COP 3502C Programming Assignment # 1: NY Times Puzzle Archive

Dynamic Memory Allocation

Please read all the pages before starting to write your code

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.

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. **Also, it's okay not to finish an assignment. I didn't complete many of my undergraduate coding assignments (Arup) simply because I couldn't figure them out.** If all students completed all of their homework, then students would learn very little. (I can explain class why this is the case.)

Objective

Give practice with:

- Dynamic Memory Allocation for:
  - an array of strings
  - array of structs
  - a struct where the struct has a dynamically allocated array of structs and an array of int in it
  - an array of struct pointers where each pointer points to a dynamically allocated struct
- Releasing memory appropriately
- Writing functions based on the given specifications

Background Story:

You have been hired by the New York Times to support their growing collection of daily puzzles. They publish several popular daily puzzles, including Wordle, Connections, and Crossword. Players around the world compete to solve these puzzles and earn scores. In this assignment, you will simulate an archive system that tracks puzzles, players, and their scores.

- Each puzzle has a **type** (e.g., Wordle) and a **puzzle number (sometimes also referred as id)**. Together, the type and puzzle number uniquely identify a puzzle.
- A puzzle can be solved by several players.
- A player can solve each puzzle **only once**, and their score for that puzzle is recorded.
- The system also keeps track of each player's **total score** across all puzzles.

The assignment has various requirements to achieve the learning goals. So, pay attention to all the requirements while writing your solution.

Required structs:

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.

```
// Represents a player in the system
typedef struct {
    char *playerName;//dynamically allocated name of the player without
wasting space
    int totalScore;    // cumulative score across all puzzles
} Player;


// Represents one puzzle instance (e.g., Wordle #751, Crossword #202)
typedef struct {
    char *puzzleType; // points to an already allocated puzzle type string
(Wordle, Crossword, etc.). No malloc/calloc for this property
    int puzzleNo;       // ID number of the puzzle (e.g., 751, 202)
    Player** players;   // dynamic array of pointers to players who played
this puzzle
    int *scores;        // dynamic array of scores corresponding to each
player (index-aligned with players)
    int playerCount;    // number of players who played this puzzle
} Puzzle;
// Represents the archive of all puzzle instances
typedef struct {
    Puzzle* puzzles;   // dynamic array of puzzles
    int puzzleCount;   // number of puzzles stored
} Archive;
```

Must maintain variables in main:

In addition, in your *main* function, you must maintain the following variables and use them:

```
char** puzzleTypes; // to store array of dynamically allocated strings for
puzzle types (e.g., {"Wordle", "Crossword", "Connections"})
int puzzleTypeCount; // number of puzzle types
// ofcourse you may need other variables and pointers to complete the tasks.
```

Function Requirements:

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

- **Function:** `char** readPuzzleTypes(int *countOut)`

```
Preconditions:
    • countOut must be a valid memory address where the number of puzzle
      types can be stored.
Postconditions:
    • Returns a dynamically allocated array of strings loaded with the puzzle
      types from the input.
    •  countOut is updated to reflect the number of puzzle types.
```

- **Function:** `Player *createPlayer(char *name);`

```
Preconditions:
    • name must be a valid non-null string representing the player's name.
Postconditions:
    • Returns a pointer to a newly allocated Player structure with the
      player's name stored and other properties initialized.
```

- **Function:** `Archive *createArchive(int puzzleCount);`

```
Preconditions:
    • puzzleCount represents the number of puzzle in the archive
Postconditions:
    • Returns a pointer to a new Archive structure containing an array of
      Puzzle structures.
```

- **Function:** `Player**  readPlayerPool(int *playerCount)`

```
Preconditions:
    • playerCount must be a valid memory address where the number of players
      can be stored.
Postconditions:
    • Returns a dynamically allocated array of pointers to Player structures.
    • Each pointer points to a dynamically allocated player.
    • playerCount is updated with the number of players read.
    • Must take the help from the createPlayer function
```

- **Function:** `char*  getPuzzleTypePtr(char **puzzleTypes, int count, char *type)`

```
Preconditions:
    • puzzleTypes must be a valid pointer to an array of strings.
    • count should represent the number of puzzle types available.
    • type must be a valid non-null string representing the type to find.
Postconditions:
    • Returns a pointer to the matching puzzle type string, or NULL if not
      found.
```

- **Function:** `Player* getPlayerPtrByName(Player **playerPool, int playerCount, char *playerName)`

```
Preconditions:
    • playerPool must be a valid pointer to an array of player pointers.
    • playerCount must indicate the number of players in playerPool.
    • playerName must be a valid non-null string to search for.
Postconditions:
    • Returns a pointer to the matching Player structure, or NULL if not
      found.
```

➢ **Function:** `void printBestScorer(Puzzle *puzzle)`

```
Preconditions:
    • puzzle must be a valid pointer to a Puzzle structure with at least one
      player.
Postconditions:
    • Displays the name and score of the top scorer for the specified puzzle.
      In case of a tie, the player who appears first in the input is
      considered the best.
```

➢ **Function:** `void printMaxTotalScorer(Archive *archive)`

```
Preconditions:
    • archive must be a valid pointer to an Archive structure with puzzles.
Postconditions:
    • Displays the player with the highest total score. In case of a tie, the
      player who appears first in the input is considered the top scorer.
```

