - getSpace[2, 2].setBoolPlayer = true
            - keep track of player row and col?
        - borders
            - set these pointers to null
            - [row = 0, all col] up is null
            - [row = 5, all col] down is null
            - [all rows, col = 0] left is null
            - [all rows, col = 5] right is null
            - should cover corners
    - deallocMap()
    - manageHealth()
    - movePlayer()
        - moves player on board, how?
        - getPlayerLocation()
    - checkitem()
        - see if item is present
    - checkleak()
        - see if leak is present
- Contains space pointers?
    - space bool valid set to false for non allocated spaces
    - space pointers
    - point to derived class objects
- allocate derived space objects
- randomly assign spaces and push onto stack?
- pointer to current space, select top of stack?

        - once moved, pop space off, set current space to new top?

Menu

- variables
- functions
    - displayMenu
    - validateMenuChoice(Game* pGame)
        - runs game functions
        - takes a game pointer
    - validateInt(key)
    - runMenu

Story

Chapter 1 I wake up. It's dark in here. Where am I? *drip* …*drip* ..*drip*... There's something leaking from one of the walls. I'm not feeling well… I need to get out of here. If I fail to stop it, maybe there's a way to escape. What should I do? move forward, backwards, left, right inspect health inspect room
Chapter 2 (starts at health <= 80) The leak is getting worse. I can feel liquid around my feet.
Chapter 3 (starts at health <= 60) It's up to my knees now. I should hurry.
Chapter 4 (starts at health <= 20) It's almost up to my shoulders. Time's running out. I have to move.
Ending (Good) Player escapes
Ending (Bad) Player dies

