Important message on plagiarism

The single most important point for you to realize before the beginning of your studies at ShanghaiTech is the meaning of “plagiarism”:

Plagiarism is the practice of taking someone else's work or ideas and passing them off as one's own. It is the misrepresentation of the work of another as your own. It is academic theft; a serious infraction of a University honor code, and the latter is your responsibility to uphold. Instances of plagiarism or any other cheating will be reported to the university leadership, and will have serious consequences. Avoiding any form of plagiarism is in your own interest. If you plagiarize and it is unveiled at a later stage only, it will not only reflect badly on the university, but also on your image/career opportunities.

Plagiarism is academic misconduct, and we take it very serious at ShanghaiTech. In the past we have had lots of problems related to plagiarism especially with newly arriving students, so it is important to get this right upfront:

You may…

• … discuss with your peers about course material.
• … discuss generally about the programming language, some features, or abstract lines of code. As long as it is not directly related to any homework, but formulated in a general, abstract way, such discussion is acceptable.
• … share test cases with each other.
• … help each other with setting up the development environment etc.

You may not …

• … read, possess, copy or submit the solution code of anyone else (including people outside this course or university)!
• … receive direct help from someone else (i.e. a direct communication of some lines of code, no matter if it is visual, verbal, or written)!
• … give direct help to someone else. Helping one of your peers by letting him read your code or communicating even just part of the solution in written or in verbal form will have equal consequences.
• … gain access to another one’s account, no matter if with or without permission.
• … give your account access to another student. It is your responsibility to keep your account safe, always log out, and choose a safe password. Do not just share access to your computer with other students without prior lock--out and disabling of automatic login functionality. Do not just leave your computer on without a lock even if it is just for the sake of a 5--minute break.
• … work in teams. You may meet to discuss generally about the material, but any work on the homework is to be done individually and in privacy. Remember, you may not allow anyone to even just read your source code.

With the Internet, "paste", and "share" are easy operations. Don’t think that it is easy to hide and that we will not find you, we have just as easy to use, fully automatic and intelligent tools that will identify any potential cases of plagiarism. And do not think that being the original author will make any difference. Sharing an original solution with others is just as unethical as using someone else’s work.

CS100 Homework 3 (Fall, 2019)

In this homework, you will practice with C strings and structures, and systematically use them to ultimately form a simple database.

Percentage of this homework over the whole score: 7%

Submission deadline:

2019-10-15 23:59

Overview

In this homework, you will ultimately implement a database for storing students' information, including their names, and their scores in the course CS999. You will put the information in student structures (part 1), store the structures using a linked list and implement the related functions to operate the list (part 2), and finally search for information in your database (part 3).

This is a step-by-step homework. In other words, finishing part 1 is necessary before you start working on part 2. You won't need to write three individual files. Just write in one file, step by step.

Important: New rules on using OJ

For this homework, there are no specific inputs or outputs, and **main() function is not required** in submission. Therefore, you won't need to worry about output formats. Moreover, you can customize your own main() function and use it for debugging. But note that **you should not submit your main function to OJ, just submit the rest of your codes, otherwise compile errors may happen.**

To encourage debugging locally and not on the OJ output (which is not right for practical programming), the OJ will not be available from the date the homework is released. It will gradually open until the submission deadline passes, and the schedule is:

Part 1 submission will be available on 10/11 at 17:00.
Part 2 submission will be available on 10/13 at 17:00.
Part 3 submission will be available on 10/14 at 00:00.

You can also see this schedule on OJ homepage.

<string.h> usage:

As this homework involves C strings, we have included <string.h> in our provided template. You can use functions like strcpy() and strlen() freely.
However, because part of the homework is writing a comparison function, we have disabled strcmp() and related functions on OJ. Using them will result in compile errors.

Part 1: Sorting C Strings and Structure for Students (25%)

Sorting C Strings:

To begin this relatively larger project to form our homework3, we need some essential preparations.

First, let's implement a function to sort strings, as we want ordered output when we use this database. The function sorts an array of strings, and it looks like this:

```
void string_sort (char* strings[], int size);
```

Your will need to sort strings in *lexicographical order*, which is based on the alphabetical order of their component letters. Here is a demo:

```
char* str[] = {"Finish", "Dog", "dog", "Egg", "Eggplant", "Fish"};
string_sort(str, 6);
//Now str[] is {"Dog", "Egg", "Eggplant", "Finish", "Fish", "dog"}
//If you feel hard to understand, explanation is below.
```

In order for you to practice, the function `strcmp` from `<string.h>` has been disabled on Online Judge, and you will be writing your own string comparison function:

```
int compare (const char* str1, const char* str2);
```

The comparison of two strings starts by comparing the first letters. If they are the same, proceed to the next letter, until you encounter two different letters or the end of one string. A string is smaller if its different letter comes before the other's in dictionary. Here are some examples:

`"Dog" < "Egg"` (`'D'` comes before `'E'`)
`"Egg" < "Eggplant"`
(`"Egg"` ends before `"Eggplant"`, or, `"Egg"` has `'\0'` after the second `'g'`, and `'\0'` comes before `'p'`)
`"Eggplant" < "Finish"` (`'E'` comes before `'F'`)
`"Finish" < "Fish"` (When comparing the third letter, `'n' < 's'`)
`"Fish" < "dog"`
(In ASCII coding, uppercase letter `'F'` comes before lowercase letter `'d'`)

In fact, directly comparing each character will be safe, as C language compares char types by their ASCII values.
Your comparison function should return 1 if str1 is greater than str2, -1 if str1 is smaller than str2, and 0 if they are equal.

