CS 240: Programming in C
Fall 2020
Homework 9

Prof. Turkstra

Due November 4, 2020 9:00 PM

1 Goals

The purpose of this assignment is to reinforce the ideas of doubly-linked lists as well as introduce you to pointers to pointers. We will be adapting the previous homework to use pointers to pointers. We will also add some additional functionality.

2 The Big Idea

As in Homework 8, in this assignment, you will be using a database that holds events of your calendar. This database contains the event title, start time, end time, and description of each event.

The structures follow:

```
typedef struct time_struct {
  int year;
  int month;
  int day;
  int hour;
  int minute;
  int id;
} time_struct_t;

typedef struct event_info {
  char *title;
  char *description;
  time_struct_t start_time;
  time_struct_t end_time;
} event_info_t;

typedef struct event {
  struct event_info *info;
  struct event *prev_event;
  struct event *next_event;
} event_t;
```

3 The Assignment

You are to write a series of functions that extract event information from a calendar using structures. This calendar contains title, description, start time, and end time of each event. In the structure of `event_info_t`, the two fields of `start_time` and `end_time` represent a booking on the half open interval `[start_time, end_time)`.

In this homework, you are going to write functions to:

- Create a new single event;
- Delete a single, separated event;
- Insert the new event into an internal calendar, a doubly-linked list, (in order)
- Delete an entire calendar (doubly-linked list);
- Find an event within the calendar of events;
- Remove (isolate) an event from the calendar of events;
- Get a list of finished events before a particular time (as a new doubly-linked list);
- Delete an entire list of finished events — just the list, not its contents.

Examine hw9.h carefully. It contains declarations for each type you will use. It also contains prototypes for the functions that you will write.

It is worth noting that each node of the list only has a pointer to the event data structure. This permits multiple lists to point to (and access) the same data without duplicating it in memory. It also allows us to access the data without always first going through a doubly-linked list node.

For the purposes of this assignment, all doubly-linked lists `event_t`s will be NULL-terminated. I.e., the tail element of the list will have a NULL `next_event`, and the head of the list will have a NULL `prev_event`

3.1 Functions You Will Write

You will write the following functions:

`void create_event(char *, char *, time_struct_t, time_struct_t, event_t **);`

~~This function should allocate and populate a new `event_t` node. Remember to set both the `next_event` and `prev_event` to NULL.~~

~~The parameters are ordered as follows: event title, event description, event starting time, and event ending time.~~

Do not forget to allocate the necessary memory for the internal pointers!

The function should modify the pointer pointed to by the last argument to point to the newly allocated `event_t`.

~~You should assert that the time fields are valid (minute should range from 0 to 59, hour from 0 to 23, day from 1 to 31, month from 1 to 12, and year should be larger than, say, 0). You should assert that both string arguments are not NULL. You can assume that the ending time is always larger than the starting time.~~

You should assert that the final argument is not NULL and that what it points to *is* NULL.

~~The ID field for this assignment is a little open-ended. We recommend that you use some form of calculation involving the actual time such that the ID is a single number that uniquely identifies the time. This allows you to compare a single field instead of five when determining the insertion order. The constants `MONTHS_IN_YEAR`, `DAYS_IN_MONTH`, `HOURS_IN_DAY` and `MINUTES_IN_HOUR` are provided in an effort to help you.~~

~~The ID field will not be examined by any portion of the test module and is entirely optional.~~

`void delete_event(event_t **);`

This function should deallocate a single doubly-linked list node and all of its associated data.

You should ~~assert that the argument is non-NULL and that what it points to is not NULL.~~ You should also assert that the node is isolated (i.e., not part of a list). In addition, you should assert that the node's info is non-NULL.

On completion, set the pointer pointed to by the first argument to NULL.

`int add_event(event_t **, event_t *);`

This function should insert the singleton `event_t` (second argument) into the calendar of events pointed to by the first argument.

Please note: if the calendar exists, the first argument is not guaranteed to point to the head!

Insert the event such that the calendar continues to be sorted by the event starting time.

Please note: A new event can be added if adding the event will not cause a double booking. A double booking happens when two events have some non-empty intersection (ie., there is some time that is common to both events.) If adding the new event causes double booking, the function should return `DOUBLE_BOOKING` error. Otherwise, this function should return the event "placement" number — the number of events traversed from the head of the calendar to the first matching event. The head event is considered position `#1`.

