# Project Report
# Systems Programming – Spring 2025

Ali Ayhan Gunder, 2021400219
Yunus Emre Ozturk, 2022400204

April 12, 2025

## Contents

## 1 Introduction

This project implements a C-based command interpreter to manage Geralt of Rivia's inventory, bestiary, and potion formulas. It simulates core mechanics from The Witcher universe by enforcing strict grammar for text-based commands, enabling operations like looting, trading, brewing, and learning. The interpreter maintains state through structured global arrays and demonstrates key programming principles including string parsing, modularity, and robust error handling.

Problem Statement: Geralt has lost his memory and needs a system to keep track of his alchemical ingredients, potions, monster trophies, knowledge of potion formulas, and bestiary information about monsters and how to defeat them.

Objectives:

- Develop a C program that can process three types of input: sentences (actions, knowledge, encounters), questions (queries about inventory, bestiary, alchemy), and an exit command.
- Implement an inventory system to store and manage alchemical ingredients, potions, and monster trophies.
- Create a bestiary to store information about monsters and effective ways to counter them.
- Develop an alchemy section to track potion formulas and brewing.
- Ensure the program adheres specific grammar rules for input processing and provides appropriate responses

Motivation: The motivation behind this project is to create a tool that simulates and manages various aspects of a Witcher's life, including gathering resources, learning about monsters, brewing potions, and tracking inventory.

## 2 Problem Description

The project requires the development of a C program that functions as an interpreter and inventory-event tracking system for Geralt of Rivia, a Witcher. The program must process three types of input: sentences, questions, and an exit command, each with specific formats and purposes.

Detailed Problem Description: The core problem is to simulate and manage Geralt's inventory, his knowledge, and his interactions within the game world. This involves:

- Inventory Management: Tracking alchemical ingredients, potions, and monster trophies.
- Knowledge Tracking: Storing information about potion formulas and effective methods (potions and signs) against monsters.
- Action Processing: Executing actions such as looting ingredients, trading trophies, brewing potions, learning new information, and encountering monsters.
- Query Responses: Answering questions about the contents of Geralt's inventory, bestiary, and alchemy knowledge.
- Input Validation: Ensuring that all input adheres to a defined grammar.

Given Constraints:

- Input Format: Input must follow the specified grammar rules. Invalid inputs should be handled by responding with "INVALID".
- Initial State: Geralt starts with no items, potions, or knowledge of potion formulas or monster weaknesses.
- Naming Conventions: Specific rules for naming entities like ingredients, monsters, potions, and signs.
- Output Format: The program should provide specific responses for each valid input, as demonstrated in the examples.
- Resource Constraints: There is a time limit of 30 seconds for program execution.

Expected Inputs and Outputs:

- **Inputs:**
  - Sentences: Instructions for actions (e.g., "Geralt loots 5 Rebis"), knowledge acquisition (e.g., "Geralt learns Black Blood potion consists of 3 Vitriol, 2 Rebis, 1 Quebrith"), or monster encounters (e.g., "Geralt encounters a Bruxa").
  - Questions: Queries about inventory (e.g., "Total ingredient Vitriol ?"), bestiary (e.g., "What is effective against Bruxa ?"), or alchemy (e.g., "What is in Black Blood ?").
  - Exit Command: "Exit" to terminate the program.
- **Outputs:**
  - Responses to actions: Confirmation messages (e.g., "Alchemy ingredients obtained", "Trade successful") or error messages (e.g., "Not enough trophies", "No formula for Black Blood").
  - Answers to questions: Information about inventory (e.g., "10 Vitriol", "None"), bestiary (e.g., "Black Blood, Yrden", "No knowledge of Wraith"), or alchemy (e.g., "3 Vitriol, 2 Rebis, 1 Quebrith", "No formula for Full Moon").
  - Error message: "INVALID" for any input that does not conform to the defined grammar.

## 3 Methodology

### 1. Data Structures

The code defines several struct types to represent the game entities:

- Ingredient: Stores the name and quantity of an ingredient.
- Potion: Stores the name and quantity of a potion.
- Trophy: Stores the monster name and quantity of a trophy.

- MonsterEntry: Stores a monster's name and the potions and signs effective against it.
- PotionFormula: Stores the potion's name and the ingredients required to brew it.

The code also declares global arrays to store collections of these entities: `ingredients_inventory`, `potions_inventory`, `trophies_inventory`, `bestiary`, and `formulas`.

