**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.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. Since the assignment required an object-oriented approach, I knew I would need separate classes for different parts of the game. I created the following files right 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
* inventory.cpp / inventory.h
* control.cpp / control.h
* action.cpp / action.h

I created these files as empty placeholders to establish the skeleton of the project. I made sure to write header guards inside each .h file to avoid duplicate inclusion errors during compilation. This was something I had learned from previous assignments: without include guards, compilation errors will occur if headers are included multiple times.

Once the files were created, I wrote a minimal main.cpp to test if the structure would compile. I included "game.h" and wrote a simple main() function that would instantiate a game object and call setup() and run() on it. The idea here is that these methods would later handle loading data and running 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 implementing the player class because it was probably not going to depend on other classes as much as the rest of the classes will. The player would need attributes like health, hunger, size, base damage, and an inventory. Since I needed default values, I implemented the constructor to initialize them.

player::player()

: player\_inventory(new inventory()), size("normal"), hunger(100), health(100), base\_damage(7) {}

I set the player’s initial health and hunger to 100, size to "normal", and base damage to 7. This may change after completing the program during balancing.  
The inventory is operated by a separate class, but is linked to player through by an inventory object, which is allocated with new inside the player constructor and deleted in the destructor.

player::~player() {

delete player\_inventory;

}

After the constructor, I made basic getter and setter functions for health, and made sure player health can’t go above nor below its normal values:

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 used the same approach for hunger:

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; }

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; }

Inventory functions all forward to the player\_inventory member. For example, to add an item:

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

if (player\_inventory->get\_items().size() >= player\_inventory->INVENTORY\_MAX\_SIZE) {

std::cout << "You can't carry any more items (max 7).\n";

return;

}

player\_inventory->add\_item(item\_id);

}

Other inventory operations:

bool player::has\_item(const std::string& item\_id) const { return player\_inventory->has\_item(item\_id); }

void player::remove\_item(const std::string& item\_id) { player\_inventory->remove\_item(item\_id); }

const std::vector<std::string>& player::get\_inventory() const { return player\_inventory->get\_items(); }

Size is just a string:

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

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

I tested the player class by instantiating a player, adding and removing items, taking and healing damage (wrote a test function to do this).  
This was the first time I managed dynamic memory for a member object in C++ and I made sure to delete the inventory in the destructor to avoid any memory leaks.

**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 some 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.  
The class is only a data container, but it has to work with both food and weapons (for restoring hunger and for combat), as well as items that alter player size.

Keeping the naming conventions I was using consistent was surprisingly difficult.

**4. Implementing the Inventory Class**

At first I had implemented the inventory functionality as part of the player class, but after taking another look at the assignment instructions, I realized that I would have to place it in its own class. So, I created the inventory class, then created the inventory itself, which is a vector of strings (item IDs), with a hard coded maximum size (7). I also added functions to add and remove items to the vector.

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

if (items.size() >= INVENTORY\_MAX\_SIZE) {

return;

}

items.push\_back(item\_id);

}

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

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

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

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

}

}

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

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

}

const std::vector<std::string>& inventory::get\_items() const {

return items;

}

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 the aforementioned gaps. (<https://en.wikipedia.org/wiki/Erase%E2%80%93remove_idiom>).

**5. Implementing the Room Class (location.cpp)**

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\_size\_required() const { return size\_required; }

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

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:

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.

I also wrote print\_room\_state to print out the available exits, items, and characters in a room as well as chest contents. I took into account whether or not the player had entered the room before, so instead of printing out the default description, it prints out a “you are back in the ” message.

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

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

std::cout << "\n" << rooms[current\_room].get\_description() << "\n";

visited\_rooms[current\_room] = true;

} else if (just\_moved) {

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

}

const auto& exits = rooms[current\_room].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.count(current\_room) && !characters\_in\_rooms[current\_room].empty()) {

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

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

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

}

}

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

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

for (const std::string& resolve\_item\_id : items\_in\_rooms[current\_room]) {

std::string desc = item\_manager->all\_items[resolve\_item\_id].get\_description();

if (desc.empty()) desc = resolve\_item\_id;

std::cout << "- " << desc << "\n";

}

}

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

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

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

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

}

}

game\_manager->show\_status();

}

This was my first time using dynamic maps and nesting containers in this way, and it required a lot of debugging to get working properly. I also had to learn how to use structured bindings like: for (const auto& [key, value] : map). (<https://www.geeksforgeeks.org/structured-binding-c/>)

**6. 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() : id(""), description(""), health(0), damage(0), peaceful(false), greeting(""), gift\_item("") {}

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) {}

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; }

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 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.

