COP 3502C Programming Assignment # 1

Dynamic Memory Allocation

Read all the pages before starting to write your code

Overview

This assignment is intended to make you work with dynamic memory allocation, pointers, and arrays of pointers. The difficulty level is not actually very high, but it might take sometimes to do it depending on your understanding of Dynamic Memory Allocation - don't wait until the weekend it's due to start it!

Your solution should follow a set of requirements to get credit.

Please include the following commented lines in the beginning of your code to declare your authorship of the code:

/* COP 3502C Assignment 1

This program is written by: Your Full Name */

Compliance with Rules: UCF Golden rules apply towards this assignment and submission. Assignment rules mentioned in syllabus, are also applied in this submission. The TA and Instructor can call any students for explaining any part of the code in order to better assess your authorship and for further clarification if needed.

Caution!!!

Sharing this assignment description (fully or partly) as well as your code (fully or partly) to anyone/anywhere is a violation of the policy. I may report to office of student conduct and an investigation can easily trace the student who shared/posted it. Also, getting a part of code from anywhere will be considered as cheating.

Deadline:

See the deadline in Mimir. The assignment will accept late submission up to 24 hours after the due date time with 10% penalty. After that the assignment submission will be locked. An assignment submitted by email will not be graded and such emails will not be replied according to the course policy.

What to do if you need clarification on the problem?

I will create a discussion thread in webcourses and I highly encourage you to ask your question in the discussion board. Maybe many students might have same question like you. Also, other students can reply and you might get your answer faster. Also, you can write an email to the TAs and put the course teacher in the cc for clarification on the requirements.

How to get help if you are stuck?

According to the course policy, all the helps should be taken during office hours. Occasionally, we might reply in email.

Problem Description: Making Smoothies

A smoothie is a blended drink concoction, with several ingredients. Each different type of smoothie has a different ratio of its ingredients. For example, the "StrawberryBreeze" has the following ratios of items:

2 units strawberry
1 unit banana
1 unit kiwi
2 units yogurt

If a store manager sells 18 pounds of Strawberry Breeze smoothie a week, then his weekly order ought to have 6 lbs of strawberries, 3 lbs of banana, 3 lbs of kiwi and 6 lbs of yogurt.

Of course, there are other types of smoothies a store manager must sell, so to figure out her weekly order, she has to go through how much of each smoothie she'll sell and each smoothie's recipe and put together these calculations.

For your program, you'll read in information about several smoothie recipes and several stores expected sales for the week, and determine what raw ingredients (and how much of each) each store should order.

Note: This assignment test dynamic memory allocation. Later in the write up, requirements will be given stating which memory needs to be dynamically allocated and freed.

The Problem

Given a list of possible smoothie ingredients, a list of smoothie recipes, and lists of sales from several stores, determine how much of each ingredient each store must order.

Input Specification

The first line will contain a single positive integer, n ($n \leq 10^5$), representing the number of possible smoothie ingredients.

The following n lines will each contain a single string of in between 1 and 20 characters (all letters, digits or underscores). The i^{th} ($0 \leq i \leq n-1$) of these will contain the name of the i^{th} smoothie ingredient. (Thus, the ingredients are numbered from 0 to n-1.)

The next line of input will contain a positive integer, s ($s \leq 10^5$), representing the number of different smoothie recipes. (Use the same numbering convention for the recipes. The recipes are numbered from 0 to s-1.)
The next s lines will contain the smoothie recipes, one per line. Each of these lines will be formatted as follows:

$m\ \ I_1\ \ R_1\ \ I_2\ \ R_2\ \ \ldots\ \ I_m\ \ R_m$

m represents the number of different ingredients in the smoothie ($1 \leq m \leq 100$)

I_1 represents the ingredient number of the first ingredient ($0 \leq I_1 < n$)

R_1 represents the number of parts (ratio) of the first ingredient ($1 \leq R_1 \leq 1000$) in the smoothie recipe

The rest of the I and R variables represent the corresponding information for the rest of the smoothie ingredients.

For example, if strawberries were ingredient 7, bananas ingredient 3, kiwis ingredient 6 and yogurt was ingredient 0, then the following line would store a recipe for the Strawberry Breeze previously mentioned:

```
4 7 2 3 1 6 1 0 2
```

The following line of input will contain a single positive integer, k ($1 \leq k \leq 100$), representing the number of stores making orders for smoothie ingredients. (**Number the stores 1 to k, in the order they appear in the input.)**

The last k lines of input will contain each store's order, one order per line.

