Witcher Tracker Project Report (C++ Version)

Systems Programming – Spring 2025

Salih Can Erer, 2022400174

Beyza Nur Deniz, 2021400285

Contents

1 Introduction

The Witcher Tracker is a command-line based inventory and knowledge tracking system inspired by the Witcher universe, where the protagonist Geralt of Rivia manages ingredients, crafts potions, learns monster weaknesses, and engages in combat encounters. Originally developed in C, this project has now been re-implemented in **C++** as part of the CMPE230 Systems Programming course at Boğaziçi University.

The new version adopts an object-oriented approach that emphasizes modularity, encapsulation, and code reusability. The system processes three types of user input: *sentences* (which perform actions or update knowledge), *questions* (which query the internal state), and an *exit command* that terminates execution. All inputs must strictly conform to the grammar provided in the assignment specification.

The main goal of the C++ rewrite is to enhance maintainability and logical separation of concerns. This is achieved by organizing the logic into dedicated classes such as `Inventory`, `Potion`, `Monster`, and `Utils`, each responsible for a specific part of the system's functionality. By leveraging STL containers like `map`, `set`, and `vector`, the system maintains an efficient, readable, and scalable structure suitable for complex input handling.

This report documents the system's requirements, design principles, implementation details, and challenges encountered during development. It also highlights the differences and improvements over the original C implementation.

2 Problem Description

Geralt, the Witcher, begins his journey having lost all memory and possessions. The user is tasked with developing a command-line based simulation that helps Geralt recover his knowledge and rebuild his inventory through structured interactions. The system operates entirely through textual input and simulates Geralt's progression in alchemy, combat preparedness, and knowledge acquisition.

The C++ implementation emphasizes an object-oriented structure that processes a continuous stream of syntactically constrained input. Each line is interpreted as either an action (e.g., loot, trade, brew), a knowledge update (e.g., learning potion effects or monster weaknesses), a query (e.g., inventory checks), or a termination command. The system validates each line against a strict grammar and responds accordingly.

The core objectives of the system are:

- Accurately parse and execute valid user commands such as looting ingredients, trading trophies, brewing potions, learning new recipes, and encountering monsters.

- Maintain a dynamic, in-memory inventory of ingredients, potions, and monster trophies using STL containers like `std::map` and `std::set`.
- Store potion formulas and bestiary knowledge in expandable data structures to simulate Geralt's gradual recollection and learning process.
- Enforce precise syntactic rules as defined by a BNF-style grammar specification; all malformed input must trigger a clear `INVALID` message.
- Respond meaningfully and consistently to all queries regarding inventory contents, bestiary entries, and potion recipes.

The program accepts lines up to 1024 characters in length and continues execution in an interactive loop until the special `Exit` command is received. All operations are performed in-memory; persistent storage is not required. This structure supports rapid user feedback and testing while maintaining logical separation between parsing, validation, and execution.

3 Methodology

The Witcher Tracker system was developed with an object-oriented, modular design to ensure clarity, scalability, and maintainability. Unlike the earlier C version which relied on procedural logic and dynamic arrays, the C++ implementation leverages encapsulated classes, STL containers, and clearly separated responsibilities across components.

Input Handling and Parsing

User interaction is conducted exclusively via command-line input. Each line is initially parsed using regular expressions by the `Utils` module, which splits tokens and determines the command category: **sentence**, **question**, or **exit**.

`Utils::detect_type()` classifies each input, and then depending on its classification:

- Sentence-type inputs (e.g., `loot`, `trade`, `brew`) are routed to the appropriate `Inventory` handler.
- Question-type inputs (e.g., `Total ingredient ?`) are interpreted using `detect_question_type()` and responded to via query methods.
- Invalid inputs are immediately flagged and responded with `INVALID`.

Class Decomposition and Responsibilities

The core system functionality is decomposed into the following classes:

- **Inventory**: Stores all inventory state including ingredients, potions, trophies, and bestiary knowledge. Routes validated commands to internal methods.
- **Potion**: Stores potion recipes using a sorted `vector` of `<ingredient, quantity>` pairs.
- **Monster**: Maintains sets of known effective potions and signs for each monster.
- **Utils (namespace)**: Implements all grammar and syntax validation, including tokenization and rule checking.

Memory and Data Structures

Instead of manual memory management or dynamic arrays, the program uses:

- `std::map<std::string, int>` for storing quantities of ingredients, potions, and trophies.
- `std::map<std::string, Potion>` and `std::map<std::string, Monster>` for storing structured entries.
- `std::set<std::string>` to ensure uniqueness of signs/potions per monster.
- `std::vector<std::pair<std::string, int>>` to maintain ordered recipe ingredients.

This use of STL ensures consistent runtime complexity, automatic memory safety, and easy sorting/filtering operations.

Execution Pipeline

The execution flow is streamlined as follows:

1. `main.cpp` reads an input line and forwards it to `execute_line()`.
2. `Utils::split_line()` tokenizes the input and classifies it.
3. Based on the classification, a method of the `Inventory` class is called to execute the action or respond to a query.
4. Results (or `INVALID`) are printed to stdout and the loop continues until an `Exit` command or EOF is received.

Grammar Enforcement and Robustness

All syntactic constraints defined in the BNF-like grammar from the assignment are strictly enforced during parsing. This prevents invalid or malformed inputs from affecting internal state. Each handler in `Inventory` assumes pre-validated input and focuses purely on logic and state mutation.

This methodology enabled the creation of a robust, scalable C++ application that cleanly separates concerns, enforces strict grammar compliance, and mirrors game-like interactions in a clear, command-driven environment.

