CST8234 – C Programming W18 (Assgn1)

Programming Exercise

The purpose of this assignment is to demonstrate that you can do

- Dynamic allocation of memory
- Singly-linked lists
- Sorting

Statement of Problem

Work with a partner to write a program that represents a Tindr-like app for rental properties.

I.e., you'll have a list of rental opportunities that you can dismiss (i.e., "swipe left)", or add to a list of properties that you'd like to further consider (i.e., "swipe right")

You will maintain **TWO singly-linked lists**. One is for unviewed properties (called "undecided"), and one for your rental properties that you'd like to consider further (called "favourites"). At any time, one of these lists will be the current list.

You will build an interface that will do the following (see "User Interface" section below, for more details)

- Let the user get a list of commands
- View the "undecided" list (possibly adding a new item to the "undecided" list)
- View "favourites" list
- Next (displays the next rental property in the current list, without making a decision)
- Discard (i.e., swipe left)
- Add to favourites (i.e., swipe right)
- Set the sorting mode, to one of four criteria
- Exit the program

Generating New Rental Properties

When the program starts, you will randomly create 6 random properties, and append them to the "undecided" list.

Every time the user enters the "View New Properties" command (i.e., ***UNDECIDED*** from the list of commands, below), there is a 50% chance (i.e., 1 in 2) that you will generate a new property, and append it to the "undecided" list. This simulates new listings that periodically come into being.

Random Generation of New Rental Properties

For each new property that you randomly generate you need to generate some random properties.

You will have a fixed list of 10 streets (you **are** allowed to define this as a statically-defined array!). Each street will have a **name** of your choosing (e.g., "Cat St.", "Tiger Boul.", "Ferret Ave."), and a randomly-generated **base distance** from campus, from 0.5km – 4km, in 100m increments (e.g., 100*random(5,40)). Clearly, you'll need an array of structs to hold all of your street information. Hint: you can use {} initialization to create your array, and then afterwards give each element a random distance.

Each rental property will be a struct. In this assignment, it will have members for the **street**, the **street number**, the **price per bedroom**, and the **number of bedrooms**.

For each new property, you will

- randomly select a street to use, and store a pointer to that street struct
- randomly generate a number between 1-200 for the street number
- randomly number of bedrooms, from 1-4
- randomly generate a rent per bedroom from $200-$600 in $50 increments (i.e., 50 * random(4,12))

The random values need to differ every time you execute the program.

Property Functions

In addition to the standard setters and getters you'll want for your structs, you'll need to define a function to calculate the distance from the College.

The distance of a property is defined by the base distance of the street, plus 20 m per street number. E.g., if "Fox St." is 600 m distant from the College, and the street address is 38, then the total distance of the rental property is 600+20*38 = 676 m.

Property Display Functions

The address of a property is listed as the street number concatenated with the street name. E.g., "38 Fox St."

Distances are stored in metres, but are display in km's, expressed as a float with **two decimals**. E.g., 676 m will be shown as "0.68 km".

You will need a function that will accept a pointer to a property structure, and print a nicely-formatted summary of a property (e.g., the formatted address, the rent/room, the total distance from campus, and the number of bedrooms).

You will also need a function that will accept a pointer to a property structure and print out the information in a form suitable for display in a table.

User Interface

The basic sequence is that the user types a brief command as listed below, and then they get some visual feedback (as described in the Details for each command).

Key	Command	Details
h	HELP	Print out a brief list all the available commands
q	QUIT	Exit the program
a	ALL	Sort the current list of rental properties ("undecided" or "favourites"), and then show them in a tabular form. After showing the user the entire list, ask what they think about the first rental property in the list. If there are no rental properties in the current list, display "There are no more rental properties".
u	UNDECIDED	Generate a new rental property (50% of the time) and add it to the "undecided" list. Set the current list to "undecided", and do an ***ALL***.
f	FAVS	Set the current list to "favourites" list, and do an ***ALL***.
s<n>	SORT	Organize the properties according to the desired sort method **<n>** **r** – Rent per Room (ascending) **d** – Distance (ascending) **n** – Number of rooms (descending) **a** – Address (ascending) And then do an ***ALL***.
n	NEXT	Either a) display the next rental property on the current list, or b) print "There are no more rental properties" if you were already at the last one on the list.
l	LEFT	Remove the just-viewed property from the current list, delete it, and either a) display the next rental property on the current list, or b) print "There are no more properties" if you were already at the last one on the list.
r	RIGHT	If you are currently viewing the favourites list, print out "This property is already on the favourites list". Otherwise, remove the just-viewed property from the undecided list, and append it to the favourites list, and either a) display the next rental property on the undecided list, or b) print "There are no more rental properties" if you were already at the last one on the list.
Anything else	UNSUPPORTED	Print out "That is not a supported command", and then do a ***HELP***.

