CMPE230 Project 1 - Favor for the Ringmaster

Bahadır Kuşcan, Yusuf Akdoğan

1) Purpose and Objectives

The aim of the project was to write a program that takes sentences as input from the user which are either questions or instructions to perform actions under optional conditions. Those actions consisted of subjects buying and selling stuff to each other and moving to different locations. The conditions would be about the amount of certain items that subjects held or the locations of certain subjects. Questions that user could ask would retrieve information about the inventories or whereabouts of subjects. So the program was supposed to have 2 main functionalities: Parsing the input and storing/managing data. Parsing functionality would need to parse the given input correctly to obtain a meaningful set of instructions while checking if the input is given in the correct format. Storing/managing functionality would need to keep track of inventories and locations of the subjects and perform the instructions obtained from parsing after checking if they are performable.

2) Implementation Process

After determining the objectives, first thing to do was planning our work. We were going to either implement 1 main functionality per person or build the whole program together. Working together seemed more logical to us since we had plenty of time together thanks to being roommates, we both wanted to work on both mediums to learn their specifics and we thought that 2 people would be more effective in avoiding bugs and finding solutions to the encountered problems as we progress.

We decided to take on parsing first since the structure of the instructions to be passed would be determined in this process. We analysed the input format and decided to go word by word. Every possible word that could come after each word was definitive so we drew a tree to model it. With the help of this model we wrote our code step by step considering each route. This model was also useful for checking validity at each step.

As we moved on our instructions structure began to form. After some brainstorming we came up with a reasonable way to store and pass the data. We decided to use 5 different structs: Person, Action, Condition, Action Sequence and Condition Sequence.

Person struct would store the name, location, item names (array), item amounts (array) and the number of different items of a subject. Name and the amount of a certain item would have the same index in their respective arrays. Each subject’s Person struct would be stored in a Person array in the main function, we would use this array to get the Person struct of a subject through its name or store a new subject.

Action struct would represent a single action and it would store the names of the subjects involved in the action (array), type of the action (buy/sell/go to/buy from/sell to), objects involved in the action (items if buy/sell location if go to) (array), amounts of the items to be traded (array), total number of subjects involved, total number of objects involved and the name of a potential trader (x in sell to x or buy from x). Just like in Person struct, name and the amount of the items would have the same index in their respective arrays. Action structs would be stored in action sequences and they would be used to perform actions.

Condition struct was almost the same as action struct, with the only difference being that condition struct did not store a potential trader. Also condition types were “has”, “has more than”, “has less than” and “at”.

Action sequence struct would represent a connected sequence of actions which are seperated by the “and” keyword. It would store the action structs it consisted of (array) and their count. Action sequences were stored in an array in the main function, which would be reset for each line of input. Condition sequences had the same logic.

Each action sequence would have the same index as the condition sequence it was connected to in their respective arrays in the main function (apart from the potential trailing action sequence which might not have a succeeding condition sequence, this possibility was handled with storing and comparing the sequence counts).

After finishing the instruction parsing, we moved on to the control and application of actions. Since the tricky part of this task was to find and create a suitable structure of instructions and it was already done in parsing, we didn’t have much left to do. We iterated the sequence arrays, checked each condition in the condition sequence if they were true, and if all of them were true we moved on to the corresponding action sequence. We used 2 functions to perform the actions in the sequence: one of them would directly adjust the inventory/location and the second one would call the other after checking if the action is performable.

Lastly, we moved on to the question parsing. We obviously differentiated the question sentences from instruction sentences by the checking the presence of the question mark (questions without a question mark would be handled automatically because of the used keywords). We used the same method in the instructions parsing and went word by word, checking the possible next words at every step. Since the format of the question sentences were pretty strict, we didn’t need a tree this time. All the answers to the given questions were obtained by iterating each person in the main function's array and searching.

After all the implementation was done, we finally continued with debugging and trying edge cases.

3) Challenges

The most common problem we encountered in the debugging process was segmentation faults. This problem appeared in various ways. One of them was quite interesting: We wrote and debugged our program in CLion IDE, at some point we decided to run the program in a docker image with GCC. When we did, we got a segmentation fault from an input that worked perfectly fine in CLion. So we had to use GDB to debug and find the issue. It turns out that some string functions in <string.h> cause an error in GCC but not in CLion when they accept NULL pointer as a parameter. We had to fix this issue by additional checks before passing values to string functions.

Dealing with the wide variety of edge cases that result in invalid inputs was quite compelling. We had to check every possibility and every necessity in each step of parsing. Some of the edge cases caused segmentation faults such as the blank input. Overcoming the issues of edge cases required quite a lot of patching.

