ESC190H1S Winter 2022: Assignment 1

Due March 10, 2022 at 23:59 on Gradescope; up to 1 hour grace period where no late penalty will be incurred. No late submissions accepted after March 11, 2022 at 00:59.

Overview

With the upsettingly high costs of dining out, Sandy and Cima decide to take matters into their own hands and open their own restaurant. Unfortunately, due to budget constraints, they have to develop their order management system on an embedded system using a PIC microcontroller. Thankfully, their PIC microcontroller has a C compiler available and they narrowly escape having to use Assembly.

Cima notes that they have to be careful with memory management, because it would be disastrous to the restaurant's operations if the ordering system crashed due to memory leaks or segmentation faults. Sandy heartily agrees; they both decide to use Valgrind to make sure there is no memory malpractice afoot and get started on development. Seeing the amount of work cut out for Sandy and Cima, you offer your help (you wonder if you might regret this later...). Excitedly, they tell you they have function-level breakdowns of the requirements for the program, which they hand to you in the **Tasks** section of this document.

Constraints

Violation of any of these constraints will result in a grade of zero due to the nature of our autotesters.

- You **MUST NOT** modify `a1.h`
- Your code **MUST** work when the defined `#define` values in `a1.h` are changed
- Your code **MUST** compile and work with the given compiler commands in this document and provided autotesters to receive any credit.
- You **MUST NOT** modify the function signatures.
- You **MUST NOT** add any additional `include` statements.
- `a1.c` **MUST NOT** include a `main(...)` function.
- Your submitted `a1.c` **MUST NOT** include any `printf(...)`, `fprintf(...)`, `scanf(...)`, or `fscanf(...)` statements, other than the ones specified in this document, or those found in functions that are already provided in the starter code.

Starter Code

Carefully review the provided `a1.h` for existing type definitions and function prototypes. Some starter functions to print out the `structs` once they are filled are provided for you in `a1.c`; you may modify these function implementations if you wish. You may add additional functions (and their prototypes) to `a1.c`. An interactive program, `restaurant.c` is provided; you may modify this however you like.

`/tests/` contains a preliminary set of auto-tests for your code. You should review these tests carefully as they demonstrate how we will call and test your code. To compile and run the tests...

```
> gcc -g -o <target> <tester .c file> a1.c
> ./<target>
> valgrind --leak-check=full
  --show-leak-kinds=all
  --track-origins=yes
  ./<target>
```

For example, to run tests on my `Order` interfaces...

```
> gcc -g -o order_tests ./tests/order_tests.c ./a1.c
> ./order_tests
> valgrind --leak-check=full
  --show-leak-kinds=all
  --track-origins=yes
  ./order_tests
```

To compile with `restaurant.c`...

```
> gcc -g -o a1 ./restaurant.c ./a1.c
> ./a1
> valgrind --leak-check=full
  --show-leak-kinds=all
  --track-origins=yes
  ./a1
```

Additional Remarks

This document is *not* written in the suggested completion order. Rather, it is written in an order that tries to facilitate understanding of the problem. You should use your judgment when dividing up work with your partner (if you choose to work with one) and deciding what order to complete the functions in.

We highly recommend that you read through the entire document and starter code before starting to write your own code, and potentially drawing pictures of what the final result(s) should look like.

You are permitted (but not required) to work in partners on this assignment. Declare your partners on Gradescope (you do not need to work with your lab partner for the assignments).

