Jon Frosch
CS 162
Final Project Program design

Requirements:
Create a text adventure-style game (similar to zork/planetfall) for one player in which the player must navigate through rooms and collect items to achieve a goal. The player must be able to hold items in an inventory, and the items must be part of the solution to winning the game. The player must be able to interact with some of the rooms. There should also be a limit on the number of turns a user is allowed to take.

The game theme is a rescue mission into a mine. The player must shut off a flow of toxic gas and turn on fans to clear the air. There are tools (a wrench and some jumper cables) that are used to achieve these goals. The turn limit is implemented a limited supply of breathing air. Walking between rooms and using/taking items have different costs in air that are deducted from the player's supply of air. When the player's air supply hits 0, the player dies and the game ends. The game detects if the win conditions are met and congratulates the player if they win.

Specific requirements:
Abstract Space class with at least 4 Space* members:

Space class contains a pure virtual destructor, so it is abstract

Space class has 6 Space* members: north, south, east, west, up, and down that point to a room's neighbors.

Game has at least 6 spaces:

The Mine Rescue game has 9 rooms in its map

At least 3 derived classes from Space that have interactive elements for the player

ToolRoom has a cabinet the player can open to reveal Items

LockRoom allows doors to be locked, preventing the player from going that direction unless they have the correct tool to bypass the lock

ValveRoom has a valve the player can shut if they have a Wrench item

FanRoom has fans the player can enable with JumperCable item

Game track the player's location:

After every action the player takes, a description of the current room. Description includes a list of the available exits.

Container for the player to carry items with a capacity limit

The Player class (which is aggregated into the current Space so the two can interact) has std::list member inventory that is allowed to hold two Item* elements representing Items in the player's hands.

Items required for solution

The player needs a Wrench to shut the valve and a JumperCables to start the fans.
These two actions are need to win the game.
The player also needs a Screwdriver to open the locked doors in order to win (the FanRoom is behind a locked door)

The game must have a time limit

The Player object has a airSupply member that starts at 100 air. Taking actions (moving, interacting with the room, taking items) costs varying amounts of air. When airSupply reaches 0, the game ends, and the player loses if they have not met the win conditions.

Declare the goal of the game:

The Game class prints a message to the console when a game starts describing the goal and theme.

No free form input (provide menus)

The Game class calls the action() function of the current space and passes it a pointer to the Player object. Action() provides the user a console menu of valid actions to take there and returns a pointer to the next space the player will be in (which may be the same as the current space).

Program Flow:
Main:
Menu to allow the user to start the game or exit the program
If play game selected, create a game object and call Game.playGame().

Void Game.playGame():
print welcome message
set location pointer to starting room
while(player is alive and player is not a winner and player has not asked to quit)
    describe the current room
    tell the player how much air they have left
    set the location pointer to the next room the player is in using location.action(&player)
end loop
check exit status (win/die/quit) and print appropriate message
return to menu in main

Space* Space.action(Player*)
Generate a list of valid actions for the player to take
Give the user a console menu of actions
Execute an action based on input
(check inventory, take an item, drop an item, move, etc.)
Return the next space the player will be in

Test Case	Input values	Driver Function	Expected outcomes	Observed outcomes
main menu input not an integer	Floating point, string	main() calling optionMenu()	Reject and reprompt	Reject and reprompt
main menu input not in range	Input < 1 or > 2	main() calling optionMenu()	Reject and reprompt	Reject and reprompt
main input ok	1 <= input <= 2	main() calling optionMenu()	Exit to terminal or start game	Exit to terminal or start game
room menu input not an integer	Floating point, string	Action(Player*) calling getSelection()	Reject and reprompt	Reject and reprompt
room menu input not in range	<1 or over highest option	Action(Player*) calling getSelection()	Reject and reprompt	Reject and reprompt
room input ok	Valid integer	Action(Player*) calling getSelection()	Take specified action	Take specified action
Correct exits	--	DescribeRoom() calling exits()	Description only displays valid exits	Description only displays valid exits
Correct actions	--	Action(Player*) calling generateMenu()	Menu only contains valid actions	Menu only contains valid actions
Move	--	Action(Player*) calling move()	Only valid exits in menu	Only valid exits in menu
Empty handed	Player inventory empty	Action(player*) calling generateMenu()	Drop not offered	Drop not offered
Drop	Player inventory size > 0	Action(player*) calling player.drop()	Selected item removed from player and added to room	Selected item removed from player and added to room
Empty room	roomItems is empty	Action(player*) calling generateMenu()	Take not offered	Take not offered
Take	roomItems size > 0	Action(Player*) calling take	Selected item removed from room and added to player	Selected item removed from room and added to player
Take/drop menu	--	Action(player*)	Only present items available	Only present items available

