COP 3502C Programming Assignment # 1

Dynamic Memory Allocation

Please read all the pages before starting to write your code

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

```
/* COP 3502C PA1
This program is written by: Your Full Name */
```

Compliance with Rules: *The UCF Golden Rules apply to this assignment and its submission. In addition, all assignment policies outlined in the syllabus also apply here. The instructor or TA may ask any student to explain any part of their code to verify authorship and ensure clarity. Sharing this assignment description or your code, in whole or in part, with anyone or anywhere is a violation of course policy. Likewise, using code obtained from any outside source, including AI tools, will be considered cheating. Your primary resources should be class notes, lectures, labs, and TAs. These rules are in place to ensure fairness, maintain academic integrity, and give every student the opportunity to develop their own problem-solving and coding skills.*

Deadline:

Please check Webcourses for the official assignment deadline. Assignments submitted by email will not be graded, and such emails will not receive a response in accordance with course policy.

What to do if you need clarification on the problem?

If you need clarification on the problem, a discussion thread will be created on Webcourses, and you are strongly encouraged to post your questions there. You may also ask questions on Discord; if TAs are available, they can respond, and other students might also contribute, helping you get answers more quickly. For further clarification on the requirements, you are welcome to visit the TAs or the instructor during office hours. You should also disable AI assistant mode of your IDE while working on this assignment.

How to get help if you are stuck?

As stated in the course policy, office hours are the primary place to seek help. While we may occasionally respond to emails, debugging requests are best addressed during office hours rather than through email.

PA1: Cat Store Management System

Image generated by ChatGPT

Objective:

- Practice dynamic memory allocation with the following concepts:
    - Allocating arrays of structs, struct pointers, integers & strings
    - Practice with pointer manipulation
    - a struct where the struct has a dynamically allocated array of structs and an array of int in it
- Properly freeing allocated memory.
- Writing functions with query handling following the given specifications.

Background story:

Cat shelters constantly grow and want to do a refresh on their data system. They want to create a centralized system online since they currently use an outdated method. You've been tasked to make the program to do the following:

- Track how many breeds are being stored.
- Store the maximum capacity for each breed per kennel.
- Monitor each kennel's max capacity and compare to valid constraints for transfers across kennels
- Store each individual cat's information.
- Perform some queries on the stored information

Variables and context:

The program you are writing manages a single cat store at once, where each store has several cat kennels across different locations. Each kennel location has a set of cats residing at it, and its occupancy is being maintained and compared against its maximum capacity. Each cat has all the info about it stored in the program, like its name, breed, age, etc.
The program also keeps track of a dictionary for the store, where the dictionary is a list of words representing cat breeds that could be in this store at any of its locations.

One of the pieces of information that is stored and managed about this cat store is a set of capacity constraints for each of its kennels of each breed type. This is represented as a double array associated with the store, where the number of rows is equal to the number of kennels, and the number of columns is equal to the number of breeds in the dictionary. Each kennel's maximum capacity is the sum of these individual breed capacities of the kennel.

For example, if we have 4 breeds, and capacities[2] = {4, 2, 1, 5}, then kennel 2 can have a maximum of 5 cats of breed type 3, and the max capacity of kennel 2 is 12, as it is the sum of its individual breed capacities.

Although there is a max capacity for each kennel, it is not guaranteed that during the inputs you will fill your kennel. An occupancy variable will keep track of how many cats there are in this kennel. Occupancy should never be higher than the max capacity of a kennel

One of the pieces of information we're keeping track of about each cat is its current status. In the cat struct, status is stored as an integer ranging from 0-2. This serves as an index to access a given constant global STATUS_CAT array. Each cat's status should be initialized to 'AVAILABLE" when first inserted into the system.

STATUS_CAT is a string array (given below, in the global constant variable section) representing the possible statuses for a cat in the store. The cat status is stored as an integer (ranges from indexes 0 to 2) in the cat struct, storing the index corresponding to the status. For example, if a cat's status is 0, then the status of this cat could be retrieved by accessing the word stored at the 0^{th} index in the STATUS_CAT array.

Lastly, the cat's breed is a char pointer that is **not to be allocated** for each cat. Instead, it will point to the corresponding string stored in the dictionary

The rest of the members of the structs are self-explanatory and are given below in the required sections.

Flow of the program:

The program will mainly run in two parts, Initialization, and Queries Processing.