Each of these lines will be formatted as follows:

$r\ \ S_1\ \ W_1\ \ S_2\ \ W_2\ \ \ldots\ \ S_r\ \ W_r$

r represents the number of different smoothies the store offers ($1 \leq r \leq s$)

S_1 represents the smoothie number of the first smoothie ($0 \leq S_1 < s$)

W_1 represents the weight sold of the first smoothie ($1 \leq W_1 \leq 1000$), in pounds.

The rest of the S and W variables represent the corresponding information for the rest of the smoothie ingredients.

Note: the sum of all the number of different smoothies all the stores offer won't exceed 10^6.

Output Specification

The output of the program must be **written to out.txt file** with exact same format as specified in the sample output. The output also needs to be displayed **in the standard output console**. For each store order, print the following header line:

```
Store #x:
```

where x is the 1-based number of the store.

This will be followed by a list of each ingredient the store must order and the amount of that ingredient (in pounds), rounded to 6 decimal places. The ingredients must be listed in their numeric order (order in the input), one ingredient per line, but instead of printing out the number of the ingredient, print out its name. For example, in the previously listed example, if a store only sold 18 pounds of the Strawberry Breeze and the numbers of the ingredients were 7 - strawberry, 3 - banana, 6 - kiwi, and 0 - yogurt, then the corresponding output (minus the header line) would be:

```
yogurt 6.000000
banana 3.000000
kiwi 3.000000
strawberry 6.000000
```

Thus, the format of each line is:

```
Ingredient_Name Weight_To_Order
```

with weight to order rounded to exactly 6 decimal places.

Print blank lines between each store. The last part of the output should have two blank lines.

Example Input and Output with explanation (note that you should prepare more test cases and test your code with more inputs)

Example in.txt file

8 //ingredients. Sequence starts from 0

yogurt // sequence 0

chocolate //1

raspberry //2

banana //3

mango //4

spinach //5

kiwi //6

strawberry //7

3 //recipes

4 7 2 3 1 6 1 0 2// recipe#0 strawberry breeze has 4 ingredients. (ingredient number#, unit or ratio), ...

3 0 2 5 1 4 2 // recipe#1 veggie mango has 3 ingredients (yogurt 2 unit, spinach 1 unit, mango 4 units)

4 0 2 4 4 3 1 1 2 // recipe#2 fruity chocolate has 4 ingredients

3 //stores. sequence starts from 1

2 1 10 2 20 // store 1 offers 2 different recipes (recipe number, total unit in pounds)

1 0 30 //store two offers 1 recipe (strawberry breeze 30 LBs)

3 0 10 1 5 2 15 //store three offers 3 recipes

Example Output (out.txt file as well as to the standard output)

Store #1:
yogurt 8.444444
chocolate 4.444444
banana 2.222222
mango 12.888889
spinach 2.000000

Store #2:
yogurt 6.000000
banana 3.000000
kiwi 3.000000
strawberry 6.000000

Store #3:
yogurt 8.666667
chocolate 3.333333
banana 3.333333
mango 8.666667
spinach 1.000000
kiwi 1.666667
strawberry 3.333333
<new line>
<new line>

Calculation hints:

Here is an example calculation of the numbers for store #1
Store#1 offers 2 types of recipes.
They want to make 10LBs of recipe#1 and 20 LBs or recipe#2
Based on recipe#1 we need the following ingredients for Store#1:
Yogurt: (2 * 10)/(2+1+2) = 4 LBs
Chocolate: 0LB
Raspberry: 0LB
Banana: 0LB
Mango: (2 * 10)/(2+1+2) = 4 LBs
Spinach: (1 * 10)/(2+1+2) = 2 LBs
kiwi 0LB
strawberry 0LB

Based on recipe#2 we need the following ingredients for Store#1 :
Yogurt: (2 * 20)/(2+4+1+2) = 4.444444 LBs

Chocolate: : (2 * 20)/(2+4+1+2) = 4.444444 LBs

Raspberry: 0LB
Banana: : (1 * 20)/(2+4+1+2) = 2.222222 LBs
Mango: (4 * 20)/(2+4+1+2) = 8.888889 LBs
Spinach: 0LB
kiwi 0LB
strawberry 0LB