Because this assignment is about use of linked lists, and not input validation, you may assume that any command that instructors enter will only be, at most, a few characters, and you are permitted to use a input buffer of 8 chars, without worrying that the instructors will type in long inputs and force buffer overruns.

Sorting

You will remember the current sorting preference (e.g., by price, by distance, by number of bedrooms), and use this mode.

When sorting by address, when you'll first sort by street name, and if the street names are equal you'll then use street number. I.e., you'll need to ensure that "22 Fox St." comes after "3 Fox St.". Hint: sort based on the struct members (street name and then street number), rather than just sorting the formatted address (e.g., "22 Fox St.").

Linked List Management

You must write functions that can do the following

- Count the number of items in a list.
  - E.g., "`int getCount( Node *pHead );`"
- Find the i-th element in a list.
  - E.g., "`Node *getNodeAtIndex( Node *pHead, int i );`"
- Append an item to the end of a list.
  - E.g., "`void appendNode( Node **ppHead, Node *pNewNode );`"
- Insert an item into a list, at position i.
  - E.g., "`void insertNode( Node **ppHead, Node *pNewNode, int i );`"
- Remove the i-th item from a list.
  - E.g., "`Node *removeNodeAtIndex( Node **ppHead, int i );`"

The functions above must be able to operate on **EITHER** your "undecided" list or "favourites" list. I.e., you may not have a two "count the number of items" function (one for "undecided" and one for "favourites").

You'll have to decide if you want a dedicated "Node" struct that has a pointer to a rental property struct, and a pointer to the next "Node", and whose only purpose is to create a singly-linked list, or whether you want to add the self-referential pointers to the rental property structure. I **strongly recommend** having a dedicated Node struct, because it neatly separates the content of the property structs from the way in which they are being organized (e.g., linked list vs arrays). This also simplifies sorting (see 'Hints' below).

Hint: getting your linked list management routines implemented can be done independently of most of the other work. E.g., it doesn't impact parsing input for menu items, or printing lists/properties (see suggestions re stub routines in the "Strategies" section of this document.)

Multiple Files

You must split your code between multiple source files. I strongly recommend that you write it this way from the beginning. I.e., think like you're using Java, and decide which functionality belongs in which class (i.e., file). Putting everything in main.c, and then splitting it up at the end "just to satisfy the instructor" is a really dumb way to approach your program's structure.

You will be marked on the sensible way you divvy up the functionality into multiple files. Conversely, you will lose a lot of marks if you just arbitrarily carve up your functions in ways that don't make sense.

You will need to define .h files for most source file that contains the 'extern' declarations to the functions that will need to be accessed from other files.

My implementation had the following files.

- main.c — initialization and input parsing
- rental.c — all functions related to properties, e.g. initialization, setters/getters, display functions
- sort.c — all functions relating to comparators, sorting and swapping
- node.c — all functions relating to linked lists, e.g., creating nodes, appending, removal, etc.

You're not obligated to use the same division of functionality into files, but you'll want to use something sensible, as opposed to randomly chucking a bunch of functions into different files and hoping it will satisfy the instructions (hint: it won't, and you'll lose marks).

Special Restrictions

You MUST work in pairs on this assignment. **Solo submissions will be docked 10% marks (e.g., mark of 73% will become 63%)**.

You must implement your lists of rental properties with linked lists**. Implementations that use arrays of rental property structs instead of linked lists WILL BE GIVEN A GRADE OF ZERO**.

In an effort to force you to pass information via parameters, rather than relying on global variables, **you are not permitted to define any global variables**.

Structs must be passed into functions (e.g., your comparison functions, or your swap routine) by ***reference*** (i.e., as pointers) rather than by ***value*** (i.e., as copies of the structs). This means that you'll end up using "->" notation, not the "." notation. Failure to use pointers will result in deducted marks. Yes, you will need to pass in double pointers to many of your linked-list manipulation routines.

