* Started by creating the basic file structure for each class and component in the game:
  + Game.cpp, Game.h
  + Character.cpp, Character.h
  + Location.cpp, Location.h
  + Control.cpp, Control.h
  + Item.cpp, Item.h
  + Inventory.cpp, Inventory.h
  + Action.cpp, Action.h  
    **Design decisions**: use of separate files for each class for modularity and making each component easier to manage and test.  
    **Challenges**: linking all the components, this will probably be modified as the project develops.  
    **Testing**: successfully compiled this basic skeleton structure of the program.  
    **Reflections**: clarity when managing many different components that need to work together as a unit is very important. Laying clearly structured foundation for this from the very beginning will help to minimize confusion as the project unfolds.
* The main entry point, `main.cpp`, initializes the `Game` object, which runs the game. The code in `main.cpp` is as follows:

#include "Game.h"  
int main()   
{  
 Game game;  
 game.play();  
 return 0;  
}  
Added a public `play()` function to the Game, this function will manage game execution.  
  
**Design decisions**: I created the Game class to handle the main execution flow, and the play() function to manage the game.  
**Challenges**: Figuring out the main flow of the program, i.e. what are the roles of the main(), the Game constructor, and the play() functions.  
**Testing**: inserted cout statements along the execution of the program to make sure it works as intended.  
**Reflections**: the main.cpp serves as a centralize entry point that creates the Game object and runs it.

* Next, I focused on implementing the game's map, specifically designing the locations. Since players must be able to navigate between locations, each `Location` object requires pointers to adjacent locations, making an adjacency list structure the ideal choice. This allows each `Location` to store directional connections to other locations, functioning as a directional graph. This design was informed by research on adjacency lists: <https://www.geeksforgeeks.org/cpp-program-to-implement-adjacency-list/>   
    
  Each `Location` object includes a private member variable of type `std::map` to store pairs representing directions (as strings) and pointers to `Location` objects. The format is as follows:  
    
  std::map<std::string, Location\*> exits;  
  The `addExit` function enables the setup of these directional links between locations:  
  void Location::addExit(const std::string& direction, Location\* location)   
  {  
   exits[direction] = location;  
  }  
    
  **Design decisions**: I implemented the map as an adjacency list using std::map in the Location class. Each Location object contains a map of exit directions, allowing for navigation between locations.  
  **Challenges**: Choosing the right data structure required some consideration. After discarding linked lists and other structures, the adjacency list provided the most suitable solution.  
  **Testing**: I tested each Location object by and verifying connections. This was done with printing out the locations with the showInfo() function to be able to visually confirm that the locations are read in and linked according to the locations.txt. **Reflections**: Adjacency list is the most suitable solution for the data structure where each node need to hold multiple connections to other nodes.
* The `Game` class includes a `loadLocations()` function, which reads data from "locations.txt", parses it, and creates `Location` objects. Each `Location` is initialized with its `exits` map, containing pointers to the appropriate exit locations. The decision for the “locations.txt” format involves the following structure, where each entry includes the location name, description, and directional connections to adjacent locations:  
  <location name>|<description>|<direction>-<exit location>,…|  
    
  In future iterations, this format might be expanded to include actions or characters present at each location.  
    
  The initial map layout includes five locations, creating a simple structure for testing. After setting up the map and completing the `loadLocations()` implementation, the program compiles successfully and outputs the correct graph representation.  
    
  Initially, I considered implementing the map as a simple text-based printed maze. However, due to the predefined room order, I realized this flexibility was unnecessary. After considering and dismissing a linked list structure, I confirmed that an adjacency list best supports multiple connections between locations.  
    
  **Design decisions**: I designed locations.txt to contain location names, descriptions, and directional connections for each location. The loadLocations() function reads this data and initializes the Location objects.  
  **Challenges**: Parsing the file and setting up each location’s connections required careful attention to the format. Handling various connections with consistency took time.  
  **Testing**: I verified the loadLocations() function by printing out each location and its exits after loading to ensure accuracy.  
  **Reflections**: Using a text file to define locations provided a convenient way to manage and adjust the map.
* I implemented the currentLocation pointer as a private variable within the Game class to keep track of the player’s position in the game. I added two public functions, getCurrentLocation() and setCurrentLocation(), which allow access to and updating of the currentLocation.  
  **Design decisions**: I added currentLocation as a private pointer in the Game class, along with getCurrentLocation() and setCurrentLocation() functions for controlled access.  
  **Challenges**: Integrating currentLocation with Control class required careful management.  
  **Testing**: I set initial current location to the Rabit Hole at the beginning of the game and added a print out with cout to show user where they are located at the beginning of the game.  
  **Reflections**: currentLocation pointer will serve as a main reference to the sate of the game, meaning when user enters a command the currentLocation will act a s reference for finding possible actions at this location.
* Next, I implemented the loadControl() function and set up the control mechanism in the Control class. To manage this, I added loadControl() to the Game class and created the control.txt file. For consistency and to simplify loading, I used a format in control.txt similar to locations.txt::  
  <action name>|<action target>,<action target>,..|  
  **Design decisions**: I added loadControl() to the Game class to initialize actions from control.txt. Using a consistent format for both control.txt and locations.txt simplified parsing and loading.  
  **Challenges**: parsing control.txt accurately and loading each action while maintaining readability of the code required careful attention to detail.  
  **Testing**: I confirmed loadControl() functionality by outputting each parsed action to ensure proper loading.  
  **Reflections**: this setup allows flexible way of adding new actions without directly modifying the code.
* To make the Control class be able of storing all possible actions, I added a private std::vector<Action\*> actions to store loaded actions. The Game::loadControl() function reads control.txt, creates an Action object for each command, and stores each action in the actions vector.  
    