Input File Structure

In this assignment, you will need to read from a file to populate the restaurant's menu. You will do this in the `load_menu(...)` function. The constraints on the file structure provided here are meant to *simplify* the file reading task so you do not have to consider extreme cases. Consider the provided example menu file, `menu.txt`. Any input files we test your code on will be guaranteed to:

- have one or more lines of data
- have no empty lines until after the last line of data (empty lines are considered to be whitespace-only, where whitespace characters include spaces, tabs, newlines, and carriage returns).
- have each line represent one menu item
- consist of `MENU_DELIM` separated-fields on each line with no spaces around `MENU_DELIM`
- consist of exactly 3 fields in the following order: item code, item name, item cost per unit
  - the item code is guaranteed to be exactly `(ITEM_CODE_LENGTH - 1)` characters in length, in valid alphanumeric characters
  - the item name is guaranteed to be no more than `(MAX_ITEM_NAME_LENGTH - 1)` characters in length, in valid alphabetic characters; the item name may include spaces but will never include tabs, newlines, or carriage returns
  - the item cost per unit is guaranteed to begin with a `$` character, and be a valid floating point value (note the precision is not guaranteed)

Lines may have excess leading or trailing whitespace, but will not have excess whitespace separating entries (i.e. you do not have to trim whitespace, e.g., in the middle of the item name). `MENU_DELIM` may be the tab character, but will never be the newline, carriage return, or space character.

Tasks

Complete these functions in `a1.c`. The function prototypes are given in `a1.h`.

```
Restaurant* initialize_restaurant(char* name);
```

Inputs `<name>` is a string literal.
Output Return a pointer to a `Restaurant` with:

- `name` field set equal to the input parameter `<name>`
- `menu` initialized to the contents of `MENU_FNAME` (see `load_menu(...)`)
- `num_completed_orders` set to `0`
- `num_pending_orders` set to `0`
- `pending_orders` intialized to an empty `Queue`. An empty `Queue` is defined as a `Queue` with its `head` and `tail` set to `NULL`

```
Menu* load_menu(char* fname);
```

Example usage (See Appendix A for a memory diagram of the resulting menu)

```
Menu* menu = load_menu("menu.txt");
```

Inputs `<fname>` is a string literal, which is the file name of a plaintext file that is guaranteed to have the format specifications provided under **Input File Structure**.
Output Return a pointer to a `Menu` initialized with the contents of the file `<fname>` with:

- `num_items` set to the number of items available on the menu.
- `item_codes`, a pointer to array of `char *`. Each item in the array is a pointer to a null-terminated character array of size `sizeof(char) * ITEM_CODE_LENGTH`. The sequence of this array must match the sequence that items appear in the file `<fname>`.[a]
- `item_names`, a pointer to array of `char *`. Each item in the array is itself a pointer to a null-terminated character array of size `sizeof(char) * MAX_ITEM_NAME_LENGTH`. The sequence of items in this array must match that of the file `<fname>`.[b]
- `item_cost_per_unit`, an array of `double`, representing an array of item costs per unit. The sequence of items in this array must match that of the file `<fname>`.

[a] `ITEM_CODE_LENGTH` includes the null terminating character
[b] `MAX_ITEM_NAME_LENGTH` includes the null terminating character

```
Order* build_order(char* items, char* quantities);
```

Example call for an order of 20 units of item A1, 11 units of B1 and 17 units of C1:

```
Order* order = build_order("A1B1C1", "20,11,17");
order->num_items; // should be 3
```

Inputs `<items>` and `<quantities>` are read in parallel. You are guaranteed that the input lengths are consistent (i.e, 3 items will have 3 respective quantities) as described below.

- `<items>` is a null-terminated string literal consisting of one or more order codes, with no separation (and no repetition of order codes). There are no whitespace characters in `items`. Each item code is exactly `(ITEM_CODE_LENGTH - 1)` characters.
- `<quantities>` is a null-terminated string literal consisting of `MENU_DELIM` delimited entries. There are no excess whitespace characters in the string (`MENU_DELIM` may be, e.g., the tab character). All characters other than `MENU_DELIM` are valid numeric characters.

