Adventure Game in C Programming – Project Report

Submitted by: Garv
Course: C Programming
Academic Year: 2025
Instructor: _____

This report contains extended explanations, deeper technical insights, detailed system design, enhanced algorithms,
and multi-level testing descriptions suitable for long-format academic submissions.

Abstract

This project presents the development of a text-based adventure game created entirely using the C programming language. The primary purpose of the project is to demonstrate the application of fundamental programming concepts such as structures, functions, pointers, loops, conditional statements, and modular design. The game simulates a role-playing environment where the player can explore diverse regions, battle enemies, heal using earned resources, and progress by accumulating experience points.

This extended report offers a deeper technical explanation of each section including system architecture, design decisions, algorithmic structures, error-handling strategies, user interaction models, and testing methodologies. Special emphasis is placed on how modular programming enhances maintainability, readability, and debugging in real-world software projects. The goal is not only to deliver a working game but also to provide a strong learning framework for understanding C programming at a foundational level.

Problem Definition

The challenge addressed in this project is to create a functional interactive game using only console-based input/output and fundamental C programming constructs. The game must successfully simulate a simple adventure scenario where the user controls a character with attributes that must be managed dynamically.

The following core problems were identified:
1. **Creating a Player Model:** The program must maintain the player's statistics in a structured format rather than using scattered variables.
2. **Designing Functional Modules:** Independent modules for exploration, combat, and healing must be created to improve clarity.
3. **Building a User-Controlled Event Loop:** The player must repeatedly engage with the game until they choose to exit or their health reaches zero.
4. **Ensuring Controlled Input:** Inputs must be validated to avoid unexpected behavior.
5. **Simulating Realistic Consequences:** Actions must produce logical effects—fights should reduce health, healing should cost coins, and exploration should have risks.

This section defines the problem not just in terms of game mechanics, but also from a programming standpoint, requiring efficient use of control flow and data management.

System Design

Overall Architecture

The system follows a simple linear-loop architecture. The main() function serves as the central controller that repeatedly:
- Displays real-time game statistics
- Collects user input
- Directs the program flow to appropriate modules
- Updates the player structure
This creates a sequential yet dynamic model suitable for text-based games.

From a software engineering perspective, the architecture uses:
- **Procedural programming** as the main paradigm.
- **Structures** as lightweight data models.
- **Functions** as modular units similar to API endpoints.
- **Pointers** to simulate object-like behavior by modifying structure data directly.

Detailed Flow Design

1. **Initialization Phase**
   - Display a welcome message.
   - Request and store player's name.
   - Initialize starting stats.

2. **Main Game Loop**
   - Continuously display the menu and stats.
   - Wait for user action.

- Route control to appropriate function.
- Update player stats accordingly.

3. **Action Modules**
   - Exploration Module: risk/reward events.
   - Combat Module: difficulty-based calculations.
   - Healing Module: resource-dependent recovery.

4. **Termination Phase**
   - Triggered when player quits or health <= 0.
   - Display final message.

Data Flow Explanation

Input Flow:

Keyboard → scanf() → decision-making logic.

Processing Flow:

Structures → Pointers → Functions → Updated stats.