  To add an Action object I implemented a Control::addAction function that adds each parsed action to the actions vector.  
    
  **Design decisions**: I used a private std::vector<Action\*> actions in Control to store all possible actions. This allowed me to add each action dynamically through Control::addAction.  
  **Challenges**: Managing memory and ensuring each action was correctly loaded and stored required carefull attention.  
  **Testing**: I confirmed the action list by adding and retrieving actions, ensuring each command was stored as expected. I added showActions() to the Control class to verify all actions are loaded properly:   
   void showActions()

{  
 for (auto it = actions.begin(); it != actions.end(); ++it)  
 {  
 (\*it)->showInfo();  
 }  
 }  
**Reflections**: storing actions in Control centalizes command handling, allowing me to manage commands centrally.

* To get everything working as a functional game, I rewrote parts of the Game class logic. When the Game object is created, the constructor calls loadGameData(), which loads and populates data for game objects. After setup, the main function calls game.play(). The play() function starts the main game loop with a while (true) statement. This loop continuously listens for user input via cin and calls control->processCommand() to handle each command. The game stops if the user enters an exit command.  
  **Design decisions**: I revised the Game class to add a while (true) loop to the play() function to continuously accept user input until an exit command is given.  
  **Challenges**: figuring out how to manage the game’s lifecycle and ensure smooth control flow  
  **Testing**: I tested the loop by entering various commands to ensure the game responds as expected and terminates with the exit command.  
  **Reflections**: adding a main game loop was essential for player interaction with the game.
* The Controll::processCommand() parses the string that was entered by the user, finds corresponding Action object and calls executeAction() for this action. In the Control::processCommand() function, I parse the user’s input, find the corresponding Action object, and call executeAction() for that action. The executeAction() function serves as a central handler, or "switchboard," that manages and executes valid actions as specified in control.txt.  
  **Design decisions**: I implemented processCommand() in Control to parse user input, locate the corresponding Action object, and execute the action if valid. I designed executeAction() as a "switchboard" to handle valid actions based on the loaded actions in control.txt. Each action triggered different game responses.  
  **Challenges**: parsing the input and ensuring it matched an available action.  
  **Testing**: I tested processCommand() by entering commands and confirming that each action executed correctly by adding extra printout statements with cout to be able to visually confirm the flow of execution.  
  **Reflections**: this function acts as a central command processor, simplifying the management of user actions.
* Next I needed to add functionality that changes value of the currentLocation pointer of the Game class object to the location indicated by the user command. Fo r this I neede to ensure that the Control class had access to the Game class’s currentLocation and locations members. I modified Control to take a reference to a Game object in its constructor. This allowed Control to call game.setCurrentLocation() and other Game methods directly.  
  class Control   
  {   
   public:   
   Control(Game& game);   
   ….  
   private:   
   Game& game;   
   ……  
  }  
  To avoid circular dependencies, I used a forward declaration for Control in Game.h and for Game in Control.h. This approach allowed Control and Game to reference each other without including their entire definitions in both headers.  
    
  In Control::executeAction, I added code to locate the target location name in Game's locations vector and set currentLocation when the user enters a "go" command.

void Control::executeAction(const Action& action)

{

    …

    else if (action.getName() == "go")

    {

        string targetLocationDirection = action.getTarget();

        Location\* currLocation =game.getCurrentLocation();

        Location\* exitLocation = currLocation->getExit(targetLocationDirection);

        if(exitLocation)

        {

            game.setCurrentLocation(exitLocation);

            cout << "You are at: " << game.getCurrentLocation()->getName() << endl;

        }

        else

        {

            cout << "You can't go that way." << endl;

        }

    }   
….  
}  
**Design decisions**: To enable Control to access Game methods, I added a reference to Game in the Control class. To handle circular dependencies, I used forward declarations for Control in Game.h and for Game in Control.h, allowing each class to reference the other without full inclusion.  
**Challenges**: In Control::executeAction, I needed to carefully parse and find the correct target location in locations, ensuring currentLocation updated correctly.  
**Testing**: I tested various "go" commands to confirm that currentLocation updated to the correct target location, and the description printed as expected.  
**Reflections**: Adding game.setCurrentLocation() in executeAction required adjustments to the class structure and dependencies. Using forward declarations was a practical way to handle dependencies, and I will use this approach as I am developing this project, since Characater, Items, and Inventory are not yet added at this point.

* Next  
  **Design decisions**:   
  **Challenges**:   
  **Testing**:   
  **Reflections**:
* Next  
  **Design decisions**:   
  **Challenges**:   
  **Testing**:   
  **Reflections**:
* Next  
  **Design decisions**:   
  **Challenges**:   
  **Testing**:   
  **Reflections**: