Rock Paper Scissors

Problem Description

The full description can be found in this site's repository.

You are provided a list of data *input.txt* that contains a list of Rock Paper Scissors games. Each line contains the enemy's move (`A` for Rock, `B` for Paper, and `C` for Scissors) and our move (`X` for Rock, `Y` for Paper, and `Z` for Scissors).

This game comes in two parts:

1. The score for a round is the score for the *shape you selected* (1 for Rock, 2 for Paper, and 3 for Scissors), plus the score of the outcome round (0 if you lose, 3 if you draw, and 6 if you win).
2. The setup of the game changes. The second column indicates how the round needs to end. `X` meaning a loss, `Y` meaning a draw, and `Z` meaning a win.

Each part's answer will be the total score if everything follows the guidelines. Submit your final answer in the form `flag{Part1_Part2}` where `Part1` and `Part2` are the answers for each part.

> *If you're still confused on what's going on here, check the full problem description. It runs some sample testing cases for the challenge so you understand how to compute the totals.*

Walkthrough

For each problem, we'll follow the same template. We'll break it down into the following steps:

1. Choose our data structure for the challenge based on our end goal.
2. Open the file and initialize our necessary structures.
3. Read each line of the file and process the data into the data structure.
4. Process the data structure to get our answer.

We'll work on each of these in turn.

Choosing a Data Structure

For this challenge, our end goal is to get the total score across the round. Since each round is independent of the other, we don't need to store each round's score. All we need is a running total score for each round. **We can do this with a single (unsigned) integer for each round.**

We'll find that when we do this challenge, it's easier to do these parts independently. There's not much comparable information other than reading the file, so we'll handle each part separately.

Initialization

In this stage, we'll prepare to read the data. We'll do this the same way we do this in *CalorieCounting*.

```
// open file
FILE* fp = fopen("input.txt", "r");
```

```
if (fp == NULL)
    exit(EXIT_FAILURE);

// initialize reader variables
char* line = NULL;
size_t len = 0;
```

Note that this time, we don't need the `ssize_t read` variable. We know the size of the data we're reading, so we don't need to store it.

Then, we just need to initialize the total score to 0.

```
uint32_t total_score = 0;
```

Reading the Data

We'll handle this one part at a time.

Part 1

For the first part, we are told the second value is the value the opponent plays. When we compare the two values, it will be easier to convert them to integers first. We can use the offset from `A` or `X` as the value for each player (meaning every move corresponds to a `0`, `1`, or `2`).

> *Why do we think to do this?* Numeric comparisons are typically easier to wrap our head around. Rather than making an `if` statement with many checks, we can check for patterns with integers.

We can do the same thing for the user score as well. Since the user's move can be seen as `1 + offset`, we can add this offset to the total score.

We'll use the same logic as the previous challenge to loop through each line:

```
while (getline(&line, &len, fp) != -1) { ... }
```

Each line, we can get the self and enemy scores. We'll convert them to integers

```
int enemy = (int)line[0] - 'A';
int self = (int)line[2] - 'X';
```

We'll add `1 + offset` for our move's portion of the total score.

```
total_score += (self + 1);
```

Then, we need to handle the three cases. Let's think about what these cases are. Let's make a table subtracting the `enemy` score from `self`.

Enemy / Self	0	1	2
0	0 [D]	1 [W]	2 [L]
1	-1 [L]	0 [D]	1 [W]
2	-2 [W]	-1 [L]	0 [D]

In this case, we notice the following:

- *Wins* happen when the result is `1` or `-2`.
- *Losses* happen when the result is `-1` or `2`.
- *Draws* happen when the result is `0`.

We can simplify this into the following checks:

```
int difference = self - enemy;

// draw case
if (difference == 0) {
    total_score += 3;
    continue;
}

// win case
if (difference == 1 || difference == -2) {
    total_score += 6;
    continue;
}

// loss case
if (difference == -1 || difference == 2) {
    total_score += 0;
    continue;
}
```

As a sanity check, we should never the code after all three checks. I put a `printf("Bad\n");` statement at the end and make sure this doesn't get printed.

For this part, once all the lines are iterated, we can return the total score for that round.

Part 2

For the second part, we are told the second value is the outcome of the game. `X` is a loss, `Y` is a draw, and `Z` is a win.

