**Assignment 4 Journal**

**1. Starting the Project**

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.cs.fsu.edu/~myers/c++/notes/compilation.html>).

After reading the assignment instructions, I started by setting up the basic file structure of the game. I created the following files away:

* main.cpp
* game.cpp / game.h
* player.cpp / player.h
* item.cpp / item.h
* characters.cpp / characters.h
* location.cpp / location.h
* gradual\_text.cpp / gradual\_text.h

I created these files as empty placeholders to create a skeleton of sorts for project. I made sure to write header guards (https://www.learncpp.com/cpp-tutorial/header-guards/) inside each .h file to ensure that I am not duplicating any #includes during compilation. This was something I learned the hard way while trying to test compile the files with simple test code inside.

Once the files were created and guards written, I rewrote a minimal main.cpp to test if the structure would compile. I wrote a simple main() function that would instantiate a game object and call setup() and run() on it, and I will later use these methods to load data and run the game loop.

#include "game.h"

int main() {

game g;

g.setup();

g.run();

return 0;

}

**2. Implementing the Player Class**

I decided to begin by creating a player class because it seemed the most intuitive. I already knew the player would need attributes like health, hunger, size, a base damage value, and an inventory, so I created these and made a constructor to intialize them.

player::player() : size("normal"), hunger(100), health(100), base\_damage(1) {}

I set the player’s initial health and hunger to 100, size to "normal", and base damage to 7.

After the constructor, I made basic getter and setter functions for health:

int player::get\_health() const { return health; }

void player::take\_damage(int amount) {

health -= amount;

if (health < 0) health = 0;

}

void player::heal(int amount) {

health += amount;

if (health > 100) health = 100;

}

I wrote take\_damage() so that health would never go below zero. I also made sure heal() could never exceed 100.

I then implemented getters and setters for base\_damage (which was also straightforward):

int player::get\_base\_damage() const { return base\_damage; }

void player::set\_base\_damage(int damage) { base\_damage = damage; }

Next, I worked on the player’s inventory. I decided to store it as an std::vector of std::strings of item IDs. I have used std::vector before, but never for this kind of object tracking.

I wrote:

void player::add\_item(const std::string& item\_id) {

inventory.push\_back(item\_id);

}

bool player::has\_item(const std::string& item\_id) const {

return std::find(inventory.begin(), inventory.end(), item\_id) != inventory.end();

}

void player::remove\_item(const std::string& item\_id) {

auto it = std::remove(inventory.begin(), inventory.end(), item\_id);

if (it != inventory.end()) {

inventory.erase(it, inventory.end());

}

}

const std::vector<std::string>& player::get\_inventory() const {

return inventory;

}

I had to research how delete an element from a vector. I found a source about std::erase, but using it leaves a gap in the array, and people on random online forums answering a question similar to mine thought it was a good idea to use something called an erase remove idiom because otherwise there would be gaps in the vector. Researching that made me understand that std::remove moves all elements not matching the target to the front and returns an iterator to the new end, but does not make the vector smaller. std::erase actually deletes them from the container. When used in conjunction they remove elements without leaving aforementioned gaps. (<https://en.wikipedia.org/wiki/Erase%E2%80%93remove_idiom>).

Originally, I forgot to check if (it != inventory.end()) before calling erase(), so nothing was being erased if the item didn’t exist. After adding the check, it worked as expected.

I then implemented size and hunger management:

void player::set\_size(const std::string& new\_size) { size = new\_size; }

std::string player::get\_size() const { return size; }

void player::change\_hunger(int amount) {

hunger += amount;

if (hunger > 100) hunger = 100;

if (hunger < 0) hunger = 0;

}

int player::get\_hunger() const { return hunger; }

Like health, I made sure hunger could only be between 0 and 100.

To test the class, I created a test main() function that created a player object, added and removed items, took and healed damage, and printed out values.

**3. Implementing the Item Class**

With the player class mostly done, I moved on to the item class. I know items need to store an ID, description, have damage value (if it’s some sort of weapon), a hunger restoration amount (if it’s a consumable), and a size change effect (if it’s one of the growing/shrinking consumables).

