# Assignment 4 journal

The first thing I did when I started this assignment is research the alice in wonderland storyline. I have read the book before, so I know the general storyline and plot, but it has been a long time and I need to refresh what the most important events were about. To do this, I used this website that provides a summary of the book (<https://www.sparknotes.com/lit/alice/summary/>).

After reading the book, there are a few main events that I want to incorporate into the game, and these are some of my ideas:

* The game should begin with Alice going down the rabbit hole
* After she falls in, she should be in a room called “hallway”
  + The hallway should be connected to multiple rooms (more than 3)
  + The hallway should have 2 items labelled drink me and eat me. They will cause Alice to either increase or decrease in size, and will be pre-requisites to enter some of the other rooms.
  + Alice should encounter the character white rabbit, and interact with him.
* If Alice goes through one of the doors, she can meet a character called Mouse. If Alice (the player) encounters Mouse, Mouse will become a companion
  + Mouse will consume ½ of any food Alice consumes, and in return can attack 1 character that the player specifies. Mouse must die in that encounter, and there is a 1/3 chance attacked character dies as well. If the attacked character does not die from the first attack, they will lose a third of their health. The only exception is the Queen of Hearts. An attack on the Queen of Hearts will remove half of her health.
  + The room in which the Mouse will spawn will be random, and there is equal possibility in all of the rooms connected to the hallway
* One of the rooms will be a chest room, where the player will be able to store 3 items.
* One of the rooms will have the Cheshire cat waiting. The Cheshire cat will guide Alice to the next room, which is the March Hare’s house. This will cost Alice 1 coin.
  + Alice cannot leave the room due to the time loop unless the character defeats the March Hare.
  + Alice will be able to throw items at the March Hare – one of which will be rocks. Rocks have a 25% chance of missing, and deal 1 damage if they successfully hit the target.
  + Alice will also be able to attack by punching. Punching deals 2 damage to the target and deals 1 hunger to Alice.
  + After defeating the March Hare, the March Hare will drop two items – Hot Tea, which can deal 5 damage if thrown, or remove 3 hunger while dealing 2 damage to Alice.
* After the March Hare is defeated, Alice can go back to the previous rooms to the hallway.
* One of the rooms will have a pigeon in it, which can only be attacked by throwing things at it
  + Important to mention that all fights will be turn-based
  + If the pigeon is defeated, Alice can move onto the next room, and the player will gain 2 eggs, which can each be consumed to decrease hunger by 2.
* The room connected to the pigeon’s room will be a dark, damp room with mushrooms in it. The mushrooms can be consumed to change Alice’s size
* Alice has 15 health and can get up to 10 hunger. If hunger reaches 10, Alice will lose health at a rate of 1 health/3 seconds (this will continue even through interactions).

For now, these are just some ideas, and they will probably change as I try to implement the game.

Creating a game of this size will be extremely difficult to do in a single file – so the first thing I want to do is research how to create multi-file c++ programs. I found a few sources, and this is the one I used (<https://www.sparknotes.com/lit/alice/summary/>).

Assignment Journal

When I initially reviewed the assignment, my immediate understanding was clear: I needed to develop a text-based adventure game inspired by "Alice's Adventures in Wonderland" by Lewis Carroll. The project would require extensive use of object-oriented principles, file handling, dynamic input normalization, alias mapping, character interaction, combat mechanics, and inventory management. The primary classes identified from the outset were Game, Room (Locations), Item, Character, Inventory, and Control. The provided assignment instructions guided my early decisions, while further details emerged organically through development and testing.

Design Decisions

My first step was to research the original "Colossal Cave Adventure" to understand the traditional mechanics of text-based adventures [source]. This helped me grasp common gameplay features such as movement commands, inventory management, interactive dialogues, random event outcomes, and text-driven feedback.

For my game, I chose to closely follow Lewis Carroll's narrative style while selectively adapting certain iconic scenes and characters. My starting location was "the meadow," where Alice first encounters the White Rabbit. This initial scene provided an immersive introduction, immediately engaging the player with recognizable elements from the source material.

