ASSIGNMENT 3 CMPT 125

SUBMISSION

This assignment consists of one problem. You will submit 1 text file containing the commented C code that solves that problem.
For the first problem you will submit a file with the name addressBook.c.

MARKING SCHEME

80% of the marks for each problem will be based on the results produced by running your code for a series of tests.

- Most of these tests will be provided with the problem in the same Canvas Module
- Some tests will be added for grading and will be provided when your graded assignment is returned.
- Any functionality, constraint or other specification of the program requested in the problem description in this document may be tested when the code is graded.
- Some functionalities, constraints, or other specifications of any program described in this document may not be tested by the provided tests.

20% of the marks will be for direct examination of the code. This will include points for

- Following the class coding standard. (see file posted with this assignment)
- Implementation of specific portions of the solution.

EXPECTATIONS

1. A series of tests will be posted with this problem.
   a. Each test will specify, in red, the values input to your code.
   b. For this assignment verification is necessary to catch many of the errors.
   c. Each test will also specify all the expected output values and the formatting of the prompts, inputs and outputs that your code will be expected to produce
2. At least one file containing the expected results of running your program will be posted. These files will define the expected details of spacing and other formatting for each test and the expected values of each output.
3. Your output for a test and the provided output, for the same test, must match character by character to receive full marks for a test. Please watch the short video tutorial "Checking your outputs" for the details of how to compare your output to the corresponding provided output file.

You will use the environment specified below to build your solutions to assignments. Why we need a common environment and how to access the environment four this course is explained in the "Remote Access tutorial".
ENVIRONMENT: Ubuntu 22.04, gcc 11.4.0, gdb 12.1, vscode 1.82.2 (may be updated before 1st class)

Problem Title: Dynamic Address Book Application in C

Design and implement an address book application in C that uses dynamic memory allocation to manage a list of contacts. Each contact should be represented by an instance of the following structure:

```
typedef struct Contact {
   char *firstName;
   char *familyName;
   long long phoneNum;  /* 10-digit phone number stored as a 64-bit integer */
   char *address;
   int age;
} Contact;
```

The variables in this structure represent:

- **firstName:** A dynamically allocated string for the Contact's first name.
- **familyName:** A dynamically allocated string for the Contact's family (last) name.
- **phoneNum:** A 10-digit phone number (stored as a 64-bit integer). The program must ensure that the phone number entered is exactly 10 digits.
- **address:** A dynamically allocated string for the Contact's address.
- **age:** An integer representing the Contact's age.

The address book must be implemented as a dynamic, NULL-terminated array of pointers to Contacts. **For all Contacts, each string allocated must be given the minimum possible length.**

DO NOT use a global variable to keep track of the number of Contacts, or pass the number of elements in the array as an argument to functions. Instead, you will implement and use a helper function countContacts that returns the current count by scanning the array until the NULL terminator is reached.

Your addressBook will be managed using a menu system described in the section **Menu System for Address Book Program**. To implement the addressBook system you will use the functions described in the section **Required Functions for the addressBook system** to do the tasks for each item in the menu. You may not change the prototypes of the required functions. You must use the required functions.

You may add additional helper functions as you see fit. These helper functions should be used by your implementations of the required functions,

You must use case statements in the main program to implement selection of choices from the supplied menus.

Your program should not have any memory leaks or hanging pointers. If you want to test if you program has leaks you can the tool valgrind. It will run your executable in a command window and will tell you if and where you have problems with memory management. Valgrind is available in DVI or on Linux machine in the Lab.

Menu System for Address Book Program

When the program starts, it presents the following menu:

```
Address Book Menu
1.  Append Contact
2.  Insert Contact in Alphabetical Order
3.  Remove Contact by Index
4.  Remove Contact by Full Name
5.  Find and Edit Contact
6.  List Contacts
7.  Print Contacts to File with the format of an input file
8.  Print Contacts to File (Human Readable)
9.  Load Contacts from File Replacing Existing Contacts
10. Append Contacts from File
11. Merge Contacts from File
12. Exit
Choose an option:
```

The user selects an option by entering the corresponding number. The menu system calls the required functions of the menu system to accomplish each of the twelve set tasks.