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.

Testing was done by creating characters, attacking, checking peaceful vs hostile logic, and verifying drops/gifts.

**7. Implementing the Game Class and Setup Sequence**

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 and it is also responsible for tying together all the other components and running the main loop. I made the game class own pointers to every other important class: player, item, character, room, action, and control. Each was allocated in the constructor with new and properly deleted in the destructor to avoid memory leaks:

game::game() : is\_running(true) {

player\_data = new player();

item\_manager = new item();

character\_manager = new character();

room\_manager = new room();

room\_manager->game\_manager = this;

action\_manager = new action(this);

control\_manager = new control(this);

room\_manager->character\_manager = character\_manager;

room\_manager->item\_manager = item\_manager;

room\_manager->player\_data = player\_data;

room\_manager->game\_manager = this;

}

game::~game(){

delete player\_data;

delete room\_manager;

delete control\_manager;

}

The destructor only deletes what the game class directly owns. Some pointers are assigned for back-referencing (like room\_manager->game\_manager = this), but aren’t owned, so they aren’t deleted here. This avoids double deletes and segmentation faults.

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, actions, 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");

action\_manager->load\_actions("actions.txt")

}

-----------------------------------copied from other version ----------------------------

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>).

-----------------------------------copied from other version ----------------------------

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);

r.character\_manager = character\_manager;

r.item\_manager = item\_manager;

r.player\_data = player\_data;

r.game\_manager = this;

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);

}

}

room\_manager->rooms[id] = r;

}

room\_manager->current\_room = "meadow";

}

Each loader works in the same way – it parses the lines using string streams, splits fields on ‘|’ or ‘,’ then builds the objects for rooms, items, characters and alias mappings. Errors (like a missing file, etc…) prints to cerr and stops the game with is\_running = false.

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.

**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);

        item\_manager->all\_items[id] = new\_item;

        if (!location.empty()) {

            room\_manager->items\_in\_rooms[location].push\_back(id);

        }

        // Aliases

        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 how big it was and all of the string conversions and alias handling.

**8. Writing the Main Game Loop**

Once data loading and object relationships were working, I wrote the game’s main loop in game::run(). The loop is responsible for displaying the game state, reading player input, updating hunger and health, and passing commands to the control system.

The function starts by clearing the screen (just printing 20 newlines—simple and portable). Then it prints an ASCII art banner and intro explaining how the game works. After that, the description for the starting room is printed. This includes the room's full description, exits, visible characters, items, and chest contents. This is a picture of the ascii from my



Here's the start of the loop setup:

void game::run() {

for (int i = 0; i < 20; ++i) std::cout << "\n";

std::cout << R"(

ASCII HERE (too big, see image)

)";

std::cout << "\nWelcome to Alice in Wonderland.\n";

std::cout << "Goal: Find all required treasures and return them to the safe room.\n";

std::cout << "Commands: look, go <direction>, inventory, take <item>, drop <item>, use <item>, attack <character>, talk to <character>, throw <item> at <character>\n";

std::cout << "Explore carefully. Some rooms require specific sizes to enter.\n\n";

    std::string input;

    if (!room\_manager->visited\_rooms[room\_manager->current\_room]) {

        std::cout << "\n" << room\_manager->rooms[room\_manager->current\_room].get\_description() << "\n";

       room\_manager->visited\_rooms[room\_manager->current\_room] = true;

    } else {

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

    }

    // Show visible exits

    const auto& exits = room\_manager->rooms[room\_manager->current\_room].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";

    }

    // Show characters

    if (room\_manager->characters\_in\_rooms.count(room\_manager->current\_room)) {

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

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

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

        }

    }

    // Show items

    if (room\_manager->items\_in\_rooms.count(room\_manager->current\_room)) {

        std::cout << "You see:\n";

        for (const std::string& resolve\_item\_id : room\_manager->items\_in\_rooms[room\_manager->current\_room]) {

