Witcher Tracker Project Report

Systems Programming – Spring 2025

Salih Can Erer, 2022400174

Beyza Nur Deniz, 2021400285

Contents

1 Introduction

The *Witcher Tracker* project is an inventory and interaction simulation system developed in the C programming language. Designed as part of the CMPE230 Systems Programming course, this project introduces students to fundamental systems-level concepts such as dynamic memory management, modular design, parsing user input, and simulating real-time state changes through a command-based interpreter.

Inspired by the rich world of the Witcher universe, the system enables tracking of alchemical ingredients, potion formulas, combat knowledge, and monster encounters for the main character, Geralt of Rivia. Each interaction mimics realistic gameplay elements: gathering loot, learning potion recipes, brewing potions, and defeating monsters based on prior knowledge.

This project not only reinforces programming with pointers, structs, and file organization in C, but also emphasizes robust input handling, memory safety, and scalability through modular design patterns. The overall goal is to implement a functional, extensible, and user-friendly simulator that adheres to a predefined grammar and responds accurately to user queries and actions.

2 Problem Description

Geralt, the Witcher, has lost his memory and all his possessions. The user is tasked with developing a text-based simulation that helps Geralt rebuild his knowledge and inventory through command-line interactions. The system must process a stream of structured commands and questions related to Geralt's activities, inventory, and knowledge base.

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 dynamically allocated inventory of ingredients, potions, and monster trophies, allowing updates and queries at runtime.
- Store potion formulas and monster-specific knowledge in expandable data structures to simulate Geralt's learning process.
- Enforce strict syntactic rules as defined in a custom Backus–Naur Form (BNF)-like grammar specification, rejecting all malformed input with a clear `INVALID` message.
- Provide meaningful and formatted responses to user queries related to current inventory state, bestiary knowledge, and potion formulae.

The input is constrained to a maximum of 1024 characters per line. Each line must follow the defined grammar and semantics to be accepted. The program should remain interactive until it receives the `Exit` command. No persistent storage is required; all operations are performed in-memory during execution.

3 Methodology

The system was designed with a modular and extensible architecture to ensure clarity, maintainability, and scalability. Each major functionality of the system was encapsulated within a separate source file, allowing us to isolate concerns and reduce coupling between modules.

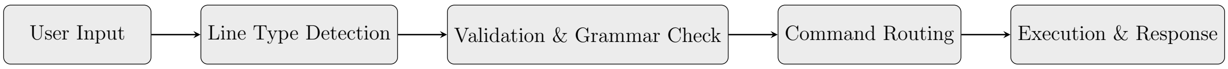
All user interactions are provided via command-line input. To process these, the system first classifies each line as either a **sentence**, a **question**, or an **exit command**. This categorization is performed by lexical analysis, implemented in the `type_detection.c` module. Once a line is classified, the program routes it to the appropriate handler for further validation and execution.

Sentences are further categorized into five types: *loot*, *trade*, *brew*, *learn*, and *encounter*. Each of these is handled by a dedicated function implemented in `sentence_handle.c`, while inventory queries and knowledge-based questions are processed by the `question_handle.c` module.

The data model was defined using multiple structured types. The core inventory entities—ingredients, potions, and trophies—are stored in dynamically allocated arrays, each governed by a corresponding capacity variable. More complex structures such as monsters and potion formulas utilize embedded arrays of other types (e.g., a monster holds arrays of effective signs and potions), allowing hierarchical information storage.

To efficiently manage memory and ensure safe operations during runtime, dynamic arrays are resized using a **doubling strategy**. When a data array reaches capacity, a new array of double size is allocated using `realloc`, and the data is preserved while expanding the available space. All such operations are encapsulated in `capacity_ensuring.c`.

This architecture allowed us to create a clear pipeline from input parsing to action execution, while enforcing the strict grammar rules defined in the project specification. Invalid inputs are promptly rejected with descriptive messages, ensuring robust handling of all edge cases without compromising user experience.



4 Implementation

The implementation is structured across multiple source files, each encapsulating a distinct subsystem of the application. This modular decomposition enables clear separation of responsibilities, testability, and easier debugging.

4.1 globals.h

The `globals.h` file defines the foundational data structures, enumerations, and global state variables used throughout the system. It also acts as a unifying interface between all modules, exposing the function declarations and shared types required for cross-file communication.

Line Categorization. To classify user input lines, the file defines the following enumerations:

- `LineType`: Indicates whether an input is a `SENTENCE`, a `QUESTION`, or an `EXIT` command.
- `Sentence`: Represents sentence-level commands issued by Geralt. These are: `LOOT`, `TRADE`, `BREW`, `LEARN`, and `ENCOUNTER`.
- `Question`: Represents query commands, either targeting specific or full listings of ingredients, potions, trophies, monsters, or potion formulas.