Initialization: You will be given all the initial information related to the store, kennels, current cats, during this part of the input. You should create the structs and store those values and information accordingly. **Note that the first set of information is guaranteed not to break any capacity constraints of the kennels.** That is, there will be no cat information given that cannot be stored at the kennel we are attempting to store it at during this stage. No printed output is needed during this stage.

Queries Processing: Next, you are asked to process a number of queries about this cat store. Those are explained in detail later in this document. It is important to know that these queries might or might not attempt to break the constraints, which is something you need to make sure you are checking for before any data transfer between kennels. Additionally, since you are a DMA expert in C and you will not waste any memory space, you need to make proper use of resizing arrays if needed, and properly keep track of pointers, so as not to create duplicates for the same cat, and so on. Printed output depends on the type of query and is explained in detail later in this document.

Required Data Structures:

You must use the following structures in this assignment. You are allowed to declare more structure if needed. However, you are not allowed to change the given structs.

```
typedef struct Cat{
   char *name; // dynamically allocated space for the name without wasting space
   int age; // specifies the age of the cat
   float weight; // stores the decimal value in weight for this specific cat
   char *breed; //points to an already allocated breed string
    (Ragdoll, Siamese, Maincoone, etc..). No malloc/calloc for this property
   int status; //specifies if a cat is adopted, pending, or available, ranges from 0 to 2, initially set to available
for newly added cats
} Cat;

typedef struct Kennel {
   char *location; // dynamically allocated space for the location without wasting space
   int occupancy; // stores the current number of cats in the kennel
   Cat **cats; // dynamically allocated array of pointers to Cats that reside in the kennel
   int maxCapacity; // specifies the max cap to the kennel
} Kennel;
```

```
typedef struct CatStore {
  int **capacities; //dynamically allocated double int array stores the breeds constraints for all kennels
  int numKenels; //specifies the total number of kennels this store owns
  Kennel *kennels; // dynamically allocated array of kennels
} CatStore;
```

Must maintain variables in main:

```
char **dictionary; //to store array of dynamically allocated strings for breeds types (e.g., {"Ragdoll", "Siamese", "Maincoone"})
int breedCount; //number of breed types
```

Must maintain global constant variable:

```
const char STATUS_CAT[3][20] = {"ADOPTED", "PENDING", "AVAILABLE"}
```

Functions Required:

You must implement and use the following functions with the specified prototypes in your solution. You are free to add additional functions as needed (and are recommended to); however, the required functions must be implemented exactly as specified, and their prototypes must **not** be modified.

```
char ** readBreeds(int *count)
```

Preconditions:

- count must be a valid memory address where the number of breeds can be stored

Postconditions:

- returns a dynamically allocated array of strings loaded with the breed types from the input.
- count is updated to reflect the number of breeds

```
char* getCharPtrByBreed(char **dictionary, char *breedName, int breedCount)
```

Preconditions

- dictionary is a strings array storing the words representing possible breeds
- breedName must be a non-null string representing the specific breed name to be found
- breedCount is a valid int that represents the number of the breed names stored in the dictionary

Postconditions:

- Returns a pointer to the matching string storing the breed if found, returns NULL if not found

```
CatStore *createStore(int kennelCount, int breedCount, char ** dictionary)
```

Preconditions:

- kennelsCount is a valid int representing the number of kennels this store has
- breedCount is a valid int that represents the number of the breed names stored in the dictionary
- dictionary is a strings array storing the words representing possible breeds

Postconditions:

- returns a valid CatStore pointer that has been allocated for, and all fields were filled entirely with proper data from the input.

```
Kennel* createKennels(int **constraints, int kennelCount, int breedCount, char **dictionary)
```

Preconditions:

- constraints is a valid 2d int array that represent the breed limit for each kennel, with dimensions kennelCount rows and breedCount columns
- kennelsCount is a valid int representing the number of kennels this store has
- breedCount is a valid int that represents the number of the breed names stored in the dictionary
- dictionary is a strings array storing the words representing possible breeds

Postconditions:

- Returns a dynamically allocated array of kennel structs of size kennelCount, with each field of every kennel struct properly filled

```
Cat **createCats(char **dictionary, int breedCount, int count)
```

Preconditions:

- dictionary is a strings array storing the words representing possible breeds
- breedCount is a valid int that represents the number of the breed names stored in the dictionary
- count in a valid int representing the number of cats to create

