**Alice in Wonderland Design Document**

**Game Concept and Overview**

This game is a text-based adventure implemented in C++ using object-oriented programming. It divides functionality across classes representing the player, items, characters, rooms, and game control logic. The player navigates rooms, interacts with characters and items, manages health and hunger, and issues commands to progress. Inputs are normalized for flexible command handling. Outputs are textual descriptions printed to the console.

Standard C++ libraries used: <iostream>, <fstream>, <sstream>, <string>, <vector>, <map>, <unordered\_map>, <algorithm>, <cstdlib>, <thread>, <chrono>.

**Game Goals**

* Defeat enemies and collect all required treasures.
* Return treasures to the designated safe room.
* Prevent death by maintaining positive health and hunger values.
* Access size-restricted rooms by using size-altering items.

**Core Gameplay Systems**

* **Rooms:** Each room is identified by an id and has a description, as well as an optional size\_required to enter, and an std map of exits linking it to the other rooms. Methods allow querying room details and managing exits.
* **Items:** Items have an id, description, damage value, hunger\_restore value, and size\_change. Items are initialized from a text file, then these properties are returned by a method.
* **Characters:** Characters also have an id, description, health, damage, a list of drop\_items, a peaceful flag, a greeting, and an sometimes a gift\_item. Characters are initialized from a text file, just like items. Multiple methods exist for combat, status queries, and interactions.
* **Player:** Tracks health, hunger, size, base\_damage, and an inventory of item IDs.
* **Inventory/Chests:** Player inventory holds 7 items; room chests store up to 3 items per room. It tracks an inventory of item IDs, and its has methods for adding, and removing items from the inventory.
* **Input Normalization:** Cleans input text by turning all of the input characters into lowercase, stripping punctuation, and mapping aliases for items, characters, etc…
* **Command Processing:** Interprets the normalized input then calls the corresponding game action.

**4. Class Breakdown with Detailed Function Explanations**

**4.1 Game**

File: game.cpp, game.h  
Functions:

* **setup()**:  
  Loads all core game data files: rooms.txt, items.txt, characters.txt, item\_aliases.txt, character\_aliases.txt, required\_treasures.txt, and actions.txt. Populates all main containers and alias maps for rooms, items, and characters. Initializes room, item, and character relationships.
* **run()**:  
  Clears the screen, prints ASCII art title, displays intro and instructions. Prints the initial room description, available exits, present characters, visible items, and chest contents in the current room. Displays player status bars (health, hunger, size). Begins the main game loop:
  + Reads input from cin, passing input to control\_manager->process\_command(input).
  + Each turn, randomly decreases hunger (1-3 points).
  + If hunger drops to zero, ends game with starvation message.
  + If hunger <30, displays a warning.
  + Continues looping while is\_running is true.
* **load\_rooms(filename)**:  
  Reads rooms from rooms.txt, ignoring empty lines and comments. For each line, parses id, description, size\_required, and exit definitions. Builds Room objects, sets exit mappings, and stores them in room\_manager->rooms. Sets the starting room to "meadow".
* **load\_items(filename)**:  
  Reads items from items.txt. Parses id, description, initial location, damage, hunger\_restore, size\_change, and aliases. Constructs Item objects, inserts them into item\_manager->all\_items. Adds item to the specified room’s inventory if a location is present. Populates item\_alias\_map with all aliases and ids.
* **load\_characters(filename)**:  
  Reads characters from characters.txt. Parses id, description, initial location, health, damage, drop\_items, aliases, peaceful flag, greeting, and gift. Builds Character objects, adds them to character\_manager->all\_characters. Populates room\_manager->characters\_in\_rooms with character ids. Adds all aliases and id to character\_alias\_map.
* **load\_item\_aliases(filename)**:  
  Reads alias mappings from file. For each line, splits id and comma-separated aliases. Each alias is normalized (lowercase, stripped of whitespace/punctuation) and added to item\_alias\_map.
* **load\_character\_aliases(filename)**:  
  Reads alias mappings from file. For each line, splits id and comma-separated aliases. Each alias is normalized and added to character\_alias\_map.
* **load\_required\_treasures(filename)**:  
  Reads required treasure ids, strips whitespace, adds each non-empty id to the required\_treasures vector.
* **resolve\_item\_id(input)**:  
  Normalizes input (lowercase, strips punctuation). Returns item id from item\_alias\_map if found, or empty string otherwise.
* **resolve\_character\_id(input)**:  
  Normalizes input (lowercase, strips punctuation). Returns character id from character\_alias\_map if found, or empty string otherwise.
* **handle\_combat(character& enemy, const std::string& enemy\_id, int damage\_dealt)**:  
  Applies damage to the enemy. If enemy is defeated, drops all items in current room and removes the enemy from the room. If enemy survives, triggers a counterattack, with a 20% chance of critical hit. If player health falls to zero, ends the game with a death message.
* **show\_status()**:  
  Displays player's health and hunger as ASCII bars (20 segments each, scaled by percentage). Prints the player's current size value.

