Bandit Baiting

Objective
Give practice with Strings in C.
Give practice with Dynamic Memory Allocation in C.

Story


The local raccoons are infamous for stealing pet food. Your employer—KAT Pet Shop—is leveraging you to help find out where the raccoons are storing the stolen food. The hope is that by finding the food thieves the KAT Pet Shop will have some clout when it comes to providing quality and care for pet services.

For now you will work on the backend of a system that will manage a collection of raccoon baits placed in several strategic locations. Your system will also be used to monitor the amount of bait taken, to help narrow down the possible location of the den of thieves.

You will have food placements with unique Identifiers that will send information to your database about its food level. Your backend will receive the Identifier and the food level. Your goal will be to compute any possible loss of food.

One problem you've noticed is that KAT Pet Shop has hired some employees to refill the bait locations. Refilling the bait means that sometimes the level will increase. However, if the level decreases this was definitely a raccoon. It might also be that the raccoon took some food and the location was refilled, but sadly you might not have reports frequent enough to capture this behavior.

The raccoons may have stolen more food than you prove using your data, since the refill and stealing could have both happened between reports. At the end you want a summary for each placement as to the least possible amount of food taken. These values will hopefully help narrow down the location of the den.

Problem
You will be given a series of data. Each line of data is composed of an ID and food level. You will compute how much food could have been taken from that data point.

At the end of your program you will print a list of all the IDs and the sum of food that could have been taken for each one.

Input
Each line of input will contain placement data described using three tokens:

- A string, ***N***, representing the name of the location 19 characters (none of which will be whitespace)
- An integer, ***D*** (-1,000,000,000 ≤ ***D*** ≤ 1,000,000,000), representing an Identifier for the bait placement within the location.

- A non-negative integer, ***F*** (***F*** ≤ 1,000,000,000), representing the food level at the time of the reading.

The tokens will be separated by a single space. Note that each location will have at most 1 bait placement with any possible Identifier.

Input will be terminated by a line with the tokens "`END -1 -1`"

Output

For each data point with exception of the "`END -1 -1`":

- If the ID within the location is new, print a line with the phrase "`New placement.`"
- If the ID within the given location has been seen before, output the least amount of food that could have been taken by a raccoon since the last report from the placement.

The data points need to be reported in the order passed in.

At the end of the program print out the least amount of food that could have been taken from each location. The location can be printed in **ANY** order.

Sample Input	Sample Output
HEC 1 5 StudentUnion 1 45 HEC 1 10 StudentUnion 1 25 CB1 5 20 HEC 2 5 CB1 5 15 HEC 1 15 END -1 -1	New placement. New placement. 0 20 New placement. New placement. 5 0 HEC 0 StudentUnion 20 CB1 5
HouseAlice 99 20 HouseBob 4 20 HouseAlice 99 0 HouseAlice 99 20 HouseAlice 99 10 HouseAlice 53 20 HouseAlice 53 10 END -1 -1	New placement. New placement. 20 0 10 New placement. 10 HouseAlice 40 HouseBob 0
MyPool 1 100 MyPool 2 99 Gutter 3 15 Gutter 3 7 END -1 -1	New placement. New placement. New placement. 8 Gutter 8 MyPool 0

Sample Explanation

FOR CASE 1
There are 4 placements.

2 of the placements are in HEC. The numerical IDs of the placements in HEC are 1 and 2.

1 placement is located in CB1. It has ID 5.

The last placement is in StudentUnion with ID 1.

Note that several placements could have the same integer ID, but different locations mean they are different.

placement 1 in HEC goes from 5 to 10 to 15. It never decreases, so it might have never had any food taken from it.

placement 2 in HEC only has 1 reading, so there is not enough information to know whether any food has been taken.

All of HEC has had 0 food units taken.

placement 5 in CB1 goes from 20 to 15. We guarantee that at least 5 units of food have been taken.

All of CB1 has had 5 food units taken.

placement 1 in StundentUnion goes from 45 to 25. We guarantee that at least 20 units of food have been taken.

All of StudentUnion has had 20 food units taken.

FOR CASE 2
There are a total of 3 placements.

AliceHouse has 2 placements
1. placement 99
2. placement 53

placement 99 in AliceHouse goes from 20 to 0 to 20 to 10.
- Going from 20 to 0 means that the raccoons took at least 20 units of food.
- Going from 0 to 20 means that the raccoons might have taken nothing.
- Going from 20 to 10 means that the racoons took at least 10 units of food.

placement 53 in AliceHouse goes from 20 to 10.
- Going from 20 to 10 means that the raccoons took 10 units of food.