The player would have to achieve the overarching goal: finding a predefined list of treasures scattered across Wonderland and bringing them back to a safe room. This clear objective offered structured gameplay and consistent motivation throughout the player's journey.

Rooms were carefully chosen based on their vividness and significance within the original story. I included well-known areas such as the Rabbit Hole (one-way entrance), the Hall of Doors, the Mad Hatter's Tea Party, the Queen's Croquet Ground, and the Cheshire Cat’s Forest. I made the decision early on to ensure at least one room (the Hall of Doors) had multiple exits (more than three), thus meeting one of the core project requirements.

Character interactions were particularly critical. I chose five key characters: the White Rabbit, Cheshire Cat, Mad Hatter, Queen of Hearts, and the Caterpillar. Each character had distinct behaviors—some hostile, some helpful. For example, the White Rabbit was peaceful and provided hints and useful items, while the Queen of Hearts aggressively initiated combat, fitting her depiction in Carroll’s book. The characters were designed to drop specific items upon defeat or interaction, such as keys and food.

Items were integral to gameplay, providing strategic benefits or fulfilling critical narrative functions. Items such as the "Eat Me" cake and "Drink Me" potion directly influenced player size and access to specific rooms, while other items like the Vorpal Sword served as weapons. Food items were critical, as they restored hunger points, preventing the player from perishing due to starvation.

I decided early on to externalize all significant text data—descriptions, aliases, and interaction prompts—into separate text files, following project requirements. This choice allowed flexible editing without needing to recompile the program.

Implementation Challenges and Solutions

One of my initial technical challenges was implementing robust input normalization. Users would naturally enter commands differently, such as "go north," "north," or "n." My normalization function needed to convert user input to a consistent form regardless of punctuation, spacing, or capitalization:

[code snippet here]

The function was successfully tested and refined iteratively. However, I encountered issues initially when punctuation was not properly removed, causing command mismatches. After consulting resources on string manipulation in C++ [source], I applied the std::remove\_if function effectively, ensuring robust input handling.

Alias mapping presented another hurdle. Initially, my alias resolution was simplistic, and the game frequently failed to recognize alternate names for characters and items. For example, typing "rabbit" or "bunny" needed to resolve correctly to "white\_rabbit." To address this, I structured my aliases clearly in external files (item\_aliases.txt, character\_aliases.txt) and employed unordered maps for quick alias resolution:

[code snippet here]

Through iterative testing, this alias system evolved into a stable solution that reliably interpreted diverse user inputs.

Combat mechanics proved complex, particularly implementing randomized outcomes and critical hits. Initially, my combat calculations were too predictable, detracting from player engagement. After research into random number generation methods in C++ [source], I implemented a random chance for critical hits and misses using rand() % 100. This significantly enhanced the combat experience, providing dynamic unpredictability:

[code snippet here]

Another issue arose regarding the management of inventory and item interactions. The player needed to effectively manage hunger, item usage, and size changes. Early implementations of inventory management allowed for duplicate items or failed to handle chest storage limitations properly. Careful adjustments to the inventory logic, such as limiting chest storage capacity and checking for item existence before actions, resolved these concerns:

[code snippet here]

Testing

Throughout development, testing was a continual, iterative process. Initially, I conducted basic unit tests for each component, such as room loading, item pickups, and character interactions. I manually verified that each command produced expected outputs.

As the game's complexity increased, I moved to scenario-based testing. For instance, testing the "Hall of Doors" extensively confirmed the size-restriction logic, ensuring players could only enter specific rooms when appropriately sized. I also tested hunger mechanics thoroughly, ensuring the player would receive appropriate notifications as hunger points diminished, eventually leading to death if not addressed.