I created two constructors: a default constructor and one that initialized all the attributes.

item::item() : damage(0), hunger\_restore(0), size\_change("") {}

item::item(const std::string& id, const std::string& description,

int damage, int hunger\_restore, const std::string& size\_change)

: id(id), description(description), damage(damage),

hunger\_restore(hunger\_restore), size\_change(size\_change) {}

After that, I wrote simple getter functions:

std::string item::get\_id() const { return id; }

std::string item::get\_description() const { return description; }

int item::get\_damage() const { return damage; }

int item::get\_hunger\_restore() const { return hunger\_restore; }

std::string item::get\_size\_change() const { return size\_change; }

To test this class, I made a temporary item object in a test main function and printed its attributes to confirm that they were stored and retrieved correctly.

Keeping the naming conventions I was using consistent was surprisingly difficult

**4. Implementing the Room Class**

After finishing the item class, I moved on to the room class in location.cpp. I know that each room needs to store:

* A unique room ID
* A description of the room
* A required size (to restrict access for small or large players)
* A collection of exits (where each command leads to another room)

I created a default constructor and one that took the room’s ID, description, and required size:

room::room() {}

room::room(const std::string& id, const std::string& desc, const std::string& required\_size)

: id(id), room\_description(desc), size\_required(required\_size) {}

I then added basic getter functions for the room’s description and size:

std::string room::get\_description() const { return room\_description; }

std::string room::get\_size\_required() const { return size\_required; }

The next step is to represent the exits from a room. Initially, I wasn’t very sure how I create a connection between a command like move forward to a destination room ID (especially so move forward only moves the player to the room in front of them – when they use it once the room in front of them changes, so using it again has to refer to a different room id). Initially I wanted to use tuples to do this, but I quickly gave up because it was ugly, convoluted, and I didn’t fully understand what I was doing. So, I researched what data structures I could use to store key-value pairs in C++ and discovered the wonderful std::map. I had never used it before, but because this is basically what it was created for I found understanding how it worked pretty simple (even if syntax was a bit complicated).(<https://cplusplus.com/reference/map/map/>).

I declared:

std::map<std::string, std::string> exits;

I then created add\_exit() to add an exit command and its destination:

void room::add\_exit(std::string command, std::string destination) {

exits[command] = destination;

}

For retrieving the next room, I wrote get\_next\_room():

std::string room::get\_next\_room(const std::string& command) const {

auto it = exits.find(command);

if (it != exits.end()) {

return it->second;

}

return "";

}

then, I wrote a function to retrieve the entire map of exits:

std::map<std::string, std::string> room::get\_all\_exits() const { return exits; }

I had to learn how .find() returns an iterator and how to dereference it to get .second. I also learned that accessing exits[key] with a missing key would insert a blank entry, so I used .find() instead to avoid modifying the map accidentally. (<https://cplusplus.com/reference/map/map/find/>).

To test, I created a room object, added exits, and printed them to confirm they stored correctly. I misunderstood how insert()acted if elements with the same keys were added, and ended up changing that to using []. After testing, I confirmed that exits could be added, retrieved, and printed properly.

**5. Implementing the Character Class**

Next, I moved on to the character class in characters.cpp. I needed this class to represent both friendly and hostile characters in the game.

Each character needs:

* An ID
* A description
* Health
* Damage
* A peaceful flag (to indicate whether they attack or not)
* A greeting message
* A gift item ID
* A list of items they drop when defeated

I started by writing the constructor:

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) {}

This was my first time initializing a std::vector field in a constructor initializer list. I had to look up how initializer lists work for vectors inside a class.

I also implemented a default constructor:

character::character() : id(""), description(""), health(0), damage(0),

peaceful(false), greeting(""), gift\_item("") {}

**Getter Functions**

I implemented getters for each attribute:

std::string character::get\_id() const { return id; }

std::string character::get\_description() const { return description; }

int character::get\_health() const { return health; }

int character::get\_damage() const { return damage; }

std::vector<std::string> character::get\_drops() const { return drop\_items; }

bool character::is\_peaceful() const { return peaceful; }

std::string character::get\_greeting() const { return greeting; }

std::string character::get\_gift() const { return gift\_item; }

I also realized that returning get\_drops() by value copies the vector, but since I am not going to be modifying it after the game starts, it probably wont matter.