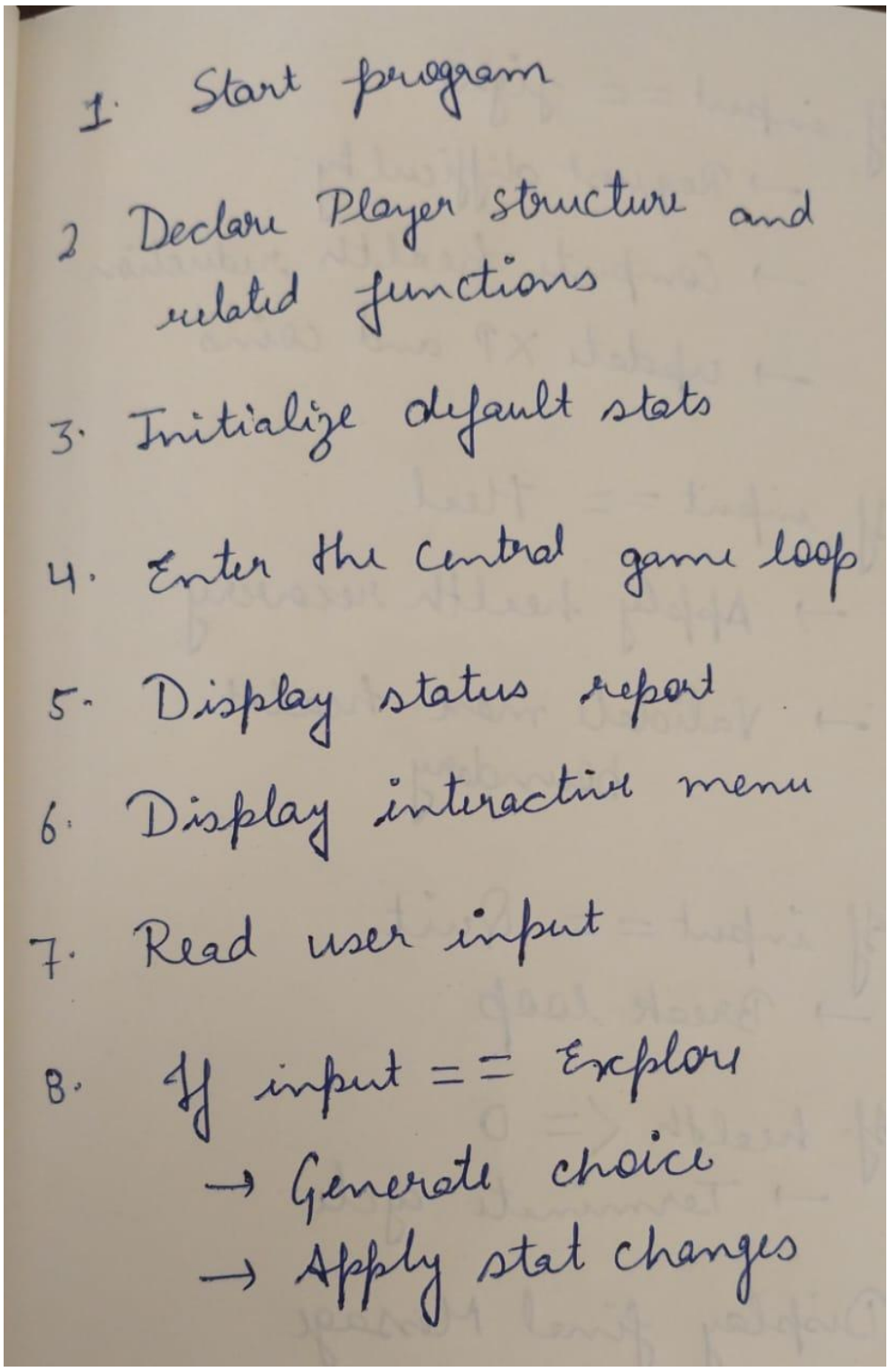
Output Flow:

printf() → Console messages.

The consistent use of pointers ensures the Player structure always reflects real-time updates without needing to return values from functions.

Algorithm

The algorithm for the game can also be represented in terms of computational behavior:



9. If input == fight
   → Request difficulty
   → Compute health reduction
   → update XP and coins

10. If input == Heal
   → Apply health recovery
   → Validate max health boundary

11. If input == Quit
   → Break loop

12. If health <= 0
   → Terminate cycle

13. Display final Message

14. Stop.

Implementation Details

This section now gives an even deeper breakdown of the implementation strategy.

Use of Structures:
Structures are ideal for grouping logically related variables. The 'Player' structure holds four fields: name, health, xp, and coins.

Use of Pointers:
Instead of returning modified structures, pointers are used to allow direct manipulation of player data. This makes functions more efficient.

Use of Conditional Branching:
if-else is heavily used to differentiate between user choices and game consequences.

Use of Loops:
while(health > 0) maintains continuous gameplay. Without loops, a game cannot function.