1. Append Contact

This option adds a new **C**ontact to the address book. The user is prompted to enter details such as the first name, family name, address, phone number, and age. The **C**ontact is added at the end of the list.

2. Insert Contact in Alphabetical Order

This option inserts a new **C**ontact into the address book while maintaining the list of **C**ontacts in alphabetical order by family name and first name. The list should be in alphabetical order based on the family name. If there are multiple entries with the same family name then the Contacts with the same family name should be ordered by first name. The user is prompted to enter details such as the first name, last name, address, phone number, and age. The pointer to the new Contact is placed into the location that assures the list will stay in alphabetical order. It will usually be necessary to move other pointers to make room for the new pointer to the new Contact.

3. Remove Contact by Index

This option removes a **C**ontact based on its index position in the list. The user enters an index, and if it is valid, the corresponding **C**ontact is deleted, and the list is adjusted accordingly. The index position is 0 for the first element in the array.

4. Remove Contact by Full Name

This option allows the user to remove a **C**ontact by specifying the first and last name. If a matching **C**ontact is found, it is removed from the list, and the memory for the removed Contact is freed.

5. Find and Edit Contact

This option allows the user to locate a **C**ontact by index and modify its details. The user can choose to edit fields such as first name, last name, address, phone number, or age. After selecting a **C**ontact, the following sub-menu is displayed:

```
1. Edit First Name
2. Edit Last Name
3. Edit Address
4. Edit Phone Number
5. Edit Age
6. Cancel
```

The user selects an option, enters new data, and the **C**ontact is updated.

6. List Contacts

This option displays all **C**ontacts stored in the address book to the screen. Each **C**ontact's details, including name, phone number, address, and age, are printed in a structured format shown in the examples and the detailed discussion of required functions below

7. Print Contacts to File (Input File Format)

This option saves the **C**ontact list to a file in a structured format suitable for reloading. Each **C**ontact's details are written in a simple, line-by-line format that can be read by the program later. The user supplies the file name in which the data will be stored. For actual format see the examples and the detailed description function descriptions below.

8. Print Contacts to File (Human Readable)

This option saves the **C**ontact list to a file in a user-friendly format. Each **C**ontact is printed in a structured way with labels, making it easy for a person to read and review. The user will provide the name of the file in which the Contacts are to be saved.

9. Load Contacts from File (Replace Existing Contacts)

This option replaces the current **C**ontact list with **C**ontacts from a file. Any existing **C**ontacts in memory are discarded, and only those from the file remain in the address book. Contacts are stored in the file being loaded in the format given in option 7.

10. Append Contacts from File

This option adds Contacts from a file to the existing Contact list without removing the current Contacts. New Contacts are added to the end of the list in the order they are read. No duplicates Contacts are added to the list. Contacts are stored in the file being loaded are in the format given in option 7.

11. Merge Contacts from File

This option adds Contacts from a file to the existing Contact list without removing the current Contacts from the existing list. New Contacts are inserted to maintain alphabetical order of the list of Contacts. Contacts are inserted in the order they are read. Duplicate Contacts are not added to the list of Contacts. Contacts are stored in the file being loaded are in the format given in option 7.

12. Exit

This option exits the program. Before exiting, all allocated memory for Contacts is freed to prevent memory leaks.

Required Functions for the addressBook system

Your program must provide the functionality needed by the tasks in the menu using the functions specified. Do not change any of the function prototypes. You may add helper functions to be used within the functions below.

countContacts

int countContacts(Contact **contacts);

Counts the number of Contacts in contacts in a NULL-terminated array of dynamically allocated Contact structures. This function returns the number of Contacts in the list.

readNewContact

Contact *readNewContact();

A call to this function (including execution of any helper functions it calls) creates a new Contact. Creating the Contact includes allocating memory, checking that memory has been allocated, prompting for values to initialize the Contact, and reading and verifying those values. If memory allocation, or data reading / verification fails then readNewContact cleans up then returns NULL. Error messages will, in some cases, be printed to identify the reason for return of a NULL pointer.