For the sorting part, you are going to use “bubble sort” algorithm, as you've implemented in homework 2. Its pseudocode is given below:

```
procedure bubble_sort (A: list of sortable items)
   n = length(A)
   for i = 0 to n - 1
      for j = 1 to n-1
         if A[j-1] > A[j] then
            swap(A[j-1], A[j])
         end if
      end for
   end for
end procedure
```

Problem Description:

You will implement the function `string_sort`. To do this, you may also need a comparison function. We do not have any restriction for the comparison function, but we have provided `compare` in our code template.

The function `string_sort` takes an array of strings and its size as parameters. It does not have a return value, but will sort the array in its position.

- **It is important that you should NOT change the name or the parameter list of this function, or it will not be recognized by Online Judge. If you feel like you need to do so, write an auxiliary function instead.**

If you want to implement the provided template of `compare`, it takes two strings as parameters, and it should return:

1 if str1 is greater than str2;
0 if str1 is precisely the same as str2;
-1 if str1 is smaller than str2.

- **We won't check this function, as it is only called by `string_sort`. You may change its name or parameters, or even write your own one instead.**

Structure for Students:

You will also need a structure for students to store the essential information, including their names and their scores. The structure should contain a C string (`char name[NAME_SIZE]`) for a student's name, and a `float` for the student's score. It may contain any other member variables if you want.

Note: In our template, NAME_SIZE is pre-defined to be 64.

Note: You can see this statement in the template:

```
typedef /*YOUR STRUCTURE*/ Student;
```

This statement is for simplifying your code. When declaring a student variable, instead of `struct /*some name*/ foo;`**, you can simply say** `Student foo;`.

Please write your structure at the position of this comment, or, before this statement and put the name of your structure at the position of this comment. You can refer to the slides for detailed information on forms when writing a structure.

Moreover, you should also implement a function `new_student` to initialize student variables. It looks like this:

```
Student* new_student(const char* _name, float _score);
```

Your function should take a C string and a `float` as parameters. It dynamically creates a `Student` variable, initializes it with given parameters, and returns a pointer to it. For example, you will use your function like so:

```
Student* geziWang = new_student("Gezi Wang", 0.0);
```

Problem Description:
You are going to implement the structure for students, as well as a function for creating a new student.
Your structure will at least contain member variables of a C string and a `float`. **Its typename should be type-defined as `Student`.**
Your function **should have the name of `new_student`, and take a C string and a floating point number as parameter. You should dynamically create a `Student` variable in your function, and return a pointer to the `Student` you created.**

Be careful that when you dynamically create the student structure, you should free it from memory in somewhere else!

How to debug (same for part 2&3):
As this homework does not require output, you should debug by testing the functions and structures you wrote in main function. For example, you can write the following lines:

```
int main()
{
    char* str[] = {"Finish", "Dog", "dog", "Egg", "Eggplant", "Fish"};
    string_sort(str, 6);
    for (int i = 0; i < 6; i++)
        printf("%s\n", str[i]);
    Student* geziWang = new_student("Gezi Wang", 0.0);
    printf("%s %f\n", geziWang->name, geziWang->score);
}
```

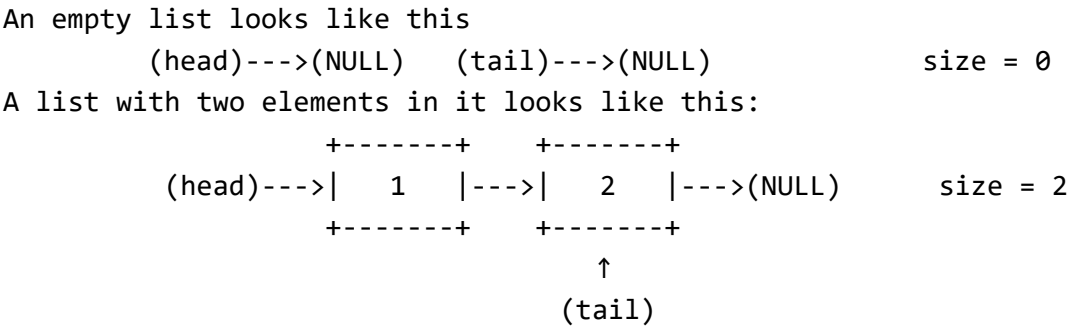
Then, you can run the code and see if it behaves well. You can also write your own test cases.