The function should set the pointer pointed to by the first argument to point to the head, if it does not already.

~~Assert that both arguments are not NULL. You should also assert that the first argument to point to a pointer that is not NULL.~~

It is recommended that you draw a number of pictures so that you can visualize how nodes will be inserted at various locations within the doubly-linked list. Be sure to handle all corner cases.

You may wish to again create your own `prepend()` and `append()` functions.

`int delete_calendar(event_t **);`

This function deallocates an entire calendar. The argument points somewhere inside of the calendar (not necessarily the head).

This function should deallocate all internal data for each event as well as every events in the calendar.

On completion, the pointer pointed to by the first argument should be set to NULL. The function should return the number of events deallocated.

We encourage you to traverse the calendar and use `delete_event()`.

Assert that the argument is non-NULL. It is not an error if the pointer it points to is NULL.

`int find_event(event_t **, char *, time_struct_t);`

This function should search through the calendar pointed to by the first argument (again, not necessarily the head) and identify the first event of the specified title (second argument) and starting time (third argument).

If an event is found, this function should return the event "placement" number – the number of events traversed from the head of the calendar to the first matching event. The head event is considered position #1.

Otherwise, the function should return `EVENT_NOT_FOUND`.

Assert that the first and second arguments are non-NULL. You should also assert that the time fields are valid. In addition, you need to assert that the first argument to point to a pointer that is not null

`void remove_event(event_t *);`

This function removes the event pointed to by the first argument from the doubly-linked list in which it currently resides. Do not `free()` the event. Only remove it.

If the argument is NULL, do nothing.

Be sure to set the removed event's `next_event` and `prev_event` to NULL.

`int get_finished_calendar_list(event_t *, time_struct_t, event_t **);`

This function should search through the provided calendar (pointed to by the first argument — not necessarily the head) for all events finished before a specified time (second argument).

This function should build a new list of `event_ts` *without duplicating the actual data payload of each* `event_t`. Put anther way, the `info` pointer for each `event_t` should point to already allocated data. The value at the address pointed to by the third argument should be set to the head of this list.

The function returns the number of events found. If no event is found, the function should return 0 and set the pointer pointed to by the third argument to NULL.

~~Assert that the first and third arguments are non-NULL. You should also assert that the time fields are valid. In addition, assert that the pointer pointed to by the third argument is NULL.~~

`int delete_list(event_t **);`

This function performs the opposite of `get_finished_calendar_list()`. Since `get_finished_calendar_list()` generates a doubly-linked list that contains payloads that point to data in the calendar, `delete_list()` should delete and deallocate the entire list of `event_ts` *without* deallocating the actual payloads.

This function differs from `delete_calendar()` in that the latter deallocates the nodes AND associated payloads while `delete_list()` only deallocates the node.

This function should return the number of events deallocated as well as set the pointer pointed to by the argument to NULL.

Assert that the argument is not NULL and what it points to is not NULL.

3.2 Input Files

We will take a break from input files on this homework.

3.3 Header Files

We provide a header file, hw9.h, for you. It contains prototypes for each of the functions that you will write as well as #definitions for the constants. You should not alter this file. We will replace it with the original when grading.

3.4 Error Codes

The following error codes should be used by your functions:

- `EVENT_NOT_FOUND` : the given event does not exist in the calendar (doubly-linked list).
- `DOUBLE_BOOKING` : the adding event causes double booking.

These Standard Rules Apply

- You may add any `#include`s you need to the top of your hw9.c file;
- You may not create any global variables other than those that are provided for you. Creation of additional global variables will impact your style grade;
- Do not look at anyone elses source code. Do not work with any other students.

Submission

To submit your program for grading, type:

```
$ make submit
```

In your hw9 directory. You can do this as often as you wish. We encourage you to submit your code as often as possible. Only your final submission will be graded.

4 Grading

The operation of your functions will be graded out of 100 points. The point breakdown will be determined by the test program.

The test program will be run many times when grading. It is your job to do the same. The lowest score will be your final grade.

This homework will also have a style grade based on 20 points, with 1 point deducted for each code standard violation found.

Your code must compile successfully using `-Wall -Werror -std=c99` **to receive any credit. Code that does not compile will be assigned an automatic score of 0.**

If your program crashes (e.g., segmentation fault) at any point during the testing process, your score will be an automatic 0.