## 2. Helper Functions

The code includes several helper functions to manage data and process input:

- `trim()`: Removes leading and trailing whitespace from a string.
- `remove_question_mark()`: Removes the trailing question mark from a query string.
- `add_ingredient()`, `get_ingredient_quantity()`, `subtract_ingredient()`: Functions to manage the ingredient inventory.
- `add_potion()`, `get_potion_quantity()`: Functions to manage the potion inventory.
- `add_trophy()`, `get_trophy_quantity()`, `remove_trophy()`: Functions to manage the trophy inventory.
- `parse_ingredient_list()`: Parses a string of ingredients and their quantities.
- `find_formula_index()`, `add_formula()`: Functions to manage potion formulas.
- `find_bestiary_index()`, `add_effectiveness()`: Functions to manage the bestiary.
- `cmp_ingredient()`, `cmp_formula()`: Comparison functions for sorting.

## 3. Command Processing

The `execute_line()` function is the core of the program. It takes a line of input, parses it, and executes the corresponding action or query.

- It uses `strncmp()` and `strcmp()` to identify the command type.
- It calls helper functions to perform actions like adding ingredients, brewing potions, trading trophies, and adding bestiary entries.
- It uses `sscanf()` and `strtok()` to parse the input string and extract data.
- It calls helper functions to answer queries about the inventory, bestiary, and alchemy.
- It uses `qsort()` to sort ingredients and potion formulas when necessary.

### 4. Input/Output

- The `main()` function handles the main input/output loop.
- It uses `fgets()` to read input from the user.
- It calls `execute_line()` to process the input.
- It uses `printf()` to display output to the user.

### 5. Error Handling

- The code returns `-1` from `execute_line()` to indicate an invalid command.
- The `main()` function prints "INVALID" when it receives `-1`.
- Helper functions like `parse_ingredient_list()` also return error codes (e.g., `-1`) to indicate invalid input.

Additional Notes:

- The code uses global arrays to store the system's state. While this is simple, it might be less organized for larger projects.
- The code assumes a maximum number of items, potions, trophies, etc. (e.g., `MAX_INGREDIENTS`).
- String manipulation is done using `strcpy()`, `strncpy()`, `strcmp()`, `strstr()`, and `strtok()`.

# 4 Implementation

## 4.1 4.1 Code Structure

The code is organized into modular components to ensure clarity, maintainability, and separation of concerns. It is implemented entirely in C using a single file structure and leverages global state management through arrays and structured types. Below are the primary components:

- Data Models: Defined using `typedef struct`, the key data types include Ingredient, Potion, Trophy, MonsterEntry, and PotionFormula. Each holds relevant fields such as names, quantities, and associated effects.
- Global Arrays and Counters: Each category of data is stored in a fixed-size global array (e.g., `ingredients_inventory[]`, `bestiary[]`) with associated counters (e.g., `ingredient_count`) to track usage.
- Inventory Management Functions:
  - `add_ingredient()`, `get_ingredient_quantity()`, and `subtract_ingredient()` manage ingredient data.

  - add_potion(), get_potion_quantity() manage potion data.
  - add_trophy(), remove_trophy(), get_trophy_quantity() manage trophy data.

- Alchemy and Formula Logic:
  - add_formula() and find_formula_index() manage known potion recipes.
  - parse_ingredient_list() validates and parses user input strings into usable ingredient arrays.

- Bestiary System:
  - add_effectiveness() and find_bestiary_index() store and retrieve effectiveness data for monsters.

- Command Parsing and Execution:
  - execute_line() acts as the interpreter, matching strings against command patterns (e.g., Geralt loots, Geralt brews) and executing corresponding logic.
  - Each command is verified for syntax and semantic correctness; invalid commands return -1 to signal errors.

- Sorting and Output:
  - Comparators such as cmp_ingredient(), cmp_ingredient_desc() allow alphabetic or quantity-based sorting.
  - qsort() is used to sort lists prior to display for user queries.

- User Interaction Loop:
  - The main() function handles user input with prompts and ends execution on the keyword Exit.

## 4.2 4.2 Sample Code

Below are several key functions from the implementation.

