Report

Ayşenur Elif Abanoz

Requirement Analysis

Problem Statement

The project involves the development of a soldier game with two teams. The game follows a turn-based combat system, and the team with the last player remaining wins.

Goals

The aim of this project is to understand the design, implementation, and testing phases while developing the game. Additionally, it aims to effectively utilize pointer and stack structures. The key objectives are as follows:

- Utilize pointer and stack structures efficiently.

- Demonstrate problem-solving skills in software engineering.

- Create a game using algorithmic skills in the C programming language, focusing on design, implementation, and testing.

Inputs

To read the data, a text file is required to enter the data. Inputs will be taken from here, and the following methods will be run:

- In the `addSoldiers` method, when 'A' is read from the text, as parameters, we will first specify the team to which we want to add soldiers, followed by the health and strength values of that soldier, respectively.

- The `fight` method will run when it reads 'F' from the text. However, this method will not take parameters.

- The `callReinforcements` method will run when 'R' is read from the text. Soldiers with randomly assigned health and strength values will be assigned within certain limits. It takes parameters for which team this soldier will be assigned to.

- The `criticalShot` method runs when 'C' is read from the text. It does not take any parameters. Whoever’s turn it is at that moment steps in and kills the opposing team’s soldier.

Outputs

A text file is needed to store the output values while the methods are running throughout the game. The game will close when one side runs out of soldiers, that is, when it loses. Here, writing to the text file will also stop. The methods produce the following outputs:

- The `addSoldiers` method will print text every time a soldier is added to the team. It will also keep the health and strength values of these soldiers.

- The `fight` method will print the damage it did to the soldier, and if the soldier died, it will be written to the text file.

- The `callReinforcements` method prints the added soldiers to the text file with their health and strength values.

- When the `criticalShot` method runs, it should produce an output, and if a soldier dies, it should also be printed.

Error Handling

In order for the game to end, a team's players must finish. When soldiers die, they will be removed from the stack using the `pop` method. If we do not check whether there are any soldiers left in the stack during the game, it will try to throw a non-pop soldier from the stack. That's why it will give an error.

Design

Design Decisions

In this project, I decided to use linked list and stack structure instead of an array. An array places elements into successive memory regions. This means we need to set the array size initially and stay at that size. On the other hand, the linked list keeps each element itself and the reference of the next element. This allows us to use memory more efficiently.

I will use 4 different methods throughout the game. These methods are: `addSoldiers`, `fight`, `callReinforcements`, and `criticalShot`. They will play an important role in determining the course of the game throughout the game. Using the `pop` structure, I will remove any dead soldiers from the structure.

I also need to keep track of what's happening in the game, these outputs.

Data Structures

I will use linked list and stack data structures in this project. I will also need data structures to keep the soldiers' health and strength values.

Algorithms

I need to use the stack data structure to add and remove soldiers, that is, to keep them in a certain place and change them easily.

Stack has `pop`, `push`, and `top` methods. `push` is used to add an element to the top of the stack, `pop` is used to pop the element at the top of the stack, and `top` returns the top element without removing. I will also use this `isEmpty` method to check if my stack is empty. I will use these methods to add and remove soldiers in my war game.

There are also 4 methods I will use in my game. My first method is `addSoldiers`, this method will allow me to add soldiers to the stack. My second method is `fight`, this method will enable the two sides to fight each other. So, when the fight command (`F`) is entered, a soldier will be selected from the stack. When a soldier dies, it will be popped from the stack. My third method is `callReinforcements`, this method assigns random health and strength values among the specified limit values and calls soldiers to the given side. And my last method is `criticalShot`, with this method, when it's the team's turn, the opposing team's player dies and is popped from the stack.

Execution Flow Between Subprograms

The process of the game will be as follows: Values will be written to the text file, and these values will be read from the text file together with the methods. If the methods have parameters, they will be read with these parameters and written back to a text file. I will add soldiers by taking the health and strength values from the text file with the `addSoldiers` method. If we consider that this method takes `A 1 100,59;22,987` as an input, for this method the `A` represents the method input, the number `1` represents the team, the value `100` represents health, and the value `59` represents strength. If we want to add how many soldiers, the values are separated with semicolons. The `fight` method will not take any parameters, there will be `F` in the input. With the damage formula, it will be determined whether the soldier will die or not. The `callReinforcements` method will take `R` as input. It will also get whichever side it wants to be added to, `1` or `2`. The `criticalShot` method will take `C` as

input. When it is the team's turn, the opposing team's player will die. What happens as a result of these methods will be written to the output text file.

Programmer Aspect

Main Data Variables

- `casualty`: A stack structure responsible for tracking eliminated soldiers.

- `totalSoldiers`: An integer variable representing the total number of soldiers in the game.

- `teamOrder`: An integer variable indicating which team's turn it is in the game. It alternates between 1 and 2 during each turn.

Algorithmic Functions

- `initializeStack`: Allocates memory for the stack structure.

- `push`: Adds a soldier to the stack.

- `pop`: Removes a soldier from the stack.

- `top`: Retrieves the top soldier from the stack.

Special Design Properties

- Dynamic memory allocation and deallocation techniques are employed to ensure efficient memory usage and prevent memory leaks.

Implementation

I added the implementation part to the zip folder.

File Checks

- Files are checked for successful opening using `fopen`.

- Proper error handling is in place if files cannot be opened.

Memory Management

- Memory allocated with `malloc` is freed using `free` to prevent memory leaks.

Checking for Invalid Entries

- Input validation is performed to ensure correct method calls.