➢ **Function:** `void freePlayerPool(Player **pool, int count)`

```
Preconditions:
    • pool must be a valid pointer to an array of player pointers.
count must be > 0 to indicate the number of players in the pool.
Postconditions:
    • Frees each player and their associated name, then frees the array of
      players.
```

➢ **Function:** `void freeArchive(Archive *archive)`

```
Preconditions:
    • archive must be a valid pointer to an Archive structure.
Postconditions:
    • Frees all puzzles in the archive and then frees the archive itself.
```

Queries:

After loading all the data into your data structures, you need to process one or more of the following queries based on the input

Query type 1: This query prints the player's name and the total score of the player with the maximum total score

Query type 2: This query prints the best scorer per puzzle

This dataset can support a wide range of interesting queries—such as identifying the hardest puzzle based on average score, finding the most popular puzzle, or determining the best player for each puzzle type based on cumulative scores. 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 and Query 2, as the primary focus of this assignment is on dynamic memory allocation.

Input

The first line of input will contain a single positive integer, **n** ($1 \leq$ **n** ≤ 10), representing the number of types of puzzles. The next **n** lines of input will each contain a single word lowercase string with a maximum of 20 characters, representing the list of puzzle types.

The next line of the input will contain a single positive integer, p ($1 \leq$ **p** $\leq 100{,}000$), representing the number of players. The next **p** lines of input will each contain a single word lower case string with a maximum of 20 characters, representing the list of player names.

The next line of the input will contain a single positive integer, z ($1 \leq$ **z** $\leq 100{,}000$), representing the number of puzzles in the archive. The next z set of inputs will cover puzzle instances of the archive.

The first line of a puzzle instance contains a single word string with a maximum of 20 characters for puzzle type, and an int representing the id (a.k.a puzzleNo) of the puzzle ($1 \leq$ **id**$\leq 100{,}000$), and an int c that represents the number of players who played this puzzle ($0 \leq$ **c** $\leq 100{,}000$). The next c line of a puzzle contains a string and an int separated by space. The string represents the name of the player, and the int is the score achieved by the student. It is guaranteed that the puzzle type of a puzzle instance will be available in the puzzle type and the player names in a puzzle instance will be available in the player list. After the inputs for the z set of puzzles, the next line contains an int representing how many queries q will be processed ($1 \leq$ **q** ≤ 2), and the next **q** lines will contain one int each line representing the query to be processed.

Output

The output depends on the query type.

Query Type 1

For query type 1, the output should be in the following format, followed by a newline:

```
Top player: <PlayerName> with total score <totalScore> <new line>
```

Here, `<PlayerName>` is the name of the player with the highest total score and `<totalScore>` is their total score.

Query Type 2

For query type 2, the output should be in the following format:

```
Top scorer per puzzle:
<type>#<id> <PlayerName> <Score>
<type>#<id> <PlayerName> <Score>
…… . .
<type>#<id> <PlayerName> <Score>
```

- `<type>` is the puzzle type.
- `<id>` is the puzzle number.
- `<PlayerName>` is the name of the player with the highest score for that puzzle.
- `<Score>` is that player's score in that puzzle.

If a puzzle has no players, print:

```
<type>#<id> No player yet for this puzzle <new line>
```

Sample Input1

```
3  <-- number of types
wordle
crossword
connections
3   <--number of player
alice
bob
charlie
2    <-- number of puzzles
wordle 751 2 <--type id player count
alice 90     <-- player and score
bob 75
crossword 202 2
alice 100
charlie 85
2           <-- number of queries
1          <-- query type number
2
```

Sample Output1

```
Top player: alice with total score 190
Top scorer per puzzle:
wordle#751 alice 90
crossword#202 alice 100
```

Sample Input2

```
3
wordle
crossword
connections
7
aisha
matthew
william
luciana
adam
xavier
hudson
4
crossword 222 4
adam 50
hudson 55
aisha 78
xavier 51
wordle 222 5
luciana 60
matthew 68
aisha 59
adam 90
hudson 85
crossword 225 0
crossword 229 1
luciana 60
2
2
1
```

Sample Output2

```
Top scorer per puzzle:
crossword#222 aisha 78
wordle#222 adam 90
crossword#225 No player yet for this puzzle
crossword#229 luciana 60
Top player: adam with total score 140
```

Important Additional Implementation Requirements/Run Time Requirements

1. The allowed run-time for this program is **O(pz),** since both can be upto 10^5, this means in practice a single case could **probably take upto 20 seconds or so**. Once I make a case close to the max case, I'll inform the class of the allowable run-time.
2. You may only use **malloc**, **calloc**, or **realloc** for memory allocation.
3. Only one instance of each player 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 player.
4. Your code design must not use triple pointers (***). You should limit the pointer usage to single or double pointers for simplicity and clarity.
5. 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.
6. You must free all the memory to receive full credit.
7. 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)
8. Make sure not to use any malloc/calloc/realloc for puzzle's puzzleType property. It should only point to an existing puzzle type
9. You do not need to comment line by line but comment every function and every "paragraph" of code.
10. You don't have to hold any particular indentation standard, but you must indent and you must do so **consistently** within your own code.
11. Make sure your file name is **`archive.c`**, so that the graders can more easily write grading scripts.
12. 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 **`archive.c`**, via Webcourses.

Hints:

- Make sure you have a pretty good idea of DMA based on the lecture and notes
- Before starting to code, draw the whole scenario based on the sample inputs and see how the structures and various variables are connected
- Don't forget to watch the recording on loading list of strings from file to a dynamically allocated array of strings