Miscellaneous Hints

I used function pointers to simplify picking which sorting comparator to use (i.e., rather than having to use multiple if-then-else / switch statements all over the place). E.g., I defined a function pointer called "pComparator" as follows:

```
int (*pComparator)( RentalProperty *a, RentalProperty *b);
```

You may want to divide up the work by logically grouping. E.g., one person can be working on the linked lists manipulation, while the other is working on the user interface.

If you implement a distinct Node data structure that is separate from the Rental Property data structure (**strongly recommended**), then you can actually implement a swap algorithm by changing the "payloads" rather than having to re-organize the linked list (much, much simpler!)… i.e., I have a swap routine that takes a pointer to the two nodes that are being "swapped" and just exchanges their payloads.

```
typedef struct _Node {
    RentalProperty  *pRental;           // 'payload' of this list node
    struct _Node    *pNext;             // pointer to next node
} Node;

void swap( Node *pNode_A, Node *pNode_B )
{
    RentalProperty temp = pNode_A->pRental;
    pNode_A->pRental = pNode_B->pRental;
    pNode_B->pRental = temp;
}
```

Implementation Strategies

This is a big assignment. This is why you've been given extra time.

Strategy #1: Find a partner, immediately. Waiting a week to start looking means you'll only have a week to implement the project, and will not be able to complete the project on time. Do it now. Check the course Discord (https://discord.gg/uGTR5gG).

If you can't find a partner immediately, start working on the project immediately, by yourself! It is YOUR responsibility is to get the assignment implemented and submitted.

I will ignore all emails about "I can't find a partner."

Strategy #2: Print out a copy of these requirements, and with a highlighter, underline ANYTHING that looks like requirement (e.g., is the rent randomized in $50 increments?). In this way, when you THINK you are done, you can go back over the requirements and make sure that everything that was highlighted has been addressed.

Strategy #3: Agree on your data environment. Are you going to upload/download completed files via a shared drive (e.g., dropbox)? Are you going to exchange files via email (yuck!)? Are you going to set up a GIT repository on bitbucket.org? It's up to you, but you need to be able to both work from a set of files, and then figure out what needs to be updated as you both work on stuff independently.

Strategy #4: Agree on a schedule, with short interim deliverables. Just saying "I'll do x, y and z, and you'll do a, b and c, and we'll get together on Sunday night before it's due to integrate our stuff is a recipe for disaster. Plan on meeting 2-3 times per week.

Strategy #5: Always, always, ALWAYS have you code in a compilable and executable state. Never share broken code with your partner. Keep a back-up of your stable code, so that if you seriously break something the next day, you have a means of going back to a known state.

Strategy #6: Have a plan for, and use redundant off-device storage. Excuses like "My hard-drive crashed" or "The VMware image got corrupted" or "my laptop got stolen" is NOT acceptable. You HAVE to have a backup (google drive? Dropbox? Bitbucket?). Keeping your source code safe is trivial to do, which is why losing files is not considered an excuse.

Strategy #7: Eliminate compiler warnings immediately, before proceeding with further development.

Strategy #8: Spit-polish the code, AS YOU IMPLEMENT IT, to maximize your coding convention marks. Don't do a clean-up at the end, because you'll introduce bugs.

Strategy #9: Mutually decide on your data structs before starting implementation. Declare them in an appropriate header file (e.g., "typedefs.h" or "node.h" and "property.h") and share them with both partners.

Strategy #10: Figure out what are the most basic things that need to be implemented first, before moving on to the more interactive stuff.

E.g., I would start out with defining my Street struct, and create a list with non-random distances.

I'd then implement a RentalProperty *createProperty(Street *pStreet, int nStreetNumber, int nRooms, int nRent) that lets me malloc and initialize a property, e.g., "createProperty(STREETS[1], 22, 3, 600);" which will be important for building/testing my linked list management.

I'd implement a Node *createNode(RentalProperty *pProperty) that mallocs and initializes a node.

This then allows me to hard-code a bunch of nodes that I can MANUALLY link together (because I haven't yet written my link list manipulation functions).

It's important to be able to use hard-coded data, because it lets you test for correct implementation (e.g., is your sort algorithm handling properties on the same street correctly? Did getNodeAtIndex() return the correct value?). Don't be too keen to immediately start using random data, as you may not be able to reproduce errors.