**State Manipulation Functions**

I implemented functions to take damage, check if the character is alive, and clear the character’s gift item (once the player accepts it, so it doesn’t just keep duping):

void character::take\_damage(int amount) {

health -= amount;

if (health < 0) health = 0;

}

bool character::is\_alive() const { return health > 0; }

void character::clear\_gift() { gift\_item = ""; }

I created a test character object, assigned values, and printed them. I received a linker error because I hadn’t written the default constructor, which was needed by some test cases.

After adding the default constructor, everything compiled.

**6. Implementing Gradual Text**

I was getting very bored with the ugly terminal and thought it would be a good idea to make a function to print text slowly.

I didn’t know how to add a timed delay between characters in C++, so I researched and found that I could use <chrono> and <thread>. I learned that std::this\_thread::sleep\_until() allows sleeping until a future time. Instead of making this myself, I just ended up using (after slightly tweaking) this guy’s code (<https://stackoverflow.com/questions/78776920/how-to-make-a-gradual-text-function-in-c>).

I used:

#include <iostream>

#include <string>

#include <thread>

#include <chrono>

void cool\_text(const std::string& text) {

auto tp = std::chrono::steady\_clock::now();

for (char letter : text) {

std::cout << letter << std::flush;

tp += std::chrono::milliseconds(50);

std::this\_thread::sleep\_until(tp);

}

}

When I tested it, it displayed beautifuly.

**7. Implementing the Game Setup**

With the important classes for the player, items, rooms, characters, and gradual text complete, I moved on to the game class in game.cpp. This class will control the actual game execution.

I started by writing the setup() function, whose purpose was to load all of the game data from text files into memory. I wrote the function to call separate loaders for rooms, items, characters, item aliases, character aliases, and the list of required treasures.

void game::setup() {

load\_rooms("rooms.txt");

load\_items("items.txt");

load\_characters("characters.txt");

load\_item\_aliases("item\_aliases.txt");

load\_character\_aliases("character\_aliases.txt");

load\_required\_treasures("required\_treasures.txt");

}

At this point, I haven’t written the actual loader functions yet, but I want setup() to show the structure of the game’s initialization. The idea is that each loader will be responsible for reading one of the text files and then populating my other functions with that data.

I wrote some test code to have them read some random text from the three files just to make sure everything compiles.

I have never written a program that read a text file then turned its contents into objects before. I knew I would still be using std::getline() to read lines as usual, but I didn’t know how to split a line into parts then turn these parts into variables or objects.

After a long time researching, I figured out how delimiters work, discovered std::istringstream, .find() and .substr() and how they worked for extracting substrings. I can use these to basically make the function open the file, read the lines, parse each one, then create the objects from the parsed lines. (<https://stackoverflow.com/questions/4533652/how-to-split-string-using-istringstream-with-other-delimiter-than-whitespace>).

**8. Implementing load\_rooms()**

The first loader function I implemented was load\_rooms(). Each line of the rooms.txt file was formatted like this:

room\_id|room\_description|required\_size|exit1=destination1,exit2=destination2,...

I needed to parse each line into:

* The room’s ID
* The description
* The size requirement
* A string containing all the exits

I wrote the function as follows:

void game::load\_rooms(const std::string& filename) {

std::ifstream file(filename);

if (!file.is\_open()) {

std::cerr << "Error: Could not open room file.\n";

is\_running = false;

return;

}

std::string line;

while (std::getline(file, line)) {

if (line.empty() || line[0] == '#') continue;

std::istringstream line\_stream(line);

std::string id, description, size\_required, exit\_string;

std::getline(line\_stream, id, '|');

std::getline(line\_stream, description, '|');

std::getline(line\_stream, size\_required, '|');

std::getline(line\_stream, exit\_string);

room r(id, description, size\_required);

std::istringstream exit\_stream(exit\_string);

std::string exit\_entry;

while (std::getline(exit\_stream, exit\_entry, ',')) {

size\_t eq\_pos = exit\_entry.find('=');

if (eq\_pos != std::string::npos) {

std::string command = exit\_entry.substr(0, eq\_pos);

std::string destination = exit\_entry.substr(eq\_pos + 1);

r.add\_exit(command, destination);