So, the total amount of non zero ingredients needed for store#1

yogurt 8.444444 (calculated from 4 + 4.444444)
chocolate 4.444444 (calculated from 0 + 4.444444)
banana 2.222222 (calculated from 0 + 2.222222)
mango 12.888889 (calculated from 4 + 8.888889)
spinach 2.000000 (calculated from 2 + 0)

Implementation Restrictions/ Run-Time/Memory Restrictions

1. The names of each of the ingredients must be stored in a dynamically allocated character array, where the memory for each individual string is ALSO dynamically allocated. (Thus, there should be one malloc.calloc at the top level and several malloc/callocs on the inner level.)

2. You must use the following structs. What these structs store is also described below:

```
typedef struct item {
    int itemID;
    int numParts;
} item;
```

This stores one component of a smoothie recipe. The itemID represents the ingredient number and numParts represents the number of parts of that ingredient.

```
typedef struct recipe {
    int numItems;
    item* itemList;
    int totalParts;
} recipe;
```

This stores one smoothie recipe. numItems stores the number of different ingredients, itemList will be a dynamically allocated array of item, where each slot of the array stores one ingredient from the recipe, and totalParts will equal the sum of the numParts of each ingredient in the smoothie. Notice that this is not an array of pointers but just an array of struct. The reason this design decision has been made is because (a) to give you practice with how deal with an array of struct, and (b) Once the data is read into this array, no changes will be made to the array (ie no swapping of elements), thus, this will work equally well for our purposes as an array of pointers.

3. You must use variables of the following types to perform the following tasks:

(a) All of the smoothies need to be stored an array of pointers to recipes:

```
recipe** smoothieList;
```

Thus, storing the smoothieList will involve one malloc/calloc for an array of pointers. Then, each of those pointers will point to a single recipe.

(b) When processing each store, you must store the amount of each ingredient in a dynamically allocated frequency array. This array should be allocated right before your program reads in the store's sales information (which smoothies it makes and how much of each one). Then, it should be freed right AFTER the calculation of how much of each ingredient needs to be ordered completes and is printed to the screen. This array should look like this:

```
double* amtOfEachItem;
```

(Note: You can figure out how much space to allocate for it. The idea is that amtOfEachItem[i] will store a double equaling the number of pounds for the order for ingredient i.

4. For full credit, you must write the following functions with the following prototypes, pre-conditions and post-conditions:

```
// Pre-condition: reference to a variable to store number of ingredients.
// Post-condition: Reads in numIngredients and that number of strings from
//                 the inputs, allocates an array of
//                 strings to store the input, and sizes each
//                 individual string dynamically to be the
//                 proper size (string length plus 1), and
//                 returns a pointer to the array.
char** readIngredients(int *numIngredients);
```

```
// Pre-condition: does not take any parameter
// Post-condition: Reads in details of a recipe such as numItems,
//                 Dynamically allocates space for a single
//                 recipe, dynamically allocates an array of
//                 item of the proper size, updates the
//                 numItems field of the struct, fills the
//                 array of items appropriately based on the
//                 input and returns a pointer to the struct
//                 dynamically allocated.
recipe* readRecipe();
```

```
// Pre-condition: reference to a variable to store number of recipes.
// Post-condition: Read number of recipes. Dynamically allocates an array of //
pointers to recipes of size numRecipes, reads numRecipes
//                 number of recipes from standard input, creates
//                 structs to store each recipe and has the
```

```
//                 pointers point to each struct, in the order
//                 the information was read in. (Should call
//                 readRecipe in a loop.)
recipe** readAllRecipes(int *numRecipes);


// Pre-condition: 0 < numSmoothies <= 100000, recipeList is
//    pointing to the list of all smoothie recipes and
//    numIngredients equals the number of total ingredients (you have        //      already
read it in the first line of the input).
// Post-condition: Reads in information from standard input
//    about numSmoothies number of smoothie orders and dynamically
//    allocates an array of doubles of size numIngredients such
//    that index i stores the # of pounds of ingredient i
//    needed to fulfill all smoothie orders and returns a pointer
//    to the array.

double* calculateOrder(int ingredientCount, int numSmoothies, recipe** recipeList);


// Pre-conditions: ingredientNames store the names of each
//                 ingredient and orderList stores the amount
//                 to order for each ingredient, and both arrays
//                 are of size numIngredients.
// Post-condition: Prints out a list, in ingredient order, of each
//                 ingredient, a space and the amount of that
//                 ingredient to order rounded to 6 decimal
//                 places. One ingredient per line.
void    printOrder(char**    ingredientNames,    double*    orderList,    int
numIngredients)


// Pre-conditions: ingredientList is an array of char* of size
//          numIngredients with each char* dynamically allocated.
// Post-condition: all the memory pointed to by ingredientList is
//                 freed.
void freeIngredients(char** ingredientList, int numIngredients);


// Pre-conditions: allRecipes is an array of recipe* of size
//                numRecipes with each recipe* dynamically allocated
//               to point to a single recipe.
// Post-condition: all the memory pointed to by allRecipes is
//                 freed.
void freeRecipes(recipe** allRecipes, int numRecipes);
```