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

        }

    }

    // Show chest contents

    if (room\_manager->chests.count(room\_manager->current\_room) && !room\_manager->chests[room\_manager->current\_room].empty()) {

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

        for (const auto& id : room\_manager->chests[room\_manager->current\_room]) {

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

        }

    }

    show\_status();

    while (is\_running) {

        std::cout << "> ";

        std::getline(std::cin, input);

        // Hunger decreases every round

        int hunger\_loss = 1 + rand() % 3

        player\_data->change\_hunger(-hunger\_loss);

        if (player\_data->get\_hunger() <= 0) {

            std::cout << "You have died of starvation.\n";

            is\_running = false;

            return;

        }

        int current\_hunger = player\_data->get\_hunger();

        if (current\_hunger < 30 && current\_hunger > 0) {

            std::cout << "You feel very hungry. Hunger: " << current\_hunger << "\n";

        }

        control\_manager->process\_command(input);

    }

}

Most of the functionality is back referenced to all of the other classes, but the messages that the player sees are mostly hard coded in run(). Status bars for health and hunger are printed using ASCII art (20-block bars), and size is shown as a label.

The main input loop is pretty simple:

* Prompt with > and read a line from cin.
* Subtract a random amount from hunger each turn:

int hunger\_loss = 1 + rand() % 3;

player\_data->change\_hunger(-hunger\_loss);

* If hunger is zero, print a starvation death message and exit.
* If hunger is below 30, warn the player.
* Pass the raw input string to control\_manager->process\_command(input).

The loop continues until is\_running is set to false, which happens if the player wins, quits, or dies.

Testing this loop was tricky at first because I had to wire up all the printing and cross-class dependencies to make sure the player could see feedback for everything they did, and I wanted to be able to see each component (room, character, item, status) in each turn.

**9. Building the Control and Action System**

After getting the main loop working, 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 tis normalize user input then route it to actions..

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)

I wrote the process\_command method which lowercases everything and strips punctuation to allow commands to be flexible. For example if the player writes “Attack” or “attack!” or “aTtAcK” it will all end up being normalized to attack and accepted as a valid command. 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;

}

For movement, 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 a function that maps things like “n”, “walk north”, “up”, and even “climb up” to the right “go <direction>” command. This lets the user input almost anything that makes sense for navigation and have it work. I decided to hardcode this simply because there are only 4 cardinal directions in addition to going up and down, and I don’t think there is really any reason to customize these or to give them any other aliases than the obvious ones. I also want to add that despite adding going up and down as possible directions, it is not part of my current plan to use them in my final game. I just saw this as an opportunity to add them so if I change my mind later or decide to come back and expand on the game’s storyline, it would be an easy thing to upgrade.

Once normalized, the input is broken into verb and parameter, and the verb is resolved using action\_manager->resolve\_action\_id. If it matches a known action (take, use, attack, etc.), the action is dispatched to the correct method in the action class.

Example from process\_command:

std::string action\_id = action\_manager->resolve\_action\_id(verb);

if (action\_id == "attack") { action\_manager->attack(param);   return; }

if (action\_id == "take")    { action\_manager->take(param);     return; }

if (action\_id == "drop")    { action\_manager->drop(param);     return; }

if (action\_id == "use")     { action\_manager->use(param);      return; }

If no action is matched, the system checks if the command is a movement alias, and then tries to move the player.

If the move is valid and the player’s size matches the target room’s requirement, the player is moved and the new room state is displayed. If the player enters the safe room, the game checks if all required treasures are present; if so, the win message is printed.

Commands for viewing items in the inventory are handled directly in control, so “inventory”, “i”, and “invent” all print the player’s carried items. If the inventory is empty, a message is shown; otherwise, the full descriptions of all items are printed.

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.

**10. Implementing Action Logic**

With input normalization working, I will write all of the logic and functionality of the game’s actions inside the action class. I will have to make: item pickup, use, drop, storing and retrieving from chests, combat, dialogue, and throwing items.

