COMP 2401 A/C – Assignment #4

Due: Thursday, November 23 at 11:59 pm

1. Goal

For this assignment, you will write a program in C, in the Ubuntu Linux environment, that allows the end user to manage a restaurant, with patrons and reservations that are stored in different types of collection structures.

2. Learning Outcomes

With this assignment, you will:

- write a program that manipulates collections using encapsulated functionality
- implement different kinds of collections, including a dynamically allocated array and a doubly linked list
- practice writing code that dynamically allocates its data and manages multiple pointers to the same data

3. Instructions

The program will present the end user with a menu of options to view restaurant data, including reservations and patrons. The user will have the option to print all the patron (customer) data, or print all the reservations at the restaurant, or print out the reservations for one patron only.

Your program must follow the programming conventions that we saw in the course lectures, including but not restricted to the correct separation of code into modular functions, basic error checking, and the documentation of each function. A more comprehensive list of constraints can be found in the *Constraints* section of this document.

3.1. Understand the data representation

You will begin by understanding the base code provided in the `a4-posted.tar` file, which is available in *Brightspace*.

The base code contains the following structure data types that your program must use:

3.1.1. the `PatronType` contains the information for one patron (customer) of the restaurant

3.1.2. the `ResvTimeType` contains the day and time information for a reservation at the restaurant

3.1.3. the `ResvType` represents one reservation at the restaurant, including the date and time for that reservation, and a pointer to the patron that made the reservation

NOTE: It is mandatory that only **one instance** of each patron and each reservation exist in the program. Your program must not create any copies of this data. Instead, it must store multiple pointers to the same instance of data, where required.

3.2. Define your structure data types

You must update the `defs.h` header file with the new data types that you create, and you must add all function prototypes as forward references to this header file as well. **Do not** make any unauthorized changes to the provided code, including the function prototypes.

You will define the following new data types:

3.2.1. The `PatronArrayType` structure type must contain the following:

(a) an elements field, which must be declared as a dynamically allocated array of `PatronType` *structures* (not pointers), as we saw in the in-class coding example of section 3.3, program #1

(b) a size field that tracks the current number of patrons in the elements array

(c) a `nextId` integer field that stores the unique identifier that will be assigned to the next patron added to the array

3.2.2. The `ResvListType` structure type defines a doubly linked list, with only a head (and no tail). The list structure must also have a `nextId` integer field that stores the unique identifier that will be assigned to the next reservation added to the list.

3.2.3. The `NodeType` structure type works with the `ResvListType` structure to implement a *doubly* linked list of `ResvType` data (as a pointer, as we saw in the in-class coding examples).

3.2.4. The `RestaurantType` structure type represents a restaurant in our program. It contains three fields: the restaurant's name as a string, a linked list of all the reservations in the restaurant, stored as a `ResvListType` structure, and an array of all the restaurant patrons, as an instance of the `PatronArrayType` type. All three fields must be allocated statically.

3.3. **Organize the code**

3.3.1. Your submission must separate the program's functions into the following source files:

- `main.c` : `main()`, `printMenu()`
- `load.c` : `loadResData()`, `loadPatronData()`
- `restaurant.c` : `initRestaurant()`, `createResv()`, `printResByPatron()`, `cleanupRestaurant()`, `validateResvTime()`
- `resv.c` : `initResvList()`, `initResv()`, `initResvTime()`, `addResv()`, `lessThan()`, `printReservations()`, `printReservation()`, `cleanupResvList()`
- `patrons.c` : `initPatronArray()`, `addPatron()`, `findPatron()`, `printPatrons()`, `cleanupPatronArray()`

3.3.2. Each source file must `#include` the `defs.h` file, in order to have access to its contents.

3.3.3. Because we have not yet covered the details of compiling and linking, we have to use a somewhat "bad" way of compiling these multiple source files together. For example, if your program contains source files `file1.c`, `file2.c`, and `file3.c`, then you can use the following command to create the `a4` executable: `gcc -o a4 file1.c file2.c file3.c`

3.4. **Implement the patron management functions**

3.4.1. Implement the `void initPatronArray(PatronArrayType *arr)` function that initializes every field of the given patron collection to default values. The elements array must be dynamically allocated in this function, using a capacity provided as a constant in the base code; the number of patrons stored in a new array is always zero; and the starting point for unique patron identifiers can be found in a provided constant.