4 Design and Implementation

In this section, we document the structure and internal logic of our C++ implementation of the Witcher Tracker. We break down the responsibilities of each translation unit, highlight the role of every core class and data structure, and explain how each component contributes to parsing, validation, and game state management. Throughout, we provide representative code snippets (limited to under ten lines) to illustrate key behaviors, while referring to the complete and submitted source code for full context. All descriptions are grounded in the actual implementation and reflect the exact functionality we developed.

4.1 Utils Namespace: Lexical Analysis & Grammar Validation

Globals and Purpose

- `Utils::words` (`std::vector<std::string>`) and `Utils::word_count` (`int`) store the tokens of the current input line.
- All functions under `Utils` operate solely on these globals and do not modify any inventory state.
- Responsibilities:
  1. **Tokenization:** Split a raw input line into discrete tokens (words, commas, question marks).
  2. **Primitive Validators:** Check if a token is alphabetical or numeric.
  3. **Potion-Name Validator:** Ensure multiword potion names contain only single spaces and no digits.
  4. **Type Detection:** Classify a line as `Exit`, `Question`, or `Sentence`.
  5. **Sentence-Category Detection:** Determine which of the seven sentence forms (loot, trade, brew, sign knowledge, potion knowledge, potion recipe, encounter) the line matches, or report invalid.

6. **Question-Category Detection:** Determine which of the eight question forms (specific/all ingredient; specific/all potion; specific/all trophy; bestiary query; alchemy query) the line matches, or report invalid.

`split_line(line)` – Tokenization

```
1    void Utils::split_line(const std::string &line) {
2      words.clear();
3      std::regex pattern(R"([^\s,?]+|[?,])");
4      auto it = std::sregex_iterator(line.begin(), line.end(),
           pattern);
5      auto end = std::sregex_iterator();
6      for (; it != end; ++it) {
7        words.push_back(it->str());
8      }
9      word_count = words.size();
10   }
```

Listing 1: Excerpt from Utils.cpp (split_line)

Explanation:

- The regex `R"([^\s,?]+[?,])"`— matches either:
  - Sequences of one or more non-space, non-comma, non-question-mark characters `[^\s,?]+`,
  - A single comma or question-mark `[?,]`.
- Each match is pushed into `Utils::words`, and `Utils::word_count` is updated.
- Downstream validators rely on these tokens to enforce grammar.

`is_alphabetical(str)` & `is_integer(str)` – Primitive Validators

```
1    bool Utils::is_alphabetical(const std::string &str) {
2      for (char c : str) if (!isalpha(c)) return false;
3      return true;
4    }
5    bool Utils::is_integer(const std::string &str) {
6      for (char c : str) if (!isdigit(c)) return false;
7      return true;
8    }
```

Listing 2: Excerpt from Utils.cpp (is_alphabetical, is_integer)

Explanation:

- `is_alphabetical` returns true only if every character is a letter (A–Z or a–z).
- `is_integer` returns true only if every character is a digit (0–9).
- Used to enforce that quantities are positive integers and names contain only letters.

`is_valid_potion_name(line, start_index, end_index)` – Multiword Potion Name Validation

```
bool Utils::is_valid_potion_name(const std::string &line, int
    start_index, int end_index) {
  std::string first = words[start_index], last = words[
     end_index];
  int start_in_line = line.find(first);
  int end_in_line = line.find(last) + (int)last.size() - 1;
  // Ensure each token is alphabetical
  for (int i = start_index; i <= end_index; ++i) {
    if (!is_alphabetical(words[i])) return false;
  }
  // Ensure no double spaces in the raw substring
  for (int i = start_in_line; i < end_in_line; ++i) {
    if (line[i] == ' ' && line[i+1] == ' ') return false;
  }
  return true;
}
```

Listing 3: Excerpt from Utils.cpp (is_valid_potion_name)

Explanation:

- `start_index` and `end_index` define the token range for the candidate potion name.
- Locate the substring in the raw `line` by finding the first and last token positions.
- Verify each token is alphabetical.
- Scan the raw substring for consecutive spaces—reject if any are found.
- Enforces that multiword potion names have exactly one space between words and no digits.

`detect_type()` – High-Level Input Classification

```
1  int Utils::detect_type() {
2    if (word_count == 0) return -1;                    //
       Empty line
3    if (words[word_count - 1] == "?") return 1;        //
       Question
4    if (words[0] == "Exit" && word_count == 1) return 2; //
       Exit
5    return 0;                                          //
       Candidate sentence
6  }
```

Listing 4: Excerpt from Utils.cpp (detect_type)

Explanation:

- Returns `-1` if there are zero tokens (empty input).
- Returns `1` if the last token is "`?`" (a question).
- Returns `2` if the single token is "`Exit`" (exit command).
- Otherwise, returns `0`, meaning "candidate sentence."

`detect_sentence_type(line)` – Sentence Validation

- Returns an integer code in {0..6} corresponding to:
  1. 0: `loot` action ("`Geralt loots ...`")
  2. 1: `trade` action ("`Geralt trades ...for ...`")
  3. 2: `brew` action ("`Geralt brews ...`")
  4. 3: `sign knowledge` ("`Geralt learns <sign> sign is effective against <monster>`")
  5. 4: `potion knowledge` ("`Geralt learns <potion> potion is effective against <monster>`")
  6. 5: `potion recipe` ("`Geralt learns <potion> potion consists of <ingredients>`")
  7. 6: `encounter` ("`Geralt encounters a <monster>`")
- Returns `-1` if none of the above match (invalid sentence).
- Internally calls helper predicates:
  - `is_valid_loot()` – verifies `<qty> <ingredient>` pairs with commas.