Another major problem we encountered was loss of data. We needed to use functions that create some structs/arrays and return the pointer to them. We knew that local variables were a potential problem but because of our lack of knowledge of C memory layout at the time, we needed to search the internet to learn more about memory allocation and find a solution which was to allocate memory for local variables. Another reason for loss of data was our lack of knowledge of reallocation. We learned that realloc function would allocate the new memory elsewhere, copy the values there and erase the values from the previous location. So calling realloc with a memory location that was pointed by multiple pointers was a problem. We needed to change the return values and parameter types of some functions to overcome this problem.

4) Usage and Examples

After the program is run, a user input from the console is needed to proceed. User gives a sentence, program outputs “INVALID” to the console if the sentence doesn’t match the format. If the sentence is an instruction sequence, “OK” is outputted. If the sentence is a question, the answer to that question is outputted. User can ask for the amount of a specific item that a subject(s) own, list of items that a subject owns with their amounts or the current location of a subject. User can make subjects buy/sell specified number of items from/to each other or from/to an infinite source and move to specified locations. User can choose these actions to depend on some specified conditions. Those conditions can be some subjects having more/less/exactly a specified number of some specified items or some subjects being at a specified location. After each input, user is asked for a new input again and again until user inputs “exit” to the console.

Example 1 : General expectations from the program.

```
$ make

$ ./ringmaster

>> Frodo go to Rivendell

OK

>> who at Shire ?

NOBODY

>> Frodo       where ?

Rivendell

>> Frodo sell 2 bread to Sam

OK

>> Sam total ?

NOTHING (Explanation: Frodo does not have enough bread to sell to Sam.)

>> Frodo and Sam buy 3 bread and 2 ring

OK

>> Sam sell 1 ring to Sauron

OK

>> Sauron where ?

NOWHERE

>> Sam total ?

3 bread and 1 ring
```

```
>> Frodo buy 2 palantir and Frodo go to NOWHERE
INVALID (Explanation: Frodo can not go to a place which is a keyword.)
>> Frodo total palantir ?
0
>> Frodo total Bread ?
0
>> Frodo go to mount_doom and Gandalf buy 3 arrow if Sauron has 1 ring and Frodo at Rivendell
OK
>> who at mount_doom ?
Frodo
>> Legolas buy 100 hairclip from Arwen if Galadriel at NOWHERE
INVALID (Explanation: Even though Galadriel is at NOWHERE, her initial location, NOWHERE is a keyword and therefore cannot be specified in sentences or questions as entities. But keywords such as NOWHERE, NOTHING, NOBODY can be at the program's output.)
>> Legolas total hairclip ?
0
>> Legolas buy 100 hairclip from Arwen if Gandalf has more than 2 arrow
OK
>> Legolas total hairclip ?
0 (Explanation: Even though the previous sentence is correct, Arwen doesn't have enough hairclips to give to Legolas.)
>> exit
```

Example 2 : Basic transaction process with one or more items.

```
$ make
$ ./ringmaster
>> Galadriel and Elrond and Cirdan buy 100 Nenya and 100 Narya and 100 Vilya
OK
>> Balrog and Saruman buy 10 Vilya and 10 Narya from Cirdan
OK
>> Cirdan total ?
100 Nenya and 80 Vilya and 80 Narya
>> Balrog total ?
10 Narya and 10 Vilya
```

```
>> Balrog and Cirdan sell 10 Nenya to Legolas
OK
>> Legolas total Nenya ?
0 (Explanation: The previous statement is correct, but since Cirdan doesn’t have any Nenya rings, the action is canceled for both.)
>> Balrog and Cirdan sell 10 Vilya to Legolas
OK
>> Legolas total Vilya ?
20
>> Balrog and Cirdan total Narya ?
90
>> exit
```

Example 3 : Entity names, question words and keywords are case-sensitive.

```
>> someone go to somewhere and somebody buy 1 somethings and 2 SoMeTHiNG if someone has less than 2 somethings and 1 nothinG and someone go to home_of_someone if someone has 10 something
OK
>> someone where ?
somewhere
>> somebody where ?
NOWHERE
>> who at home_of_someone ?
NOBODY
>> everyone and somebody and someone go to everywhere and this is invalid if everyone at everywhere
INVALID
>> everyone where ?
NOWHERE
>> exit
```

Example 4 : When there are not enough items, even if the other items are sufficient, the action is not carried out.

```
>> Galadriel and Elrond buy 3 bread and 2 map
OK
>> Galadriel sell 3 bread and 3 map
```

```
OK

>> Galadriel total ?

3 bread and 2 map

>> Galadriel sell 3 bread and 3 map to Cirdan

OK

>> Galadriel total ?

3 bread and 2 map

>> Cirdan total ?

NOTHING

>> exit
```