Combat testing required careful calibration. Testing scenarios involved repeatedly engaging in combat to ensure balanced difficulty. Adjustments were frequently necessary to avoid overly harsh outcomes or trivial victories. By tuning enemy damage output and health values, I eventually achieved a balanced, engaging difficulty level.

I discovered an intermittent bug during extensive testing involving the "throw" command. Occasionally, thrown items would inexplicably fail to register a hit or miss accurately. Through debugging, I traced the issue to faulty conditional logic handling random outcomes. Correcting the logic resolved the bug, and further tests confirmed consistent behavior.

Final Thoughts

Reflecting on this project, I encountered numerous technical and design challenges that significantly enhanced my understanding of C++, object-oriented programming, and interactive game design. Each problem required systematic troubleshooting and iterative refinement, underscoring the importance of robust initial design and comprehensive testing.

Despite the complexity, implementing dynamic input handling, extensive alias mapping, randomized combat mechanics, and comprehensive inventory management proved rewarding. These elements substantially enriched gameplay, creating an immersive and responsive experience.

Future improvements could include expanding character dialogues, enhancing descriptive depth, and implementing more sophisticated combat mechanics or puzzle interactions.

Overall, this project has substantially strengthened my coding proficiency and my capacity to design engaging, text-based user experiences. I am confident in the robustness and effectiveness of my final game and proud of the extensive work completed.

Assignment Journal

When I first read through the assignment instructions, I understood clearly that the goal was to develop a detailed text-based adventure game inspired by Lewis Carroll's "Alice’s Adventures in Wonderland." This seemed like an exciting project due to the combination of narrative elements and technical programming requirements. My plan was to implement this game step-by-step, ensuring each feature was thoroughly tested and debugged before moving on to the next.

Setup and File Loading

Initially, I began by setting up the fundamental framework for my game, starting with the setup() function. This function was straightforward: it involved loading necessary data from external files—rooms, items, characters, item aliases, character aliases, and required treasures. I created separate text files (rooms.txt, items.txt, characters.txt, etc.) to store this data, adhering closely to the assignment requirements. Early on, I encountered issues with file handling—particularly checking if files successfully opened or not. To address this, I added error handling with simple if (!file.is\_open()) checks, printing an error message and exiting the game if any file failed to open.

One initial challenge was parsing the room data effectively. Each line of the room file needed to be split correctly by delimiters (e.g., '|'), which occasionally caused problems when room descriptions themselves contained special characters. To handle this robustly, I used std::getline with a delimiter to split each line accurately.

Inventory Management

Next, I worked on the inventory command functionality. The game required an inventory system where players could carry, use, drop, store, and retrieve items. I designed the inventory as a vector of item IDs stored within a PlayerData class. Initially, the inventory allowed unlimited items, but realizing gameplay might become unrealistic, I later added a limit to the number of items that could be stored in specific locations (chests).

When implementing chest storage, I encountered a bug where players could exceed the maximum storage limit. Debugging revealed I had overlooked adding a proper check before inserting items into the chest. Adding a simple conditional statement to enforce this limit solved the issue.

Input Normalization

To enhance gameplay, user input had to be intuitive. Users might enter commands in various formats, such as "go north," "north," or simply "n." This required robust input normalization. Initially, my normalization only converted input to lowercase, but I quickly noticed that commands with punctuation or extra spaces caused unexpected failures. I researched string manipulation techniques in C++ [source] and implemented a normalization function using std::transform and std::remove\_if to remove punctuation and excessive whitespace. After rigorous testing, this function reliably interpreted user commands regardless of formatting variations.

Alias Resolution

The next significant feature was implementing alias resolution for items and characters. Players could reference items and characters by multiple names (e.g., "rabbit" or "bunny" for the "white\_rabbit"). Initially, my alias resolution was rudimentary, causing frequent mismatches. I resolved this by employing unordered maps for efficient alias lookup and storing aliases in external files. Testing revealed an issue when aliases included spaces or punctuation. Enhancing the alias parsing logic in my loading functions solved these issues effectively.