There are many ways to approach this challenge. What made the most sense to me was to differentiate the win, loss, and draw cases. This time, because we're not comparing the two values, it doesn't make too much

sense to convert them to integers.

We'll use the same logic as the previous challenge to loop through each line:

```
while (getline(&line, &len, fp) != -1) { ... }
```

Each line, we can get the self and enemy scores. We'll leave them as characters.

```
char enemy = line[0];
char self = line[2];
```

Since `self` represents the score this time, we can add the score results to the total score. The score is three times the value of the score (since the score is `0`, `1`, or `2`).

```
total_score += 3 * ((int)self - 'X');
```

Now, we can handle the three cases.

- If the result is a loss (`X`), and the enemy put `A`, we put `C` (score of `2`). For `B`, we put `A`; for `C`, we put `B`.

  ```
  if (self == 'X') {
      if (enemy == 'B' || enemy == 'C')
          total_score += ((int)enemy - 'A');
      else if (enemy == 'A')
          total_score += 3;
      else
          printf("Bad 1\n");
  }
  ```

- If the result is a draw, we can just use their score for ours.

  ```
  else if (self == 'Y') { // total score is what enemy put
      total_score += (((int)enemy - 'A') + 1);
  }
  ```

- If the result is a win, we'll follow a similar strategy to the loss case.

  ```
  else if (self == 'Z') {
      if (enemy == 'A' || enemy == 'B')
          total_score += (((int)enemy - 'A') + 2);
      else if (enemy == 'C')
          total_score += 1;
  ```

```
        else
            printf("Bad 2\n");
    }
```

- We shouldn't ever hit the `else` case, but we'll put one just in case.

```
    else {
        printf("Bad 3\n");
    }
```

Once we've iterated the lines, we can return the total score.

Putting it Together

I chose to make a function for each part. We'll define the `main` function to call each part's function and print the result.

```
int main(void)
{
    uint32_t first = part01();
    uint32_t second = part02();

    // print the total score
    printf("flag{%d_%d}\n", first, second);

    return 0;
}
```

If we run this, we'll get the following output:

```
$ gcc -o solve solve.c && ./solve
flag{12794_14979}
```

Full Solution

The full solution is below:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

uint32_t part01()
{
    // open file
    FILE* fp = fopen("input.txt", "r");
```

```
    if (fp == NULL)
        exit(EXIT_FAILURE);

    // initialize reader variables
    char* line = NULL;
    size_t len = 0;

    // read the file
    uint32_t total_score = 0;
    while (getline(&line, &len, fp) != -1) {
        int enemy = (int)line[0] - 'A';
        int self = (int)line[2] - 'X';

        total_score += (self + 1); // score for self move
        int difference = self - enemy;

        // draw case
        if (difference == 0) {
            total_score += 3;
            continue;
        }

        // win case
        if (difference == 1 || difference == -2) {
            total_score += 6;
            continue;
        }

        // loss case
        if (difference == -1 || difference == 2) {
            total_score += 0;
            continue;
        }

        printf("Bad\n");
    }

    // print the total score
    return total_score;
}

uint32_t part02()
{
    // open file
    FILE* fp = fopen("input.txt", "r");
    if (fp == NULL)
        exit(EXIT_FAILURE);

    // initialize reader variables
    char* line = NULL;
    size_t len = 0;

    // read the file
    uint32_t total_score = 0;
```

```
    while (getline(&line, &len, fp) != -1) {
        char enemy = line[0];
        char self = line[2];

        total_score += 3 * ((int)self - 'X'); // score for match result

        if (self == 'X') {
            if (enemy == 'B' || enemy == 'C')
                total_score += ((int)enemy - 'A');
            else if (enemy == 'A')
                total_score += 3;
            else
                printf("Bad 1\n");
        } else if (self == 'Y') { // total score is what enemy put
            total_score += (((int)enemy - 'A') + 1);
        } else if (self == 'Z') {
            if (enemy == 'A' || enemy == 'B')
                total_score += (((int)enemy - 'A') + 2);
            else if (enemy == 'C')
                total_score += 1;
            else
                printf("Bad 2\n");
        } else {
            printf("Bad 3\n");
        }
    }

    return total_score;
}

int main(void)
{
    uint32_t first = part01();
    uint32_t second = part02();

    // print the total score
    printf("flag{%d_%d}\n", first, second);

    return 0;
}
```