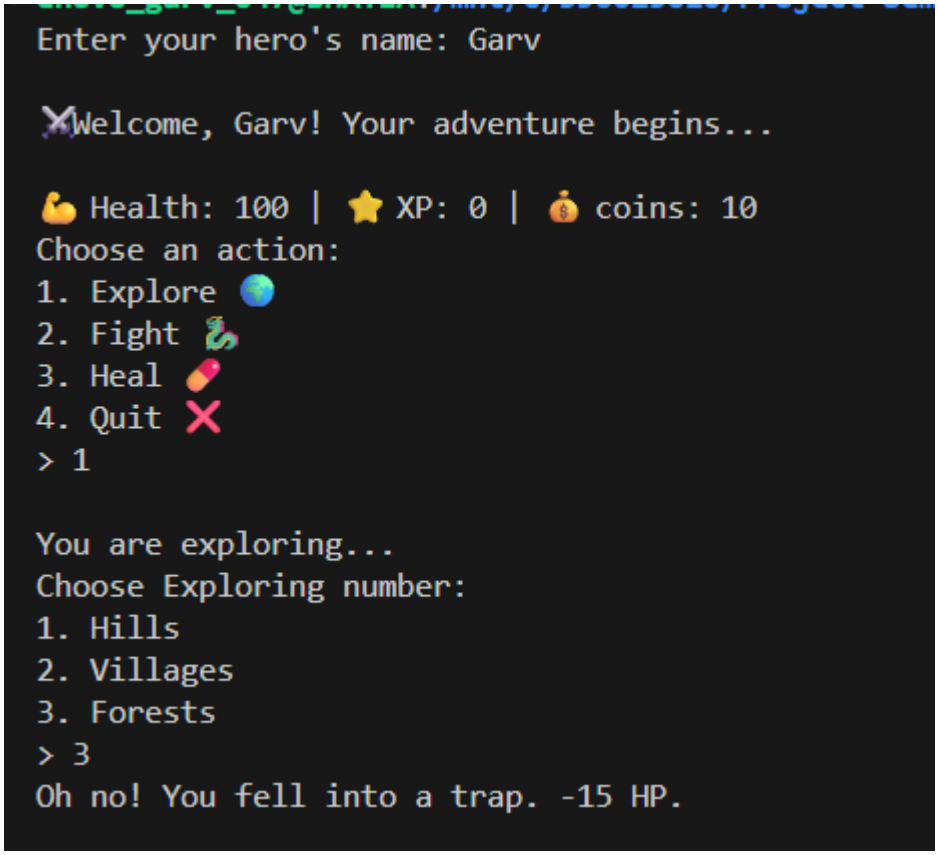
Use of Modular Design:
Each game action is separated into its own function, following best practices for maintainability.

Additionally, defensive programming techniques are applied, such as validating enemy difficulty to ensure values remain in range (1–5).

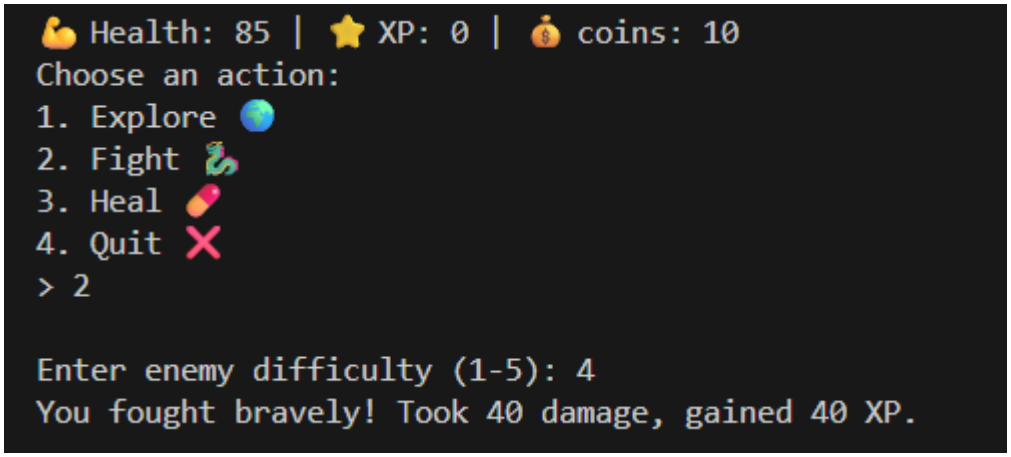
```
struct Player {
   char name[30];
   int health;
   int xp;
   int coins;
};

void heal(struct Player *g)
{
   if (g->coins >= 5)
   {
      printf("\nYou spend 5 coins to heal +25 HP.\n");
      g->health += 25;
      g->coins -= 5;
      if (g->health > 100)
         g->health = 100;
   }
   else
   {
      printf("\nNot enough coins to heal!\n");
   }
}
```