Core Entity Structures. The following `structs` model the project objects and their states:

- `Ingredient`, `Potion`, and `Trophy`: Each consists of a name and an associated quantity.
- `Sign`: Represents a magical sign effective against certain monsters.
- `Monster`: Composed of dynamic arrays of signs and potions, each with capacity and count fields.
- `PotionFormula`: Represents a learned potion recipe, containing a dynamic list of required ingredients.

```c
typedef struct {
  char name[MAX_WORD_LEN];
  int quantity;
} Ingredient;
```

Listing 1: Struct for an Ingredient

Dynamic Memory Management. To manage dynamic resizing, each entity type has an associated capacity variable and a last-used index. These variables are declared as global to allow for seamless access across modules:

- `ingredient_capacity`, `last_added_ingredient_index`, etc., are used to govern array resizing.
- Resizing follows a doubling strategy, handled in `capacity_ensuring.c`.

Global Variables. All entity arrays are declared as global pointers:

- `Ingredient *ingredients;`
- `Potion *potions;`
- `Trophy *trophies;`
- `PotionFormula *formulas;`
- `Monster *monsters;`
- `Sign *signs;`

These arrays are allocated and grown at runtime, depending on the number of entities encountered or learned by Geralt. Their usage requires careful memory handling, which is enforced via encapsulated capacity-check functions.

Function Declarations. The header also includes declarations for all utility, sentence, question, and parsing-related functions. This includes:

- Input sanitation (`remove_trailing_newline`, `split_into_words`, etc.)
- Type detection (`detect_type`, `detect_sentence`, etc.)
- Sentence execution (`handle_encounter`, `trade`, `brew_potion`, etc.)
- Query handling (`handle_all_trophies_query`, etc.)
- Capacity growth functions (`ensure_ingredient_capacity`, etc.)

This centralized declaration model enhances modularity and compile-time safety. All other source files rely on this header to ensure type consistency and maintain a unified global state.

4.2 type_detections.c

The `type_detections.c` module is responsible for the lexical classification of user input. It provides a lightweight parsing layer that determines the overall category of the input—sentence, question, or exit command—and further refines the classification into more specific subtypes.

This early detection step plays a vital role in the program's control flow, as it guides the main input dispatcher (`execute_line`) to route the input to the appropriate handling function. All parsing is done via pure functions with no side effects or memory modifications.

Detecting Line Type. The function `detect_type` determines whether the input is a sentence, a question, or an exit command.

- Returns `EXIT` if the input is a single word: `"Exit"`.
- Returns `QUESTION` if the last word ends with a `?`.
- Returns `SENTENCE` otherwise.

```c
if (word_count == 1 && strcmp(words[0], "Exit") == 0)
return EXIT;

char *last_word = words[word_count - 1];
if (last_word[strlen(last_word) - 1] == '?')
return QUESTION;
```

Listing 2: Exit and Question Detection

Detecting Sentence Type. Once a line is classified as a sentence, the `detect_sentence` function attempts to determine its specific category. It inspects the second word of the input and matches it against known sentence verbs:

- `loots` → `LOOT`
- `trades` → `TRADE`
- `brews` → `BREW`
- `learns` → `LEARN`
- `encounters` → `ENCOUNTER`

If the verb is unrecognized, the function returns `-1`, indicating an invalid sentence.

```c
if (strcmp(words[1], "brews") == 0)
return BREW;
```

Listing 3: Detecting the Sentence Verb

Detecting Question Type. The detect_question function contains the most complex classification logic. It parses questions of the form:

- Total ingredient Rebis? → INGREDIENT
- Total potion Cat? → POTION
- Total trophy Drowner? → TROPHY
- Total ingredient? → ALL_INGREDIENTS
- What is effective against Ghoul? → MONSTER
- What is in Cat potion? → POTION_FORMULA

The function first ensures that the last word is a question mark, then pattern-matches the initial keywords and checks alphabetic consistency in the relevant segments.

```c
if (strcmp(words[0], "What") == 0 &&
strcmp(words[1], "is") == 0 &&
strcmp(words[2], "effective") == 0 &&
strcmp(words[3], "against") == 0)
{
  return MONSTER;
}
```

Listing 4: Monster Question Detection

Each question type is hardcoded and mutually exclusive. This ensures full coverage without ambiguity.

Design Considerations. This modular parsing architecture makes the system highly readable and extendable. Adding new sentence or question types only requires updating this file and the associated handling logic, with minimal changes to the dispatcher.