  - `is_valid_trade()` – verifies `<qty> <monster> trophy` "for" `<qty> <ingredient>` pairs.
  - `is_valid_potion_name(line,2,word_count-1)` – ensures the potion name is valid for `brew`.
  - `is_valid_sign_knowledge()` – checks exactly eight tokens: `Geralt learns <sign> sign is effective against <monster>`.
  - `is_valid_potion_knowledge(curr_index)` – verifies the pattern after "`<potion> potion is effective against <monster>`."
  - `is_valid_potion_recipe(curr_index+3)` – verifies the ingredient list after "`<potion> potion consists of`."
  - `is_valid_encounter()` – checks exactly four tokens: `Geralt encounters a <monster>`.

`detect_question_type(line)` – Question Validation

- Returns an integer code in {0..7} corresponding to:
  1. 0: "`Total ingredient <name> ?`" – specific ingredient.
  2. 1: "`Total ingredient ?`" – all ingredients.
  3. 2: "`Total potion <name> ?`" – specific potion (handles multiword name).
  4. 3: "`Total potion ?`" – all potions.
  5. 4: "`Total trophy <name> ?`" – specific trophy.
  6. 5: "`Total trophy ?`" – all trophies.
  7. 6: "`What is effective against <monster> ?`" – bestiary query.
  8. 7: "`What is in <potion> ?`" – alchemy query (multiword potion).
- Returns `-1` if no pattern matches.
- Uses:
  - `is_alphabetical()` for single-word names.
  - `is_valid_potion_name(line,3,curr_index-1)` for multiword potion names in "`What is in`" queries.

4.2 Potion Class: Recipe Abstraction

Data Members

- `std::vector<std::pair<std::string,int>> ingredients;`
- Stores the set of ingredients (name and quantity) required for the potion.

Member Functions

```
1    std::vector<std::pair<std::string,int>> get_ingredients();
2    void set_ingredients(std::vector<std::pair<std::string,int>>
        new_ingredients);
```

Listing 5: Excerpt from Potion.h

Explanation:

- `get_ingredients()` sorts `ingredients` in descending quantity, then ascending name, and returns the sorted vector.
- `set_ingredients(...)` replaces the internal vector (called once when a new recipe is learned).

Implementation of `get_ingredients()`

```
1    std::vector<std::pair<std::string,int>> Potion::
        get_ingredients() {
2      std::sort(ingredients.begin(), ingredients.end(),
3      [](auto &a, auto &b) {
4        if (a.second == b.second) return a.first < b.first;
5        return a.second > b.second;
6      });
7      return ingredients;
8    }
```

Listing 6: Excerpt from Potion.cpp

Explanation:

- Uses `std::sort` with a lambda comparator:
  - If quantities are equal, compare names ascending.
  - Otherwise, sort by descending quantity.
- Returns a copy of the sorted vector. Callers iterate over this to print the recipe.

4.3 Monster Class: Bestiary Entry

Data Members

- `std::set<std::string> signs_against;`
- `std::set<std::string> potions_against;`
- Each set stores unique entries (alphabetically sorted) of signs and potions effective against this monster.

Member Functions

```
    std::set<std::string> get_signs();
    std::set<std::string> get_potions();
    void add_sign(std::string sign);
    void add_potion(std::string potion);
```

Listing 7: Excerpt from Monster.h

Explanation:

- `get_signs()` and `get_potions()` return copies of the respective sets.
- `add_sign(sign)` inserts `sign` into `signs_against`. If it already exists, insertion is a no-op.
- `add_potion(potion)` inserts `potion` into `potions_against`. Duplicate insertion has no effect.
- Using `std::set` ensures uniqueness and alphabetical ordering automatically.

Implementation of `add_sign` / `add_potion`

```
    void Monster::add_sign(std::string sign) {
      signs_against.insert(std::move(sign));
    }
    void Monster::add_potion(std::string potion) {
      potions_against.insert(std::move(potion));
    }
```

Listing 8: Excerpt from Monster.cpp

Explanation:

- Uses `std::move` to efficiently insert without copying.
- The underlying `std::set` organizes items in alphabetical order.

4.4 Inventory Class: Core State & Command Handlers

Header (Inventory.h) Data Members

```
1     std::map<std::string,int> ingredients;
2     std::map<std::string,int> trophies;
3     std::map<std::string,int> potion_counts;
4     std::map<std::string,Potion> potions;
5     std::map<std::string,Monster> monsters;
```

Listing 9: Excerpt from Inventory.h

Explanation:

- `ingredients` maps each ingredient name to its current quantity (sorted by name).
- `trophies` maps each monster name to the number of trophies owned.
- `potion_counts` maps each potion name to the inventory count.
- `potions` maps potion names to their `Potion` objects (recipes).
- `monsters` maps monster names to their `Monster` objects (bestiary info).
- All `std::map` operations (`operator[]`, `insert`, `erase`) run in $O(\log n)$.

Helper Functions (Inventory.h)

```
1     void add_ingredient(std::string name, int count);
2     void decrease_trophy(std::string name, int count);
3     void use_ingredient(std::string name, int count);
4     void use_one_potion_each(std::string monster_name);
```

Listing 10: Excerpt from Inventory.h

Explanation:

- `add_ingredient(name,count)` increments `ingredients[name]` by `count`.
- `decrease_trophy(name,count)` subtracts `count` from `trophies[name]`, erasing if result ≤ 0.
- `use_ingredient(name,count)` subtracts `count` from `ingredients[name]`, erasing if result ≤ 0.
- `use_one_potion_each(monster_name)` iterates over all known effective potions for that monster, decrementing `potion_counts[p]` by one if > 0, and erasing if zero.

