CANKAYA UNIVERSITY
Software Engineering Department

Name and Surname : Barış Cem Bayburtlu
Identity Number : 202228009
Course : SENG271
Experiment : Experiment 1
Subject : Final Report
Data Due : 9 November 2023, 11:59 PM
E-Mail : c2228009@student.cankaya.edu.tr

Software Usage

There are some **requirements** to use this software, our software gets its inputs from a file system, so you need to create one to start with. To proceed, please create **a file** named *'input.txt'* in the **same folder** as our application. After creating it, you can add your desired inputs to the file to use the software. There are **4 inputs** to use.

Inputs

- With **'A'** command, give soldiers to the side you specified. You can specify soldiers' health and strength respectively. Here is an example command:
  ```
  A 1 100,59;22,987;75,647;21,123;12,312
  ```
- With **'F'** command, you can make sides to fight with a turn based system. Use it without parameters.
- With **'C'** command, you can make a side to attack with a critical damage which will cause the enemy soldier to get killed instantly. Use it without parameters.
- With **'R'** command, you can call reinforcements to a side. You can specify the side that you want to call reinforcements for. Here is an example command:
  ```
  R 2
  ```

After creating your *'input.txt'* file, open your application and software will give you your output in your terminal or if you open the application directly it will create a new file named *'output.txt'* which will give you the results.

Here, you can see an **example** *input.txt* and *output.txt* respectively.

```
A 1 100,59;22,987;75,647;21,123;12,312
A 2 100,59;22,987;75,647
F
C
R 1
R 2
```

```
add soldier to side 1
S- H:100 S:59
S- H:22 S:987
S- H:75 S:647
S- H:21 S:123
S- H:12 S:312
add soldier to side 2
S- H:100 S:59
S- H:22 S:987
S- H:75 S:647
1 hit 33 damage
Critical shot
2 has a casualty
Called reinforcements to the side 1
S- H:100 S:276
Called reinforcements to the side 2
S- H:61 S:7
```

Errors & Outputs

In this section, I will try to *explain* outputs of the inputs in this with their error handlings (if it's catched).

Let's start with what happens if tanks (stacks) are over the soldier limit. If user is using command 'A' or 'R' but if the program **exceeds** the soldier limit, it will give an error output like this:

```
Please do not add more than 100 soldiers to one side!
```

If user is using command ‘A’ or ‘R’ (in this example, we think user is using A command, if it was R, ‘soldiers’ text should be switched to ‘reinforcements’) but specified side is **not valid** (not 1 or 2), it will give an error output like this:

```
Because you didn't enter a valid side number (1 or 2), we were unable to add your soldier(s). Please enter a correct side number!
```

If user is using command ‘A’ but specified **wrong integers** for health or strength, it will give an error output like this:

```
You entered stats invalid for a soldier, please check your soldier stats!
```

But if it isn't over the soldier limit nor incorrectly specified, it will give an output like this (in this example, I used “`A 1 10,10`“ command):

```
add soldier to side 1
S- H:1 S:1
```

Using command “`R`” **without any problems** should give us an output like this (in this example, I used “`R 1`” command):

```
Called reinforcements to the side 1
S- H:19 S:638
```

If user is using command ‘F’ or ‘C’ but if there is **no soldier** in tanks (in this example, we think that first side is empty), it will give an error output like this:

```
You forget to add soldiers to the first side, please add some!
```

But if stacks are **not empty**, using command “`F`” will give an output like this:

```
1 hit 50 damage
2 has a casualty
```

and also, using command “`C`” will give an output like this:

```
Critical shot
1 has a casualty
```

If there is no file named ‘input.txt’ in the folder, it will give an error output like this:

```
Error opening file.
```

If any team wins, it will give an output like this (in this example, i decided to make winning side the first side):

```
Team 1 wins
```

Implementation

For this project, I started implementing the stack system into C. I didn't know how to implement it at first so I had to check out on the internet to get more information. After getting more info, I created a Base for soldiers and a Stack system for storing the soldiers.

```
struct Base {
  int health;
  int strength;
};

struct Stack {
  struct Base *data;
  int top;
  int capacity;
};
```

After this, I started thinking about how to use this stack and base. When I researched it a little bit, I had a new problem to encounter. Memory. I implemented an initialize method for stacks so the program should start without any problems (especially memory leak problems). I implemented the method like this:

```
struct Stack *initialize(int capacity) {
  struct Stack *stack = (struct Stack *)malloc(sizeof(struct Stack));
  if (!stack) {
    perror("Stack memory allocation failed");
    exit(1);
  };

  stack->data = (struct Base *)malloc(capacity * sizeof(struct Base));
  if (!stack->data) {
    free(stack);
    perror("Stack data memory allocation failed");
    exit(1);
  };

  stack->top = -1;
  stack->capacity = capacity;
  return stack;
};
```

After implementing this, I need to free the memory as well. I did it with another method named destroy which looks like this:

```
void destroy(struct Stack *stack) {
  free(stack->data);
  free(stack);
};
```