When submitting part 1:
We strongly recommend you submit your homework to OJ **only after you have thoroughly tested your own main function locally**. You should submit all your codes for the work above, **EXCEPT the main function**. Please make sure you do not change the name of functions `string_sort` and `new_student`, and that your structure has been type-defined as `Student`.

Part 2: Constructing a Linked List for Students (40%)

In part 2, you are required to implement a singly linked-list. Our singly linked lists have at least these members: a pointer "head" pointing to the first node, and a pointer "tail" pointing to the last node, and a "size" indicating the number of nodes. In addition, the "next" pointer of the last node should be set to NULL, indicating the end of the list. For interior nodes of the list, it should have a pointer "next" pointing to its next node in list, which should also be pointed to by its previous node using its "next" pointer.

The basic structure of our list:



Note: You can see this statement in the template:

```
typedef /*YOUR STRUCTURE*/ LinkedList;
```

Please write your structure at the position of this comment, or, before this statement and put the name of your structure at the position of this comment. You can refer to the slides for detailed information on forms when writing a structure.

Note: The `head` and `tail` in this structure are pointers to nodes. You need to initialize the head and tail to NULL when the list is created and before any node is added.

For each node in list, it should contain:

(1) A pointer to the data (which should be a Student).

(2) A pointer to next node.

Note: You can see this statement in the template:

```
typedef /*YOUR STRUCTURE*/ Node;
```

Please write your structure at the position of this comment, or, before this statement and put the name of your structure at the position of this comment. You can refer to the slides for detailed information on forms when writing a structure.

More detailed, you need to implement these functions in this part:

- `void list_init(LinkedList* l);`

In this function, you should initialize your list, the size should be initialized to 0, the head and tail should be NULL.

It does not have a return value.

- `void list_insert(LinkedList* l, Student* stu_ptr);`

In this function, you should create a new node contains the given student, and insert this node into your list **immediately after the last node (which tail points to).** Also, you need to update tail to the new position.

It does not have a return value.

Note: Do not forget to update size of the list.

- `void list_erase(LinkedList* l);`

In this function, you should erase the whole linked list (i.e. remove all nodes and data in them), and **free all memory you allocated.**

It does not have a return value.

Note: Do not forget to update size of the list.

- `Node* list_remove(LinkedList* l, Node* target);`

In this function, the additional parameter is a pointer to a node (if well-behaved, this node should be in the list). First, you should traverse the list and check if the target is in the list. If it is not found, return NULL. If it is found, you should remove it from list and re-link **involved pointers(*)**, and make sure you don't forget to **free the memory you allocated**! If you have successfully removed a node, this function should **return the (originally) next node of target node**.

Note: Do not forget to update size of the list.

(*) When you delete a node, there are many questions that you should ask yourself. Take the following one as a hint:

Is it a node in the middle of the linked list, or maybe in a special position? Is there any pointer that the special position may affect?

Problem Description:

You will need to implement a structure of a LinkedList. To do this, you may also need a structure of a Node.

You also need to implement 4 LinkedList operations as shown. Please pay attention to parameter lists and return types of these functions.

The type name of your linked list structure and node structure should be type-defined to be `LinkedList` and `Node`, respectively.

You should not change the name of any function listed above. Otherwise compile errors may happen.

When submitting Part 2:

As in part 1, you should not submit the main function. Your submission should include the implementation of LinkedList, the above functions, and also the implementation for Student (part 1). A simple and safe way is to just submit all your codes, except the main function.

Part 3: Searching Data (35%)

In this part, you will use the linked list you implemented in part 2, and search for useful data which is already stored into your database.
You may need to search for the Student structure given his/her name. In this way, you can check a student's score. You may also need to search for all students in a given score range.

In particular, you will need to implement the following two functions:

```
- Node* list_search_by_name(LinkedList* l, char* search_name);
```

In this function, you should search the whole list, and find student that has same name with the given "search_name". If you find this student, **return a pointer to the node that contains it.** Return NULL if no result is found or invalid arguments (i.e. "search_name == NULL").

Note: There are no duplicate names in our list.

```
- int list_search_by_score(LinkedList* l,
                           float lower_bound,
                           float upper_bound,
                           char* result[RES_SIZE]);
```

In this function, you should search the whole list, and find student(s) that has score in the range `[lower_bound, upper_bound]`. You should find all students with score in this range, and **copy their names** into "`result`".
Then, you should use the `string_sort` function you implemented in part 1 to sort `result`.
Finally, this function returns the number of students in `result`. Return 0 if no result is found or the arguments are invalid. Also, please note that when the function returns, the names in result should be ordered.

Note: If your implementation is correct, the number of students you found will not exceed the size of the result, so don't worry.

Note: In our template, RES_SIZE is pre-defined to be 128.

Problem Description:
You will need to implement two functions. Please pay attention to parameter lists and return types of these functions.
You should not change the name of any function listed above. Otherwise compile errors may happen.

When submitting part 3:
You need to submit codes for all three parts together. A simple and safe way is to just submit all your codes, except the main function.