ÇANKAYA UNIVERSITY

SOFTWARE ENGINEERING DEPARTMENT

SOFTWARE PROJECT I

Name Surname	**Duru Karacan**
Identity Number	**202128022**
Course	**SENG 271**
Experiment	**Experiment 1**
E-mail	**c2128022@student.cankaya.edu.tr**

Programming Assignment Goals:

Aim of This Project: Introduction to pointer basics and a simple data structure; **stack**

Understanding and applying the basic stack operations push, pop and top.

What Is Expected From Me: Implementing a battle game in the C programming language with a dynamic data structure **'stack'.**

General Content/Explanation of the Game: There are 2 sides. Each side has their own soldiers. Each soldier has its own strength and health value. Considering these values and the 4 basic functions in the game, a war is simulated and one of the sides wins.

What Are the Normal Inputs to the Program:

Standard Input File (input.txt): The program's normal input is a standart input file containing commands of this game.

Commands: According to the game's functions, there are 4 different types of commands. These commands are; Add Soldier, Fight, Call Reinforcement, and Critical Shot. And these are also functions of the game. Each command has a specific format to use. For instance, the command of Add Soldier includes the data of health and strength value.

What Output Should the Program Create:

Expected Output: Each command has a standard output that shows what's going on in the game right now. For instance, when you use the Fight function, the output will be; "1 hit 65 damage". The output tells us, It's side 1's turn to shoot. There is an expected output chart for each command down below.

Expected Output Chart:

Command	**Input file**	**Output**
Add Soldier	A 1 100,59 ; 22,987	add soldiers to side 1 S- H:100 S:59 S- H:22 S:987
Fight	F	1 hit 65 damage
Call Reinforcement	R 1	Called reinforcements to side 1 S- H:100 S:59
Critical Shot	C	Critical shot 1 has a casualty

What Is the Error Handling Requirements:

Error Handling: Normally, ***********************************when the program gets an invalid command or incorrect input formats from user, show an relevant error message to user. But this program gets input in a file (input.txt), so program show an relevant error message and needs to be started from scratch.

Design Decisions:

- It was decided to use the stack data structure to simulate the war between the two sides.
- Separate stacks were created for both sides, and these stacks were used to store the soldiers.
- It was decided to store the soldiers health and strength values as integer. This was used to represent the soldiers in the stacks.
- We calculate a damage value according to a given formula for each side attack.
- When the program receives the add soldier or call reinforcement command, it adds new soldier with health and strength value to the stack with the push function.
- When the program receives the fight command, it calculates the damage and subtracts it from the health value. If the health value becomes to zero or falls below zero, the program uses the pop function, and the soldier is removed from the stack.
- When the program receives the Critical Shot command, the opponent's soldier is removed from the stack with the 'pop' function, and no damage is calculated.

What Data Structures Are Used:

Fundamental Data Structure: is Stack for to store the integer values representing soldiers' health and strength.

What Algorithms Are Used:

Mathematical Formula: is used for calculate the damage.

$\text{Damage} = (\text{Strength1}-\text{Strength2}) \times 0.05 + 50$

rand() Function: is utilized to generate new random soldiers when the "Call Reinforcement" command is issued.

Pros and Cons of Choices Above:

Pros:

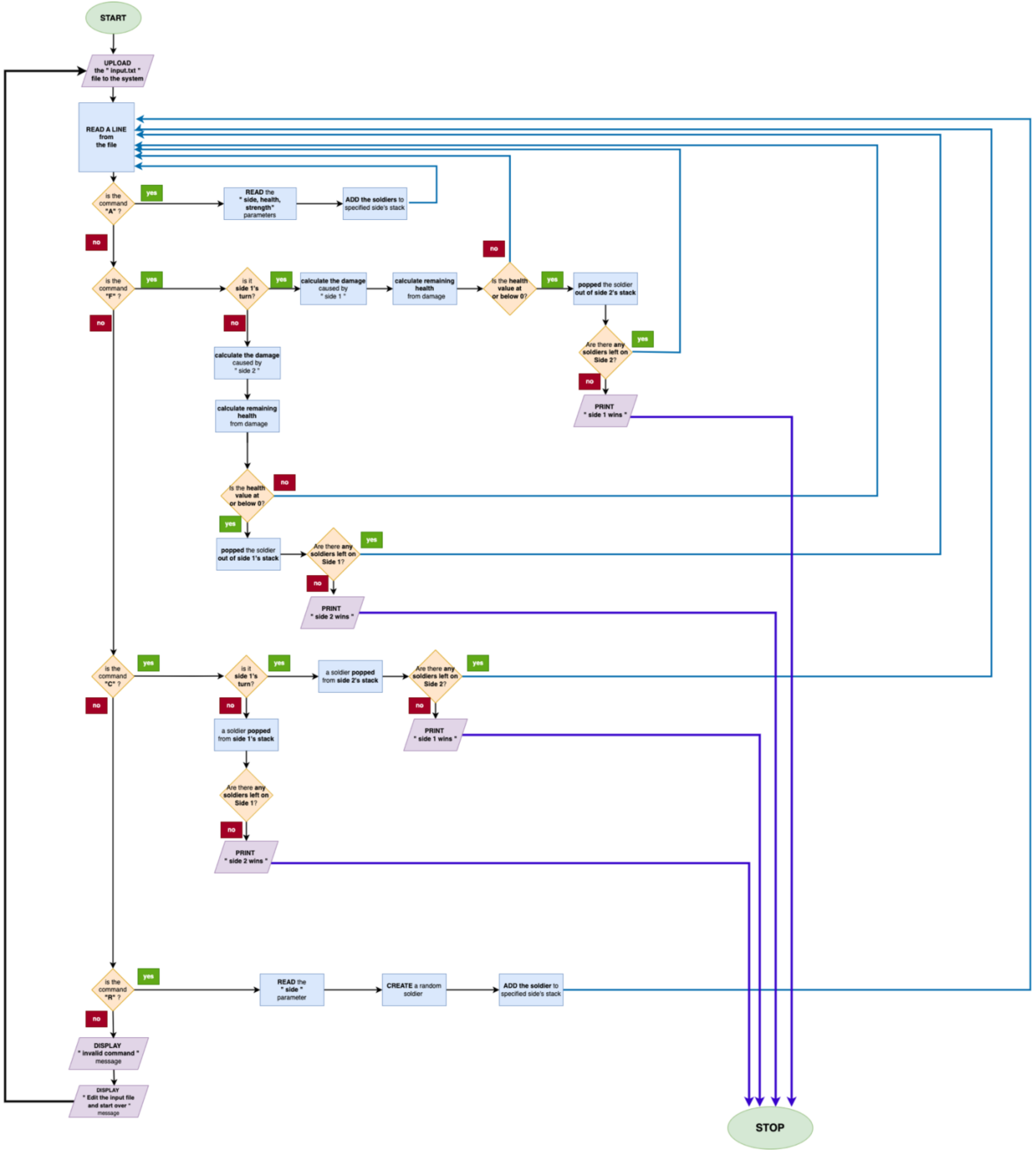
- Storing the soldiers' attributes(Health, Strength) as integer values optimizes memory usage.
- Calculating damage using a simple mathematical formula makes the calculations fast and comprehensible.
- Using a dynamic data structure like a stack to store the soldiers' data is more convenient than keeping them inside an array. Provides us with Dynamic Sizing, Efficient Insertions and Deletions, and Reduced Memory Wastage.

Cons:

- Because a simple war model is used, it may not be possible to simulate complex war strategies.
- Memory allocation and memory management can increase the complexity of the program.
- Although this is a war game, it is not an interactive game.