All detection functions operate without modifying global state, maintaining strict separation of concerns and enabling unit testing.

4.3 utils.c

This module provides core utility routines for input preprocessing, word parsing, validation, and sorting. These functions are stateless and operate on primitive data types or shared data structures. As such, they are used widely throughout the system and should be defined before any high-level logic (e.g., sentence handling or main program flow).

Input Sanitization

- `remove_trailing_newline(char *line)`
  Removes the final `\n` character, if present. Called immediately after reading input from `fgets`.

- `remove_trailing_spaces(char *line)`
  Cleans up redundant whitespace (spaces/tabs) at the end of a string.

Tokenization and Parsing

- `split_into_words(char *line, char words[][MAX_WORD_LEN])`
  Tokenizes input based on space, comma, and question mark. Returns the number of tokens. It ensures:
  - Delimiters (',' and '?') are treated as separate tokens.
  - Repeated spaces are collapsed.

Validation Helpers

- `is_alphabetic_custom(const char *token)`
  Verifies the token contains only letters.

- `is_digit_custom(const char *token)`
  Confirms that the token is a positive integer without leading zeros.

- `is_valid_potion_name_spacing(const char *line, const char *name)`
  Ensures that multi-word potion names don't have consecutive spaces between words in the input string.

Custom Comparators for Sorting

- `cmpIngredient`, `cmpPotion`, `cmpTrophy`
  Used to sort respective arrays alphabetically (via `strcmp`).

- `cmp`
  Generic comparator for arrays of `char*` strings. Used e.g., when sorting lists of signs or potions related to monsters.

- `cmpForRecipe`
  Sorts ingredients in decreasing order of quantity; ties are broken alphabetically. Used in potion recipe queries.

Reflection. These functions allow decoupling of low-level operations from high-level logic, promoting reusability, clarity, and testing ease. Their modular nature enables clean abstraction between raw text parsing and semantic interpretation.

4.4 `capacity_ensuring.c`

This module ensures that all dynamically allocated arrays in the system (such as ingredients, potions, monsters, etc.) can grow as needed. Each 'ensure_*_capacity' function checks if the associated array has reached its limit, and if so, reallocates memory with double the previous capacity.

This approach is central to avoiding segmentation faults or undefined behavior due to out-of-bound writes in dynamically growing collections.

Global Arrays

The following functions handle capacity management for the global arrays:

- `ensure_ingredient_capacity()` — Resizes the `ingredients` array and updates `ingredient_capacity`.
- `ensure_potion_capacity()` — Resizes the `potions` array and updates `potion_capacity`.
- `ensure_trophy_capacity()` — Resizes the `trophies` array and updates `trophy_capacity`.
- `ensure_monster_capacity()` — Resizes the `monsters` array and updates `monster_capacity`.
- `ensure_formula_capacity()` — Resizes the `formulas` array and updates `formula_capacity`.
- `ensure_sign_capacity()` — Resizes the `signs` array and updates `sign_capacity`.

Each of these functions follows the same doubling strategy:

1. Checks whether the current index + 1 exceeds capacity.
2. If so, doubles the capacity.
3. Reallocates memory using `realloc`.

Monster Subarrays

Some arrays are nested within the `Monster` structure, and require specialized handling:

- `ensure_monster_sign_capacity(Monster *m)` — Ensures capacity for the `signs` array inside a specific `Monster`.
- `ensure_monster_potion_capacity(Monster *m)` — Ensures capacity for the `potions` array inside a specific `Monster`.

These functions double the local capacities (`m->sign_capacity`, `m->potion_capacity`) and use `realloc` to expand the subarrays, preserving existing data.

Design Rationale. This module abstracts away all raw memory management from the high-level logic. Instead of duplicating `realloc` calls throughout the program, capacity checks are centralized in these utility functions — a major win for readability, maintainability, and future extensibility.

4.5 `main.c`

The `main.c` module acts as the central control unit of the system. It is responsible for initializing the global state, handling the input loop, and orchestrating the full processing pipeline: from input parsing to command execution. The core logic is centered around the `execute_line` function, which delegates input to specific subsystems based on its classification.

Global State Initialization. The program maintains dynamic arrays for all runtime entities—ingredients, potions, trophies, potion formulas, monsters, and magical signs. Their capacities and last-used indices are globally declared and initialized as follows:

```c
int last_added_ingredient_index = -1;
int ingredient_capacity = 2;
Ingredient *ingredients = malloc(sizeof(Ingredient) *
    ingredient_capacity);
```

Listing 5: Initial Global Setup

This pattern is repeated for each data type, allowing flexible and scalable memory usage.