Combat System

One of the most intricate features was designing a dynamic combat system. Characters had different health and damage attributes, and combat outcomes were randomized, including the possibility of critical hits. Implementing randomness required careful calibration. My initial implementation used basic random number generation (rand()), but the combat felt predictable. After further research [source], I incorporated randomness into critical hits and enemy responses, significantly improving the gameplay experience.

Testing the combat system was time-consuming. I frequently needed to rebalance enemy health and damage values to maintain fair yet challenging combat. Through iterative testing, I found a good balance that kept encounters exciting without becoming frustrating.

Hunger and Health Mechanics

To add depth, I introduced hunger and health mechanics. Players lost hunger points over time, eventually leading to death by starvation if they did not regularly consume food items. Initially, hunger points decreased too rapidly, frustrating early testers. Adjusting the hunger depletion rate and clearly communicating hunger status through ASCII health bars greatly improved player experience.

Command Processing

The process\_command() function managed player commands. Implementing this involved extensive branching logic. Initially, command processing became difficult to maintain due to repetitive code. I streamlined this by encapsulating common actions into helper functions. One specific issue arose with the "throw" command, occasionally failing to register item throws correctly. This issue turned out to be incorrect substring logic, resolved by refining the string parsing conditions.

Room Navigation

Movement between rooms was central to gameplay. Each room had specific exits, and certain rooms required the player to be a specific size. I implemented size checks and enforced entry restrictions, ensuring immersive gameplay. Testing these mechanics revealed a logic flaw: players could sometimes bypass size restrictions unintentionally. Revising conditional checks fixed this issue, providing reliable and intuitive navigation.

Loading External Data

Throughout development, managing external data files required meticulous attention. Errors in formatting could cause unexpected crashes or gameplay issues. To mitigate this, I implemented comprehensive validation during data loading, including checking for empty strings and invalid numerical values. Thorough testing with intentionally malformed data files confirmed the robustness of these validations.

Alias and Direction Normalization

I encountered initial trouble handling directional commands. Players input commands differently, and the game needed to interpret all directional inputs consistently. Implementing a dedicated normalization map for directional aliases significantly improved responsiveness. Testing directional commands exhaustively helped catch subtle inconsistencies, which I resolved systematically.

Character Interactions

Implementing interactive dialogues posed its own challenges. Peaceful characters provided hints or items, while hostile characters initiated combat. Initially, dialogue interactions were too simple and lacked depth. Adding diverse greetings and conditional gift logic based on player inventory enriched these interactions substantially.

Final Testing and Debugging

Before finalizing, I conducted comprehensive end-to-end gameplay tests, systematically identifying and fixing remaining bugs. These tests highlighted minor issues, such as incorrect status bar displays or missing exit descriptions. Systematic debugging addressed each problem methodically, ensuring polished and immersive gameplay.

Reflecting on Development

Throughout development, each technical hurdle provided valuable learning opportunities, significantly enhancing my programming skills. The iterative debugging process improved my problem-solving abilities, and implementing complex features like dynamic input handling, combat randomness, and comprehensive inventory management offered significant personal growth.

In retrospect, the project's complexity initially felt daunting, but careful planning and incremental development made it manageable. This experience has strengthened my confidence and competence as a programmer, preparing me effectively for future software development challenges.

## **Journal Outline: Alice in Wonderland Game Development**

### **1. Project Setup and Initial Planning**

* Decision to base game on Alice in Wonderland
* Reading the assignment instructions and identifying key requirements
* Planning out class architecture: Game, Room, Item, Character, Inventory, Control, etc.
* Early design notes on room layout, character interactions, item mechanics
* Overview of external files: rooms.txt, items.txt, characters.txt, etc.