**4.2 Player**

**Files:** player.cpp, player.h  
**Functions:**

* **player()**:  
  Default constructor.
  + Allocates a new inventory object for player inventory management.
  + Initializes size to "normal".
  + Sets hunger to 100 and health to 100.
  + Sets base\_damage to 7.
* **~player()**:  
  Destructor. Deletes the dynamically allocated inventory object.
* **get\_health() const**:  
  Returns the player’s current health value (int).
* **take\_damage(int amount)**:  
  Subtracts the specified amount from player’s health. Health cannot drop below zero.
* **heal(int amount)**:  
  Increases player’s health by the specified amount, not exceeding 100.
* **get\_base\_damage() const**:  
  Returns the player’s base damage value (int).
* **set\_base\_damage(int damage)**:  
  Sets the player’s base damage to the given value.
* **add\_item(const std::string& item\_id)**:  
  Adds an item to the player’s inventory, if inventory size limit (INVENTORY\_MAX\_SIZE, typically 7) is not reached. Prints a message if inventory is full.
* **has\_item(const std::string& item\_id) const**:  
  Returns true if the player inventory contains the specified item.
* **remove\_item(const std::string& item\_id)**:  
  Removes the specified item from the player’s inventory.
* **get\_inventory() const**:  
  Returns a const reference to the player’s inventory vector of item ids.
* **set\_size(const std::string& new\_size)**:  
  Sets the player’s current size to the given value.
* **get\_size() const**:  
  Returns the player’s current size as a string.
* **change\_hunger(int amount)**:  
  Adds the specified amount to the player’s hunger. Clamps the value between 0 and 100.
* **get\_hunger() const**:  
  Returns the player’s current hunger value (int).

**Data Members:**

* player\_inventory: Pointer to inventory object, manages item ids held by the player.
* size: String representing player size ("normal", "small", etc.).
* hunger: Integer representing player’s current hunger (0–100).
* health: Integer representing player’s current health (0–100).
* base\_damage: Integer representing the base damage the player deals.

**4.3 Character**

**Files:** characters.cpp, characters.h  
**Functions:**

* **character()**:  
  Default constructor. Initializes all data members to default values:
  + id and description set to empty strings
  + health and damage set to 0
  + peaceful set to false
  + greeting and gift\_item set to empty strings
  + drop\_items initialized as empty vector
* **character::character(const std::string& id, const std::string& description, int health, int damage, const std::vector<std::string>& drop\_items, bool is\_peaceful, const std::string& greeting\_text, const std::string& gift\_item\_id) : id(id), description(description), health(health), damage(damage), drop\_items(drop\_items), peaceful(is\_peaceful), greeting(greeting\_text), gift\_item(gift\_item\_id) {}:** Constructs a character with the given properties and initializes all data members.
* **get\_id() const**:  
  Returns the character's id string.
* **get\_description() const**:  
  Returns the character's description string.
* **get\_health() const**:  
  Returns the character's current health (int).
* **get\_damage() const**:  
  Returns the character's base damage value (int).
* **get\_drops() const**:  
  Returns a vector of strings with item ids that this character will drop on defeat.
* **take\_damage(int amount)**:  
  Subtracts the specified damage amount from the character's health (cannot drop below zero).
* **is\_alive() const**:  
  Returns true if the character's health is above zero; otherwise false.
* **is\_peaceful() const**:  
  Returns true if the character is peaceful; otherwise false.
* **get\_greeting() const**:  
  Returns the character's greeting string (used for dialogue).
* **get\_gift() const**:  
  Returns the id of the gift item associated with this character (if any).
* **clear\_gift()**:  
  Removes the gift item from the character by setting gift\_item to an empty string.