Output Return a pointer to `Order` with:

- `num_items` set to the number of unique item codes in the order.
- `item_codes`, a pointer to array of `char *`, set to the item codes in the order. The sequence of this array must match the sequence that items appear in the input `<items>`.
- `item_quantities`, an array of `int`, set to the quantity of each item code in the order. The sequence of this array must match the sequence that items appear in the input `<quantities>`.

```
void enqueue_order(Order* order, Restaurant* restaurant);
```

Example usage:

```
Restaurant* restaurant = initialize_restaurant("McBonalbs");

Order* order_1 = build_order("A1B1", "12,13");
Order* order_2 = build_order("A1B1C1", "12,10,9");

enqueue_order(order_1, restaurant);
enqueue_order(order_2, restaurant);

Order* dq_order_1 = dequeue_order(restaurant);
Order* dq_order_2 = dequeue_order(restaurant);
```

All fields for `order_1` and `dq_order_1` should have identical values (not necessarily identical memory addresses, but they may be). The same holds for `order_2` and `dq_order_2`.

Inputs

- `<order>` is the pointer to the `Order` to enqueue.
- `<restaurant>` is the pointer to the `Restaurant` that `<order>` should be enqueued to.

Output No return value.

- Modify the `Queue` referenced by `<restaurant>`'s `pending_orders` field, respecting FIFO order, to enqueue `<order>` as the cargo of a `QueueNode`.
- Update the `<restaurant>`'s `num_pending_orders` field.

```
Order* dequeue_order(Restaurant* restaurant);
```

See `enqueue_order(...)` for example usage.

Inputs `<restaurant>` is a pointer to the `Restaurant` to dequeue from. You may assume the `Restaurant`'s `pending_orders` filed refers to a `Queue` that is **NOT** empty, i.e. there is at least one `QueueNode` available to dequeue.

Output

- Return a pointer to `Order`, which is the `Order` found as the cargo of the `QueueNode` deuqueued from the `Queue` referred to by `<restaurant>`'s `pending_orders` field. This dequeue operation must maintain FIFO order of the `Queue`. Make sure that if you empty the `Queue`, the `head` and `tail` are set to `NULL`.
- Update the `<restaurant>`'s `num_pending_orders` field.
- Update the `<restaurant>`'s `num_completed_orders` field.

```
double get_item_cost(char* item_code, Menu* menu);
```

Example usage:

```
Menu* menu = load_menu("menu.txt");
get_item_cost("L1", menu); // should return 5.99
```

Inputs

- `<item_code>` is a null-terminated string literal of length `(ITEM_CODE_LENGTH - 1)`. You may assume the item code is present in the given `<menu>`'s `item_codes`.
- `<menu>` is a `Menu` loaded in through a call to `load_menu(...)`.

Output Return the floating point cost of the item `<item_code>`, as given by `<menu>`.

```
double get_order_subtotal(Order* order, Menu* menu);
```

Example usage:

```
Menu* m = load_menu("menu.txt");
Order* o = build_order("L2B2", "2,1");
get_order_subtotal(Order* o, Menu* m); // should return 12.5
```

Inputs

- `<order>` is a pointer to an `Order` created through a call to `build_order(...)`. You may assume that all items in the `<order>` are present in the given `<menu>`.
- `<menu>` is a `Menu` loaded in through a call to `load_menu(...)`.

Output Return the floating point cost of all the items in the order `<order>` with their respective quantities, with item costs given by `<menu>`.

```
double get_order_total(Order* order, Menu* menu);
```

Example usage:

```
Menu* m = load_menu("menu.txt");
Order* o = build_order("L2B2", "2,1");
get_order_total(Order* o, Menu* m); // should return 14.125
```

Inputs

- `<order>` is a pointer to an `Order` created through a call to `build_order(...)`. You may assume that all items in the `<order>` are present in the given `<menu>`.
- `<menu>` is a `Menu` loaded in through a call to `load_menu(...)`.