### **2.** game::game() **– Game Constructor**

* Initialization of flags (e.g., is\_running = true)
* Simple logic, no research needed

### **3.** game::setup() **– Data File Loaders**

* Purpose of loading all required game data from files
* Problems with file structure and parsing inconsistencies
* Discuss std::ifstream and std::istringstream [source]
* Custom logic for parsing |-delimited text files
* Input validation and file existence checks

#### Subsections:

* load\_rooms()
  + How room data and exits are parsed and stored
  + Building exit maps with direction commands → room IDs
  + Use of std::map and std::getline() parsing [source]
* load\_items()
  + Parsing complex item attributes: damage, hunger restoration, aliases
  + Dealing with optional fields like size\_change
  + Populating items\_in\_rooms with map of vectors [source]
* load\_characters()
  + Handling drops (vector of strings), alias processing, peaceful/aggressive flags
  + Multi-value parsing for lists within one field [source]
* load\_item\_aliases() and load\_character\_aliases()
  + Mapping aliases to canonical item/character IDs
  + Stripping punctuation and whitespace using std::transform() and std::remove\_if() [source]
* load\_required\_treasures()
  + Simple line-based load into a vector

### **4.** game::run() **– Main Game Loop**

* Setting up the game loop structure
* Adding ASCII art and status screen
* Handling dynamic hunger depletion over time
  + Use of rand() and simple RNG-based events [source]
* Introductory messages and goal reminder
* Loop design: std::getline, then process\_command()

### **5.** game::process\_command() **– Core Input Handler**

* Input normalization via normalize\_input() [source]
* Command parsing and dispatch
* Issues with ambiguous commands or substring matching

#### Key commands (each subcommand becomes a subheading):

* "look" – Reprints room description
* "inventory" / "i" / "invent" – Inventory display
* "quit" – Flag toggle

##### attack <character>

* Character ID resolution
* Combat system: base damage, critical hit logic with rand() [source]
* Character health, defeat, drops

##### talk to <character>

* Greeting logic, gift giving, hostile response conditions

##### take <item> / drop <item>

* Inventory modifications
* Alias resolution and item presence checks

##### use <item>

* Effects of food or size-changing items
* Hunger restoration and size state tracking

##### store <item> / retrieve <item>

* Chest interaction: up to 3 items per room
* Vector size checks, insertion/removal logic

##### throw <item> at <character>

* Hit/miss chance with rand() [source]
* Damage resolution and counterattack

##### Movement Commands (go north, n, etc.)

* Size restrictions, alias normalization
* Destination check, get\_next\_room()

##### normalize\_direction()

* Alias map setup for flexible directional input [source]

### **6.** game::print\_room\_state()

* Rendering room, exits, items, characters, chests
* Reuse of description logic based on visit history

### **7.** game::resolve\_item\_id() **/** resolve\_character\_id()

* Text normalization, stripping punctuation and mapping to canonical form [source]

### **8.** game::normalize\_input()

* Lowercasing and punctuation removal
* Handling different user typing styles [source]

### **9.** game::handle\_combat()

* Extracted from process\_command()
* Encapsulated combat resolution, including critical strikes and retaliation
* Character removal from map after defeat

### **10.** game::show\_status()

* ASCII bars for health and hunger
* Unicode block characters (\u2588) for display [source]
* Decision to represent player's current size

### **11. Integration and Testing**

* Problems with alias mismatches
* Testing different input variations
* Debugging logic for character drops, item limits, hunger death
* Crash causes and fixes (e.g., file errors, empty data)

### **12. Final Features Added**

* Game win condition (returning all required treasures to the safe room)
* Victory and death messages
* Descriptive flavor messages and immersion

### **13. Challenges and Bugs Faced**

* Input ambiguity issues
* File parsing errors and crash handling
* Combat logic errors (negative health, incorrect removal)
* Missed aliases and failed command recognition

### **14. Final Polish and Documentation**

* Rechecking all function flows
* ASCII title formatting
* Final code cleanup, comment pass, and spacing

### **15. Lessons Learned**

* Value of modular design
* Input normalization complexity
* File parsing robustness
* Importance of structured testing phases