Prompts the user to input a new Contact's details. The following prompts should be used

```
Enter the first name:
Enter the family name:
Enter the address:
```

Enter 10-digit phone number that must not start with 0:
Enter the age:

Valid input for phone number is a **10 digit integer that does not have a first digit 0.** If invalid input is entered for the phone number, the user is prompted to try again using the prompt below. The user is prompted a maximum of 5 times (including the original prompt).
Error: Invalid phone number. Try again:
Valid input for age is **1<=age<=150**. If invalid input is entered for the age, the user is prompted to try again using the prompt below.
Error: Invalid age. Try again:
The user is prompted a maximum of 5 times (including the original prompt) for each variable.
If the read fails 5 times for phoneNumber then phoneNumber is set to 0 and an error message is printed
Error: Could not read a valid phone number
If the read fails 5 times for age, age is set to 0 and an error message is printed
Error: Could not read a valid age
If memory allocation fails the function returns a NULL pointer and it prints one of the following:
Error: Memory allocation failed for Contact in readNewContact
Error: unable to allocate memory for the first name string
Error: unable to allocate memory for the family name string
Error: unable to allocate memory for the address string

appendContact

Contact **appendContact(Contact **contacts, Contact *newContact);

The array, contacts, is a NULL terminated array of pointers to Contact Structures. This function extends the NULL terminated array of pointers to **C**ontacts to hold one more **C**ontact. Then this function places the pointer to newContact into the first empty location in the extended array of contacts (the last element of the NULL terminated array of pointers before it was extended].

This function expects an existing array of pointers to contacts as its first argument, and a pointer to an existing Contact as its second argument. **If the second** argument is NULL the function will return the pointer to a pointer to a Contact passed into the function (contacts).

If memory reallocation fails, it prints the message below then cleans up and exits:
Error: Memory reallocation error in appendContact
Otherwise, it prints the message below:
Contact appended successfully by appendContact

insertContactAlphabetical

Contact **insertContactAlphabetical(Contact **contacts, Contact *newContact);

The array, contacts, is a NULL terminated array of pointers to Contact Structures. This function extends the NULL terminated array of pointers to **C**ontacts to hold one more **C**ontact. Then this function determines the index such that contacts[index] is the location where the pointer to newContact should be placed. It moves other pointers to Contacts to make room in the location

contacts[index] and to assure the array contacts remains NULL terminated. Then, this function places the pointer to the new Contact into contacts[index].

This function expects an existing array of Contacts as its first argument, and an existing Contact as its second argument. If **the second** argument is NULL the function will return the pointer to a pointer to a Contact passed into the function (Contacts).
If memory reallocation fails, it prints:
Error: Memory reallocation error in insertContactAlphabetical
Otherwise, it prints:
Contact was successfully added in alphabetical order

removeContactByIndex

Contact **removeContactByIndex(Contact **contacts);

If the pointer to a pointer to a Contact receives a value of NULL from the calling function then immediately print the following message and return NULL.
Error: value of addressBook received in removeContactByIndex was NULL

The function prompts the user for an index to remove, the first element in the array is at index 0.
Removing a Contact by index
Enter index to remove (0 based):

Then it tries to read the value. If the value supplied by the user will not result in reading an integer (using scanf). Then the function will print the error
Error: Value of index supplied could not be read.

If the index is out of range, it prints the error message below then returns the pointer to a pointer contacts :
Error: Index out of range in removeContactByIndex

After the Contact is removed and some of the other pointers to Contacts have moved to fill in the empty array element that used to hold the deleted Contact, the array will be reallocated to be the correct size for reduced size of the array.
If memory reallocation fails, it prints:
Error: Memory reallocation failed in removeContactByIndex
Otherwise, it prints:
Contact removed successfully by removeContactByIndex

The function will then return a pointer to the modified array of pointers to **C**ontacts

removeContactByFullName

int removeContactByFullName(Contact ***contacts);

If the pointer to the pointer to the array of Contacts receives a value of NULL from the calling function then immediately print the following message and then return **0**.
Error: value of contacts received in removeContactByFullName was NULL
Unnecessary lines deleted

Prompts the user for the first name then the **family** name in the Contact to remove:
Enter first name:
Enter family name:

The first name and family name must both match in the Contact and in the list for the Contact to be removed. A call to removeContactByFullName will remove only the first matching Contact it finds. After the **C**ontact has been removed and its memory has been freed the pointers to some other Contacts will need to be moved to fill in the empty array location without changing the order of the remaining Contacts.