}

}

rooms[id] = r;

}

current\_room = "meadow";

}

This function was the most complicated function I had written up to this point. It involved multiple layers of parsing: First, splitting the line by | delimiters, then splitting the exit\_string by , delimiters then splitting each exit pair by =.

When I first tested the function, it crashed. I discovered that blank lines and comment lines in the file caused the parsing code to run even though they didn’t contain valid data, so I added

if (line.empty() || line[0] == '#') continue;

I also accidentally discovere that a missing = in an exit entry caused substr() (which puts the split string into another string) to crash because find() returned the special value for the end of a string npos. I fixed this by adding a check for if (eq\_pos != std::string::npos) before using substr().

I tested the loader by printing the loaded rooms and their exits to monfirm that rooms were being created correctly, their descriptions were stored, and that their exits where mapped properly.

**9. Implementing load\_items()**

Once I got load\_rooms() working, I moved on to load\_items(), which was similar but had even more fields.

Each line of items.txt was formatted like this:

item\_id|description|location|damage|hunger\_restore|size\_change|alias1,alias2,...

I wrote the function like this:

void game::load\_items(const std::string& filename) {

std::ifstream file(filename);

if (!file.is\_open()) {

std::cerr << "Error: Could not open item file.\n";

is\_running = false;

return;

}

std::string line;

while (std::getline(file, line)) {

if (line.empty() || line[0] == '#') continue;

std::istringstream line\_stream(line);

std::string id, description, location, damage\_str, hunger\_str, size\_change, aliases\_str;

std::getline(line\_stream, id, '|');

std::getline(line\_stream, description, '|');

std::getline(line\_stream, location, '|');

std::getline(line\_stream, damage\_str, '|');

std::getline(line\_stream, hunger\_str, '|');

std::getline(line\_stream, size\_change, '|');

std::getline(line\_stream, aliases\_str);

int damage = std::stoi(damage\_str);

int hunger\_restore = std::stoi(hunger\_str);

item new\_item(id, description, damage, hunger\_restore, size\_change);

all\_items[id] = new\_item;

if (!location.empty()) {

items\_in\_rooms[location].push\_back(id);

}

std::transform(id.begin(), id.end(), id.begin(), ::tolower);

item\_alias\_map[id] = id;

std::istringstream alias\_stream(aliases\_str);

std::string alias;

while (std::getline(alias\_stream, alias, ',')) {

alias.erase(std::remove\_if(alias.begin(), alias.end(), ::isspace), alias.end());

std::transform(alias.begin(), alias.end(), alias.begin(), ::tolower);

if (!alias.empty()) item\_alias\_map[alias] = id;

}

}

}

This function required additional steps I hadn’t encountered before:

* Converting damage\_str and hunger\_str from strings to integers with std::stoi()(<https://cplusplus.com/reference/string/stoi/>)
* Adding an item ID to a vector inside a std::map for the room’s inventory
* Normalizing the ID and aliases to lowercase
* Using std::transform and std::remove\_if to turn to lowercase and remove whitespace

I also found that if a line in the file was missing a field, the parser would just continue working but leave variables uninitialized, so I just validated that all required fields were present.

I tested by printing the loaded items and verifying that each item was stored under its correct ID and also appeared in the correct room’s inventory.

This function took a long time to debug because of the sheer number of fields, string conversions, and alias handling.

**10. Implementing Input Handling**

After finishing the loading functions, I will start working input handling. I need a way for the player to type commands and have the game process them. For this, I created the control class in control.cpp, whose job was to normalize and interpret user input.

