# Project Report

Systems Programming – Spring 2025
(C++ Implementation with Object-Oriented Programming)

Serdar ŞEN, 2022205027
Yağmur KARACA, 2022400129

June 1, 2025

## 1 Introduction

This project implements an interpreter and an inventory-event tracking system for the character Geralt of Rivia from the popular fantasy series *The Witcher*. The system allows the tracking of Geralt's actions and interactions with the world around him, such as looting ingredients, learning potion recipes, brewing potions, and encountering monsters.

This version of the project has been reimplemented in C++ using **object-oriented programming** principles as the third assignment of the CMPE230 course. Compared to the initial C version, this version provides modularity due to class-based implementation, data encapsulation, and improved code structure through abstraction.

## 2 Problem Description

The system processes three types of input:

- Sentences: Define inventory actions (looting, trading, brewing), knowledge acquisition, or monster encounters.
- Questions: Retrieve information about Geralt's inventory, bestiary, or alchemy.
- Exit Command: Terminates the program.

Inputs follow grammar rules and generate valid responses or "INVALID" if malformed.

## 3 Methodology

The C++ implementation utilizes OOP principles to design modular, reusable, and maintainable components. Key concepts such as encapsulation, data abstraction, and class-based modularity were applied.

1. **Class Design:** Core entities such as Inventory, Alchemy, Bestiary, and RecipeBook are implemented as C++ classes.
2. **Command Parsing:** Input is read from stdin and parsed using string streams. Tokenized commands are mapped to specific methods in corresponding objects.

3. **Execution Flow:** Validated commands are dispatched through object methods. Invalid commands result in an "INVALID" response.

# 4 Implementation

## 4.1 Code Structure (OOP)

1. **main.cpp**: Handles the main loop, command parsing, and flow control.
2. **Inventory.cpp/.h**: 'Inventory' class for managing ingredients, potions, and trophies.
3. **Alchemy.cpp/.h**: 'Alchemy' and 'PotionFormula' classes for brewing logic and recipe management.
4. **Bestiary.cpp/.h**: 'Bestiary' class for tracking monsters and effective items.

## 4.2 Execution Flow

1. The program reads input through stdin.
2. Splits the input string into tokens.
3. Then adds each meaningful token (words, numbers and commas) in the finalTokens vector.
4. Uses an if-else logic to control the execution flow.
5. Asserts the correctness of grammar by invalidating any unnecessary / flawed tokens. Prints "INVALID" when they are detected.
6. Asks for input until the input is "Exit".

## 4.3 Data Structures

Mostly used data structures are vectors and maps. Maps are used for the implementation of inventories, where the key is a string (item name) and the value is an integer (the quantity of that item). Maps were preferred because they are quite efficient since they provide O(1) insertion and O(1) search. On the other hand, vectors were used often where there is a need for a list and search efficiency is not the first concern. For example, potion recipes are vectors that store pairs of strings and integers because no searches are performed on a recipe.

## 4.4 Sample Code

**Learning Potion Recipes**
learnNewRecipe function below first checks if the potion already has a known recipe and adds the recipe to the recipeBook map if it does not.

```
void Alchemy::learnNewRecipe(string potionName, vector<pair<string,int
    >> ingredientList) {
    auto itr = recipeBook.find(potionName);
    if (itr != recipeBook.end()) {
        cout << "Already known formula" << endl;
    }
    else {
        recipeBook[potionName] = ingredientList;
        cout << "New alchemy formula obtained: " << potionName << endl;
    }
}
```

### Checking List Validity

listIsValid function checks the grammar of the ingredient or trophy lists during looting, trading and learning recipe actions. It asserts the following grammar:

s → <number> <alphaWord> | <number> <alphaWord> , S

```
bool listIsValid(int startPos, int endPos, const vector<string>& tokens
    ) {

    if (startPos > endPos) {
        return false;
    }

    int i = startPos;

    while (i <= endPos) {
        // must start with a number
        if (!isAllDigits(tokens[i]) || stoi(tokens[i]) <= 0) {
            return false;
        }
        i++;

        // must be followed by a valid ingredient
        if (i > endPos || !isAllAlpha(tokens[i])) {
            return false;
        }
        i++;

        // if end of list, break (valid)
        if (i > endPos) {
            return true;
        }

        // expect a comma
        if (tokens[i] != ",") {
            return false;
        }
        i++;

        // must be followed by another quantity
        if (i > endPos) {
            return false;
        }
    }

    return true;
```

```
}
```