3.4.2. Implement the `void addPatron(PatronArrayType *arr, char *n)` function that adds a new patron to the back of the given patron collection. Because it consists of an array of structures (and not pointers), adding to the array means initializing the fields of the next element after the last initialized one. The new patron's name is set to the given `n` parameter, and its id is initialized from the next available id in the collection, so that unique ids are generated in a sequential manner.

3.4.3. Implement the `int findPatron(PatronArrayType *arr, int id, PatronType **p)` function that searches through the given patron collection to find the patron with the given id, and "returns" that patron using the `p` parameter. The function returns an error flag if the patron is not found, or a success flag otherwise.

3.4.4. Implement the `void printPatrons(PatronArrayType *arr)` function that prints out to the screen all the details of every patron in the given patron collection.

3.4.5. Implement the `void cleanupPatronArray(PatronArrayType *arr)` function that deallocates all the required dynamically allocated memory for the given patron collection.

3.5. **Implement the reservation management functions**

3.5.1. Implement the `void initResvList(ResvListType *list)` function that initializes both fields of the given `list` parameter to default values. Collections always start out empty, and the starting point for unique reservation identifiers can be found in a provided constant.

3.5.2. Implement the `void initResvTime(ResvTimeType **rt, int yr, int mth, int day, int hr, int min)` function that dynamically allocates memory for a `ResvTimeType` structure, initializes its fields to the given parameters, and "returns" this new structure using the `rt` parameter.

3.5.3. Implement the `void initResv(ResvType **r, PatronType *p, ResvTimeType *rt)` function that dynamically allocates memory for a `ResvType` structure, initializes its patron and reservation time fields to the given parameters, and "returns" this new structure using the `r` parameter. The reservation id must be set to a temporary value of zero until the new reservation is added to a reservation list in another function.

3.5.4. Implement the `int lessThan(ResvTimeType *r1, ResvTimeType *r2)` function that compares two reservation times, and returns the constant associated with true if the first reservation begins earlier in date and time than the second one, and the constant for false otherwise.

3.5.5. Implement the `void addResv(ResvListType *list, ResvType *r)` function that adds the reservation `r` *in its correct position* in the given `list`, so that the list remains ordered in ascending (increasing) order by reservation time. The linked list must be implemented *in accordance with the in-class coding examples*, with no dummy nodes.

This function must ensure that a unique identifier is generated automatically, in a sequentially increasing manner, and assigned to the new reservation.

3.5.6. Implement the `void printReservation(ResvType *r)` function that prints out all the details of the given reservation. The output must match the workshop video, and every field must be aligned with the other reservations. The output for the month, day, hours, and minutes fields must be padded with zeros, as shown in the video.

3.5.7. Implement the `void printReservations(ResvListType *list)` function that prints out the details of every reservation in the given list. Because the list is doubly linked, it must be printed out **twice**: once in the forward direction, using the `next` links, and again in the backward direction, using the `prev` links. This step is mandatory to demonstrate that the links are correctly initialized when reservations are added to the list.

3.5.8. Implement the `void cleanupResvList(ResvListType *list)` function that deallocates all the required dynamically allocated memory for the given list.

3.6. Implement the restaurant management functions

3.6.1. Implement the `void initRestaurant(RestaurantType *r, char *n)` function that initializes the three fields of the given restaurant structure. The name must initialized from the given parameter, and the initialization of the other fields must be performed by calling existing functions.

3.6.2. Implement the `int validateResvTime(int yr, int mth, int day, int hr, int min)` function that checks that the parameters represent a valid date and time. The year must be the current year or a future one, and the other parameters must be within valid ranges. The function returns an error flag if one or more of the fields is invalid, or a success flag otherwise.

3.6.3. Implement the `void createResv(RestaurantType *r, int pId, int yr, int mth, int day, int hr, int min)` function that creates a new reservation and adds it to the given restaurant, as follows:

(a) validate the date and time parameters

(b) find the patron with the given id in the restaurant

(c) if an error is encountered in either step above, a detailed error message must be printed out, and the reservation cannot be created

(d) create and initialize a new reservation time, using the given parameters

(e) create and initialize a new reservation, using the new reservation time and the found patron

(f) add the new reservation to the restaurant

3.6.4. Implement the `void printResByPatron(RestaurantType *r, int id)` function that prints out the restaurant name and the details of every reservation made by the patron with the given id.