At first, I thought handling input would be pretty simple (checking for strings like "go north" or "take item"), but I realized that doing that for all of the input variations would be a bad idea – so I decided to clean and normalize the input before matching it to any commands like with my load functions. Unlike these, though, I needed to remove punctuation (go north!, for eg.), so I used ::ispunct with removeif. (https://www.programiz.com/c-programming/library-function/ctype.h/ispunct)

Here is what I wrote:

std::string control::normalize\_input(const std::string& raw) {

std::string cleaned = raw;

std::transform(cleaned.begin(), cleaned.end(), cleaned.begin(), ::tolower);

cleaned.erase(std::remove\_if(cleaned.begin(), cleaned.end(), ::ispunct), cleaned.end());

return cleaned;

}

**Handling Direction Aliases**

Next, I wanted the game to recognize other words for movement directions. For example, n should be treated the same as go north. A little research on how to do that yeilded results on using std::unorder\_map for aliases. (<https://cplusplus.com/reference/unordered_map/unordered_map/>)

I wrote:

std::string control::normalize\_direction(const std::string& input) {

static std::unordered\_map<std::string, std::string> direction\_aliases = {

{"north", "go north"}, {"n", "go north"},

{"south", "go south"}, {"s", "go south"},

{"east", "go east"}, {"e", "go east"},

{"west", "go west"}, {"w", "go west"},

{"up", "go up"}, {"u", "go up"},

{"down", "go down"}, {"d", "go down"}

};

std::string cleaned = normalize\_input(input);

if (direction\_aliases.count(cleaned)) {

return direction\_aliases[cleaned];

}

return cleaned;

}

When testing, I realised that using direction\_aliases[cleaned] without checking if whatever was being cleaned already existed in the map basically inserted what I was trying to access in the map if it wasn’t in the map (had a similar problem with the normal (ordered) map which I was using for rooms, before realizing that the [] operator can both access and insert stuff in the map. After that, used insert() for insertion and [] only for access after a check). I fixed this by using .count() to make sure I wasn’t modifying my map accidentally mid game.

**Implementing Command Processing**

Finally, I wrote the main process\_command() function inside control.cpp to handle each possible command. The function would normalize the input, check for known commands, and execute actions accordingly.

I started by writing simple commands like "look", "inventory", and "quit":

void control::process\_command(const std::string& input) {

std::string cleaned = normalize\_direction(normalize\_input(input));

if (cleaned == "look") {

print\_room\_state(false);

} else if (cleaned == "inventory" || cleaned == "i" || cleaned == "invent") {

const auto& inv = player\_data.get\_inventory();

if (inv.empty()) {

std::cout << "Your inventory is empty.\n";

} else {

for (const auto& item\_id : inv) {

std::cout << "- " << all\_items[item\_id].get\_description() << "\n";

}

}

} else if (cleaned == "quit") {

is\_running = false;

std::cout << "Goodbye.\n";

}

// Additional command handling here...

}

Adding more complex commands like "take item", "use item", "attack character", and "throw item at character" required me to parse input strings to extract IDs and targets, just like with loading characters.

For example, parsing "throw rock at goblin":

size\_t at\_pos = cleaned.find(" at ");

if (at\_pos != std::string::npos) {

std::string item\_id = resolve\_item\_id(cleaned.substr(6, at\_pos - 6));

std::string target\_id = resolve\_character\_id(cleaned.substr(at\_pos + 4));

// process throw action...

}

Stuff kept breaking when testing until I realised the problem was with me using substr(6, at\_pos) instead of substr(6, at\_pos - 6).

Each command follows a similar structure: normalize input, parse IDs, check if the IDs exist, and then perform the action.

**11. Implementing Combat Handling**

Next, I focused on combat interactions. I wanted to handle fighting an enemy, dealing damage, checking for defeat, triggering drops, and handling retaliation. I also did not want to create a system for weapons, and Alice is like 10 years old anyway and probably couldn’t use a sword so I just decided to make combat punch-only for melee and throwing stuff at characters for ranged attacks.