**Data Members:**

* id: String identifier for the character.
* description: Character's narrative or visual description.
* health: Integer, current health value.
* damage: Integer, base damage dealt by the character.
* drop\_items: Vector of item ids dropped when defeated.
* peaceful: Boolean, true if character is non-hostile.
* greeting: String, used when the player talks to the character.
* gift\_item: String, id of the gift item (if any) the character can give.

**4.4 Item**

**Files:** item.cpp, item.h  
**Functions:**

* **item()**:  
  Default constructor. Initializes data members: damage to 0, hunger\_restore to 0, and size\_change to an empty string.
* **item(const std::string& id, const std::string& description, int damage, int hunger\_restore, const std::string& size\_change)**:  
  Constructs an item with the specified id, description, damage value, hunger restoration value, and size change string.
* **get\_id() const**:  
  Returns the item's id string.
* **get\_description() const**:  
  Returns the item's description string.
* **get\_damage() const**:  
  Returns the item's damage value (int).
* **get\_hunger\_restore() const**:  
  Returns the item's hunger restoration value (int).
* **get\_size\_change() const**:  
  Returns the item's size change string (can be empty if item does not affect size).

**Data Members:**

* id: string identifier for the item.
* description: Item's narrative or functional description.
* damage: Integer amount of damage the item deals (if used as a weapon).
* hunger\_restore: Integer amount of hunger restored when item is consumed or used.
* size\_change: String representing the size effect of the item on the player (e.g., "small", "large", or empty if none).

**4.5 Room**

**Files:** location.cpp, location.h  
**Functions:**

* **room()**:  
  Default constructor. Initializes room data members.
* **~room()**:  
  Destructor. No specific resource management.
* **room(const std::string& id, const std::string& desc, const std::string& required\_size)**:  
  Constructs a room with given id, description, and required player size.
* **get\_size\_required() const**:  
  Returns the required player size (string) for entering this room.
* **get\_description() const**:  
  Returns the full room description as a string.
* **get\_next\_room(const std::string& command) const**:  
  Given a direction/command, returns the next room’s id if there is an exit in that direction, or empty string if none.
* **add\_exit(std::string command, std::string destination)**:  
  Adds a mapping from a direction/command string to a destination room id.
* **get\_all\_exits() const**:  
  Returns a map of all exits (command string to destination room id).
* **print\_room\_state(bool just\_moved)**:  
  Prints the current room’s full description on first visit, or a short message if revisiting or moving between rooms.
  + Prints visible exits using get\_all\_exits().
  + Prints all visible characters using their description (calls into character\_manager).
  + Prints all visible items in the room using their descriptions (calls into item\_manager).
  + Prints chest contents if the room has a non-empty chest.
  + Calls game\_manager->show\_status() at the end.

**Data Members:**

* id: Room identifier string.
* room\_description: Room’s narrative text.
* size\_required: Size required to enter (string, can be empty for no restriction).
* exits: Map from command/direction string to room id.
* Pointers to managers: character\_manager, item\_manager, player\_data, game\_manager.
* Global structures: visited\_rooms, rooms, current\_room, characters\_in\_rooms, items\_in\_rooms, chests (these are managed at the room\_manager level, not as instance members of individual Room objects).

**4.6 Control**

**Files:** control.cpp, control.h  
**Functions:**

* **control(game *g)***:  
  Constructor. Sets pointers to core game components:
  + game\_manager: main game instance
  + player\_data: player instance
  + room\_manager: room manager instance
  + item\_manager: item manager instance
  + character\_manager: character manager instance
  + action\_manager: action manager instance
* **~control()**:  
  Destructor. No explicit resource management.
* **process\_command(const std::string& input)**:  
  Main input handling and command dispatch logic:
  + Normalizes input using normalize\_input().
  + Handles room re-entry checks and prints updated room state if necessary.
  + Handles look command to print the current room state.
  + Handles quit command, sets is\_running to false, and prints a goodbye message.
  + Handles inventory, i, and invent commands:
    - If inventory is empty, prints "Your inventory is empty."
    - Otherwise, prints descriptions of all carried items.
  + Splits input into verb and parameter (space-separated). Resolves the action using action\_manager->resolve\_action\_id(verb).
  + Dispatches recognized actions to the relevant action manager methods (attack, take, drop, use, store, retrieve, talk, throw).
  + For movement commands, normalizes direction using normalize\_direction(), resolves the next room, and handles size restrictions and room transitions.
  + After moving, prints the room state.
  + On entering the "safe\_room", checks if the player possesses all required treasures; if so, prints win message and ends the game.
* **normalize\_input(const std::string& raw)**:  
  Converts input to lowercase and removes punctuation.
* **normalize\_direction(const std::string& input)**:  
  Converts common direction and movement aliases to "go <direction>" commands, returns normalized direction string.