I'd also implement printList(RentalProperty *pHead) at this point so that I've got a way to see the contents of my list(s) and make sure they are correct. Make sure that your printList handles empty lists.

Strategy #11: Use stub functions so that you can work independently on your code. The issue is you're not sure at the outset who is going to implement which chunks of functionality, and one of you may discover that their section is harder than expected and so the other person has to jump in and pick up some of the tasks. So you want to have code that at any time will run and allow your partner to continue the implementation.

E.g., when implementing the user interface, create functions for each of the methods that initially just have a printf as a body. E.g., some of your functions might be

```
void setCurrentList() {
   printf( "set current list\n" );
}
void swipeLeft() {
   printf( "swipe left\n" );
}
void swipeLeft() {
   printf( "swipe left\n" );
}
void setSort() {
   printf( "set sort algorithm\n" );
}
```

This means that you can build and test your input validating and processing (by jumping to each of the stub functions), even if the functions don't do anything meaningful yet. I.e., you can verify that when you enter 'r' you see the message "swipe right". You can then successively implement each stub.

Once you start refining your implementation, you'll realize you may need to parse more carefully and pass in other parameters. E.g., you may start parsing the inputs "sd" vs "sn" and change your stub to be the following.

```
void setSort( RentalProperty *p, SortMethod eSortMethod ) {
   printf( "set sort algorithm to %d\n", eSortMethod );
}
```

I.e., it's still a stub, but now it's a more accurate stub!

In a similar manner, if your list management functions start out as stubs, the UI component can start calling them, even if they do nothing other than say "delete node 5 from current list".

You'll then be able to replace your stub code (e.g., a printf) by the actually implementation, one-by-one. And both of you will have an idea of how much of the program is still undone and adjust your time management to suit.

Strategy #12: Replace your hard-coded values with randomly-generated ones in the FINAL stages of implementation… after you've tested everything with known quantities. E.g., give the streets random distances, replace the hard-coded initial list with randomly generated streets, numbers, rooms and rent.

Strategy #13: Spit-polish the output. Look for typos, spacing and alignment errors. Check the sample below to see what we expect it to look like. Part of the implementation marks are the demonstration that you can use many of printf's formatting modifiers to get professional-looking output (albeit, for a command line application!)

Strategy #14: Always test your code to see if you can BREAK it, not just see if it works under perfect conditions.

Strategy #15: Have a friend test your code, and offer to test theirs. Try to break it, and give feedback. Thank your friends for any feedback they offer.

Strategy #16: Once you've packaged your submission, unpack it and test it ONE LAST TIME before actually submitting it, making sure that you've stripped out unnecessary files and not omitted anything important (like the makefile) and that your zip contains a correctly-named folder. Double check with your partner that everything looks good before actually clicking the "Submit" button.

Submission

When you are complete submit your programs to Brightspace.

But… before you do, please check that you've satisfied the following submission requirements

1. Did you confirm that your zipped submission is a ".zip" file (not a '.rar' or '.tar.gz' or '.7z' file)?
2. Did you zip up a folder that includes both partner's *user names*, and the lab/assignment indicator? I.e., if you open up your own zipped submission, you should see a *folder* called (for example) "smit9112_sing0001". If you just see *file(s)*, you've done it wrong, and you'll need to go to the *parent* folder, and try zipping your lab folder.
3. Did you remove all the unnecessary files from the folder contained in your zipped submission? I.e., you open up your own zipped submission, and click on the folder called (for example) "smit9112_sing0001", you should see *just your makefile and your source files, and include files.*
4. Is your makefile actually called 'makefile' or 'Makefile' (without any filename extensions?)

If you don't satisfy submission requirements #1 or #2 you will lose 10% on this assignment. If you realize afterwards that you've made a mistake, don't panic! you are allowed to correct your mistake and re-submit.