I wrote a handle\_combat() function inside character.cpp to create that behavior:

void character::handle\_combat(character& enemy, const std::string& enemy\_id, int damage\_dealt) {

enemy.take\_damage(damage\_dealt);

if (!enemy.is\_alive()) {

std::cout << enemy\_id << " is defeated!\n";

for (const auto& drop : enemy.get\_drops()) {

items\_in\_rooms[current\_room].push\_back(drop);

std::cout << enemy\_id << " dropped a " << drop << ".\n";

}

auto& room\_characters = characters\_in\_rooms[current\_room];

auto it = std::find(room\_characters.begin(), room\_characters.end(), enemy\_id);

if (it != room\_characters.end()) {

room\_characters.erase(it);

}

return;

}

int retaliation = enemy.get\_damage();

bool crit = (rand() % 100) < 20;

if (crit) {

retaliation = static\_cast<int>(retaliation \* 1.5);

std::cout << "Critical hit from " << enemy\_id << "!\n";

}

player\_data.take\_damage(retaliation);

std::cout << enemy\_id << " strikes back for " << retaliation << " damage!\n";

if (player\_data.get\_health() <= 0) {

std::cout << "You have been fatally wounded...\n";

is\_running = false;

}

}

This function was my first time using rand() to generate random numbers. I had to research how to generate a percentage chance for a critical hit. (<https://cplusplus.com/reference/cstdlib/rand/>)

I tested combat by manually creating a hostile character in the room, setting their health low, and attacking them. The output confirmed the enemy was defeated, dropped items, and was removed from the room.

**12. Implementing Room Display**

Next I am going to make print\_room\_state() inside location.cpp to handle what the player sees when entering or revisiting a room.

I want it to print:

* The room description (only the first time)
* Visible exits
* Visible characters
* Visible items
* Chest contents
* Player’s status bar

I wrote:

void room::print\_room\_state(bool just\_moved) {

if (!visited\_rooms[current\_room]) {

std::cout << "\n" << room\_description << "\n";

visited\_rooms[current\_room] = true;

} else if (just\_moved) {

std::cout << "\nYou are back in the " << current\_room << ".\n";

}

const auto& exits = get\_all\_exits();

if (!exits.empty()) {

std::cout << "Exits visible: ";

bool first = true;

for (const auto& [dir, \_] : exits) {

if (!first) std::cout << ", ";

std::cout << dir;

first = false;

}

std::cout << ".\n";

}

if (!characters\_in\_rooms[current\_room].empty()) {

std::cout << "You see someone:\n";

for (const auto& char\_id : characters\_in\_rooms[current\_room]) {

std::cout << "- " << all\_characters[char\_id].get\_description() << "\n";

}

}

if (!items\_in\_rooms[current\_room].empty()) {

std::cout << "You see:\n";

for (const auto& item\_id : items\_in\_rooms[current\_room]) {

std::cout << "- " << all\_items[item\_id].get\_description() << "\n";

}

}

if (!chests[current\_room].empty()) {

std::cout << "The chest contains:\n";

for (const auto& item\_id : chests[current\_room]) {

std::cout << "- " << all\_items[item\_id].get\_description() << "\n";

}

}

show\_status();

}

I also learned how to use structured bindings like: for (const auto& [key, value] : map). (https://www.geeksforgeeks.org/structured-binding-c/)

I also added the visited\_rooms[current\_room] check and had to add .empty() checks to avoid printing blank lines when no characters or items were present.

After testing different rooms, the function correctly printed all visible exits, items, characters, and the status bar.

**Assignment 4 Journal (continued)**

**13. Implementing Alias Loading**

After completing the core game systems, I turned my attention to handling aliases. I wanted the game to recognize different names or synonyms for the same item or character. For example, both “cake” and “tea cake” should refer to the same underlying item ID “teacake”. To achieve this, I needed to load alias mappings from external text files.

I implemented two functions: load\_item\_aliases() and load\_character\_aliases(). These functions read a text file where each line maps an ID to a comma-separated list of aliases, separated by a |.