After the Contact is removed and some of the other pointers to Contacts have moved to fill in the empty array element that used to hold the deleted Contact, the array will be reallocated to be the correct size for reduced size of the array.
If the **C**ontact is found and removed, it prints:
Contact '<firstName> <lastName>' removed successfully
If no match is found, it prints:
Contact '<firstName> <lastName>' not found

The function will return **1** if any changes were made **and 2 if no matching link was found.** If the **C**ontact was not found the function will return the pointer to the original array of pointers passed into the function

listContacts

void listContacts(Contact **contacts);

Prints all Contacts stored in contacts array to the screen. Use the countContacts function to determine the number of Contacts to print. The 1-D array contacts is a NULL-terminated dynamic array of pointers to Contacts.
If the **C**ontact list is empty, it prints:
Error: No contacts available.

Otherwise, it prints each **C**ontact in the following format, note that the number identifying each **C**ontact is the array index where the pointer to the Contact is stored plus 1:

```
1. John Doe
   Phone: 1234567890
   Address: 123 Elm St
   Age: 30
2. Alice Smith
   Phone: 1234234555
   Address: 55 Snake Ave
   Age: 19
```

saveContactsToFile

```
void saveContactsToFile(Contact **contacts, char *filename);
```

This option saves the Contact list to a file in a structured format suitable for reloading. Each Contact's details are written in a simple, line-by-line format that can be read by the program later.
The user supplies two arguments, the name of the file in which the data will be stored and the pointer to the array of pointers to Contacts.
If the pointer to a character supplied to the function is passed the value NULL print the following message then return to the calling program.

```
Error: filename formal parameter passed value NULL in saveContactsToFile
```

If the pointer to the array of pointers to Contacts is passed the value NULL print the following message then return to the calling program.

```
Error: contacts formal parameter  passed value NULL in saveContactsToFile
```

If the pointer to the array of pointer to Contacts pa

The file with the name assed in in the parameter filename is opened. The function checks to assure that the file did open. If it did not the error below is printed and then the function returns.

```
Error: file not opened in saveContactsToFile
```

Otherwise, it prints each Contact in the following format

```
2
Jane
Smith
456 Oak St
9876543210
25
Alice
Johnson
789 Maple Ave
5551234567
40
```

printContactsToFile

```
void printContactsToFile(Contact **contacts, char *filename);
```

This option prints the Contact list to a file in a format that is clear and concise for human reading.
The user supplies two arguments, the name of the file in which the data will be stored and the pointer to the array of pointers to Contacts.
If the pointer to a character supplied to the function is passed the value NULL print the following message then return to the calling program.

```
Error: filename formal parameter passed value NULL in printContactsToFile
```

If the pointer to the array of pointers to Contacts is passed the value NULL print the following message then return to the calling program.

```
Error: contacts formal parameter  passed value NULL in printContactsToFile
```

The file with the name passed in in the parameter filename is opened. The function also checks to assure that the file did open. If it did not open the error below is printed and then the function returns.
Error: file not opened in printContactsToFile

Then the Contacts are printed into the file using the following format. Note the numbers counting each Contact are the index of the element containing the pointer to the Contact + 1

```
Address Book Report
-------------------
1. John Doe
   Phone: 1234567890
   Address: 123 Elm St
   Age: 30

2. Jane Smith
   Phone: 9876543210
   Address: 456 Oak St
   Age: 25

-------------------
Total Contacts: 2
```

loadContactsFromFile

```
Contact **loadContactsFromFile(Contact ** addressBook,
                               char *filename);
```

Loads the contacts from a file containing Contacts with the format given in saveContactsToFile. Contacts in the loaded file replace the contents presently in the list.