IMPLEMENTATION DETAILS

FLOWCHART


START
UPLOAD the " input.txt " file to the system
READ A LINE from the file
is the command "A" ?
yes
READ the " side, health, strength" parameters
ADD the soldiers to specified side's stack
no
is the command "F" ?
yes
is it side 1's turn?
yes
calculate the damage caused by " side 1 "
calculate remaining health from damage
Is the health value at or below 0?
no
yes
popped the soldier out of side 2's stack
Are there any soldiers left on Side 2?
yes
no
PRINT " side 1 wins "
no
no
calculate the damage caused by " side 2 "
calculate remaining health from damage
Is the health value at or below 0?
no
yes
popped the soldier out of side 1's stack
Are there any soldiers left on Side 1?
yes
no
PRINT " side 2 wins "
is the command "C" ?
yes
is it side 1's turn?
yes
a soldier popped from side 2's stack
Are there any soldiers left on Side 2?
yes
no
PRINT " side 1 wins "
no
no
a soldier popped from side 1's stack
Are there any soldiers left on Side 1?
no
PRINT " side 2 wins "
is the command "R" ?
yes
READ the " side " parameter
CREATE a random soldier
ADD the soldier to specified side's stack
no
DISPLAY " invalid command " message
DISPLAY " Edit the input file and start over " message
STOP

What sample code did you start with?

- In the main part of the code, I tried to read the commands from the input file, as I mentioned in the flowchart. I determined a case according to the symbol of each command.

```
int main() {
    srand(time(NULL));
    Stack side1, side2;
    side1.top = -1;
    side2.top = -1;
    int isBattleHappend = 0;

    int turn = 1;

    FILE *file = fopen("input.txt", "r");
    if (file == NULL) {
        perror("Error opening file");
        return 1;
    }
    char command[100];

    while (fscanf(file, "%s", command) != EOF) {
        if (command[0] == 'A' || command[0] == 'C' || command[0] == 'F' || command[0] == 'R') {
         switch(command[0]) {
            case 'A':
            {
                int side;
                Soldier soldier;
                fscanf(file, " %d %d,%d", &side, &soldier.health, &soldier.strength);
                if (side == 1) {
                    push(&side1, soldier);
                    printf("Add soldiers to side 1\n");
                    printf("S- H:%d S:%d\n", soldier.health, soldier.strength);
                } else {
                    push(&side2, soldier);
                    printf("Add soldiers to side 2\n");
                    printf("S- H:%d S:%d\n", soldier.health, soldier.strength);
                }
                break;
            }
            case 'F':
            {
                if (isEmpty(&side1) || isEmpty(&side2)) {
                    printf("One of the sides has no soldiers left. Can't fight.\n");
                    continue;;
```

```c
    }
    isBattleHappend = 1;
    if (turn == 1) {
        Soldier s1 = top(&side1);
        Soldier s2 = top(&side2);
        int damage = damageCalculation(s1.strength, s2.strength);
        printf("1 hit  %d damage\n", damage);
        s2.health -= damage;
        if (s2.health <= 0) {
            printf("->2 has a casualty\n");
            pop(&side2);
        }
        turn = 2;
    } else {
        Soldier s1 = top(&side1);
        Soldier s2 = top(&side2);
        int damage = damageCalculation(s2.strength, s1.strength);
        printf("2 hit %d damage\n", damage);
        s1.health -= damage;
        if (s1.health <= 0) {
            printf("->1 has a casualty\n");
            pop(&side1);
        }
        turn = 1;
    }
    break; }
case 'R':
{
    int side;
    fscanf(file, " %d", &side);
    Soldier s = randomSoldier();
    if (side == 1) {
        printf("Called reinforcements to side 1\n");
        printf("S- H:%d S:%d\n", s.health, s.strength);
        push(&side1, s);
    } else {
        printf("Called reinforcements to side 2\n");
        printf("S- H:%d S:%d\n", s.health, s.strength);
        push(&side2, s);
    }
    break;
}
case 'C':
    criticalShot(&side1, &side2, &turn);
    break;
```

```
            default:
                printf("invalid command: %s\n", command);
                printf("please edit your input file and start over!");
                break;
         }
        }
        else{
                printf("Invalid command: %s\n", command);
                printf("Game ends due to invalid command.\n");
                printf("please edit your input file and start over!");
                break;
        }
    }

    if (isBattleHappend) {
        Summary(&side1, &side2);
    } else {
        printf("\n");
    }
    fclose(file);
    return 0;
}
```

How did you extend or adapt this code?