3.6.5. Implement the `void cleanupRestaurant(RestaurantType *r)` function that cleans up all the dynamically allocated memory for the given restaurant `r`. This includes the patron collection, and the reservation list.

3.7. **Implement the main control flow**

Implement the `main()` function as follows:

3.7.1. Declare a local `RestaurantType` structure variable to store all the restaurant data in the program. You may allocate the restaurant either statically or dynamically.

3.7.2. Initialize the local `RestaurantType` structure by calling an existing function.

3.7.3. Load the provided data into the local restaurant structure, by calling two functions given in the base code.

3.7.4. Repeatedly print out the main menu by calling the provided `printMenu()` function, and process each user selection as described below, until the user chooses to exit. Verify that the user enters a valid menu option. If they enter an invalid option, they must be prompted for a new selection.

3.7.5. The "print patrons" and "print reservations" functionality must both print out the restaurant name before calling an existing function.

3.7.6. The "print reservations by patron" functionality must prompt the user to enter a patron id before calling an existing function.

3.7.7. At the end of the program, the restaurant data must be cleaned up.

If any errors are encountered in the above, a detailed error message must be printed out to the end user. Existing functions must be reused everywhere possible.

3.8. **Document your program**

You must document your program, as we covered in the course material, section 1.2.4. Every function except `main()` must have a block of comments before the function to indicate: (1) the function purpose, (2) the correct role for **each** of its parameters (covered in section 1.2.3 of the course material), and (3) the possible return value(s) if applicable. The program as a whole is documented in a `README` file. **Do not** use inline comments as a substitute, as they cannot correctly document a procedural design.

3.9. **Package your files**

3.9.1. Your program must be separated into five (5) source files, as indicated in instruction 3.3.1.

3.9.2. You must provide a plain text `README` file that includes:

(a) a preamble (program author, purpose, list of source, header, and data files, if applicable)

(b) compilation and launching instructions, including any command line arguments

3.9.3. Use either the `tar` or the `zip` utility in Linux to package the files above into one `.tar` or `.zip` file.

3.9.4. Do not include any additional files or directories in your submission.

4. Constraints

Your design and implementation must comply with all the rules of correct software development and programming conventions that we have learned during the course lectures, including but not restricted to:

4.1. Your program must be correctly separated into modular, reusable functions, and it must adhere to the correct programming conventions for C programming in Linux.

4.2. The code must be written using the C11 standard that is supported by the course VM, and it must compile and execute in that environment. It must not require the installation of libraries or packages or any software not already provided in the course VM.

4.3. Your program must not use any library functions or techniques not covered in this course, unless otherwise permitted in the instructions.

4.4. Do not use any global variables.

4.5. If base code is provided, do not make any unauthorized changes to it.

4.6. Your program must reuse the functions that you implemented, as well as predefined constants, everywhere possible.

4.7. Your program must perform all basic error checking.

4.8. Do not use recursion where iteration is the better choice.

4.9. Compound data types must always be passed by reference, never by value.

4.10. Return values must be used only to indicate function status (success or failure). Except where otherwise permitted in the instructions, data must be returned using parameters, and never using the return value.

4.11. All dynamically allocated memory must be explicitly deallocated.

4.12. You may implement helper functions, but only if the design is consistent with the provided instructions. Design and implementation choices that undermine the learning outcomes will not earn any marks.

4.13. All functions must be documented, as described in the course material, section 1.2.

4.14. To earn full program execution marks, your code must compile and execute without warnings or errors or memory leaks, and it must be implemented in accordance with all instructions.

5. Submission Requirements

5.1. You will submit in *Brightspace*, before the due date and time, one `tar` or `zip` file that includes all your program files, as described in the **Package your files** instruction.

5.2. Late submissions will be subject to the late penalty described in the course outline. No exceptions will be made, including for technical and/or connectivity issues. Do not wait until the last day to submit.

5.3. Only files uploaded into *Brightspace* will be graded. Submissions that contain incorrect, corrupt, or missing files will be graded as is. Corrections to submissions will not be accepted after the due date and time, for any reason.

6. Grading Criteria

- 35 marks: code quality and design
- 35 marks: coding approach and implementation
- 25 marks: program execution
- 5 marks: program packaging