### Checking and Updating Monster-Effective Items

checkAndUpdateEffectiveness function first checks the bestiary map, where keys are strings and value is a pair of vectors (signs and potions, resp.), to see if the monster already has known weaknesses. If so, prints an appropriate output and adds the new item to the appropriate vector. If it is the first time, it initializes the vectors and prints a fitting output.

```
void Bestiary::checkAndUpdateEffectiveness(const string& type, const
    string& itemName, const string& monsterName) {
    auto itr = bestiary.find(monsterName);
    if (itr == bestiary.end()) { // if the monster is not known, adds
        it to bestiary
        cout << "New bestiary entry added: " << monsterName << endl;
        if (type == "sign") {
            vector<string> signVector;
            vector<string> potionVector;
            signVector.push_back(itemName);
            pair<vector<string>,vector<string>> pair;
            pair.first = signVector;
            pair.second = potionVector;
            bestiary[monsterName] = pair; // initializes the bestiary
        }
        else if (type == "potion") {
            vector<string> signVector;
            vector<string> potionVector;
            potionVector.push_back(itemName);
            pair<vector<string>,vector<string>> pair;
            pair.first = signVector;
            pair.second = potionVector;
            bestiary[monsterName] = pair; // initializes the bestiary
        }
    }
    else { // if the monster is known.
        if (type == "sign") {
            vector<string> &signVector = bestiary[monsterName].first;
                // first vector holds sign type items
            auto signItr = find(signVector.begin(), signVector.end(),
                itemName);
            if (signItr == signVector.end()) {
                signVector.push_back(itemName);
                cout << "Bestiary entry updated: " << monsterName <<
                    endl;
            }
            else {
                cout << "Already known effectiveness" << endl;
            }
        }
        else if (type == "potion") {
            vector<string> &potionVector = bestiary[monsterName].second
                ; // second vector holds potion type items
            auto potionItr = find(potionVector.begin(), potionVector.
                end(), itemName);
            if (potionItr == potionVector.end()) {
                potionVector.push_back(itemName);
                cout << "Bestiary entry updated: " << monsterName <<
                    endl;
```

```
42                }
43                else {
44                    cout << "Already known effectiveness" << endl;
45                }
46            }
47        }
48    }
```

# 5 Results

```
>> Geralt loots 5 Rebis, 3 Vitriol
Alchemy ingredients obtained
>> Total ingredient
INVALID
>> Total ingredient ?
5 Rebis, 3 Vitriol
>> Geralt brews Black Blood
No formula for Black Blood
>> Geralt encounters Harpy
INVALID
>> Geralt encounters a Harpy
Geralt is unprepared and barely escapes with his life
>> Geralt learns Black Blood potion is effective against Harpy
New bestiary entry added: Harpy
>> Geralt learns Black Blood potion consists of 6 Rebis, 1 Vitriol
New alchemy formula obtained: Black Blood
>> Geralt brews Black  Blood
INVALID
>> Geralt brews Black Blood
Not enough ingredients
>> Geralt learns Igni sign is effective against Drowner
New bestiary entry added: Drowner
>> Geralt encounters a Drowner
Geralt defeats Drowner
>> Geralt trades 1 Drowner trophy for 1 Rebis
Trade successful
>> Geralt brews Black Blood
Alchemy item created: Black Blood
>> Total potion ?
1 Black Blood
>> Geralt encounters a Harpy
Geralt defeats Harpy
>> Total trophy Harpy?
1
```

# 6 Discussion

The restructured C++ implementation improves code organization, reusability, and abstraction. Object-oriented design helped isolate concerns such as inventory management, alchemy logic, and monster interaction into separate classes.

The use of classes allowed a more natural modeling of entities like 'PotionFormula' and 'Inventory', with better memory safety and data encapsulation. However, the learning curve of managing dynamic memory (e.g., linked structures in C++) remains a notable challenge.

## 6.1 Challanges and Solutions

- Detecting multiple spaces in multiple-word potion names were quite difficult since the tokenization at the start of the program does not consider spaces. We overcame this problem by writing a getExactPotionName function that returns the portion of the input string that is supposed to consist the potion name, and then performed spacesAreValid function which detects if there are multiple spaces in that name.
- Considering many other invalid cases that were not covered by the test case files was difficult to accomplish. We used AI assistants to provide invalid sentences and performed our tests on those.

# 7 Conclusion

The Witcher Tracker project successfully implements a command interpreter and inventory tracking system. The C++ version demonstrates better modularity and maintainability.

The transition to C++ enabled the use of object-oriented features such as:

- **Encapsulation:** Each module (Inventory, Alchemy, Bestiary) manages its own data.
- **Modularity:** Header/source separation and encapsulated logic improve reuse.
- **Constructor/Destructor Use:** Resources are initialized and cleaned up properly.

Future enhancements could include:

- Adding persistent storage to save state between sessions
- Developing a graphical user interface to enhance user interaction
- Implementing a more comprehensive testing framework

## Comparison Between 2 Versions: C vs. C++

- Better modularity through classes and namespaces
- Built-in constructors/destructors
- More reliable, better implemented parsing

# AI Assistants

AI Assistants such as ChatGPT and Claude were used in this project for the following purposes:

- English grammar correction in comments & documentation,
- New logic (without code) on how to implement some functions
- Change of variable names in the entire program when a variable name is decided to be changed
- Generation of invalid inputs for testing purposes