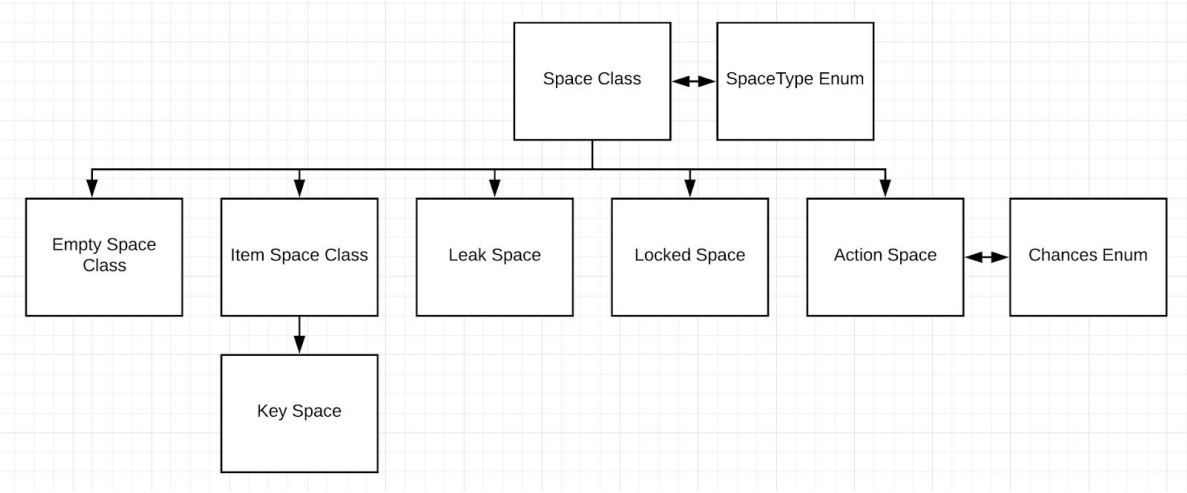
After first move

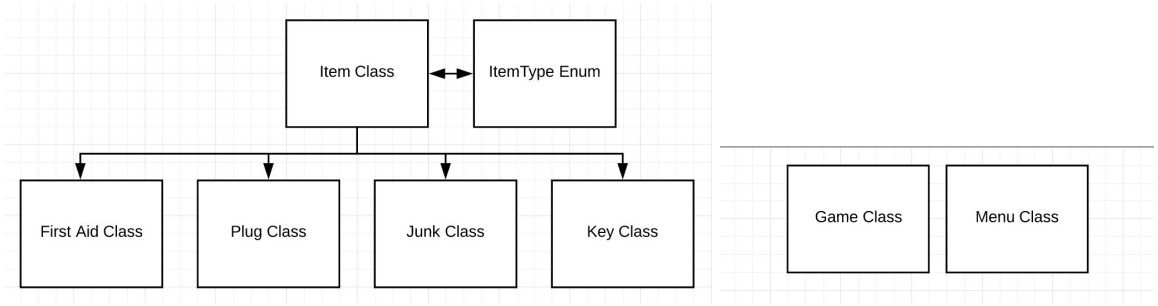
- The leak is getting worse. I can feel liquid around my feet.
- get space description
- What should I do?

Map

	0	1	2	3	4
0	Key	Item	Leak 1	Key	Exit
1		Item	Item	Item	Door
2	Exit	Leak 2	START	Item	Item
3	Item		Leak 3	Item	Leak 4
4	Hatch	Item	Tunnel	Key	Exit

Class Hierarchy





Test Plan

Validate Start Menu Choices

Test Cases	Input	Functions	Expected Output
Invalid input (string)	“test”	startCheck() validateStartChoice()	Display error message, prompt for input
Invalid input (empty)	empty	startCheck() validateStartChoice()	Display error message, prompt for input
Invalid input (signed integer)	-1	startCheck() validateStartChoice()	Display error message, prompt for input
Invalid input (float value)	1.5	startCheck() validateStartChoice()	Display error message, prompt for input
Invalid input (large integer)	100000	startCheck() validateStartChoice()	Display error message, prompt for input
Valid input	‘s’ or ‘q’	startCheck() validateStartChoice()	Continue to Play Game

Validate Move Choices

Test Cases	Input	Functions	Expected Output
Invalid input (string)	“test”	move() displayMoveChoices() validateMoveKey() validateInt()	Display error message, prompt for input
Invalid input (empty)	empty	move() displayMoveChoices() validateMoveKey() validateInt()	Display error message, prompt for input

Invalid input (signed integer)	-1	move() displayMoveChoices() validateMoveKey() validateInt()	Display error message, prompt for input
Invalid input (float value)	1.5	move() displayMoveChoices() validateMoveKey() validateInt()	Display error message, prompt for input
Invalid input (large integer)	100000	move() displayMoveChoices() validateMoveKey() validateInt()	Display error message, prompt for input
Valid input	1-4	move() displayMoveChoices() validateMoveKey() validateInt()	Move character in selected direction on map

Validate Item Choice

Test Cases	Input	Functions	Expected Output
Invalid input (string)	“test”	move() validateItemChoice() validateInt()	Display error message, prompt for input
Invalid input (empty)	empty	move() validateItemChoice() validateInt()	Display error message, prompt for input
Invalid input (signed integer)	-1	move() validateItemChoice() validateInt()	Display error message, prompt for input
Invalid input (float value)	1.5	move() validateItemChoice() validateInt()	Display error message, prompt for input
Invalid input (large integer)	100000	move() validateItemChoice() validateInt()	Display error message, prompt for input
Valid input	1-3	move() validateItemChoice() validateInt()	Use selected item appropriately

Validate Play Menu Choice

Test Cases	Input	Functions	Expected Output
Invalid input (string)	“test”	validateInt() move() addLeaks() adjustHealth() printMap() inspectRoom() displayItems() itemMenu()	Display error message, prompt for input
Invalid input (empty)	empty	validateInt() move() addLeaks() adjustHealth() printMap() inspectRoom() displayItems() itemMenu()	Display error message, prompt for input
Invalid input (signed integer)	-1	validateInt() move() addLeaks() adjustHealth() printMap() inspectRoom() displayItems() itemMenu()	Display error message, prompt for input
Invalid input (float value)	1.5	validateInt() move() addLeaks() adjustHealth() printMap() inspectRoom() displayItems() itemMenu()	Display error message, prompt for input
Invalid input (large integer)	100000	validateInt() move() addLeaks() adjustHealth() printMap() inspectRoom() displayItems() itemMenu()	Display error message, prompt for input