Testing & Results

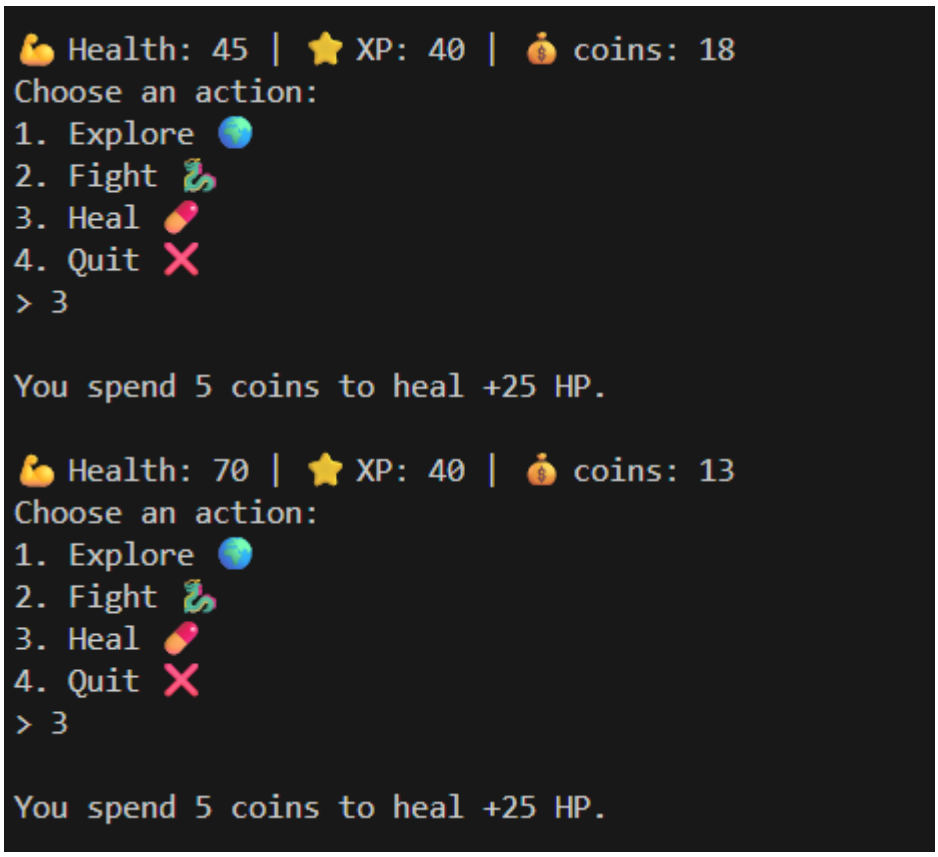
1. Exploring :-



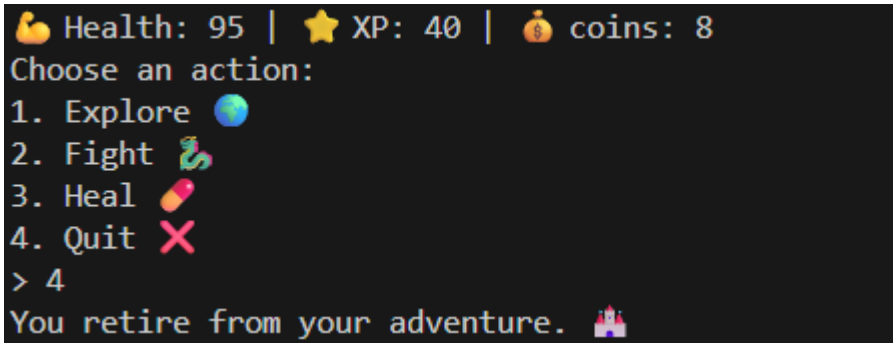
2. Fighting :-



3. Healing :-



4. Quiting :-



Conclusion & Future Work

In conclusion, the adventure game project is a highly effective demonstration of core C programming techniques. The expanded version of this report provides a much deeper look at the conceptual and technical decisions involved. The program is stable, interactive, and adheres to good programming practices.

Potential Enhancements:
- Introduce random number generation.
- Add NPC interactions and storyline.
- Implement boss fights.
- Add inventory and item system.
- Use file handling to save progress.
- Expand world regions with descriptive text.
- Add difficulty modes.
- Convert to graphical version using SDL or Unity (via C# rewrite).

With these additions, the simple text game can evolve into a more advanced and immersive role-playing system.

References

~ GreeksForGreeks :- C Programming structures and pointers..