I implemented load\_item\_aliases() like this:

void game::load\_item\_aliases(const std::string& filename) {

std::ifstream file(filename);

if (!file.is\_open()) {

std::cerr << "Could not open " << filename << "\n";

return;

}

std::string line;

while (std::getline(file, line)) {

if (line.empty() || line[0] == '#') continue;

std::istringstream iss(line);

std::string id, aliases;

if (std::getline(iss, id, '|') && std::getline(iss, aliases)) {

std::istringstream alias\_stream(aliases);

std::string alias;

while (std::getline(alias\_stream, alias, ',')) {

alias.erase(std::remove\_if(alias.begin(), alias.end(), ::isspace), alias.end());

std::transform(alias.begin(), alias.end(), alias.begin(), ::tolower);

alias.erase(std::remove\_if(alias.begin(), alias.end(), ::ispunct), alias.end());

if (!alias.empty()) {

item\_alias\_map[alias] = id;

}

}

}

}

}

When testing, I discovered that if an alias contained leading or trailing spaces, it wasn’t matching player input. I fixed this by explicitly removing spaces during loading.

I wrote load\_character\_aliases() with the same logic but storing aliases in character\_alias\_map.

I tested alias loading by printing the alias map after loading, confirming that all aliases correctly pointed to their IDs.

**14. Implementing ID Resolution**

With aliases loaded into memory, I needed a way to resolve user input to the internal ID. For example, if the user typed "stone ", it needed to resolve to "rock".

I implemented two resolver functions: resolve\_item\_id() and resolve\_character\_id(). Each function took a string input and looked it up in the corresponding alias map.

I had to make sure the input was normalized the same way as the aliases during loading. That meant stripping punctuation, converting to lowercase, and removing spaces.

I implemented resolve\_item\_id() like this:

std::string game::resolve\_item\_id(const std::string& user\_input) {

std::string key = user\_input;

std::transform(key.begin(), key.end(), key.begin(), ::tolower);

key.erase(std::remove\_if(key.begin(), key.end(), ::ispunct), key.end());

key.erase(std::remove\_if(key.begin(), key.end(), ::isspace), key.end());

if (item\_alias\_map.count(key)) {

return item\_alias\_map[key];

}

return "";

}

I wrote resolve\_character\_id() the same way, using character\_alias\_map.

**15. Final Testing and Tweaks**

With all the systems implemented—data loading, player, items, characters, rooms, command handling, combat, aliases—I entered the final testing phase.

I wrote a test plan consisting of scenarios to cover every command:

* Moving in all directions
* Looking around in each room
* Taking, dropping, using items
* Attacking and defeating enemies
* Talking to friendly characters
* Storing and retrieving items from chests
* Resolving aliases for items and characters
* Winning the game by bringing treasures to the safe room

During testing, I discovered several bugs:

* Hunger decreased too quickly; the player starved within a few turns. I adjusted the hunger decay rate.
* "talk to" command failed if user typed "talk rabbit" without "to". I modified parsing to accept both "talk rabbit" and "talk to rabbit".
* Throwing an item failed if " at " was missing spaces ("throw rockatgoblin"). I added an error message to enforce correct syntax.
* Enemies dropped duplicate items when defeated. I traced this to a loop that added drops twice and removed the extra call.
* Chests allowed more than three items. I added a check:

if (chests[current\_room].size() >= 3) {

std::cout << "The chest is full.\n";

return;

}

* Room descriptions printed every time the player entered. I added a visited\_rooms flag to suppress repeat descriptions.
* ASCII art was misaligned due to inconsistent spacing in cout. I adjusted spacing and added padding.
* Win condition failed because the required\_treasures file omitted one treasure ID. I updated the text file to match.

I replayed the entire game several times, testing different paths, command variations, and edge cases. After each run, I fixed issues and retested until no critical bugs remained.

I also tuned game balance:

* Increased critical hit chance from 10% to 20%
* Reduced base enemy damage for weak enemies
* Slowed hunger decay by increasing delay between decrements

Finally, I added an introduction using cool\_text() to print an opening story gradually, followed by an ASCII title screen.

At this point, I confirmed:

* All input commands worked
* Aliases resolved correctly
* All required treasures could be found and stored
* The win condition triggered when returning to the safe room with all treasures
* Combat, inventory, chests, and interactions worked end-to-end

This marked the completion of the project.