Input Handling Loop. Inside the `main` function, the program enters an infinite loop and waits for user input:

```c
while (1) {
  printf(">>␣");
  fflush(stdout);
  if (!fgets(line, sizeof(line), stdin))
  break;
  execute_line(line);
}
```

Listing 6: REPL Loop for Input Handling

Execution Pipeline. The `execute_line` function is the heart of command processing. It executes the following steps:

1. **Sanitization:** Removes trailing newline and spaces.
2. **Tokenization:** Splits input into words and determines its `LineType`.
3. **Dispatch:** Based on the type, the function redirects to sentence, question, or exit handlers.

Sentence Processing. If the line is a sentence (`SENTENCE`), the system:

- Verifies the sentence starts with `"Geralt"` and adheres to the grammar.
- Detects the sentence type (`LOOT`, `TRADE`, etc.).
- Validates the sentence using specific functions (e.g., `is_valid_ingredient_sentence`).
- Executes the appropriate logic (e.g., call to `add_ingredient` or `handle_encounter`).

Each sentence type is handled in its dedicated block, with dynamic memory resizing when capacity limits are reached. Example:

```c
if (last_added_ingredient_index + 1 >= ingredient_capacity) {
  ingredient_capacity *= 2;
  ingredients = realloc(ingredients, ingredient_capacity *
     sizeof(Ingredient));
}
add_ingredient(ingredients, ingredient_name, quantity);
```

Listing 7: Doubling Capacity and Adding Ingredient

Question Processing. If the input is a QUESTION, the program detects the question type and delegates to the appropriate query function, such as:

- handle_specific_ingredient_query
- handle_all_trophies_query
- handle_monster_query

Each handler retrieves and formats the relevant information based on current state.

Memory Deallocation. At the end of execution, all dynamically allocated memory is freed in a disciplined manner:

```
free(ingredients);
free(potions);
free(trophies);
free(formulas);
free(monsters);
free(signs);
```

Listing 8: Memory Cleanup

Design Impact. This design isolates low-level memory operations to the main control file, while sentence- and question-specific logic is modularized into separate source files. This architecture results in:

- Clear separation of responsibilities
- Minimal coupling between modules
- Easy testing and extension of functionality

The main.c file thus acts not only as the driver, but also as a guardian of the overall state integrity and memory safety.

4.6 sentence_handle.c

The sentence_handle.c file processes all command sentences that affect Geralt's state, including inventory manipulation, learning actions, potion creation, and monster encounters. Each sentence type has a distinct parsing logic and is handled by a dedicated function, with rigorous validation steps to ensure syntactic correctness.

4.6.1 LOOT: Gathering Alchemy Ingredients

The `LOOT` command allows Geralt to collect various alchemical ingredients, which are essential for potion brewing. Sentences of this form are parsed and processed in `sentence_handle.c` via a two-stage process: syntactic validation and semantic state update.

Validation. The sentence must conform to a grammar pattern resembling:

```
Geralt loots <quantity> <ingredient>, ...
```

This format is verified by the function `is_valid_ingredient_sentence`, which performs the following steps:

- Skips the initial two words (`Geralt loots`).
- Iterates through each pair of tokens, checking that:
  - Quantities are non-zero positive integers without leading zeros.
  - Ingredient names are strictly alphabetic.
  - Every pair is separated by a comma, and no trailing comma exists.
- If any rule is violated, the input is rejected as `INVALID`.

```c
if (!is_digit_custom(words[curr_index]))
return FALSE;
if (!is_alphabetic_custom(words[curr_index + 1]))
return FALSE;
if (strcmp(words[curr_index], ",") != 0)
return FALSE;
```

Listing 9: Validation of Loot Sentence

Execution. Once validated, the sentence is processed to update Geralt's inventory. The function `add_ingredient` is responsible for inserting or updating an ingredient entry. If the ingredient already exists, its quantity is increased. Otherwise, the array is expanded if needed and the new ingredient is appended.

```c
for (int i = 0; i <= last_added_ingredient_index; i++) {
  if (strcmp(ingredients[i].name, name) == 0) {
    ingredients[i].quantity += quantity;
    return;
  }
}
// Not found: add new entry
ensure_ingredient_capacity();
last_added_ingredient_index++;
strcpy(ingredients[last_added_ingredient_index].name, name);
ingredients[last_added_ingredient_index].quantity = quantity;
```

Listing 10: Adding an Ingredient or Updating Its Quantity

The function `ensure_ingredient_capacity` doubles the capacity of the ingredients array when needed. All operations are memory-safe and maintain array consistency.