`handle_loot()` – Adding Ingredients

```
void Inventory::handle_loot() {
  int idx = 2;
  while (idx < Utils::word_count - 1) {
    int cnt = std::stoi(Utils::words[idx]);
    std::string ing = Utils::words[idx + 1];
    add_ingredient(ing, cnt);
    idx += 3;
  }
  std::cout << "Alchemy␣ingredients␣obtained\n";
}
void Inventory::add_ingredient(std::string name, int count) {
  ingredients[name] += count; // default 0 if absent
}
```

Listing 11: Excerpt from Inventory.cpp (handle_loot)

Explanation:

- In a valid loot sentence, tokens appear as: `Geralt loots <qty> <ingredient> [, <qty> <ingredient>]*`.
- `idx = 2` skips the "`Geralt loots`" prefix.
- In the loop:
  1. Convert `words[idx]` to integer (`cnt`).
  2. `words[idx+1]` is the ingredient name (alphabetical).
  3. Call `add_ingredient()`, which increments the map entry (inserting if absent).
  4. Advance `idx` ← `idx`+3 to skip "`<qty> <ingredient> ,`" or "`<qty> <ingredient>`" if final.
- Finally, print the fixed "`Alchemy ingredients obtained`" message.

`handle_trade()` – Trading Trophies for Ingredients

```
void Inventory::handle_trade() {
  int idx = 2;
  std::map<std::string,int> trophies_to_trade;
  // Phase 1: parse trophy list
  while (true) {
    int cnt = std::stoi(Utils::words[idx]);
    std::string trophy = Utils::words[idx + 1];
    if (trophies[trophy] < cnt) {
```

```
9           std::cout << "Not enough trophies\n";
10          return;
11        }
12        trophies_to_trade[trophy] = cnt;
13        if (Utils::words[idx + 3] == "for") {
14          idx += 4;
15          break;
16        }
17        idx += 3;
18      }
19      // Phase 2: deduct trophies
20      for (auto &p : trophies_to_trade) {
21        decrease_trophy(p.first, p.second);
22      }
23      // Phase 3: add ingredients (same as handle_loot)
24      while (idx < Utils::word_count - 1) {
25        int cnt = std::stoi(Utils::words[idx]);
26        std::string ing = Utils::words[idx + 1];
27        add_ingredient(ing, cnt);
28        idx += 3;
29      }
30      std::cout << "Trade successful\n";
31    }
32    void Inventory::decrease_trophy(std::string name, int count)
        {
33      trophies[name] -= count;
34      if (trophies[name] <= 0) trophies.erase(name);
35    }
```

Listing 12: Excerpt from Inventory.cpp (handle_trade)

Explanation:

1. Skip "`Geralt trades`" (`idx = 2`).
2. **Phase 1:** Build `trophies_to_trade`:
   - Read `cnt = stoi(words[idx])`, `trophy = words[idx+1]`.
   - If `trophies[trophy] < cnt`, print "`Not enough trophies`" and return without changes.
   - Record `trophies_to_trade[trophy] = cnt`.
   - If next token (`words[idx+3]`) is "`for`", advance `idx += 4` to skip "`for`" and break.

- Otherwise, skip the comma/continue pattern via `idx += 3`.

3. **Phase 2:** Deduct trophies:
   - For each $(trophy, cnt)$ in `trophies_to_trade`, call `decrease_trophy(trophy,cnt)`, which subtracts and erases if ≤ 0.
4. **Phase 3:** Parse ingredient list identically to `handle_loot()`, calling `add_ingredient()` for each pair.
5. Finally, print "`Trade successful`."

`handle_brew()` – Brewing a Potion

```
void Inventory::handle_brew() {
  // Phase 1: reconstruct multiword potion name
  std::string potion_name;
  for (int i = 2; i <= Utils::word_count - 1; ++i) {
    potion_name += Utils::words[i];
    if (i < Utils::word_count - 1) potion_name += " ";
  }
  // Phase 2: check if formula exists
  if (potions.find(potion_name) == potions.end()) {
    std::cout << "No formula for " << potion_name << "\n";
    return;
  }
  auto needed = potions[potion_name].get_ingredients();
  // Phase 3: verify all ingredients are available
  for (auto &pr : needed) {
    if (ingredients[pr.first] < pr.second) {
      std::cout << "Not enough ingredients\n";
      return;
    }
  }
  // Phase 4: deduct ingredients
  for (auto &pr : needed) {
    use_ingredient(pr.first, pr.second);
  }
  // Phase 5: increment potion count
  potion_counts[potion_name]++;
  std::cout << "Alchemy item created: " << potion_name << "\n
     ";
}
void Inventory::use_ingredient(std::string name, int count) {
```

```
30        ingredients[name] -= count;
31        if (ingredients[name] <= 0) ingredients.erase(name);
32      }
```

Listing 13: Excerpt from Inventory.cpp (handle_brew)

Explanation:

- **Phase 1:** Concatenate tokens `words[2..end]` separated by spaces to form `potion_name`.
- **Phase 2:** If `potions.find(potion_name) == potions.end()`, the formula is unknown → print "`No formula for <potion>`" and return.
- **Phase 3:** Retrieve sorted vector `needed` = $potions[potion\_name].get\_ingredients()$. For each (ing, cnt), check `ingredients[ing] >= cnt`. If any fail, print "`Not enough ingredients`" and return.
- **Phase 4:** For each (ing, cnt), call `use_ingredient(ing,cnt)` which subtracts and erases if ≤ 0.
- **Phase 5:** Increment `potion_counts[potion_name]` and print "`Alchemy item created: <potion>`."