Postconditions:

- Returns a dynamically allocated and properly filled array of Cat pointers of size count
- Each pointer points to a new Cat with data properly stored in struct members

```
Cat* createSingleCat(char **dictionary, int breedCount)
```

Preconditions:

- dictionary is a strings array storing the words representing possible breeds
- breedCount is a valid int that represents the number of the breed names stored in the dictionary

Postconditions:

- Creates a single cat structure, reads in its data, initializes cat's status to the integer representing 'AVAILABLE' as explained above, points to the corresponding breed from dictionary
- Returns a pointer to the allocated Cat structure

```
int canMoveTo(CatStore *s, char *location, char *breed, char **dictionary, int breedCount)
```

Preconditions:

- s is a properly allocated and filled CatStore pointer
- location is a valid non null string representing the location of the kennel we are checking if we can move this cat to
- breed is a valid string that represents the breed of the cat we want to move to the location
- dictionary is a strings array storing the words representing possible breeds
- breedCount is a valid int that represents the number of the breed names stored in the dictionary

Postconditions:

- Returns an integer representing if a cat of the given breed can be moved to the specific location.
- If can be moved returns 1, if not, returns 0
- A cat of a given breed can only be moved to the location if the location maximum capacity is not exceeded and the breed follows the capacity constraint this location has for this breed

```
Kennel *getKennelByCat(CatStore *s, Cat *cat)
```

Preconditions:

- s is a properly allocated and filled CatStore pointer
- cat is a properly allocated and filled Cat pointer

Postconditions:

- Returns the pointer of the kennel holding this cat if found, otherwise, returns NULL

```
int getCatPosi(Kennel *home, Cat *cat)
```

Preconditions:

- home is a valid Kennel pointer
- cat is a properly allocated and filled Cat pointer

Postconditions:

- Returns an integer that represents what index the Cat is found at in the home kennel, if not found, returns -1

```
Cat *getCatByName(CatStore *s, char *catName)
```

Preconditions:

- s is a properly allocated and filled CatStore pointer

- catName is a valid non null string representing the cat name we're looking for

Postconditions:

- searches the store and returns the pointer to the cat with the name catName

```
void removeCatFromKennel(Kennel *k, Cat *cat)
```

Preconditions:

- k is a properly allocated and filled kennel pointer
- cat is a pointer of the cat to be removed from kennel k

Postconditions:

- cat is removed from kennel k, elements are shifted to fill in the gap, kennel's size and cat array it adjusted
- cat is **not to be freed** in this function

```
void runQueries(CatStore *s, char **dictionary, int breedCount,
int numQueries)
```

Preconditions:

- s is a properly allocated and filled CatStore pointer
- dictionary is a strings array storing the words representing possible breeds
- breedCount is a valid int that represents the number of the breed names stored in the dictionary
- numQueries is a valid int that represents the number of questions the function will handle

Postconditions:

- Handles each query accordingly (more details below), it is recommended that you write a separate function for each query and call them from this function

```
void freeBreeds(char **dictionary, int breedCount)
```

Preconditions:

- dictionary is a strings array storing the words representing possible breeds
- breedCount is a valid int that represents the number of the breed names stored in the dictionary

Postconditions:

- All memory associated with dictionary is freed

```
void freeStore(int count, CatStore *store)
```

Preconditions:

- count is an in integer that represents the number of kennels in the store
- store is a properly allocated and filled CatStore pointer

Postconditions:

- All dynamically allocated memory within CatStore struct and its associated struct members is freed

- Frees cats in every kennel first, then frees kennels, then frees the capacities and the store

Queries:

This dataset can support a wide range of interesting queries; Such, as finding all available cats, identifying overcrowded kennels, or determining adoption rates by breed. In fact, we initially included several more complex and engaging queries. However, we realized that the assignment was becoming too lengthy. Therefore, we've decided to limit the scope to just Query 1 through Query 3, as the primary focus of this assignment is on dynamic memory allocation.

After storing all the data into the data structures, queries will need to be processed based on the inputs.

Query Type 1: This query prints all cats' information of a given breed, displaying their name, weight, age, location, and status.

Query Type 2: This query updates the status of a specific cat given its name. If the new status of the cat is adopted, the cat needs to be entirely removed from the system, and the cat array must be adjusted accordingly.