TEST

Bugs and Software Reliability

- Code is checked for potential bugs, and edge cases

In the testing phase, special attention is given to identifying potential issues that could lead to program failure or incorrect behavior. The following scenarios are considered:

Edge Cases: The program is tested with extreme or boundary values for health, strength, and team numbers to ensure it handles these cases correctly.

Invalid Inputs: Various combinations of incorrect inputs are provided to verify that the program handles them gracefully. This includes providing invalid commands or parameters.

Concurrency Issues: In a multi-threaded environment, the program is tested to ensure that it handles concurrent operations correctly, without race conditions or deadlock.

Memory Leaks: Memory allocation and deallocation are scrutinized to confirm that there are no memory leaks, which could lead to performance degradation or program crashes over time.

Stack Integrity: The integrity of the stack is crucial. Tests are conducted to confirm that soldiers are pushed and popped in the correct order, and that the stack does not become corrupted during operations.

Critical Shot Scenarios: The critical shot method is thoroughly tested to ensure it properly identifies valid recipients and eliminates opposing team soldiers as expected.

Error Handling: The program's response to unexpected situations is examined. This includes testing scenarios where soldiers are called for reinforcements when the stack is full, or when the critical shot method is called without any valid recipients.

Some examples of Error Handling in program:

File Checks:

Before opening files with fopen files, it checks whether the files have been opened successfully. If the file cannot be opened, NULL is returned:

```
FILE *fileInput = fopen("input.txt", "r");
fileOutput = fopen("output.txt", "w");

if (fileInput == NULL || fileOutput == NULL) {
   perror("The file could not be opened!");
   return 1;
}
```

Checking for Invalid Entries:

To validate input received from the user, determine how invalid input will be handled.

A check can be added in the takenInput function with if-else:

```
void takenInput(char input[]) {
   char letter;
   int team, health, strength;

   sscanf(input, "%c", &letter);
```

```
    if (letter == 'A') {
        sscanf(input, "%c %d %d,%d", &letter, &team, &health, &strength);
        addSoldiers(team, health, strength);
    } else if (letter == 'R') {
        sscanf(input, "%c %d", &letter, &team);
        callReinforcement(team);
    } else if (letter == 'F') {
        fight();
    } else if (letter == 'C') {
        criticalShot();
    } else {
        printf("Invalid entry!");
    }
}
```

Check Soldiers:

If there are no soldiers left in the game, the totalSoldiers value is checked:

```
void criticalShot() {
    if (totalSoldiers == 0) {
        printf("No soldiers on the game!\n");
        fprintf(fileOutput, "No soldiers on the game!\n");
```

```
        return;
    }
```

If Recipient is Not Found:

"Invalid recipient" message is printed when a specific recipient cannot be found in the criticalShot function.

```
    int recipient = -1;
    for (int i = 0; i < totalSoldiers; i++) {
        if (soldiers[i].team != teamOrder && soldiers[i].isAlive) {
            recipient = i;
            break;
        }
    }
    if (recipient != -1) {
        soldiers[recipient].isAlive = false;
        printf("Critical shot:\n");
        printf("On team %d has a casualty\n", soldiers[recipient].team);
        fprintf(fileOutput, "Critical shot:\n");
        fprintf(fileOutput, "On team %d has a casualty\n",
soldiers[recipient].team);

         struct Soldier *newNode = malloc(sizeof(struct Soldier));
        newNode->team = soldiers[recipient].team;
        newNode->next = NULL;
```

```
        push(casualty, newNode);
    } else {
        fprintf(fileOutput, "Invalid recipient");
        printf("Invalid recipient");
    }
```

Software Extendibility and Upgradeability

The design of the soldier game implementation allows for easy extendibility and upgradeability. New features or functionalities can be added by simply creating new methods and integrating them into the existing codebase. Additionally, the use of dynamic data structures like linked lists and stacks provides flexibility in handling different scenarios within the game.

Performance Considerations

Software Extendibility and Upgradeability:

The design of the soldier game implementation allows for easy extendibility and upgradeability. New features or functionalities can be added by simply creating new methods and integrating them into the existing codebase. Additionally, the use of dynamic data structures like linked lists and stacks provides flexibility in handling different scenarios within the game.

Stack Operations Efficiency:

- The stack operations (push, pop) have a fixed time complexity, ensuring efficient memory management and consistent performance regardless of the number of soldiers in the game.

Randomization Impact:

- The callReinforcements method introduces random health and strength values. Consideration should be given to the impact of these randomizations on the overall game balance and fairness.

Critical Shot Handling:

- The criticalShot method may lead to abrupt game endings. Performance implications of critical shots should be assessed to ensure a balanced and enjoyable gameplay experience.

Memory Usage with Pointers:

- As pointers are used extensively, memory usage should be monitored to avoid potential memory leaks or excessive memory consumption.

Comments:

- The implementation effectively utilizes linked lists and stacks to manage soldiers and casualties, demonstrating a solid understanding of data structures and algorithms.

- The use of pointer-based memory management allows for dynamic allocation and deallocation of memory, showcasing proficiency in memory management techniques.

- Error handling is appropriately implemented to ensure robustness in file operations and memory allocation.

- The implementation considers various scenarios, such as reinforcements, critical shots, and regular fights, providing a diverse gameplay experience.

- The program follows a clear and structured design, making it easy to understand and maintain.

Overall, the soldier game implementation exhibits strong programming skills, effective use of data structures, and thoughtful consideration of potential edge cases. With appropriate testing and validation, this implementation has the potential to provide an engaging gaming experience.