`handle_sign_knowledge()` – Learning a Sign's Effectiveness

```
1     void Inventory::handle_sign_knowledge() {
2       std::string sign_name = Utils::words[2];
3       std::string monster_name = Utils::words[7];
4       if (monsters.find(monster_name) == monsters.end()) {
5         std::cout << "New␣bestiary␣entry␣added:␣" << monster_name
              << "\n";
6         Monster m; m.add_sign(sign_name);
7         monsters[monster_name] = std::move(m);
8       } else {
9         auto signs = monsters[monster_name].get_signs();
10        if (signs.find(sign_name) == signs.end()) {
11          std::cout << "Bestiary␣entry␣updated:␣" << monster_name
                << "\n";
12          monsters[monster_name].add_sign(sign_name);
13        } else {
14          std::cout << "Already␣known␣effectiveness\n";
15        }
16      }
17    }
```

Listing 14: Excerpt from Inventory.cpp (handle_sign_knowledge)

Explanation:

1. Extract `sign_name = words[2]` and `monster_name = words[7]` from "`Geralt learns <sign> sign is effective against <monster>.`"
2. If `monsters.find(monster_name) == monsters.end()`, this monster is new:
   - Print "`New bestiary entry added: <monster>.`"
   - Create `Monster m`, call `m.add_sign(sign_name)`, and insert into `monsters[monster_name]`.
3. Otherwise, retrieve `signs = monsters[monster_name].get_signs()`. If `sign_name` is not present:
   - Print "`Bestiary entry updated: <monster>.`"
   - Call `monsters[monster_name].add_sign(sign_name)`.
4. If `sign_name` is already known, print "`Already known effectiveness`" (no state change).

`handle_potion_knowledge()` – Learning a Potion's Effectiveness

```
void Inventory::handle_potion_knowledge() {
  // Phase 1: reconstruct multiword potion name
  std::string potion_name;
  int idx = 2;
  while (Utils::words[idx] != "potion") {
    potion_name += Utils::words[idx];
    if (Utils::words[idx+1] != "potion") potion_name += "␣";
    idx++;
  }
  // Phase 2: monster name at idx+4
  std::string monster_name = Utils::words[idx+4];
  if (monsters.find(monster_name) == monsters.end()) {
    std::cout << "New␣bestiary␣entry␣added:␣" << monster_name
        << "\n";
    Monster m;
    m.add_potion(potion_name);
    monsters[monster_name] = std::move(m);
  } else {
    auto m_pots = monsters[monster_name].get_potions();
    if (m_pots.find(potion_name) == m_pots.end()) {
      std::cout << "Bestiary␣entry␣updated:␣" << monster_name
          << "\n";
      monsters[monster_name].add_potion(potion_name);
```

```
22          } else {
23            std::cout << "Already known effectiveness\n";
24          }
25        }
26      }
```

Listing 15: Excerpt from Inventory.cpp (handle_potion_knowledge)

Explanation:

- **Phase 1:** Build `potion_name` from tokens until the literal "`potion`" is reached. Insert a space between tokens except before "`potion`."
- **Phase 2:** The monster name appears four tokens after "`potion` is effective against."
- If this monster is new, print "`New bestiary entry added: <monster>`," create `Monster m`, call `m.add_potion(potion_name)`, and insert.
- Otherwise, retrieve existing potions set. If `potion_name` is not already known, add it, print "`Bestiary entry updated: <monster>`." If already present, print "`Already known effectiveness`."

`handle_potion_recipe()` – Learning a Potion Formula

```
1     void Inventory::handle_potion_recipe() {
2       // Phase 1: reconstruct multiword potion name
3       std::string potion_name;
4       int idx = 2;
5       while (Utils::words[idx] != "potion") {
6         potion_name += Utils::words[idx];
7         if (Utils::words[idx+1] != "potion") potion_name += " ";
8         idx++;
9       }
10      // If formula already known
11      if (!potions[potion_name].get_ingredients().empty()) {
12        std::cout << "Already known formula\n";
13        return;
14      }
15      // Phase 2: parse <ingredient_list> starting at idx+3
16      std::vector<std::pair<std::string,int>> formula;
17      idx += 3; // skip "potion consists of"
18      while (idx < Utils::word_count - 1) {
19        int cnt = std::stoi(Utils::words[idx]);
20        std::string ing = Utils::words[idx + 1];
21        formula.push_back({ing, cnt});
```

```
22          idx += 3;
23        }
24        // Phase 3: store new recipe
25        Potion p;
26        p.set_ingredients(std::move(formula));
27        potions[potion_name] = std::move(p);
28        std::cout << "New alchemy formula obtained: " <<
              potion_name << "\n";
29      }
```

Listing 16: Excerpt from Inventory.cpp (handle_potion_recipe)

Explanation:

1. **Phase 1:** Build the multiword `potion_name` until encountering the token "`potion`."
2. If `potions[potion_name].get_ingredients()` is nonempty, a formula already exists → print "`Already known formula`" and return.
3. **Phase 2:** Starting three tokens after "`potion consists of`," parse triples `<qty> <ingredient> ,` (or final `<qty> <ingredient>`) into `formula`.
4. **Phase 3:** Create a `Potion p`, call `p.set_ingredients(move(formula))`, and insert into `potions[potion_name]`. Print "`New alchemy formula obtained: <potion>`."

`handle_encounter()` – Monster Encounter Logic

```
1     void Inventory::handle_encounter() {
2       std::string monster_name = Utils::words[3];
3       // Phase 1: check knowledge
4       if (monsters.find(monster_name) == monsters.end() ||
5       (monsters[monster_name].get_signs().empty() &&
6       monsters[monster_name].get_potions().empty())) {
7         std::cout << "Geralt is unprepared and barely escapes
              with his life\n";
8         return;
9       }
10      // Phase 2: if no known signs, ensure at least one
            effective potion exists
11      if (monsters[monster_name].get_signs().empty()) {
12        bool found = false;
13        for (auto &p : monsters[monster_name].get_potions()) {
14          if (potion_counts[p] > 0) { found = true; break; }
15        }
16        if (!found) {
```

```
17            std::cout << "Geralt is unprepared and barely escapes
                 with his life\n";
18            return;
19          }
20          use_one_potion_each(monster_name);
21        } else {
22          // Known sign: defeat and still consume any known potions
23          use_one_potion_each(monster_name);
24        }
25        // Phase 3: award trophy
26        std::cout << "Geralt defeats " << monster_name << "\n";
27        trophies[monster_name]++;
28      }
```