**Command Execution** (execute_line)

```c
int execute_line(const char *line) {
    char cmd[1024];
    strcpy(cmd, line);

    size_t len = strlen(cmd);
    if (len > 0 && cmd[len - 1] == '\n') {
        cmd[len - 1] = '\0';
    }

    if (strncmp(cmd, "Geralt loots ", 13) == 0) {
        const char *rest = cmd + 13;
        Ingredient arr[10];
        int cnt = parse_ingredient_list(rest, arr);
        if (cnt < 0) return -1;
        for (int i = 0; i < cnt; i++) {
            add_ingredient(arr[i].name, arr[i].quantity);
        }
        printf("Alchemy ingredients obtained\n");
        return 0;
    }

    /* ... other action handlers ... */

    return -1;
}
```

**Parsing an Ingredient List (parse_ingredient_list)**

```
int parse_ingredient_list(const char *str, Ingredient *arr) {
    char buf[256];
    strncpy(buf, str, sizeof(buf));
    buf[sizeof(buf)-1] = '\0';
    char *tok = strtok(buf, ",");
    int count = 0;
    while (tok && count < MAX_INGS) {
        int qty;
        char name[64];
        if (sscanf(tok, "%d %63s", &qty, name) != 2) return -1;
        arr[count].quantity = qty;
        strcpy(arr[count].name, name);
        count++;
        tok = strtok(NULL, ",");
    }
    return count;
}
```

**Managing Potion Formulas**

```
int find_formula_index(const char *name) {
    for (int i = 0; i < formula_count; i++) {
        if (strcmp(formulas[i].name, name) == 0) return i;
    }
    return -1;
}

void add_formula(const char *name, Ingredient *ings, int count) {
    int idx = find_formula_index(name);
    if (idx < 0) idx = formula_count++;
    strcpy(formulas[idx].name, name);
    formulas[idx].ingredient_count = count;
    for (int i = 0; i < count; i++) {
        formulas[idx].ingredients[i] = ings[i];
    }
}
```

**Recording Bestiary Effects (`add_effectiveness`)**

```c
void add_effectiveness(const char *monster, const char *effect) {
    int idx = find_bestiary_index(monster);
    if (idx < 0) idx = add_bestiary_entry(monster);
    strcpy(bestiary[idx].effects[bestiary[idx].effect_count++],
       effect);
}
```

**Handling a Query (e.g., Total Ingredient)**

```c
if (strncmp(cmd, "Total ingredient ", 17) == 0) {
    char name[64];
    sscanf(cmd + 17, "%63s", name);
    int qty = get_ingredient_quantity(name);
    if (qty > 0)
        printf("%d %s\n", qty, name);
    else
        printf("None\n");
    return 0;
}
```

## 5 Results

Tested with sample input commands. Successful cases output expected confirmation messages; invalid input returns "INVALID". The grammar enforcement and structured output meet the project specifications. I added two more testcase to given testcase and achive 100 percent succes in grader after some corrections.

## 6 Discussion

The interpreter meets its core goals and efficiently maintains state. However, the use of fixed-size arrays limits scalability. Future improvements could include persistent storage, dynamic data structures, and support for more advanced grammar. Additionally, using a wider variety of data structures may enhance performance and reduce runtime. Nonetheless, the current implementation is sufficient for programs with small input sizes.

## 7 Conclusion

The Witcher Tracker project demonstrates the effective use of C programming for building a text-based, stateful interpreter that faithfully models key aspects of Geralt's journey. By combining a clean, modular design with rigorous input validation and error handling, the system not only meets all functional requirements—inventory management, bestiary tracking, alchemical formula processing, and query answering—but also provides clear, user-friendly feedback in response to both valid and invalid commands.

Throughout development, we adhered to strict grammar rules and leveraged a suite of helper functions to encapsulate low-level details, resulting in code that is easy to read, maintain, and extend. The use of fixed-size global arrays simplified state management and ensured predictable performance under the 30-second execution constraint, while sorting routines and formatted output guaranteed consistent, alphabetically ordered listings when queried.

Looking ahead, several avenues for enhancement present themselves. Integrating dynamic data structures (e.g., linked lists or hash tables) would remove hard limits on the number of ingredients, potions, and trophies. Persistent storage—through file I/O or a lightweight database could allow Geralt's progress to be saved and resumed across sessions.

In conclusion, the Witcher Tracker achieves its goal of simulating core Witcher mechanics in C, balancing simplicity and robustness. The project serves as a solid foundation for further experimentation with language parsing, data persistence, and user interface design bringing us one step closer to a fully immersive text-based Witcher experience.

## References

- Lecture - Problem Session Slides
- Q/A messages in Piazza
- Ritchie, Dennis M., Brian W. Kernighan, and Michael E. Lesk. The C programming language. Englewood Cliffs: Prentice Hall, 1988

## AI Assistants

AI tools such as ChatGPT were used for the following:

- Clarifying C programming concepts
- Debugging support for code
- Testcase Generation for testing with grader
- Grammar and spelling correction in documentation
- Enriching latex syntax and format

All implementation decisions and coding were performed by the project authors.