**Taking and Dropping Items:**  
When the player takes an item, the action method needs to check if that the item exists in the room (also that the inventory isn’t full, but that is already done elsewhere). It also needs to use the alias map to resolve what the player typed to a real item ID. If the inventory isn’t full, the item is moved to the player and removed from the room (which is the inventory class’s job). Dropping is the reverse: check inventory, remove from player, add to the current room.

This is the take method I wrote:

void action::take(const std::string& raw) {

    std::string item\_id = game\_manager->resolve\_item\_id(raw);

    if (item\_id.empty()) {

        std::cout << "There is no item called '" << raw << "' here.\n";

        return;

    }

    auto& items\_here = room\_manager->items\_in\_rooms[room\_manager->current\_room];

    auto it = std::find(items\_here.begin(), items\_here.end(), item\_id);

    if (it != items\_here.end()) {

        player\_data->add\_item(item\_id);

        items\_here.erase(it);

        std::cout << "You picked up the " << item\_id << ".\n";

    } else {

        std::cout << "There is no " << item\_id << " here.\n";

    }

}

**Using Items:**  
To use an item, I first need to confirm that the player has it. Then:

* If the item restores hunger, the value is added and the item is removed.
* If the item changes size, the player’s size is updated and a message is printed.
* After use, the item is always removed from inventory.

I also need to make sure that some items cannot be used. For example, for an item like rock, if the player uses consume, they should get a message that the item is not consumable.

HERE PUT CODE OF FOR ITEMS THAT CANNOT BE CONSUMED AND ITEMS IN GENERAL

**Storing and Retrieving Items (Chests):**  
the chest room has a chest with a max of 3 items.

* To store, check inventory and that the chest isn’t full, then move the item.
* To retrieve, check the chest and that inventory isn’t full, then move the item back.

**Combat (Attacking and Throwing):**  
For attack, the player chooses a target (again, resolved through aliases).

* If the character is present and alive, damage is dealt—critical hit chance is handled with rand() % 100 < 25.
* If the enemy dies, it drops items to the room and the character is removed.
* If not, the enemy counterattacks (also with critical chance).

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/>)

Throwing an item is similar but checks if the item can be used as a weapon (i.e., has nonzero damage). Whether or not the throw hits has a randomized chance (70% chance), and then damage is applied. If the character is defeated, they drop items as in regular combat. Any item that is thrown is removed (regardless of whether it hits or not).

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.

**Dialogue and Gifts:**  
Talking to a character works differently depending on if they are peaceful:

* If peaceful, the character greets the player and may hand over a gift (but only once; gift is cleared after).
* If hostile, talking counts as provoking an attack, and the character damages the player.

All action results are printed directly whether it was a success or failure, why it failed, and the results. I wrote everything so that no command fails silently: the player should always be shown what happened, even if the answer is just “You can’t do that.”

I tested every action by using typos, invalid aliases, and invalid objects to make sure edge cases gave a sensible error message or fallback response.

**11. Finalizing Game Balance, Win/Loss, and Output Polish**

After the command system and all major actions were working, I focused on tuning the game’s difficulty and making the user experience clean. I wanted the game to be hard but fair, with clear feedback for every action, and no way for the player to get stuck due to a bug or unclear error message.

**Balancing Health, Hunger, and Damage:**  
I adjusted the starting player health and hunger to 100 each. Every turn, hunger drops by 1–3 points randomly, and if it hits 0, the player dies immediately (no grace period). This mechanic forces the player to search for food and manage inventory space carefully—there are only so many consumable items on the map.  
All weapons and food values were tuned so that it’s possible (but not easy) to survive and win if the player explores, uses resources smartly, and avoids unnecessary fights.

Combat is intentionally dangerous:

* Enemy attacks use the same damage system as the player, with critical hits possible for both sides.
* The system always prints updated health and hunger bars after every attack, using 20-block ASCII graphics for clarity.
* Player death (from either hunger or combat) is always final, with a unique printed message before the program ends.

**Inventory and Chest Limits:**  
I hardcoded the player inventory limit to 7 items. Most playtesters wanted to carry everything, but this limit forces choices, especially since food, treasures, and weapons all compete for space.  
Room chests can hold up to 3 items each. I implemented this to give the player a way to offload treasures or nonessential gear and come back for it later.