Listing 17: Excerpt from Inventory.cpp (handle_encounter)

Explanation:

- `monster_name = words[3]` from "`Geralt encounters a <monster>`."
- **Phase 1:** If `monsters` has no entry for `monster_name` or both its sign and potion sets are empty, Geralt cannot win → print failure message and return.
- **Phase 2:** If no known signs, check if any known effective potion has count ¿ 0. If none, failure. Otherwise, call `use_one_potion_each(monster_name)` to consume one of each effective potion. If at least one sign is known, directly consume potions (if any) and succeed.
- **Phase 3:** Print victory and increment `trophies[monster_name]`.

`use_one_potion_each(monster_name)` – Helper to Consume Potions

```
1     void Inventory::use_one_potion_each(std::string monster_name)
          {
2       for (const auto &p : monsters[monster_name].get_potions())
           {
3         if (potion_counts[p] > 0) {
4           potion_counts[p]--;
5           if (potion_counts[p] == 0) potion_counts.erase(p);
6         }
7       }
8     }
```

Listing 18: Excerpt from Inventory.cpp (use_one_potion_each)

Explanation:

- Iterates over all known effective potions from `monsters[monster_name].get_potions()`.
- If `potion_counts[p] > 0`, decrement by one; if zero afterward, erase the map entry.

Query Methods – Inventory State Reporting

- `get_ingredient_count(name)` – Return `ingredients[name]` or 0 if absent.
- `print_ingredients()` – If `ingredients` is empty, print "`None.`" Otherwise, iterate over `ingredients` (sorted by name) and print "`<qty> <ingredient>[, ]`."
- `get_potion_count()` – Reconstruct multiword name from `words[2..word_count-2]` and return `potion_counts[name]` or 0 if absent.
- `print_potions()` – Similar to `print_ingredients()`, but iterates over `potion_counts`.
- `get_trophy_count(name)` – Return `trophies[name]` or 0 if absent.
- `print_trophies()` – Similar pattern over `trophies`.
- `print_monster_knowledge(monster_name)`:
  - If `monsters[monster_name]` is absent or both sets empty, print "`No knowledge of <monster>`."
  - Otherwise, merge `get_signs()` and `get_potions()` (two sorted sets) into a single `std::vector`, then print comma-separated.
- `print_potion_formula()`:
  - Reconstruct multiword potion name from `words[3..word_count-2]`.
  - If `potions[potion_name]` is absent or yields an empty `get_ingredients()`, print "`No formula for <potion>`."
  - Otherwise, retrieve sorted vector from `get_ingredients()` and print "`<qty> <ingredient>[, ]`."

4.5 main.cpp: REPL and Dispatch Logic

```
1  int main() {
2    std::string line;
3    while (true) {
4      std::cout << ">> ";
5      std::getline(std::cin, line);
```

```
6         if (std::cin.eof() || line == "Exit") break;
7         execute_line(line);
8       }
9       return 0;
10    }
11    void execute_line(const std::string &line) {
12      Utils::split_line(line);
13      Utils::word_count = (int)Utils::words.size();
14      int type = Utils::detect_type();
15      if (type == 2) exit(0);
16      if (type == 1) {
17        int qtype = Utils::detect_question\_type(line);
18        if (qtype == -1) {
19          std::cout << "INVALID\n";
20          return;
21        }
22        switch(qtype) {
23          case 0: { int cnt = inventory.get_ingredient_count(
                Utils::words[2]);
24            std::cout << cnt << "\n"; break; }
25          case 1: inventory.print_ingredients(); break;
26          case 2: { int cnt = inventory.get_potion_count();
27            std::cout << cnt << "\n"; break; }
28          case 3: inventory.print_potions(); break;
29          case 4: { int cnt = inventory.get_trophy_count(Utils::
                words[2]);
30            std::cout << cnt << "\n"; break; }
31          case 5: inventory.print_trophies(); break;
32          case 6: inventory.print_monster_knowledge(Utils::words
                [4]); break;
33          case 7: inventory.print_potion_formula(); break;
34          default: std::cout << "INVALID\n"; break;
35        }
36      } else if (type == 0) {
37        int stype = Utils::detect_sentence\_type(line);
38        if (stype == -1) {
39          std::cout << "INVALID\n";
40          return;
41        }
42        switch(stype) {
43          case 0: inventory.handle_loot(); break;
```

```
44            case 1: inventory.handle_trade(); break;
45            case 2: inventory.handle_brew(); break;
46            case 3: inventory.handle_sign_knowledge(); break;
47            case 4: inventory.handle_potion_knowledge(); break;
48            case 5: inventory.handle_potion_recipe(); break;
49            case 6: inventory.handle_encounter(); break;
50            default: std::cout << "INVALID\n"; break;
51          }
52        } else {
53          std::cout << "INVALID\n";
54        }
55      }
```

Listing 19: Excerpt from main.cpp

Explanation:

1. In `main()`, loop until EOF or "`Exit`" is read. Prompt with "`>> `", read a full line.
2. Call `execute_line(line)`:
   - `split_line(line)` tokenizes input; update `word_count`.
   - `detect_type()` returns {-1,0,1,2}:
     - `2`: exit immediately (`exit(0)`).
     - `1`: question → call `detect_question_type(line)`, get `qtype`.
       - If `qtype == -1`, print "`INVALID`" and return.
       - Else, switch on `qtype` to call the appropriate `Inventory` query.
     - `0`: candidate sentence → call `detect_sentence_type(line)`, get `stype`.
       - If `stype == -1`, print "`INVALID`" and return.
       - Else, switch on `stype` to invoke `Inventory` handler.
     - `-1`: empty input → print "`INVALID`."