Query Type 3: This query attempts to move a cat from one kennel to another, if the transfer is valid based on capacity constraints, the cat is removed from the source kennel, the cats array in the kennel is resized to one less cat, the destination kennel's cats array is resized to one more, then the moved cat is added to the destination kennel's cats array. Occupancies are adjusted accordingly.
Note: a transfer is only valid if it doesn't break any of the maximum capacity **and** the breed specific capacity constraints.

Input:

The first line of input will contain a single positive integer, **n (1 ≤ n ≤ 10)**, representing the number of breeds. The next ***n*** lines of input will each contain a single word ***breedname*** a string of maximum length of 25 representing the name of a cat breed the store could have.

```
n
<breedname_1>
<breedname_2>
....
<breedname_n>
```

The next line of input will contain a single positive integer, **k (1 ≤ k ≤ 100)**, representing the number of kennels this store has. The next ***k*** lines will each contain ***n*** integers representing the capacity constraints for each breed at that kennel as explained in the background section. The maximum overall capacity of a kennel can be computed as the sum of all its breed capacities, ig; the sum of the numbers on a single line.

```
k
```

```
<capacity_0> <capacity_1> <capacity_2> ..... <capacity_n>
<capacity_0> <capacity_1> <capacity_2> ..... <capacity_n>
...
<capacity_0> <capacity_1> <capacity_2> ..... <capacity_n>
```

For each kennel, there is a set of input representing that kennel's location, number of cats currently in it, and info about the cats, in the following format, repeated ***k*** number of times:

A single line containing a unique string of maximum length of 25 ***location*** representing the location name of this kennel, followed by an integer ***c*** **(0 ≤ c ≤ 1,000)** representing the number of cats currently at that kennel. The next ***c*** lines will each contain the cats information, with each line containing a string ***name*** of maximum length of 25 for a unique name, int for age, float for weight, and string for breed. All cats are initialized with status "AVAILABLE"

```
<location_1> <c>
<name_1> <age_1> <weight_1> <breed_1>
<name_2> <age_2> <weight_2> <breed_2>
....
<name_c> <age_c> <weight_c> <breed_c>

<location_2> <c>
<name_1> <age_1> <weight_1> <breed_1>
<name_2> <age_2> <weight_2> <breed_2>
....
<name_c> <age_c> <weight_c> <breed_c>

....

<location_k> <c>
<name_1> <age_1> <weight_1> <breed_1>
<name_2> <age_2> <weight_2> <breed_2>
....
<name_c> <age_c> <weight_c> <breed_c>
```

After all the kennel data, the next line contains an integer ***q*** **(1 ≤ q ≤ 100)** representing how many queries will be processed. Each line of the ***q*** lines starts with an integer (1-3) to indicate the type of query this line is asking for. The input on the rest of the line depends on what type of query is being processed.

Query Type 1:

`1 <breed>` – breed is a string that provides the breed name; it is guaranteed that the breed is in the dictionary. However, it is possible that there may be no cat of that breed in the store.

Query Type 2:

2 <num> <name> - num is an int ranging from 0 to 2 that represents the new status of the cat, name is a string for the name of the cat being updated; it is guaranteed that a cat with this name is currently in the store.

Query Type 3:

3 <name> <location> – name is a string for the name of the cat being updated, and location is a string for the location to move the cat to. It is guaranteed that a cat with this name and a kennel with this location are in the store.

Output:

The output depends on the query type.

Query Type 1:

For every cat of the requested breed, the cat information should be printed in the following format on a single line, printed in the order they were added to the store:

```
<name> <weight> <age> <location> <status>
```

<name> represents the name of a cat.
<weight> represents the weight of a cat to 2 decimal places.
<age> represents the age of the cat.
<location> is the character pointer holding the location of the cat.
<status> is the "ADOPTED", "PENDING", or "AVAILABLE" of the cat.

However, if there is no cat with this breed, you should print the following information in a single line.

No cat with breed <breed>

Query Type 2:

A single line output in the following format followed by a new line

```
<name> is now <status>!
```

Where name is the <name> of the cat, and <status> is either "ADOPTED*",* *"*PENDING*",* *"*AVAILABLE" based on the input. The global array STATUS_CAT must be utilized for this print statement.