Sample Session

```
$ ./ass1.exe

Undecided Rental Properties
Address                          # Rooms  Rent/Room   Distance
-------------------------        -------  ---------  ---------
102 Mouse Ave.                         4        350    4.94 km
 22 Fox St.                            2        400    3.04 km
 57 Coyote Crt.                        4        400    4.54 km
  5 Cat St.                            4        400    3.00 km
 91 Cat St.                            1        500    4.72 km
168 Squirrel Cres.                     2        600    4.56 km

What do you think about this rental property?
        Addr: 102 Mouse Ave., # Rooms: 4, Rent/Room: $350, Distance: 4.94 km

Command ('h' for help): h

Valid commands are:
        h  - display this help
        a  - display all the rental properties on the current list
        f  - switch to the favourites list
        u  - switch to the undecided list
        l  - 'swipe left' on the current rental property
        r  - 'swipe right' on the current rental property
        n  - skip to the next rental property
        sa - set the sorting to 'by address'
        sn - set the sorting to 'by number of rooms'
        sr - set the sorting to 'by rent'
        sd - set the sorting to 'by distance'
        q  - quit the program

Command ('h' for help): sd

Undecided Rental Properties
Address                          # Rooms  Rent/Room   Distance
-------------------------        -------  ---------  ---------
  5 Cat St.                            4        400    3.00 km
 22 Fox St.                            2        400    3.04 km
 57 Coyote Crt.                        4        400    4.54 km
168 Squirrel Cres.                     2        600    4.56 km
 91 Cat St.                            1        500    4.72 km
102 Mouse Ave.                         4        350    4.94 km

What do you think about this rental property?
        Addr: 5 Cat St., # Rooms: 4, Rent/Room: $400, Distance: 3.00 km

Command ('h' for help): r
```

```
Rental property moved to your favourites list

What do you think about this rental property?
        Addr: 22 Fox St., # Rooms: 2, Rent/Room: $400, Distance: 3.04 km

Command ('h' for help): l

Rental property deleted

What do you think about this rental property?
        Addr: 57 Coyote Crt., # Rooms: 4, Rent/Room: $400, Distance: 4.54 km

Command ('h' for help): l

Rental property deleted

What do you think about this rental property?
        Addr: 168 Squirrel Cres., # Rooms: 2, Rent/Room: $600, Distance: 4.56 km

Command ('h' for help): r

Rental property moved to your favourites list

What do you think about this rental property?
        Addr: 91 Cat St., # Rooms: 1, Rent/Room: $500, Distance: 4.72 km

Command ('h' for help): r

Rental property moved to your favourites list

What do you think about this rental property?
        Addr: 102 Mouse Ave., # Rooms: 4, Rent/Room: $350, Distance: 4.94 km

Command ('h' for help): l

Rental property deleted

No more rental properties

Command ('h' for help): f

Favourite Rental Properties
Address                          # Rooms  Rent/Room   Distance
-------------------------        -------  ---------  ---------
  5 Cat St.                            4        400    3.00 km
168 Squirrel Cres.                     2        600    4.56 km
 91 Cat St.                            1        500    4.72 km
```

```
What do you think about this rental property?
        Addr: 5 Cat St., # Rooms: 4, Rent/Room: $400, Distance: 3.00 km

Command ('h' for help): sr

Favourite Rental Properties
Address                           # Rooms  Rent/Room   Distance
--------------------------        -------  ---------  ---------
  5 Cat St.                             4        400    3.00 km
 91 Cat St.                             1        500    4.72 km
168 Squirrel Cres.                      2        600    4.56 km

What do you think about this rental property?
        Addr: 5 Cat St., # Rooms: 4, Rent/Room: $400, Distance: 3.00 km

Command ('h' for help): r

This rental property is already on your favourites list

What do you think about this rental property?
        Addr: 5 Cat St., # Rooms: 4, Rent/Room: $400, Distance: 3.00 km

Command ('h' for help): n

What do you think about this rental property?
        Addr: 91 Cat St., # Rooms: 1, Rent/Room: $500, Distance: 4.72 km

Command ('h' for help): l

Rental property deleted

What do you think about this rental property?
        Addr: 168 Squirrel Cres., # Rooms: 2, Rent/Room: $600, Distance: 4.56 km

Command ('h' for help): a

Favourite Rental Properties
Address                           # Rooms  Rent/Room   Distance
--------------------------        -------  ---------  ---------
  5 Cat St.                             4        400    3.00 km
168 Squirrel Cres.                      2        600    4.56 km

What do you think about this rental property?
        Addr: 5 Cat St., # Rooms: 4, Rent/Room: $400, Distance: 3.00 km

Command ('h' for help): q
```