Then, I implemented the isEmpty, isFull, top, pop and push methods. I used 1 and 0 to include if it's empty or not in isEmpty. I could have done that as well with boolean but it is better and easier to do with integers. I only check the top element if it's -1 or not (if there is none, it will be -1 which is the index of the top). I implemented the method like this:

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

Similar to isFull, I also used integers to work like booleans. It checks the top element's index and if it's one less than its capacity, it gives 1 (which is full). I implemented it like this:

```
int isFull(struct Stack *stack) {
  return stack->top == stack->capacity - 1;
};
```

After that, I implemented the top, pop and push methods. Code is actually very straightforward. I create/change the index of top or change values inside of it.

```
void push(struct Stack *stack, int health, int strength) {
  if (stack->top == MAX_STACK_SIZE - 1) {
    printf("Stack is currently full, couldn't push. %d %d \n", health, strength);
  } else {
    stack->data[++stack->top].health = health;
    stack->data[stack->top].strength = strength;
  }
};

struct Base pop(struct Stack *stack) {
  struct Base emptyBase = { .health = 0, .strength = 0 };
  if (isEmpty(stack)) {                                    // checking empty
    printf("Stack is empty, cannot pop.\n");
    return emptyBase;                                      // returns empty base
  } else {
    return stack->data[stack->top--];
  }
};

struct Base top(struct Stack *stack) {
  struct Base emptyBase = { .health = 0, .strength = 0 };
  if (isEmpty(stack)) {
    printf("Stack is empty, there is no top.\n");
    return emptyBase;
  } else {
    return stack->data[stack->top];
  }
};
```

I get every line's first character to see if it is command/input related. I put it into a while and with if/else statements, I functionize those commands.

```
char type;
while (fscanf(fp, "%c", &type) != EOF) {
    if (type == 'A') {
      ...
    }
};
```

For the fight command, I got top values of two stacks. I choose which team to become a victim or attacker. Also I apply damage to the health of the victim stack's top soldier's health. After dealing damage, it checks if the victim's health is less than 1. If it is, that means sadly our victim is dead :(For dealing with dead bodies, I use the pop method to delete it from the stack itself.

```
struct Stack *team1 = initialize(100);
struct Stack *team2 = initialize(100);

struct Base attacker = top(team1); // team 1 attacks
struct Base victim = top(team2);   // team 2 gets attacked
int damage = (attacker.strength - victim.strength) * 0.05 + 50;
victim.health -= damage;

if (victim.health < 1) {
  pop(team2);
};
```

If the command is critical attack, I directly pop the stack. So that means the soldier automatically dies.

```
...
int damage = (attacker.strength - victim.strength) * 0.05 + 50;
printf("Critical shot\n");

pop(team2);
...
```

For reinforcement function, I chose randomly health and strength. After I get the values, I push them to the selected team which is given by the player.

```
int num;
fscanf(fp, " %d\n", &num);

int random_health = rand() % 100 + 1;
int random_strength = rand() % 999 + 1;

if (num == 1) {
  push(team1, random_health, random_strength);
} else if (num == 2) {
  push(team2, random_health, random_strength);
};
```

Development Timeline

Design: **4 days**
Development: **7 days**
Testing: **4 days**

Software Design

This software is a game that has two sides that we can call sides, tanks. We put soldiers inside of these tanks and we will make them fight with our commands.

Problem

In this experiment, a **game** is requested from us. We are expected to implement a soldiers' fight game in which there are two sides of soldiers and they are going to *challenge* each other for a battle.

Solution

Since all soldiers will have different characteristics, the battle will give us the result of the battle. The fighters of the current battle will be selected from the sides and they will fight one by one. When a soldier is dead, we will select another soldier (who is the top soldier) on that side to continue their fight. This will happen until there are no soldiers left on one side and the other side will be the winner and we can give an output saying that side won. Since we are going to make them fight one at a time, we can only check the top of the stack to see a soldier's details and calculate the damage.

Algorithm

I used a linked list stack algorithm to keep and control the soldiers. With the pop and push methods in it, we will be able to control the entry and exit of the soldiers in the stack we have created. A life cycle of our program should look like this:

1. Make initialization.
    a. Open input: *'input.txt'*, output: *'output.txt'*.
    b. Seed randomization method with time (for reinforcement).
    c. Create two stacks which are named **1** and 2.
2. For every command in input.txt
    a. If the command is **A**,
        i. Add the *soldiers* as specified.
        ii. Write output of *soldiers* that is added.
    b. If the command is **F**,
        i. Get the turn and specify attacker as which side got the turn
        ii. Get attacker and victim stack's top element and calculate damage
        iii. Deal damage to the victim stack's top element
        iv. Give output about it
    c. If the command is **C**,
        i. Get the turn, delete one soldier from the victim stack.
        ii. Give output about it
    d. If the command is **R**,
        i. Create a random soldier and add it to the specified side
        ii. Give output about the soldier
3. If game finishes (one stack is empty), give us an output about winning side

Software Test

In the testing section, I tried to do my best to solve bugs which I mentioned in the last document about testing and I made lots of bugs to be clear.