Example. Input:

```
Geralt loots 2 DrownerBrain, 3 Rebis
```

Results in the inventory being updated to:

- `DrownerBrain: 2`
- `Rebis: 3`

If an ingredient already exists, its quantity is aggregated rather than overwritten.

4.6.2 TRADE: Converting Trophies into Ingredients

The `TRADE` command enables Geralt to exchange trophies—gained from defeating monsters—for alchemical ingredients. These trades must follow a strict syntactic structure and pass multiple validation stages to ensure both semantic correctness and inventory consistency.

Input Format. The general form of a valid trade sentence is:

```
Geralt trades <quantity> <trophy> trophy, ... for <quantity>
<ingredient>, ...
```

Step 1: Sentence Validation. The function `is_valid_trade_sentence` ensures the input adheres to the expected grammar. The validation involves two sequentially parsed sections:

- A comma-separated list of `<quantity> <name> trophy` blocks before the `for` keyword.
- A comma-separated list of `<quantity> <ingredient>` blocks after the `for` keyword.

Each quantity must be a positive integer, and each name must consist of alphabetic characters. Any deviation results in an `INVALID` response.

```c
if (!is_digit_custom(words[curr_index]))
return FALSE;
if (!is_alphabetic_custom(words[curr_index + 1]))
return FALSE;
if (strcmp(words[curr_index + 2], "trophy") != 0)
return FALSE;
```

Listing 11: Example Trophy Section Check

Step 2: Trade Validity. After syntactic checks, the function `check_valid_trade` ensures that Geralt has sufficient quantities of the listed trophies to complete the trade.

This function performs a nested search:

- For each trophy listed in the trade,
- Find the matching trophy in the global inventory,
- Confirm that the quantity to be traded does not exceed the available quantity.

If any check fails, the trade is aborted with a `Not enough trophies` message.

```c
if (trophies_to_trade[i].quantity > trophies[j].quantity)
return FALSE;
```

Listing 12: Trophy Quantity Validation

Step 3: Trade Execution. The function `trade` updates both the ingredients and the trophies arrays:

- Each traded trophy's quantity is decremented.
- Each received ingredient is added (or updated) in the inventory.

- If ingredient capacity is reached, it is doubled via `ensure_ingredient_capacity`.

```c
if (!found) {
  ensure_ingredient_capacity();
  last_added_ingredient_index++;
  strcpy(ingredients[last_added_ingredient_index].name,
     ingredient_name);
  ingredients[last_added_ingredient_index].quantity = quantity;
}
```

Listing 13: Adding Traded Ingredients

This guarantees that the inventory is kept consistent and accurate, without duplications or memory issues.

Example. Given the input:

```
Geralt trades 1 Drowner trophy, 2 Ghoul trophy for 3 Rebis, 1 Vitriol
```

...the system:

- Verifies both Drowner and Ghoul trophies exist in sufficient quantities,
- Decrements their quantities,
- Adds 3 Rebis and 1 Vitriol to the inventory.

If successful, the following message is printed:

Output

```
Trade successful
```

Design Impact. The `TRADE` flow demonstrates tight coupling between inventory logic and memory management. It emphasizes defensive programming with multiple stages of input filtering and array capacity control, ensuring resilience to malformed or malicious input.

4.6.3 BREW: Crafting Potions from Ingredients

The `BREW` command allows Geralt to synthesize potions, provided that he has already learned the formula and possesses all necessary ingredients. This feature simulates real-time inventory management, requiring both structural and semantic validation before any modification is made.

Input Format. A valid brew command takes the form:

`Geralt brews <potion_name>`

The potion name may consist of multiple words (e.g., `Full Moon`), and it is extracted and checked accordingly.

Step 1: Sentence Validation. The `is_valid_brew_sentence` function ensures that all words after `Geralt brews` are alphabetic, and the potion name is well-formed. Extra spaces or trailing non-alphabetic tokens result in `INVALID`.

```c
for (int i = 2; i < word_count; i++) {
  strcat(potion_name, words[i]);
  if (i < word_count - 1)
  strcat(potion_name, "␣");
}
```

Listing 14: Potion Name Construction

The function also calls `is_valid_potion_name_spacing` to ensure consistency with spacing and capitalization rules enforced elsewhere.

Step 2: Formula and Inventory Check. Once the sentence is validated, two critical checks are performed:

- `has_formula(potion_name)`: Verifies that a potion formula with the given name exists.
- `can_brew(potion_name, inventory, formulas)`: Confirms that Geralt's inventory contains all required ingredients in sufficient quantities.