The filename passed in is for the filename containing the contacts. That file is opened. If the file fails to open the error message below is printed and the function loadContactsFromFile returns NULL.
Error: File to load not found

The number of Contacts in the file is then read. If the number of Contacts is not read correctly then the file is closed the error message below is printed and then the function returns NULL.
Error: Memory allocation error, addressBook in loadContactsFromFile

If the addressBook passed into the function is not NULL, then, the addressBook passed in freed.

For each contact in the file a new contact is created.
If the memory allocation for one of the new Contacts fails then the error message below is printed, the %d is replaced with the index of the Contact whose allocation failed. After printing the message clean up and return NULL.
Error: Memory allocation error, Contact %d in loadContactsFromFile

After successfully allocating a new Contact read the values to initialize it from the file. Data read should be verified. The following errors may occur if data verification fails.

```
Error: Invalid phone number.
Error: Invalid age.
Error: Memory allocation error, memory for string in Contact %d not allocated
```

These messages should result in a value of 0 in the contact for phone number or age respectively. The memory allocation error should cause the program to exit.

If the function loads all Contacts successfully it should print

```
Contacts loaded from file:  filename
```

Where filename is the name of the file that you loaded the Contacts from

appendContactsFromFile

```
Contact **appendContactsFromFile(Contact **contacts, char *filename);
```

Loads the contacts from a file containing Contacts with the format given in saveContactsToFile. Contacts in the loaded file are appended one by one into the existing list. Duplicate nodes (nodes with the same first name AND the same family name) are not added to the list a second time.

After appending the Contacts from the file prints the message below. The %s in the message should be replaced by the name of the file from which the contacts are being loaded.

```
Appended contacts from %s
```

mergeContactsFromFile

```
Contact **mergeContactsFromFile(Contact **contacts, char *filename);
```

Loads the contacts from a file containing Contacts with the format given in saveContactsToFile. Contacts in the loaded file are inserted one by one into the existing list. Duplicate nodes (nodes with the same first name AND the same family name) are not added to the list a second time.

If the list of Contacts already in the system is not ordered, do not sort them. The merged Contacts should be inserted as if the list were sorted. That is, any Contact will be placed immediately before the first contact in the list that is alphabetically after the link being inserted.

After appending the Contacts from the file prints the message below. The %s in the message should be replaced by the name of the file from which the contacts are being loaded.

```
Appended contacts from %s
```

editContact

`Contact **editContact(Contact **contacts, int index );`
The second argument, index, gives the index in the array that holds the pointer to the contact you want to edit. (0 based)
The function first counts the available Contacts in the array of points to contacts.

If there are no pointers to Contacts in the array of pointers to Contacts then the following information message will be printed and then the pointer to the Array of pointers to Contacts passed into the function will be returned to the calling program.
`No contacts available to edit`

The program will then prompt for the index of the Contact to be edited, using the following prompt
Enter index of contact to edit (0-%d):
Where %d indicates the maximum index of a Contact in the array.

If the user enters a character that is not a digit as the first character in their response, or enters an integer that is out of range then the following error is printed and the program prints the base menu again.
`Error: Invalid Index`

If the index is valid then the information message below is printed
`Editing contact: %s %s`
The first %s represents the first name in the selected Contact and the second %s represents the family name in the selected Contact.

Next, a second menu is printed, which allows the user to specify which of the values in the Contact are to be edited.

```
1. Edit First Name
2. Edit Last Name
3. Edit Address
4. Edit Phone Number
5. Edit Age
6. Cancel
Choose an option:
```

Each of the first 5 options produces one of the following prompts

```
Enter new first name:
Enter new family name:
Enter new address:
Enter new phone number: Enter 10-digit phone number that must not
start with 0:
Enter new age:
```

For option 6 the information message is printed followed by the first menu
`Edit cancelled`

The user provided information will be read and verified.

Memory allocation errors for strings will produce the following error then clean up and exit the program.

```
Error: Memory allocation error for string in editContact
```

Errors in reading phone numbers and ages will produce the same messages seen in readNewContact.
If no valid phone number or no valid age is read, be sure the age or phone number is unchanged, print the second menu and continue.