Bugs & Software Reliability

Our application is creating its memory and its place for it. Also our program is not always open, it gets its needs (inputs), gets the job done and exits the program so I don't think we will get a memory leak problem. I fixed all the bugs that were found in the testing section.

Software Extendibility and Upgradeability

We can add so many things into the game and we can make it as detailed as possible. Let me write some of my ideas about it.

- New fight techniques
  - Bomb/bazookas: Area damage (will affect all soldiers) but it will damage a little less than normal fight (or it can be better)
  - Stun attack: Stuns the other side for a specified time
  - Shield: We can count this as a defensive attack, we can add shields and this attack can create a shield for themself to get protected from a specified damage next turn
- Soldier types
  - We can do that by modifying health and strength values with specified values we want and after that we can give them a name to make these soldiers in a standard value
- Soldier effects (buffs)
  - x2 damage: Self-exp, buff makes the damage x2

Our program is approximately **260 lines** and **12 hours** of work time.

Software Input Tests

In the normal inputs section, I will give my program the normal inputs (normally an user's input without any errors) to see how it reacts. I also wrote some comments about my plans.

Giving this as our input (without any winning):

```
A 1 100,59;22,987;75,647;21,123;12,312
A 2 100,59;22,987;75,647
F
C
R 1
R 2
```

Will return this as our output *as expected* (expected - no winning side, only lowering health):

```
add soldier to side 1
S- H:100 S:59
S- H:22 S:987
S- H:75 S:647
S- H:21 S:123
S- H:12 S:312
add soldier to side 2
```

```
S- H:100 S:59
S- H:22 S:987
S- H:75 S:647
1 hit 33 damage
Critical shot
2 has a casualty
Called reinforcements to the side 1
S- H:18 S:211
Called reinforcements to the side 2
S- H:36 S:125
```

Giving this as our input (with winning):

```
A 1 1,1
A 2 2,2
C
```

Will return this as our output *as expected* (expected - one side is winning):

```
add soldier to side 1
S- H:1 S:1
add soldier to side 2
S- H:2 S:2
Critical shot
2 has a casualty
Team 1 wins
```

Handling Errors

You can see that in edge cases I have some problems. In this section, I will talk about how I will fix them and how my program handles its own errors.

Stack's Availability Status

When the first team attacks the second team, I check if second team's stack is empty or not with this method:

```
if (isEmpty(team2)) {
  printf("Team 1 wins\n");
  fprintf(fo, "Team 1 wins\n");
};
```

Method itself returns a boolean type int (0 or 1) and I can use it in if conditions.

Also, I checked when I pop the stack with same method:

```
struct Base pop(struct Stack *stack) {
  struct Base emptyBase = { .health = 0, .strength = 0 };
  if (isEmpty(stack)) {
    printf("Stack is empty, cannot pop.\n");
    return emptyBase;
  } else {
    return stack->data[stack->top--];
  }
};
```

I also checked if stack's size is more than we expected with this code:

```
void push(struct Stack *stack, int health, int strength) {
  if (stack->top == MAX_STACK_SIZE - 1) {
    printf("Stack is currently full, couldn't push. %d %d \n", health, strength);
  } else {
    stack->data[++stack->top].health = health;
    stack->data[stack->top].strength = strength;
  }
};
```

`MAX_STACK_SIZE` is defined with `#define MAX_STACK_SIZE 100`

File Error Handling

I have 'fp' as our input file and 'fo' variables as our output file. I only need to control the input file because if there is no output file, it creates one itself and if there is no input file, the program wouldn't start because it is a requirement.

```
fp = fopen("input.txt", "r");
fo = fopen("output.txt", "w");
if (fp == NULL) {
  printf("Error opening file.\n");
  return 1;
}
```

Memory Control

I control programs memory by initializing two stacks and allocating its memories before inputs. I allocated stacks and its data separately to not create any problems with memory.

```
struct Stack *initialize(int capacity) {
  struct Stack *stack = (struct Stack *)malloc(sizeof(struct Stack));
  if (!stack) {
    perror("Stack memory allocation failed");
    exit(1);
  }

  stack->data = (struct Base *)malloc(capacity * sizeof(struct Base));
  if (!stack->data) {
    free(stack);
    perror("Stack data memory allocation failed");
    exit(1);
  }

  stack->top = -1;
  stack->capacity = capacity;
  return stack;
};
```

Comments

At my first implementation, I forget to break out of the while loop if *any side wins*. I detected the problem and fixed it. I fixed lots of typos and all of the error handling which they were breaking the program.

If I do this project again, I would create a proper fight function to use it but I couldn't find a solution using such function inside of a while loop so I hardcoded it and I would create in a better way because it is a little bit unreadable right now. I don't say it looks bad but it may be confusing to the new people who want to look at this.