- I used struct for soldier and stack.

```
typedef struct soldier {
    int health;
    int strength;
} Soldier;

typedef struct stack {
    Soldier soldiers[100];
    int top;
} Stack;
```

- I created and used pop, push, top functions

```
void push(Stack *s, Soldier soldier) {
    if(s->top < 99) {
        s->top++;
        s->soldiers[s->top] = soldier;
    } else {
        printf("Stack is full, can't add soldier.\n");
    }
}


Soldier pop(Stack *s) {
    if(!isEmpty(s)) {
        return s->soldiers[s->top--];
    } else {
        printf("Stack is empty, can't pop soldier.\n");
        Soldier emptySoldier = {-1, -1};
        return emptySoldier;
    }
}

Soldier top(Stack *s) {
    return s->soldiers[s->top];

}
```

- I extend my code with these funcitons:

```
int isEmpty(Stack* s) {
    return s->top == -1;
}

int damageCalculation(int strength1, int strength2) {
    return (strength1 - strength2) * 0.05 + 50;
```

```c
}

Soldier randomSoldier() {
    Soldier s;
    s.health = (rand() % 100) + 1;
    s.strength = (rand() % 100) + 1;
    return s;
}

void criticalShot(Stack *s1, Stack *s2, int *turn) {
    if (*turn == 1) {
        printf("Critical shot by 1\n~> 2 has a casualty!\n");
        while (!isEmpty(s2)) {
            pop(s2);
        }
        *turn = 2
    } else {
        printf("Critical shot by 2\n~> 1 has a casualty!\n");
        while (!isEmpty(s1)) {
            pop(s1);
        }
        *turn = 1;
    }
}



void Summary(Stack *s1, Stack *s2) {
    printf("-*-*-*-*-*-finish-*-*-*-*-*-\n");
    if (s1->top > s2->top) {
        printf("Side 1 wins \n");
    } else if (s1->top < s2->top) {
        printf("Side 2 wins \n");
    } else {
        printf("It's a draw\n");
    }
}
```

What was your the development timeline?

I've been trying to implement this code since it was first assigned.(3 weeks)

In the first week, I tried to understand the requirements of this project. I read the rules given to me and thought about what I should do. I thought about how I could simplify the project and the requirements.

In the second week, I thought about the design, and I tried to create the template of the code.

And the third week, I made progress on this template, and my code was drafted, also I tested my code and corrected the errors that I made before.

TESTING NOTES

Describe how you tested your program?

- At first, I created a sample input.txt file. I ensured to enter the commands and parameters correctly. I didn't use the 'call reinforcement' command for traceability because it creates random soldiers, and it's impossible to trace.
- Then, I traced the program by making calculations on paper. I calculated the damages and I created an expected output according to the input.txt
- Then, I executed my program and, I checked to see if the results of both are the same, and when the results were not the same, I corrected the program.

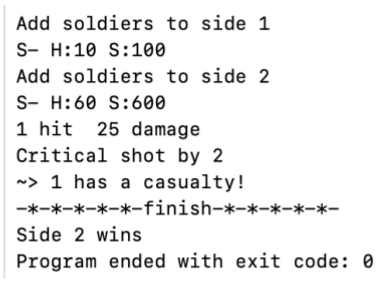
What were the normal inputs you used?

My first sample input.txt file:

```
A 1 10,100
A 2 60,600
F
```

C

Output of my first input.txt file: It gave the output as I expected

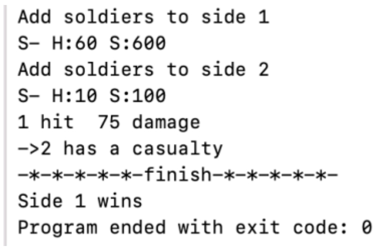


```
Add soldiers to side 1
S- H:10 S:100
Add soldiers to side 2
S- H:60 S:600
1 hit  25 damage
Critical shot by 2
~> 1 has a casualty!
-*-*-*-*-*-finish-*-*-*-*-*-
Side 2 wins
Program ended with exit code: 0
```

My second sample input.txt file:

A 1 60,600
A 2 10,100
F

Output of my second input.txt file: It gave the output as I expected

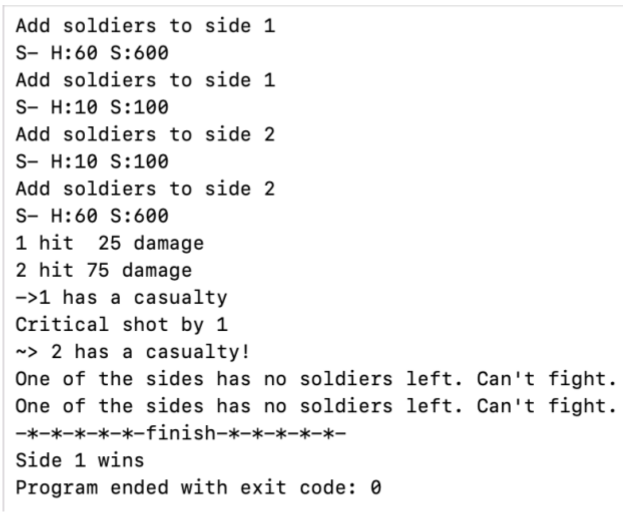


```
Add soldiers to side 1
S- H:60 S:600
Add soldiers to side 2
S- H:10 S:100
1 hit  75 damage
->2 has a casualty
-*-*-*-*-*-finish-*-*-*-*-*-
Side 1 wins
Program ended with exit code: 0
```

My third sample input.txt file:

A 1 60,600
A 1 10,100
A 2 10,100
A 2 60,600
F
F
C
F
F

Output of my second input.txt file: It gave the output as I expected



```
Add soldiers to side 1
S- H:60 S:600
Add soldiers to side 1
S- H:10 S:100
Add soldiers to side 2
S- H:10 S:100
Add soldiers to side 2
S- H:60 S:600
1 hit  25 damage
2 hit 75 damage
->1 has a casualty
Critical shot by 1
~> 2 has a casualty!
One of the sides has no soldiers left. Can't fight.
One of the sides has no soldiers left. Can't fight.
-*-*-*-*-*-finish-*-*-*-*-*-
Side 1 wins
Program ended with exit code: 0
```

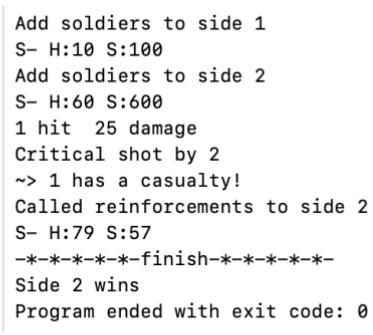
What were the special cases you tested?

I changed the input file and checked the formats' of the command to see default part of my code was working or not.

I updated my sample input.txt file again to check if it was working correctly and also **used the ‘call reinforcement’ command.**

My updated sample input.txt file: I add 'call reinforcement' command 'R 2'
A 1 10,100
A 2 60,600
F
C
R 2

Output of my updated sample input.txt file: It gave the output as I expected

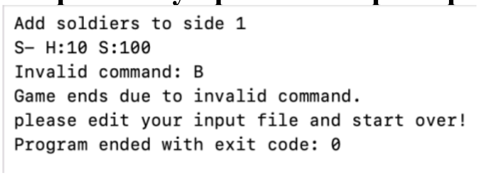


```
Add soldiers to side 1
S- H:10 S:100
Add soldiers to side 2
S- H:60 S:600
1 hit  25 damage
Critical shot by 2
~> 1 has a casualty!
Called reinforcements to side 2
S- H:79 S:57
-*-*-*-*-*-finish-*-*-*-*-*-
Side 2 wins
Program ended with exit code: 0
```

My updated sample input.txt file: I used invalid command in the file 'B 2 60,600'
A 1 10,100