Valid input	1-4	validateInt() move() addLeaks() adjustHealth() printMap() inspectRoom() displayItems() itemMenu()	1. Move player, add leaks, adjust health 2. Print map, inspect room, pickup items, add leaks 3. Display items in backpack 4. Display items to use, use items, add leaks

Player Item Use

Test Cases	Functions	Expected Output
3 Items Use item 1	move() useItem() updateItem() validateInt()	Remove item 1, update remaining
3 Items Use item 2	move() useItem() updateItem() validateInt()	Remove item 2, update remaining
3 Items Use item 3	move() useItem() updateItem() validateInt()	Remove item 3, update remaining
2 Items Use item 1	move() useItem() updateItem() validateInt()	Remove item 1, update remaining
2 Items Use item 2	move() useItem() updateItem() validateInt()	Remove item 2, update remaining
1 Item Use item 1	move() useItem() updateItem() validateInt()	Remove item 1, update remaining to empty

Reflection

Design Changes

The main design change made was how the item container was implemented. At first, the item container was implemented using a STL stack, but issues occurred when items were being used by the player. There wasn't an efficient way for the player to use an item in the middle of the stack, and update the stack appropriately. Most of the debugging and design time for the project went into solving this problem.

The item container was finally implemented using a STL vector. The vector allowed the player to access item objects in the middle of the vector. It also allowed the implementation of an iterator for the vector, which allowed the program to call the erase() function to update the vector accordingly when an item was used. Using the advance() function, the iterator's position could be changed, and the according item pointer in the vector could be erased.

Other design changes included implementing multiple bool variables in the Space class to keep track of the player's location, items, leaks, player inspection, and player traversal. These bool values allowed the map to be printed correctly with useful information. The bool values also were useful in implementing the logic for specific events in the game.

Enums were also very valuable for returning space types, item types, chances for action spaces. The enums allowed a much more efficient and clear way to communicate space and item types. These types were used in implementing the logic of events, player actions.

Action spaces were also implemented to include more variety in the environment. The allowed more story elements and actions for the user to experience. The actions on each space included special events that the player could interact with. The player would either gain an item, escape, die, lose health, return to the initial position, add a leak. 3 types were included: hatch, tunnel, and door.

Problems Encountered

Implementing the container for the backpack, which holds item pointers
The main problem encountered while implementing the project was finding an STL container to use to hold item pointers. Using a stack did not work efficiently when items were used by the player, and the container had to be updated. Using a queue also resulted in the same issues. There was no efficient way to remove an item, and update the container.

Validating integers for player movement, item usage

At first, it was difficult to figure out a way to validate user input for integers when displaying menus for player movement, item usage, play menu options. It was hard to find a range that would suite all menus, especially when player movement was being limited by a border. Item options for player usage were also difficult to validate, as the range of acceptable integers would constantly change as players picked up new items and used old items.

Uninitialized values in functions throughout the program
Uninitialized values caused a multitude of memory errors when debugging with valgrind. Uninitialized values were present in validateInt(), move(), derived space classes, derived item classes. These errors were more challenging to resolve than memory leaks in the program.

Solutions to Problems

Implementing the container for the backpack, which holds item pointers
The solution to implementing the container was to use a vector. A vector allowed the item container to be accessed freely and updated appropriately with an iterator. The iterator was very powerful when used with the advance and erase functions. These functions allowed the items to be deleted when used, even if the items were not the first item in the container. This resolved the issues encountered when using a stack and queue.

Validating integers for player movement, item usage
This issue was solved by using multiple keys in the validateInt() function. These keys were determined by the limiting factors such as nullptrs for adjacent spaces using the validateMoveKey() function. By generating keys with these functions, the appropriate range of integers could be validated for each menu option, player action.

Uninitialized values in functions throughout the program
The uninitialized values were resolved by compiling with the -g option, and running valgrind --leak-check=full. The output from valgrind displayed line numbers where the uninitialized values were located. Going to these line numbers allowed for the correction and initialization of the values when they were declared. Integers were set to 0, spaces were set to a default space type, items set to a default item type.