4.6 Complexity Analysis

- **Tokenization (`split_line`):** $O(k)$ where k is the length of the input line (at most 1024).
- **Grammar Checks (`detect_*`):** Each runs in $O(t \cdot \ell)$ where t is the number of tokens and ℓ is average token length. In practice, t, ℓ are small.
- **Map/Set Operations:** $O(\log n)$ per insertion/lookup, where n is number of distinct names (ingredients, potions, monsters). Typically $n < 100$.

- **Sorting Recipe Ingredients:** $O(m \log m)$ with m = number of ingredients in a recipe (usually ≤ 10).
- **Merging Two Sets:** $O(a + b)$, where a, b number of known signs/potions for a monster (usually small).
- **Overall:** Each command completes in well under a millisecond, ensuring interactivity well within any time constraints.

4.7 Robustness & Error Handling

- **Centralized Grammar Checks:** All syntax validation is performed in `Utils`. `Inventory` handlers only receive pre-validated input.
- **Transactional State Updates:** Handlers verify prerequisites before mutating any state. E.g., `handle_trade()` checks trophy availability before deducting.
- **No Manual Memory Management:** All dynamic allocation is handled by STL containers, guaranteeing no memory leaks.
- **Consistent `INVALID` for Malformed Input:** `execute_line()` prints `INVALID` whenever any `detect_*` returns `-1`.
- **Edge Cases Covered:**
  - Missing or non-positive quantities → caught by `is_integer()` and `is_valid_*` predicates.
  - Double spaces in multiword names → caught by `is_valid_potion_name()`.
  - Missing keywords (e.g. "`for`" or "`trophy`") → caught by `is_valid_trade()`.
  - Queries without '?' or with incorrect token counts → caught by `detect_question_type()`.

5 Results

This section presents sample test cases and the corresponding outputs obtained from the implementation. The test cases are designed to cover different features of the system, such as sentence validation, potion brewing, trading mechanics, and inventory queries. Outputs are directly produced by the program and are included for clarity.

5.1 Custom All Correct Case

Input:

```
Geralt loots 5 Aether
Geralt loots 3 Quebrith, 2 Rebis
Geralt learns Cat potion consists of 2 Aether, 2 Quebrith
Geralt brews Cat
Geralt learns Aard sign is effective against Foglet
Geralt encounters a Foglet
Total ingredient?
Total potion Cat?
Total trophy Foglet?
Geralt trades 1 Foglet trophy for 5 Aether, 1 Quebrith
Total ingredient?
What is in Cat?
What is effective against Foglet?
```

Output:

```
>> Alchemy ingredients obtained
>> Alchemy ingredients obtained
>> New alchemy formula obtained: Cat
>> Alchemy item created: Cat
>> New bestiary entry added: Foglet
>> Geralt defeats Foglet
>> 3 Aether, 1 Quebrith, 2 Rebis
>> 1
>> 1
>> Trade successful
>> 8 Aether, 2 Quebrith, 2 Rebis
>> 2 Aether, 2 Quebrith
>> Aard
```

5.2 Custom All Incorrect Case

Input:

```
Geralt loot 5 Rebis
Geralt loots 0 Vitriol
```

```
Geralt loots , 2 Quebrith
Geralt trades Harpy trophy for 5 Vitriol
Geralt trades 1 Harpy trophy for Vitriol, 3 Rebis
Geralt learns Black Blood potion contains 2 Vitriol, 1 Rebis
Geralt learns potion Black Blood consists of 2 Vitriol, 1 Rebis
Geralt brews Black  Blood
Geralt brews
What is in Black     Blood?
Total potion Black  Blood ?
Geralt learns Igni is effective against Harpy
Geralt learns sign Igni is effective against Harpy
Geralt encounters Harpy
Exit now
```

Output:

```
>> INVALID
>> INVALID
>> INVALID
>> INVALID
>> INVALID
>> INVALID
>> INVALID
>> INVALID
>> INVALID
>> INVALID
>> INVALID
>> INVALID
>> INVALID
>> INVALID
>> INVALID
```

5.3 Custom Mixed Test Case

Input:

```
Geralt loots 3 Rebis, 4 Quebrith
Geralt loots 2 Vitriol
Geralt learns Swallow potion consists of 3 Vitriol, 2 Rebis
Geralt brews Swallow
```

```
Geralt learns Igni sign is effective against Wyvern
Geralt encounters a Wyvern
Geralt trades 1 Wyvern trophy for 5 Vitriol, 2 Rebis
Total ingredient?
Geralt brews Swallow
Geralt learns Swallow potion consists of 3 Vitriol, 2 Rebis
Geralt trades 1 Swamp trophy for 4 Quebrith
Geralt learns White Honey potion consists of 1 Vitriol, 1 Quebrith
Geralt brews White Honey
Geralt loots 0 Vitriol
Geralt brews Swallow
Geralt learns potion White Honey consists of 1 Vitriol, 1 Quebrith
Total potion Swallow?
What is in White Honey?
What is effective against Wyvern?
Geralt encounters Wyvern
Geralt learns Aard sign is effective against Wyvern
What is effective against Wyvern?
Geralt learns Igni sign is effective against
Geralt loots , 2 Rebis
Total trophy Swamp?
Total ingredient?
Geralt loots 5 Quebrith, 2 Vitriol
Geralt brews White Honey
```