**Data Members:**

* game\_manager: Pointer to the main game instance.
* player\_data: Pointer to the player instance.
* room\_manager: Pointer to the room/location manager.
* item\_manager: Pointer to the item manager.
* character\_manager: Pointer to the character manager.
* action\_manager: Pointer to the action manager.

**Files:** inventory.cpp, inventory.h  
**Functions:**

* **add\_item(const std::string& item\_id)**:  
  Adds the specified item id to the inventory if the inventory size limit (INVENTORY\_MAX\_SIZE) is not exceeded.
  + If inventory is full, the item is not added.
* **remove\_item(const std::string& item\_id)**:  
  Removes all instances of the specified item id from the inventory vector.
* **has\_item(const std::string& item\_id) const**:  
  Returns true if the specified item id exists in the inventory; false otherwise.
* **get\_items() const**:  
  Returns a const reference to the vector of item ids currently held in the inventory.

**Data Members:**

* items: Vector of strings, each representing an item id in the inventory.
* INVENTORY\_MAX\_SIZE: Static or constant integer value representing the maximum number of items the inventory can hold (typically set to 7).

**4.7 Inventory**

**Files:** inventory.cpp, inventory.h  
**Functions:**

* **add\_item(const std::string& item\_id)**:  
  Adds the specified item id to the inventory if the inventory size limit (INVENTORY\_MAX\_SIZE) is not exceeded.
  + If inventory is full, the item is not added.
* **remove\_item(const std::string& item\_id)**:  
  Removes all instances of the specified item id from the inventory vector.
* **has\_item(const std::string& item\_id) const**:  
  Returns true if the specified item id exists in the inventory; false otherwise.
* **get\_items() const**:  
  Returns a const reference to the vector of item ids currently held in the inventory.

**Data Members:**

* items: Vector of strings, each representing an item id in the inventory.
* INVENTORY\_MAX\_SIZE: Static or constant integer value representing the maximum number of items the inventory can hold (typically set to 7).

**4.8 Main**

**Files:** main.cpp  
**Functions:**

* **main()**:  
  Entry point for the program.
  + Instantiates a game object.
  + Calls game.setup() to initialize all game data and state.
  + Calls game.run() to start and execute the main game loop.
  + Returns 0 when finished.

**Data Members:**

* None. Main function only handles instantiation and execution flow.

**5. Data File Formats**

Each loader parses fixed-format text files corresponding to its class:

* rooms.txt: id|description|required\_size|exit1=dest1,...
* items.txt: id|description|location|damage|hunger\_restore|size\_change|aliases
* characters.txt: id|description|location|health|damage|drops|aliases|peaceful|greeting|gift
* item\_aliases.txt: id|alias1,alias2,...
* character\_aliases.txt: id|alias1,alias2,...
* required\_treasures.txt: list of item IDs

**6. Execution Flow**

1. setup(): loads all data.
2. Prints intro.
3. Prints starting room.
4. run(): loop input → process → update state → output.
5. Checks victory in safe room with all treasures.

**7. Design Justification**

* **Structured into multiple files:** Classes encapsulate data and behavior per game entity, making it easy to debug. Header files allow the program to be organized and easy to work on and improve.
* **Content read from txt files:** allows for easy customization of the game.
* **All other design decisions are discussed briefly in the class breakdown, much more detailed discussion in the assignment journal.**

**8. Planned Extensions**

In the future, I want to add item-based puzzles, multi-turn dialogues, random encounters & special event triggers. This is expanded upon more in the last section of the journal.