If either condition fails, a descriptive error message is printed:

`No formula for Cat` or `Not enough ingredients`

Step 3: Brewing Process. If all checks pass, the `brew_potion` function is called. It performs the following steps:

- Decreases the inventory quantities for each ingredient in the potion formula.
- Adds the potion to the potion array, or increases its quantity if it already exists.
- Expands the potion array if capacity is reached, using `ensure_potion_capacity`.

```c
if (strcmp(inventory[j].name, formula->ingredients[i].name) == 0) {
  inventory[j].quantity -= formula->ingredients[i].quantity;
}
```

Listing 15: Inventory Reduction During Brewing

After the potion is successfully brewed, a message is displayed:

Output

```
Alchemy item created: <potion_name>
```

Design Notes. Unlike `LOOT` or `TRADE`, the `BREW` command is contingent upon prior learning via the `LEARN` command. This dependency chain adds a layer of interaction between subsystems, encouraging dynamic state transitions and memory-safe inventory modifications.

All memory allocations are guarded via capacity-checking macros, ensuring robustness in repeated brewing operations.

4.6.4 LEARN: Acquiring Alchemical Knowledge

The `LEARN` command allows Geralt to either acquire new potion formulas or record the effectiveness of known potions or signs against specific monsters. This makes the LEARN command structurally polymorphic—it supports two distinct subtypes of learning operations.

We divide the parsing and execution flow into two main branches:

1. Learning a potion formula
2. Learning a potion/sign effectiveness

1. Learning a Potion Formula

A potion formula defines the ingredient composition of a named potion. The input is expected in the following form:

```
Geralt learns Cat potion consists of 2 Rebis, 1 Vitriol
```

Validation. This format is validated inside the `is_valid_learn_sentence` function, which:

- Extracts the potion name by detecting the word `potion`.

- Ensures the phrase `potion consists of` is present.
- Checks that the ingredients list follows a comma-separated `<quantity> <name>` pattern.

```c
if (strcmp(words[potion_idx + 1], "consists") == 0 &&
strcmp(words[potion_idx + 2], "of") == 0)
{
  // begin formula parsing
}
```

Listing 16: Basic Formula Structure

Execution. Once validated, the `learn_potion_formula` function performs the following actions:

- Assembles the full potion name and checks for duplicates using `has_formula`.
- Dynamically allocates a new `PotionFormula` entry with expandable ingredient capacity.
- Iterates through the ingredient list, adding them to the formula array.

```c
formula->ingredients = malloc(sizeof(Ingredient) * formula->
    ingredient_capacity);
...
if (formula->ingredient_count >= formula->ingredient_capacity)
formula->ingredients = realloc(...);
```

Listing 17: Dynamic Allocation of Formula

Upon success, a message is printed:

Output

```
New alchemy formula obtained: Cat
```

Example.

`Geralt learns Cat potion consists of 2 Rebis, 1 Vitriol`

Result:

- A new formula named `Cat` is created.
- The ingredient list is stored: `Rebis (2)`, `Vitriol (1)`.

2. Learning Effectiveness Against Monsters

In this mode, Geralt learns that a potion or sign is effective against a monster. The input form is:

```
Geralt learns Igni sign is effective against Drowner
```

Validation. This is validated via the same `is_valid_learn_sentence` function, but using different rules:

- Ensures the input matches one of:
  - `<name>sign is effective against <monster>`
  - `<name>potion is effective against <monster>`
- Ensures the effectiveness target is a known word (i.e., strictly alphabetic).

Execution. The effectiveness is registered in the `learn_effectiveness` function:

- If the monster does not exist, a new entry is created in the `monsters` array.
- If the potion or sign is already recorded for that monster, the function aborts with a message.
- Otherwise, the entry is added, and the monster's internal sign or potion arrays are resized if needed.

```
if (monster_index == -1) {
  ensure_monster_capacity();
  last_added_monster_index++;
  Monster *m = &monsters[last_added_monster_index];
  strcpy(m->name, monster_name);
  ...
}
```

Listing 18: Monster Entry Allocation

Messages. Depending on the case, the function prints:

- `New bestiary entry added: Drowner`
- `Bestiary entry updated: Drowner`
- `Already known effectiveness`

Example.

```
Geralt learns Moon Dust potion is effective against Necrophage
```

Result:

- If the monster `Necrophage` does not exist, it is created.
- The potion `Moon Dust` is added to its effectiveness list.

Design Insight. The `LEARN` command is one of the most expressive features of the system, reflecting knowledge acquisition and long-term memory. It supports dynamic memory growth, separation of declarative logic (formulas) vs operational logic (combat), and maintains idempotence in learning to avoid redundant entries.