Inventory empty	--	Action(player*)	Display correct messageon inventory check	Display correct messageon inventory check
Check inventory	--	Action(player*) calling Player.descInventory()	Display held items	Display held items
Quit	Quit selected from menu	Action(player*)	Exit to main menu	Exit to main menu
Tool Cabinet	Open cabinet selected	Action(player*)	Contents of cabinet available to be taken	Contents of cabinet available to be taken
Cabinet closed	Player in tool room with cabinet closed	Action(player*)	Cabinet contents invisible to player	Cabinet contents invisible to player
Lock display	--	describeRoom() calling locks()	List of locked doors/available exits is correct	List of locked doors/available exits correct
Lock menu	Open door selected	Action(Player*) calling unlock()	List of locked doors is correct	List of locked doors is correct
No screwdriver	Screwdriver not in players inventory	Action(Player*) calling unlock()	Door stays locked	Door stays locked
Has screwdriver	Player is holding Scredriver	Action(Player*) calling unlock()	Selected door becomes an exit	Selected door becomes an exit
No wrench	Wrench not in players inventory	Action(player*) calling player.hasItem()	Valve not operable	Valve not operable
Wrench	Wrench in players inventory	Action(player*) calling player.hasItem()	Valve shuts	Valve shuts
No JC	JumperCables not in players inventory	Action(player*) calling player.hasItem()	fan not operable	fan not operable
JC	JumperCables in players inventory	Action(player*) calling player.hasItem()	fan shuts	fan shuts
Win	ValveOff and fanOn	PlayGame()	Display win message	Display win message
Die	airSupply <1 and !Win	PlayGame()	Display death message	Display death message

Reflection:

I found this assignment to be fairly challenging. I struggled for a while to come up with a theme that would allow me to create a game that would meet all the project requirements. When I finally came up with the mine rescue idea, I next tried to work out how the puzzles would work. I came up with several ideas for puzzles, and I eventually settled on the locked door, valve, and fan puzzles that appear in the game.

Next, I came up with my main concept for how the program would function, which is the action() function in the Space class. It made sense to me to have Space handle creating the menu for itself, since different rooms would have different actions that could be taken.

To test this idea out, I created the Corridor derived class, which is a basic room with no special actions. To start testing, I also wrote the setup functions so the Game constructor could build a world map and assign the n,s,e,w,u,d pointers in each Space. I used a map to map characters to the space pointers so I could have one setNeighbor() function that would handle all directions based on a human-understandable argument.

I created the world map using the Game constructor and debugged the movement functionality.

Next, I worked on the inventory system. I decided to use std::list to store pointers to item object for my inventory system. List seemed like a good choice because I wanted to be able to have arbitrary insertion and deletion, and sequential access was ok because of the limited overall number of items in the game. Each room has a list to store items if the player drops them and the Player object has a list capped at 2 items to carry items as an inventory from room to room.

When I got the inventory working correctly, I began creating the puzzle rooms with special interactions. In the Game constructor, I would change out a given Corridor for the new special room and created/tested them one at a time.

In creating the special rooms, I made a few alterations to the Space class so that I could reuse code more efficiently. These changes included splitting generateMenu() and getSelection() out of action(Player*) so that derived Spaces could call the generic version of the function and then add their special actions to the menu. This saved having to copy and paste code, which helped prevent versions for the function getting out of date/sync.

Following this process helped me not get too overwhelmed by the magnitude of the project and simplified testing by letting me focus on one thing a time. Having subsystems that worked that I could then extend and build on helped me feel like I was making progress. I think having things be semi-independent in this manner helped prevent me from making my code hard to follow, as different functionalities did not tend to get mixed.