The total amount taken from AliceHouse is 30 + 10 = 40.

BobHouse has 1 placement.

FOR CASE 3

There are 3 placements

- 2 at MyPool with IDs of 1 and 2
- 1 at Gutter with ID of 3

The MyPool placements all have only 1 reading each, so they won't have any noticeable raccoons. The Gutter placement has 2 readings that show a drop of 8 units. So the result is that MyPool has 0 units taken and Gutter has 8 units taken.

Hints

Placement Struct: I recommend using a placement struct that stores the identifier number, the previous food level, and optionally how much food has been taken at the corresponding placement.

Location Struct: I recommend using a location struct that tracks its name and a list of existing Placements, and optionally the sum of how much food has been taken from the corresponding location.

Array List of Locations: Since you don't know how many locations you can have I recommend using an Array List to hold on to them.

Linear Search: I recommend using a linear search (loop through all locations in order from start to end) to walk through the locations looking for the name of your desired location. If you don't find the location in the search, you can append it to the end.

BIG Numbers: Your answer can be well over the size of a maximum int. Food level and sums of food should be stored in an integer with more precision; use the `long long int`. Below is an example of how you can create and use a `long long int`

```
int main() // The main function
{
   long long int x; // Declare the long long int

   scanf("%lld", &x); // Read in the long long int
                      // Use the long long decimal format specifier

   x *= 1000000000; // Multiply by big number

   printf("%lld\n", x); // Print the long long int

   return 0; // Exit program
}
```

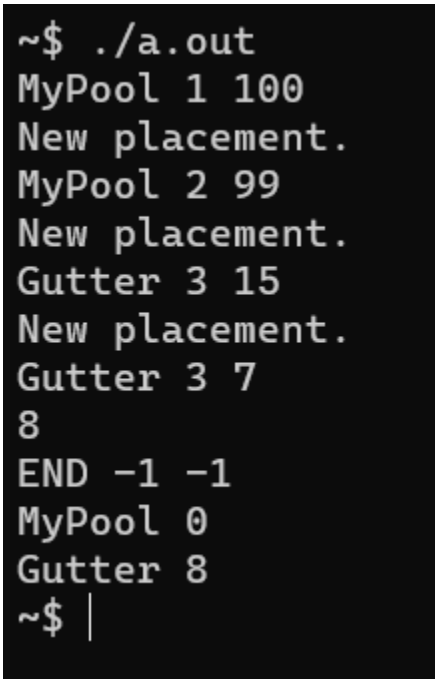
strcmp: You can use strcmp to check if a name of a location matches an existing one.

NO PROMPTS: Do NOT prompt for input. Do not print messages like, "Please enter the number of cats:" or the like.

NO LABELS: Do NOT label the output. Do not print messages like, "The cat in cozy cage 1 is…".

Hints Continued on the next page…

Input Output Syncing: You DO NOT need to sync the input and output. Below is an example of what the execution of the program could look like,



The lines typed into the program are the 1st ("`MyPool 1 100`"), 3rd, 5th, 7th, and 9th. All other lines were output by the programs or shell prompts. Note that the lines with the location sums can be printed in any order.

Grading Criteria

- Read/Write from/to **standard** input/output (e.g. scanf/printf and no FILE *)
    - 5 points
- Good comments, whitespace, and variable names
    - 15 points
- No extra input output (e.g. input prompts, “Please enter the number of cats:”)
    - 5 points
- Loops to the end input
    - 5 points
- Check if a given location already exists
    - 5 points
- Can add a new location
    - 5 points
- Checks if an ID exists within a given location (w/o looking at IDs from other locations)
    - 5 points
- Stores the last food level for each Unique ID for each location
    - 5 points.
- Programs will be tested on 10 cases
    - 5 points each

No points will be awarded to programs that do not compile using “gcc -std=gnu11 -lm”.

Sometimes a requested technique will be given, and solutions without the requested technique will have their maximum points total reduced. For this problem use dynamic memory. ***Without this programs will earn at most 50 points!***

Any case that causes a program to return a non-zero return code will be treated as wrong. Additionally, any case that takes longer than the maximum allowed time (the max of {5 times my solutions time, 10 seconds}) will also be treated as wrong.

No partial credit will be awarded for an incorrect case.