Output Return the floating point cost of all the items in the order `<order>` with their respective quantities, with item costs given by `<menu>`, after applying the sales tax of `TAX_RATE` to the order subtotal. Note that `TAX_RATE` is given as a percentage out of 100. Do **NOT** round the result.

```
int get_num_completed_orders(Restaurant* restaurant);
int get_num_pending_orders(Restaurant* restaurant);
```

Example usage:

```
Restaurant* restaurant = initialize_restaurant("McBonalbs");

Order* order_1 = build_order("A1B1", "12,13");
Order* order_2 = build_order("A1B1C1", "12,10,9");

get_num_completed_orders(restaurant); // should be 0
get_num_pending_orders(restaurant); // should be 0

enqueue_order(order_1, restaurant);

get_num_completed_orders(restaurant); // should be 0
get_num_pending_orders(restaurant); // should be 1

enqueue_order(order_2, restaurant);

get_num_completed_orders(restaurant); // should be 0
get_num_pending_orders(restaurant); // should be 2

Order* dq_order_1 = dequeue_order(restaurant);
get_num_completed_orders(restaurant); // should be 1
get_num_pending_orders(restaurant); // should be 1

Order* dq_order_2 = dequeue_order(restaurant);
get_num_completed_orders(restaurant); // should be 2
get_num_pending_orders(restaurant); // should be 0
```

Inputs `<restaurant>` is the pointer to the `Restaurant` that you should return the number of completed/pending orders for.

Output Return the number of completed/pending orders indicated by the `completed_orders` / `pending_orders` field of the given `<restaurant>`, for `get_completed_orders` / `get_pending_orders`, respectively.

```
void clear_order(Order** order);
```

Example usage:

```
Order* o = build_order("L2B2", "2,1");
clear_order(&o);
```

Inputs

- `<order>` is a pointer to pointer to an `Order` created through a call to `build_order(...)`.

Output No return value. Clear all the memory associated with `<*order>`. You should call `free(*order)` and set the value of `<*order>` to `NULL` here.

```
void clear_menu(Menu** menu);
```

Example usage:

```
Menu* m = load_menu("menu.txt");
clear_menu(&m);
```

Inputs

- `<menu>` is a pointer to pointer to a `Menu` created through a call to `load_menu(...)`.

Output No return value. Clear all the memory associated with `<*menu>`. You should call `free(*menu)` and set the value of `<*menu>` to `NULL` here.

```
void close_restaurant(Restaurant** restaurant);
```

Example usage:

```
Restaurant* restaurant = initialize_restaurant("McBonalbs");
close_restaurant(&restaurant);
```

Inputs

- `<restaurant>` is a pointer to pointer to a `Restaurant` created through a call to `intialize_restaurant(...)`.

Output No return value. Clear all the memory associated with `<*restaurant>`, including all `QueueNodes` and `Orders`. You should call `free(*restaurant)` and set the value of `<*restaurant>` to `NULL` here.

Grading Scheme

The starter tests provided on Gradescope are worth 5 out of the allocated 15% for this assignment. Overall, your grade for this assignment will be broken down as follows:

- 70% of your grade is allocated to functionality; i.e., producing the correct result when run.
- 30% of your grade is allocated to your code running without memory leaks, segmentation faults, invalid reads, and any another memory malpractices through *valid usage* of your code. Valid usage is demonstrated in the test files and `a1.c`

Make sure you thoroughly test your code, as there are (many) hidden test cases we will use to evaluate your work.

Submission Checklist

Submit the *only* the following file(s) on Gradescope; make sure the cAsInG, spelling, and whitespace (or lack thereof) in the filename(s) is identical to those listed below:

- `a1.c`

Appendix A

Memory model of the returned pointer to `menu` struct for the function call `load_menu("menu.txt")`.

&num_items	6

item_codes	L	1	\0
	L	2	\0
	L	3	\0
	D	1	\0
	B	1	\0
	B	2	\0

item_cost_per_unit	5.99
	4.50
	8.50
	12.99
	2.99
	3.50

item_names	C	h	i	c	k	e	n		···	\0
	E	g	g		S	a	l	a	···	\0
	F	r	e	s	h		G	r	···	\0
	S	t	e	a	k		a	n	···	\0
	C	o	f	f	e	e	\0	\0	···	\0
	H	o	t		C	h	o	c	···	\0