Query Type 3:

If the cat can be moved to the requested location, the following message is printed, followed by a new line

```
<name> moved successfully to <location>
```

Otherwise, the following message is printed followed by a new line

```
<location> cannot take a <breed> cat!
```

Sample Program Run:

Example Input	Example Output (standard output)
5 Ragdoll Maincoone Siamese Tuxedo Taby 3 12 5 10 6 4 18 17 13 6 9 0 5 3 20 5 Polterland 4 Charlie 12 52.55 Ragdoll Caesar 2 512.2 Siamese Cleo 4 8.6 Tuxedo Patra 4 11.15 Ragdoll Auburndale 2 Coco 14 23.2 Ragdoll Channel 13 18.3 Ragdoll Narnia 1 Sad 18 92.00 Taby 9 1 Ragdoll 2 1 Charlie 2 0 Charlie 1 Ragdoll 3 Caesar Auburndale 3 Sad Narnia 3 Coco Narnia 2 0 Cleo 1 Tuxedo	Charlie 52.55 12 Polterland AVAILABLE Patra 11.15 4 Polterland AVAILABLE Coco 23.20 14 Auburndale AVAILABLE Channel 18.30 13 Auburndale AVAILABLE Charlie is now PENDING! Charlie is now ADOPTED! Patra 11.15 4 Polterland AVAILABLE Coco 23.20 14 Auburndale AVAILABLE Channel 18.30 13 Auburndale AVAILABLE Caesar moved successfully to Auburndale Sad moved successfully to Narnia Narnia cannot take a Ragdoll cat! Cleo is now ADOPTED! No cat with breed Tuxedo

Important Additional Implementation Requirements/Run Time Requirements:

- Use standard input (scanf/printf). No file I/O is allowed or accepted.
- You may only use **malloc**, **calloc**, or **realloc** for memory allocation
- Ensure all required function prototypes are implemented within your program.
- Only one instance of each cat should be created, and their pointers should be used consistently throughout the program. In other words, there should be only one malloc call for each cat
- Your code design must not use triple pointers (***). You should limit the pointer usage to single or double pointers for simplicity and clarity

- **For strings, you must first read the input into a static char array, then allocate the exact amount of memory** needed based on its length and copy the string using strcpy function as demonstrated in class. Any other approach will result in a **-100 penalty**.
- You must free all the memory to receive full credit.
- You are not allowed to write a function that will receive or return a whole structure (instead, only an address or pointer to a struct is allowed)
- Make sure not to use any malloc/calloc/realloc for cat's breed property. It should only point to an existing breed in the dictionary
- Your code must have no memory leaks at the end of its execution.
- Only one instance of each breed should be created in the dictionary, and their pointers should be used consistently throughout the program. In other words, there should be only one malloc call for each breed string. Cat breed pointers should point to the dictionary entry, not allocate new memory
- Make sure your file name is main.c, otherwise it will not compile on codegrade.
- Make sure to test your code thoroughly using your own test cases in addition to the provided ones. During grading, we will run additional test cases to evaluate your code. Therefore, passing all sample test cases does not guarantee full credit if your code fails the grading test cases. Such failures indicate that your code contains bugs or does not handle certain situations described in the assignment properly

Deliverables:

- Please submit a single source file, `main.c`, via Webcourses->Codegrade.

Hints:

- Before starting your code, draw out the logic and scenario of the program first to help with logic implementation
- Make sure to test your code gradually as you implement functions to avoid complex debugging later.
- Update elements as needed to maintain proper functionality.
- Consider how the capacities 2D array relates to each kennel and how to validate transfers
- Make sure you have a pretty good idea of DMA based on the lecture and notes before starting
- Don't forget to watch the recording on loading list of strings from file to a dynamically allocated array of strings

Tentative Rubric:

Rubric (subject to change):

According to the Syllabus, the code must work on codegrade to receive credit. If your code does not compile on codegrade, we conclude that your code has a compiler error 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 below. 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 a compiler error cannot get more than 30%, even though most of the codes are correct
2. If you modify or do not use the required structure: you may get 0
3. Not using dynamic memory allocation for storing data will result in 0
4. There is no grade for a well-indented and well-commented code. But a badly indented code will receive 20% penalty. Not putting a comment in some important block of code -10%
5. Implementing required functions and other requirements: 35%
6. Freeing up memory properly with zero memory leak (if all the required malloc implemented): (15%)
7. Passing test cases with exact matching (We are expecting to test your code with 10 test cases, including the sample test cases provided to you): 50%