B 2 60,600
F
C

Output of my updated sample input.txt file: It gave the output as I expected

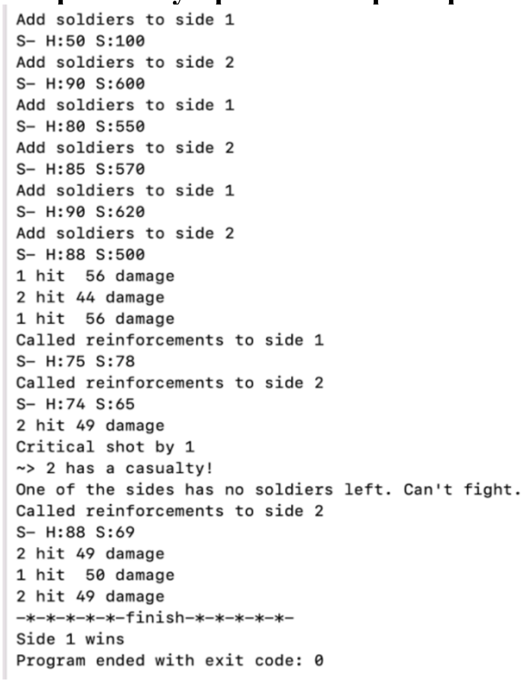


```
Add soldiers to side 1
S- H:10 S:100
Invalid command: B
Game ends due to invalid command.
please edit your input file and start over!
Program ended with exit code: 0
```

My updated sample input.txt file: I made my input file long

A 1 50,100
A 2 90,600
A 1 80,550
A 2 85,570
A 1 90,620
A 2 88,500
F
F
F
R 1
R 2
F
C
F
R 2
F
F
F

Output of my updated sample input.txt file: It gave the output as I expected



```
Add soldiers to side 1
S- H:50 S:100
Add soldiers to side 2
S- H:90 S:600
Add soldiers to side 1
S- H:80 S:550
Add soldiers to side 2
S- H:85 S:570
Add soldiers to side 1
S- H:90 S:620
Add soldiers to side 2
S- H:88 S:500
1 hit  56 damage
2 hit 44 damage
1 hit  56 damage
Called reinforcements to side 1
S- H:75 S:78
Called reinforcements to side 2
S- H:74 S:65
2 hit 49 damage
Critical shot by 1
~> 2 has a casualty!
One of the sides has no soldiers left. Can't fight.
Called reinforcements to side 2
S- H:88 S:69
2 hit 49 damage
1 hit  50 damage
2 hit 49 damage
-*-*-*-*-*-finish-*-*-*-*-*-
Side 1 wins
Program ended with exit code: 0
```

Did everything work as expected?

No, Everything did not pass as I expected, I found a solution to some of the following problems.

- The program assumes that the input file is well-formed and doesn't perform thorough input validation. It would be a good practice to add more checks to ensure that the input format is correct. For instance, you could validate that 'A' commands have the correct format, and 'F', 'R', and 'C' commands are used appropriately.

I changed the format of 'A',because my program can't read the ' ; ' sign;
The format given to me is this: A 1 10,100;60,600
Changed as:

```
A 1 10,100
A 1 60,100
```

- The program lacks comprehensive error handling. For example, when attempting to push a soldier onto a full stack or popping from an empty stack, it just prints an error message but doesn't handle the error more gracefully. But, I couldn't do anything about this.

COMMENTS

- **Describe the overall result of the assignment:**

  The overall result of the assignment is a functional C program that successfully implements the specified project goals. It simulates a battle game using a stack data structure, with the ability to add soldiers, simulate battles, call reinforcements, and execute critical shots. The code accurately processes commands and generates the expected outputs as outlined in the project requirements.

- **Was the programming project a success?**

  The programming project can be considered a success, as it accomplishes its intended objectives. It provides a working example of a battle game simulation in C, incorporating pointer basics and stack data structures effectively. The code adheres to the given project requirements and produces the expected outputs.

- **What would you do same or differently next time?**

  Same:

  1. Continue to focus on code simplicity and clarity.
  2. Continue documenting the code effectively.

  Differently:

  3. Implement comprehensive error handling for graceful handling of invalid input.
  4. Extend the functionality to support more complex war strategies and tactics.
  5. Explore alternative data structures or algorithms that may offer efficiency advantages.