4.6.5 ENCOUNTER: Combat and Trophy Collection

The `ENCOUNTER` command represents a critical gameplay mechanic—Geralt comes face-to-face with a monster. Based on his preparation (learned signs and brewed potions), he either defeats the monster or escapes narrowly.

The command is one of the most dynamic in terms of logic, as it combines all previous subsystems: bestiary entries, potion inventory, and global state transitions.

Input Format. The encounter command must strictly follow this format:

```
Geralt encounters a <monster_name>
```

Validation. The `is_valid_encounter_sentence` function ensures:

- The input contains exactly four words.
- The third word is the article `"a"`.
- The final word is a valid alphabetic monster name.

Incorrect or malformed inputs are rejected immediately with the message: `INVALID`

Execution Logic. The `handle_encounter` function processes the battle outcome through the following stages:

1. Searches for the given monster in the global `monsters` array.
2. If not found, prints: `Geralt is unprepared and barely escapes with his life`
3. If found, checks if Geralt has either:

   - At least one **sign** that is effective against the monster, or
   - At least one **potion** (with quantity ≥ 1) that is effective against the monster.

4. If neither is true, Geralt escapes.

5. Otherwise, he defeats the monster and:

   - Consumes all effective potions (by reducing their quantities).
   - Gains a `Trophy` named after the monster (adds to inventory or increments if it already exists).

```
if (!has_effective_sign && !has_effective_potion) {
  printf("Geralt␣is␣unprepared␣and␣barely␣escapes␣with␣his␣life\n");
  return;
}
```

Listing 19: Victory or Defeat Decision

Output Scenarios. The function produces one of the following outputs:

- `Geralt is unprepared and barely escapes with his life`
- `Geralt defeats <monster_name>`

Post-Battle Inventory Changes.

- All matching effective potions in Geralt's inventory are decremented.
- The monster's name is added to the `trophies` array, or its quantity is incremented.

```
if (strcmp(m->potions[i].name, potions[j].name) == 0 &&
potions[j].quantity > 0)
{
  potions[j].quantity--;
}
```

Listing 20: Potion Decrement

Example.

```
Geralt encounters a Nekker
```

If Geralt previously learned:

- That `Igni sign` is effective against `Nekker`, or
- That `Moon Dust potion` is effective and has it in inventory

He will win the battle and receive:

Output

```
Geralt defeats Nekker
```

If not:

Output

```
Geralt is unprepared and barely escapes with his life
```

Design Summary. The `ENCOUNTER` logic tightly integrates knowledge-based components (from LEARN) with resource constraints (potions inventory). It reflects a realistic combat scenario, requiring prior preparation and smart resource use.

4.7 question_handle.c

The `QUESTION` command family handles various queries that inquire about Geralt's inventory or knowledge. The system supports a diverse set of queries that are parsed and validated prior to dispatch. All queries end with a `"?"` character and follow rigid structural patterns.

We classify questions into two main types:

- **Quantitative Queries**: Ask for the quantity of a specific or all resources.
- **Informational Queries**: Ask for learned knowledge (e.g., recipes or monster weaknesses).

We discuss each supported query type and its implementation details below.

1. Specific Ingredient Query

Example: `Total ingredient Mandrake?`

Execution: `handle_specific_ingredient_query`

- Searches the `ingredients` array.
- Prints the quantity or 0 if not found.

```
3
```

2. Specific Potion Query

Example: `Total potion White Honey?`

Execution: `handle_specific_potion_query`

- Extracts the potion name (multi-word).
- Validates its spacing using `is_valid_potion_name_spacing`.
- Returns quantity or `INVALID` if name is malformed.

3. Specific Trophy Query

Example: `Total trophy Leshen?`

Execution: `handle_specific_trophy_query`

- Searches the `trophies` array.
- Returns the quantity or 0.

4. All Ingredients Query

Example: `Total ingredient?`

Execution: `handle_all_ingredients_query`

- Sorts ingredients alphabetically (`cmpIngredient`).
- Filters out 0-quantity entries.
- If no ingredients, prints `None`.

5. All Potions Query

Example: `Total potion?`

Execution: `handle_all_potions_query`

- Similar to ingredients query.
- Uses `cmpPotion` for sorting.

6. All Trophies Query

Example: `Total trophy?`

Execution: `handle_all_trophies_query`

- Uses `cmpTrophy` for alphabetical ordering.
- Returns `None` if no trophies found.