5. You can declare the file pointers as a global variable. No other global variable is allowed.

6. It is not a requirement, but could be a good idea to have a wrapper function for processing the outputs:

```
void processOutput(char** ingredientList, int ingredientCount, recipe** allRecipes, int
recipeCount);
```

7. The allowable run-time for this program is a bit tricky to state due to the fact that there are many variables. Let n = # of ingredients, k = # of stores ordering, ss = sum of the number of different smoothies in all the orders, and m = max number of ingredients in a smoothie. Then, for full credit, your program must run in $O(n*k + ss*m)$. The program involves one part where you loop through all of the orders and for each order allocate an array of size n, and loop through that array. It also involves, over the course of those k orders, looping through ss total smoothies, each of which could have upto m ingredients. More about the concept of run-time will be discussed in the lecture.

8. You have to use memory leak detector code as shown the lab as well as explained in webcourses

- You must #include "leak_detector_c.h" in your code, and
- You must call atexit(report_mem_leak) as the first line of your main().
- leak_detector_c.h and leak_detector_c.c are available in webcourse.

- You do not need to comment line by line, but comment every function and every "paragraph" of code.
- You don't have to hold any particular indentation standard, but you must indent and you must do so consistently within your own code.

Deliverables

You must submit four files over mimir:

1) A source file, *main.c*.

2) A file describing your testing strategy, *lastname_Testing.doc(x) or lastname_Testing.pdf*. This document discusses your strategy to create test cases to ensure that your program is working correctly. If you used code to create your test cases, just describe at a high level, what your code does, no need to include it.

3) You have to submit at least two input test files developed by you. The file names should be in1.txt and in2.txt with out1.txt and out2.txt generated by your code, respectively. **(Note: Hand made test cases should be fine. It's not necessary to make a max case to get full credit here.)**

4) leak_detector_c.h and leak_detector_c.c files need to be submitted as well.

Rubric (subject to change):

According to the Syllabus, the code will be compiled and tested in Mimir Platform for grading. If your code does not compile in Mimir, we conclude that your code is not compiling and it will be graded accordingly. We will apply a set of test cases to check whether your code can produce the expected output or not. Failing each test case will reduce some grade based on the rubric given bellow. If you hardcode the output, you will get -200% for the assignment. Note that we will apply more test cases while grading. So, passing the sample test cases might not guarantee that your code will also pass other test cases. So, thoroughly test your code.

1. If a code does not compile the code may get 0. However, some partial credit maybe awarded. A code having compiler error cannot get more than 50% even most of the codes are correct
2. If you modify or do not use the required structure: 0

3. Not using dynamic memory allocation for storing data will receive 0
4. There is no grade for a well indented and well commented code. But a bad indented code will receive 20% penalty. Not putting comment in some important block of code -10%
5. Implementing required functions and other requirements: 30%
6. Freeing up memory properly with zero memory leak (if all the required malloc implemented): (20%)
7. Passing test cases: 50%

Some hints:

- **Make sure you have a very good understanding of Dynamic memory allocation based on the lecture, exercises, and labs**
- **The core concepts of the example of dynamically allocating array of structure pointer, dynamically allocating array of strings, and the Lab2 code would be very useful before starting this assignment.**
- **Start the assignment as soon as possible**
- **Break it down by drawing and designing,**
- **Write each load function and test whether your data is loaded properly**
- **Then gradually implement functions one by one and test your code gradually.**
- **Do not wait till the end to test your code.**
- **Do not hesitate to take help during all of our office hours.**

Good Luck!