Output:

```
>> Alchemy ingredients obtained
>> Alchemy ingredients obtained
>> New alchemy formula obtained: Swallow
>> Not enough ingredients
>> New bestiary entry added: Wyvern
>> Geralt defeats Wyvern
>> Trade successful
>> 4 Quebrith, 5 Rebis, 7 Vitriol
>> Alchemy item created: Swallow
>> Already known formula
>> Not enough trophies
>> New alchemy formula obtained: White Honey
```

```
>> Alchemy item created: White Honey
>> INVALID
>> Alchemy item created: Swallow
>> INVALID
>> 2
>> 1 Quebrith, 1 Vitriol
>> Igni
>> INVALID
>> Bestiary entry updated: Wyvern
>> Aard, Igni
>> INVALID
>> INVALID
>> 0
>> 3 Quebrith, 1 Rebis,
>> Alchemy ingredients obtained
>> Alchemy item created: White Honey
```

6 Discussion

The implemented Witcher Tracker in C++ achieves its primary objective of interpreting and executing structured commands in a domain-specific, Witcher-themed language. It supports a wide range of operations including ingredient looting, potion brewing, knowledge acquisition, trophy trading, and monster encounters, along with inventory and knowledge queries. Compared to the previous C-based version, this implementation offers clearer separation of concerns, better encapsulation of logic, and improved safety through the use of STL containers and class abstractions.

Performance

Thanks to the use of STL data structures like `std::map`, `std::set`, and `std::vector`, most operations—such as inserting or querying ingredients, trophies, potions, or monsters—execute in logarithmic or constant time, depending on the container. This results in smooth and responsive behavior, even as the inventory grows.

Recipe lookups and inventory updates during brewing or trading are performed via direct map access, ensuring $O(\log n)$ performance per item. Sorting operations, such as listing potion ingredients or outputting inventory contents in a defined order, are handled via `std::sort` or natural ordering in `std::map`, making them both efficient and concise. No manual memory management is required, which eliminates the risk of memory leaks

and improves reliability.

Limitations

Despite the improvements in structure and safety, the system still exhibits several limitations:

- **Error Message Granularity:** The parser reports all malformed lines uniformly as `INVALID`, without specifying the nature of the error. This can hinder debugging, especially in complex multiword inputs or subtle grammar mismatches.
- **No Undo Mechanism:** Once a command is executed—such as trading or brewing—it directly mutates the game state. There is no support for rolling back incorrect actions, which could be problematic in user testing scenarios.
- **Token Context Reliance:** Sentence type detection relies heavily on exact token positions (e.g., expecting "for" at a specific index during trades), which limits flexibility and reduces tolerance to minor formatting variations.
- **No Persistent Storage:** All data resides in memory during a single session. Upon exiting, the entire state is lost. While acceptable for this assignment, it restricts continuity across runs.

Possible Improvements

Several enhancements could make the system more robust, user-friendly, and extensible:

- **Detailed Error Reporting:** Modifying the validator functions to return descriptive error types or messages (e.g., "Expected 'for' keyword", "Unknown potion name", "Double space in name") would significantly improve user experience and debugging.
- **Input Suggestions:** When an input fails, the program could suggest the closest matching valid sentence, leveraging Levenshtein distance or rule-based heuristics.
- **Enhanced Interactive Shell:** Features like command history, auto-completion, or contextual help (e.g., "help loot", "list monsters") could be added to make the REPL loop more informative and beginner-friendly.
- **State Serialization:** Introducing file I/O to save and load the game state (ingredients, potions, trophies, bestiary) would allow session persistence and open the door for more advanced gameplay extensions.

7 Conclusion

In this project, we designed and implemented a modular, object-oriented system that simulates alchemical and bestiary operations in a fantasy world inspired by The Witcher universe. The system is fully interactive and text-driven, interpreting structured commands to manage an evolving internal game state. By organizing functionality into well-encapsulated classes—such as `Inventory`, `Potion`, `Monster`, and `Utils`—we achieved a clean separation of concerns and a high degree of code reusability.

Compared to our earlier C implementation, the transition to C++ significantly improved maintainability, clarity, and robustness. STL containers such as `map`, `set`, and `vector` were used extensively to manage dynamic game state without manual memory handling. Each command, whether it's looting, trading, brewing, or querying, is routed through a well-defined pipeline of validation and execution, ensuring both correctness and performance.

Throughout development, we prioritized strict input validation, consistent user feedback, and scalable design. The system handles a diverse set of user instructions and gracefully rejects malformed input using centralized grammar enforcement. While the program currently operates entirely in-memory and lacks persistent state or undo functionality, its architecture lays a strong foundation for future extensions.

Potential improvements include enriching error messages, supporting command suggestions, introducing save/load functionality, and enhancing the command-line interface with features like history or tab-completion. These enhancements would elevate the project from a robust interpreter to a fully-fledged interactive application.

Overall, this project provided a comprehensive exercise in applying object-oriented principles to a system-level application. It deepened our understanding of encapsulation, STL usage, command parsing, and interactive program flow—key skills in both academic and real-world software development contexts.

AI Assistants

AI assistants were used during this project strictly within the boundaries defined by the course policy. The use of such tools was limited to support and clarification purposes and did not replace independent implementation or understanding.

Throughout the development process, ChatGPT was employed in the following ways:

- **Debugging Assistance:** The assistant provided help in interpreting complex error messages and understanding subtle issues related to dynamic memory management and string manipulation in C. These insights were used to guide manual debugging efforts and code refinement.
- **Documentation Support:** ChatGPT was extensively used to enhance the clarity, coherence, and academic tone of the project's written report. While all content was originally authored by us, the assistant helped rephrase and formalize sections to ensure the language met academic standards.
- **Concept Clarification:** For reinforcing understanding of C programming concepts—such as pointer handling, and token parsing—ChatGPT was consulted as a learning aid. This helped solidify implementation decisions through clearer conceptual grounding.