7. Monster Effectiveness Query

Example: `What is effective against Drowner?`

Execution: `handle_monster_query`

- Searches for the given monster.
- If found, returns a sorted list of known effective signs/potions.
- If not found, prints:

```
No knowledge of Drowner
```

8. Potion Recipe Query

Example: `What is in Cat potion?`

Execution: `handle_potion_recipe_query`

- Extracts and validates potion name.
- Searches for corresponding formula.
- Sorts ingredients with `cmpForRecipe` — by decreasing quantity, then alphabetically.
- Returns `No formula for <name>` or `INVALID` if malformed.

Implementation Reflection. Each handler function:

- Is modular and single-responsibility.
- Makes use of global arrays declared in `globals.h`.
- Outputs to standard output in compact, consistent format.

This design allows for the system's current state to be queried in a human-readable and well-structured way. It supports both precise targeting and comprehensive overviews—critical for planning and knowledge management.

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
```

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 solution successfully fulfills its primary objective: parsing and executing commands written in a domain-specific, Witcher-themed language. The program handles various types of inputs such as looting, brewing, learning, trading, and combat encounters, along with query-based questions. The modular structure, consisting of functions categorized by sentence type and utilities, enabled organized development and facilitated extensibility.

Performance

In terms of performance, the program exhibits fast execution for small to moderately sized inputs, owing to the use of efficient memory management strategies such as dynamic array resizing and in-place updates. Most operations such as lookups, inserts, and condition checks run in linear time with respect to the size of relevant arrays. Sorting operations, used primarily in the query handlers, are handled using the implemented function `qsort`, which provides acceptable performance for the data sizes involved.

Limitations

Despite the program's robustness, several limitations persist:

- **Static Memory Assumptions:** Word and line limits (e.g., `MAX_WORD_LEN`, `MAX_LINE_LENGTH`) are fixed, which might limit flexibility for edge cases or long input sentences.
- **Error Reporting Granularity:** Currently, invalid lines are simply marked as "INVALID" without specifying the exact reason (e.g., invalid structure, spacing issues, or missing keywords). This limits debuggability for users.
- **No Input History or Undo:** Once a sentence is executed, its effects are irreversible. A rollback or undo mechanism could be beneficial, particularly for invalid trade or brew operations.
- **Global State Reliance:** Most operations depend heavily on global variables (e.g., `ingredients`, `formulas`, `monsters`), which can hinder modularity and complicate future improvements.

Possible Improvements

Several improvements could enhance the utility, maintainability, and user experience of the program:

- **Improved Error Feedback:** Enhancing the parser to return detailed error messages (e.g., "Expected comma after ingredient" or "Invalid spacing in potion name") would be helpful for users and testers.
- **Configuration File Support:** Allow certain constants like 'MAX_WORD_LEN' or initial capacities to be set via a configuration file rather than hardcoded constants.

- **Command Suggestion System:** When an input is invalid, the program could offer suggestions based on edit distance or known command templates (e.g., "Did you mean 'Geralt brews Swallow'?").
- **Interactive Shell Enhancements:** Enhance the REPL (read-eval-print loop) with features like tab completion, command history, or built-in help commands (e.g., 'help trades' or 'list potions').

In summary, the project presents a functional and modular interpreter capable of understanding and executing a custom set of structured inputs. With moderate enhancements in user interaction, error handling, and architectural design, it could be developed into a more sophisticated and user-friendly system.

7 Conclusion

In this project, we designed and implemented an interactive text-based system that simulates alchemical operations in a fantasy world inspired by The Witcher universe. Through a combination of modular C programming, careful memory management, and robust parsing logic, the system effectively interprets and executes a wide range of commands issued by the user. These include actions such as looting ingredients, learning potion recipes, brewing potions, trading trophies, and encountering monsters—all while maintaining an evolving internal game state.

The program demonstrates a strong emphasis on input validation, dynamic memory allocation, and efficient data management. Each major functionality is encapsulated within separate modules, contributing to the code's overall readability and maintainability. Furthermore, the testing process through diverse input cases has shown the system to be capable of handling both well-formed and malformed instructions gracefully, ensuring user feedback remains informative and consistent.

Despite the system's capabilities, several areas for enhancement remain. For instance, the absence of persistent storage, limited natural language flexibility, and minimal user interface can be seen as opportunities for future development. By implementing features such as file-based save/load mechanics, interactive CLI improvements, or even a graphical frontend, the program could evolve into a more comprehensive and engaging application.

Overall, this project successfully combines systems-level programming with a touch of game-like logic and language processing, resulting in a satisfying and technically rich experience. The design and development process have also reinforced key concepts such as memory safety, modular design, and structured parsing—core competencies for any systems programmer.

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.