Witcher Tracker Project Report

Systems Programming – Spring 2025

Üstün Yilmaz, 2023400108

June 1, 2025

1 Introduction

In this project, we are asked to implement a parser and an interpreter with background logic to handle any action in the popular roleplaying game, "the Witcher"; utilizing the C++ programming language. We are asked to do it in an Object Oriented way adhering to the best programming principles.

Since Geralt starts from scratch in our so-called game (his memory is wiped), we are asked to start with an empty knowledge base and as we keep handling action commands and queries, our game state develops.

2 Problem Description

The problem is a two-step one, in which the first step is to check the validity of the input command and the second step is to actually process the input commands and ensure the required modifications are done in Geralt's inventory. Our tracker has three types of input:

- **Actions:** The actions consist of looting ingredients, brewing potions, trading trophies for ingredients, learning effectiveness and potion formulas; and defeating beasts. This is the specific type of input which modifies our knowledge base, i.e. the only one with side effects.
- **Queries:** The queries work on Geralt's inventory to return and output the current state of our knowledge base. They can be about total items, potion formulas and effectiveness.
- **Exit:** The exit command terminates the instance. After it is executed our knowledge base is also terminated and reset with the game.

All commands are expected to follow the rules of the given BNF and if an input does not adhere to the BNF, we are expected to output `INVALID`. Any valid commands are constrained by:

- **Loot:** Always succeeds and increments ingredient count.
  Expected input: `Geralt loots <ingredient_list>`
  Expected output: `''Alchemy ingredients obtained''`
- **Trade:** Succeeds if Geralt has sufficient trophies and fails otherwise.
  Expected input: `Geralt trades <trophy_list> trophy for <ingredient_list>`

Expected output: `''Not enough trophies'' or ''Trade successful''`

- **Brew:** Succeeds if Geralt knows the formula of a potion and has enough ingredients.
  Expected input: `Geralt brews <potion>`
  Expected output: `''No formula for potion'' or ''Not enough ingredients'' or ''Alchemy item created: <potion>''`
- **Learn:** Succeeds if the available information is not already in the knowledge base of Geralt.
  Expected input: `Geralt learns <sign/potion> sign is effective against <monster>`
  OR
  Expected input: `Geralt learns <potion> potion consists of <ingredient_list>`
  Expected output: `''Already known effectiveness'' or ''New bestiary entry added: <monster>'' or ''Bestiary entry updated: <monster>''`
  OR
  Expected output: `''Already known formula'' or ''New alchemy formula obtained: <potion>''`
- **Encounter:** Succeeds if Geralt has any/enough effective counters against the beast encountered.
  Expected input: `Geralt encounters a <monster>`
  Expected output: `''Geralt is unprepared and barely escapes with his life'' or ''Geralt defeats <monster>''`
- **Queries:** Since no side effects are possible, the only constraint is to return the result of some specific queries (like formula and total general ? queries) in a sorted order.
  Expected input: `Total (ingredient/potion/trophy) ?`
  OR
  Expected input: `Total (ingredient/trophy) <ingredient/potion/trophy> ?`
  OR
  Expected input: `Total potion <potion> ?`
  OR
  Expected input: `What is effective against <monster> ?`
  OR
  Expected input: `What is in <potion> ?`
  Expected output: `''<quantity>'' OR ''<sorted_item_list>'' OR ''<sorted_counter_list>'' OR ''<sorted_ingredient_list>''`