**Winning and Losing:**  
The win condition is only checked when the player enters the "safe room." The code checks if the player's inventory contains all required treasures (loaded from a separate file). If so, a clear win message is printed and is\_running is set to false, ending the game.  
No other room allows the player to win, and there are no soft win conditions—either all treasures are returned or the player loses (starves or dies in combat).

**Output and UX Details:**  
I added instructions, help text, and clear command prompts. Every error is explained: wrong item, invalid move, chest full, inventory full, already used an item, etc.  
All command aliases are loaded from file, so I could tweak language and add synonyms without touching the code.

The final polish step was just playing the game end-to-end several times, intentionally making mistakes, and fixing any missing or confusing output. I wanted it to feel like a classic text adventure: clear, a little harsh, but always logical and fair.

**12. Known Limitations and Lessons Learned**

While the game is fully playable and meets the assignment requirements, there are still several limitations and lessons that came out of development:

* **No Save/Load System:**  
  There is no save feature. If the player dies or exits, all progress is lost. Adding save/load would require serializing the state of every manager and all game data, which would be a major project in itself.
* **Data Integrity:**  
  All data is loaded from plain text files, with no validation beyond basic error prints. If a file is missing, corrupted, or has a wrong format, the game will either fail to start or behave unpredictably. Adding robust error handling and file validation would make the system more user-proof.
* **AI and Character Behavior:**  
  Characters are either peaceful or hostile, with hardcoded logic for dialogue or combat. There is no dynamic AI, movement, or state changes for NPCs beyond death/gift.
* **Input System:**  
  While the alias and normalization system is robust, some complex multi-step or chained commands are not supported. Only the most common synonyms are included. If the player invents a new phrasing not in the alias files, it simply won't work.
* **No Puzzle Logic or Dynamic Events:**  
  The game world is static. There are no time-based events, random encounters, or room logic beyond size restrictions and treasures.

**Lessons Learned:**

* I finally got comfortable with multi-file C++ projects and cross-file class dependencies. Debugging circular includes and header guard issues is now second nature.
* File I/O and parsing by hand (with stringstreams and STL) is tedious but gave me full control over data formats.
* Explicit memory management (via new and delete) is error-prone. Modern C++ would use smart pointers or even stack objects, but I followed the assignment's style.
* Testing every feature in isolation and as part of full runs was key to not missing edge cases. I started writing test "walkthroughs" for myself, which caught many output bugs.
* User experience matters: if a command fails, the game needs to tell the player exactly why, not just fail silently.
* For future work, I would add more in-game feedback and prompts, especially for first-time players.

**13. Planned Extensions and Further Ideas**

After finishing the base game, I started brainstorming ways to expand the project, especially if I wanted to turn it into a bigger text adventure or if the assignment allowed extra features.

* **Puzzles and Multi-Step Challenges:**  
  Adding rooms or situations that require using specific items in sequence to solve problems (e.g., needing to shrink to enter a tunnel, then finding a key only accessible while small, then growing again to fight a boss). This would require tracking puzzle state and adding new command/action logic.
* **Multi-Turn Dialogue and Branching Interactions:**  
  Right now, character dialogue is a single greeting or combat response. To make characters more interesting, I would add multi-turn conversations, branching choices, and context-aware responses (tracking which topics have already been discussed).
* **Random Events and Encounters:**  
  To make each playthrough different, I could introduce random enemy spawns, chance-based item placement, or timed hazards. This would force the player to adapt and would make the world less predictable.
* **Save and Load Support:**  
  Serializing all game state to disk would allow the player to pause and resume their adventure, or experiment with risky strategies.
* **Expanded Room Features:**  
  I’d like to add secret passages, one-way doors, environmental hazards (e.g., poison mushrooms or slippery floors), and item-based room unlocking.
* **Better Inventory Management and UI:**  
  The player currently has no way to inspect item stats except for the description printout. An "examine" command would show full item details. Inventory sorting or filtering might be helpful as the number of items increases.
* **Visual Output and Accessibility:**  
  If I was allowed to use external libraries, I’d consider adding color-coded output, simple graphics, or even a basic windowed interface for clarity.

If I come back to this codebase, these are the first areas I’d address. The current engine is stable, and the data-driven approach means I could add or change items, rooms, characters, and commands without breaking the underlying logic.