The BNF constraints' regexes are below:

```
// Regexes for validity checking
static const std::string word = "[A-Za-z]+"; // Ingredients, signs and monster names consist of single alphabetical words
static const std::string potionWord = "[A-Za-z]+(?: [A-Za-z]+)*";
// Potion names consist of 1+ words split by single spaces
static const std::string quantity = "[1-9]\\d*"; // Quantities are positive integers without leading zeroes

static const std::string ingredientToken = quantity + "\\s+" + word; // <quantity> <ingredient>
static const std::string ingredientList = ingredientToken + "(?:\\s*,\\s*" + ingredientToken + ")*";
// <quantity> <ingredient> | <quantity> <ingredient> <ingredient_list>

// Same as ingredients just with the word "trophy at the end"
static const std::string qtyMonster = quantity + "\\s+" + word;
```

```
13 static const std::string trophyList = qtyMonster + "(?:\\s*,\\s*" +
       qtyMonster + ")*\\s+trophy";
14
15 // Valid statements and their regular expression patterns
16 static const std::vector<std::regex> validStatements = {
17         std::regex("^\\s*(?:Exit|exit)\\s*$"), // Exit command
18
19         // Action commands
20         std::regex("^\\s*Geralt\\s+loots\\s+" + ingredientList + "\\s*$
               "),
21         std::regex("^\\s*Geralt\\s+trades\\s+" + trophyList + "\\s+for
               \\s+" + ingredientList + "\\s*$"),
22         std::regex("^\\s*Geralt\\s+brews\\s+" + potionWord + "\\s*$"),
23         std::regex("^\\s*Geralt\\s+learns\\s+" + word + "\\s+sign\\s+is
               \\s+effective\\s+against\\s+" + word + "\\s*$"),
24         std::regex("^\\s*Geralt\\s+learns\\s+" + potionWord + "\\s+
               potion\\s+is\\s+effective\\s+against\\s+" + word + "\\s*$"),
25         std::regex("^\\s*Geralt\\s+learns\\s+" + potionWord + "\\s+
               potion\\s+consists\\s+of\\s+" + ingredientList + "\\s*$"),
26         std::regex("^\\s*Geralt\\s+encounters\\s+a\\s+" + word + "\\s*$
               "),
27
28         // Query commands
29         std::regex(R"(^\s*Total\s+(?:ingredient|potion|trophy)\s*\?\s*$
               )"),
30         std::regex(R"(^\s*Total\s+(?:ingredient|trophy)\s+)" + word + R
               "(\s*\?\s*$)"),
31         std::regex(R"(^\s*Total\s+potion\s+)" + potionWord + R"(\s*\?\s
               *$)"),
32         std::regex("^\\s*What\\s+is\\s+effective\\s+against\\s+" + word
                + R"(\s*\?\s*$)"),
33         std::regex("^\\s*What\\s+is\\s+in\\s+" + potionWord + R"(\s*\?\
               s*$)")
34     };
```

3 Methodology

To solve these problems, a BNF based parser utilizing `std::regex` is employed and the knowledge base/actions/queries are handled through a modular, OOP based interpreter. The algorithmic flow can be described as:

1. **Input parser:** We keep reading a line from stdin until the exit command is encountered, and send the read line to our regex validation logic.
2. **Validity checker:** The validation logic consists of a constant set of regexes which are then matched to our input command. If no match is found, we output `INVALID`. If a match is found, we send it to its respective handler method.
3. **Handler methods:** The handler methods are located inside our `Game` class, which in itself utilizes the methods we have defined in our `Inventory` class; adhering to best OOP practices and encapsulating the whole background flow. They tokenize the sent input commands and after standardizing them, process and apply the needed side effects to the knowledge base. Then they output their respective outputs.

Pseudocode for our main loop:

```
while (true) {
```

```
    print ">> ";
    read inputLine;
    if (inputLine is empty) { print "INVALID"; continue; }
    if (!isValid(inputLine)) { print "INVALID"; continue; }
    tokens = tokenize(inputLine);
    if (tokens[0] == "Exit" or "exit") break;
    game.classifyAndExecute(tokens);
}
```

Pseudocode for our handler methods:

```
standardize(tokens);
tokenize(standardizedTokens);
if(inputLine == action) modifyInventory;
if (!fail) print successfulOutput;
else print failedOutput;
```

4 Implementation and Code Structure

The implementation of the project consists of 3 C++ classes, which ensure proper OOP practices are utilized. The names of our classes and their usages are as follows:

1. `main.cpp`: This is the main class where the input is handled and its validity is checked using a set of defined regexes. Once a command is affirmed as valid, the `classifyAndExecute` method is executed on the global `Game` instance, which is where our operations reside.
2. `Game.cpp and Game.h`: This is the `Game` class where all our handler functions are defined. The class utilizes the methods defined in `Inventory.cpp` which govern our knowledge base and ensures that proper changes are applied. To better emphasize on its purpose, we can analyze how ingredient looting is handled:
   - The "Geralt loots" input command is sent to the `handleLoot` method, which first constructs a standardized version of the input command by making use of C++'s own String Builder (`istringstream`). It then parses the standardized string again and adds the ingredient to Geralt's inventory by utilizing the `addIngredient` method defined in `Inventory.cpp`. This is a really good example of how OOP is utilized in the project.
   - After handling the knowledge base utilizing the `addIngredient` method of `Inventory.cpp`, the `handleLoot` method prints its output and finishes.
3. `Inventory.cpp and Inventory.h`: This is the `Inventory` class and it is the lowest leveled class in our project logic. It includes **two main data structures, `<map>` and `<vector>`** to store items like ingredients, potions, bestiary, formulas etc. To better emphasize on its purpose, we can continue our analysis on how ingredient looting is handled in the background:
   - The `addIngredient` method defined in the class accepts two parameters: the name of the ingredient and how much we are looting.
   - It then instantiates a map entry with the name of our ingredient in the `map<string, int> ingredients`, and increments its quantity by the amount we have specified (quantity).

- Every method is handled similarly using either vectors or maps. We also have other methods like `createList` for sorting a map of ingredients/potions/trophies, but the overall workflow is the same.

These classes are the backbone of our whole project and each operation has their own respective handling logic, which are pretty much the same as `handleLoot`, with some exceptions being `handleEncounter` checking two maps of signs and potions and then checking Geralt's inventory in advance; but in the end, it all revolves around **2 primary data structures: `<vector>`s and `<map>`s**.

As I have decided to include implementation and code structure in the same section since in a modular OOP based project like this, everything is best explained in unification; I have also decided to explain the functions and modules using the whole workflow logic of handling loot commands. As seen by the sample code below, proper OOP practices and adequate commenting are both applied.

4.1 Sample Code of Loot Action

```
int main() {
    ...
    if (valid) {
        witcher.classifyAndExecute(parsedLine);
    }
    ...
}
------- Game.cpp -------
/*
 * Method to split a reconstructed uniform parsedLine and remove those
    nasty commas from it
 */
std::vector<std::string> Game::splitAndTrimCommas(const std::string &
   input) {
    // Declare the needed structures
    std::vector<std::string> splitLine;
    std::istringstream iStringStream(input);
    std::string token;

    // Trim the spaces and delimit by comma
    while (std::getline(iStringStream, token, ',')) {
        trimSpaces(token);
        if (!token.empty()) splitLine.push_back(token);
    }
    return splitLine;
}
/*
 * Method to handle a loot statement
 */
void Game::handleLoot(const std::vector<std::string> &parsedLine) {
    std::ostringstream stringBuilder; // To form a uniform version of
       the command with commas handled
    for (size_t i = 2; i < parsedLine.size(); ++i) {
        stringBuilder << parsedLine[i] << ' ';
    }
    std::string afterLootStatement = stringBuilder.str();
    std::vector<std::string> lineWithoutCommas = splitAndTrimCommas(
       afterLootStatement);
    // We will remove the commas here
```

```
36
37      // Parse the quantity and ingredient name to be added to the
            inventory here
38      for (std::string &token: lineWithoutCommas) {
39          trimSpaces(token);
40          std::istringstream iStringStream(token);
41          int quantity;
42          iStringStream >> quantity; // Put the quantity in the string
                buffer
43          std::string ingredientName;
44          std::getline(iStringStream, ingredientName);
45          trimSpaces(ingredientName); // Trim spaces
46          inventory.addIngredient(ingredientName, quantity); // Add the
                ingredient
47      }
48
49      std::cout << "Alchemy ingredients obtained" << std::endl; //
            Operation done
50  }
51  ------- Inventory.cpp --------
52  /*
53   * Method to add an ingredient to the inventory
54   */
55  void Inventory::addIngredient(const std::string &ingredientName, int
        quantity) {
56      ingredients[ingredientName] += quantity;
57  }
```

5 Results

Since we have a concrete error-proof regex-based validation logic, a standardization based reparsing logic and solid data structures like `map` and `vector`; the results were surprisingly accurate as the previous assignment (Assignment 1)'s testcases were passed with 100% accuracy. For example, the sample input for `input1.txt` and its respective output are:

Input:

```
Total ingredient Darkbloom?
Geralt loots 14 Silverspore,2 Bloodmoss
Geralt loots 4 Serpentine,2 Ghostwort,3 Crowseye,7 Sagewort
Geralt learns Elder Blood potion consists of 7 Voidfrost,2 Wolvenrot,2 Nightpearl
,3 Felandaris,2 Earthshard
Total ingredient Dryadsbark?
Geralt trades 10 Draugir,3 Dagon,9 Tribunal,3 Arachas,3 Vampire,
7 Rotfiend,2 Cemetaur,8 Archespore,18 Ghoul trophy for 6 Ghoulgrass
What is effective against Gaunter?
Geralt brews Vampire Kiss
Geralt loots 3 Shiverfern
Geralt loots 3 Frostwisp
Geralt loots 3 Ruffles,6 Necrobloom,11 Sunspire,13 Gloompetal,1 Ashthorn,12 Firefern
Geralt brews Wolven Decoction
Geralt encounters a Noonwraith
Geralt loots 1 Moonflower,1 Bruxaleaf
Geralt brews Ekhidna Decoction
```

```
Geralt loots 32 Frostwisp,3 Frostcap,2 Voidorchid,11 Vermilion,
1 Winterberry,11 Witchhazel,7 Drakelily
Geralt learns Archgriffin Decoction potion consists of 14 Mistvale,
5 Feylily,3 Aether,8 Wispeye,4 Shimmerleaf,
3 Silvertear,5 Tearleaf,3 Bloodmoss,11 Duskthorn,
12 Emberleaf,1 Dimeritium,9 Trollweed
Geralt brews Spirits Solace
Geralt encounters a Botchling
Total potion White Raffards Decay?
Exit
```

Output:

```
0
Alchemy ingredients obtained
Alchemy ingredients obtained
New alchemy formula obtained: Elder Blood
0
Not enough trophies
No knowledge of Gaunter
No formula for Vampire Kiss
Alchemy ingredients obtained
Alchemy ingredients obtained
Alchemy ingredients obtained
No formula for Wolven Decoction
Geralt is unprepared and barely escapes with his life
Alchemy ingredients obtained
No formula for Ekhidna Decoction
Alchemy ingredients obtained
New alchemy formula obtained: Archgriffin Decoction
No formula for Spirits Solace
Geralt is unprepared and barely escapes with his life
0
```

6 Discussion

The project implementation performs its duty in less than half a second and in perfect precision. The biggest challenge when implementing this project was to learn the C++ libraries and best OOP paradigms and practices. After that was accomplished, the implementation was much easier thanks to C++'s safe memory allocation and safer data structures with many utilities to make our job easier like lambda functions.

The ease of usage is one of the project's strengths since the input lines are read from stdin and processed outputs are outputted to stdout. We simply type the input to stdin after instantiating ./witchertracker and when we hit enter the command is processed immediately.

7 Conclusion

In conclusion, the project helped us learn proper OOP concepts and to utilize them. It was identical as the first project, so similar conclusions and future enhancements can be

inferred from there, which are:

- The project shows us how role-playing games like The Witcher may have worked in their early stages.
- The project gives us an idea on how game programming and state-storing works.
- A possible future enhancement would be to implement the BNF handling in a recursive way, but the regular expressions are rapid enough for test cases not exceeding 10000+ words.

References

[1] g217. Trim whitespace from a string [duplicate]. *StackOverflow*, 1(1):All pages, 2015.

AI Assistants

The AI assistants used in this project were ChatGPT and DeepSeek, which were utilized in debugging, adhering to best OOP practices, and helping format my LaTeXproject report.

- DeepSeek was used for final edge-case testing and debugging of the project.
- ChatGPT was used for helping achieve a better understanding of OOP concepts, input parsing and building regexes in C++ (in which I wrote all of the regexes myself after I learned the logic behind), utilizing string utilities of C++ like istringstream and ostringstream; and finally, helping format my LaTeXproject report by making it fancier and removing minor grammatical errors.