CMSC 216 Project #4 Fall 2021

Calendar **Due date posted on Piazza**

1 Overview

In this project you use dynamic memory allocation techniques to implement a calendar application. You also write your own tests and a Makefile.

The objectives are to practice dynamic memory allocation, function pointers, linked lists, and developing tests.

IMPORTANT: Implement this project individually. Do not work with others.

2 Grading Criteria

Your project grade will be determined as follows:

Results of public tests	20 pts
Results of release tests	32 pts
Results of secret tests	25 pts
Student tests	10 pts
Code Style	8 pts
Makefile	5 pts

IMPORTANT: Your makefile must use the standard flags to compile your code. If you don't, you will lose most of the points of this project.

3 Academic integrity statement

Please **carefully read** the academic honesty section of the course syllabus. We take academic integrity matters seriously. Please do not post assignment solutions online (e.g., Chegg, github) where others can see your work. This can lead to you being reported to the Office of Student Conduct.

4 Project Files

This project description is in the 216public project_descriptions directory. Copy the folder project4 from the 216public projects directory to your 216 directory. The files in the distribution, other than `.submit`, are:

1. `calendar.h`: Defines a Calendar struct, symbolic constants, and prototypes for the functions you must implement. **Do not modify this file.**
2. `event.h`: Defines an Event struct. **Do not modify this file.**
3. .h and .c files associated with the memory checker tool.
4. `Makefile`: Define your project's Makefile here. Don't change the name; it must start with capital M.
5. public01.c, public02.c, public03.c, public04.c, public05.c, public06.c: The public tests for this project.
6. student_tests.c: Write your student tests in this file.

5 Specifications

5.1 Makefile

Your `Makefile` should build all the programs **using separate compilation** (i.e., source files are turned into object files, which are then linked together in a separate step). It must also **avoid unnecessary compilation** (hint: use the touch command to check your makefile).

It must have at least the following targets:

1. `all`: build all executables

2. `public01`, `public02`, `public03`, `public04`, `public05`, `public06`: each builds a public test
3. `student_tests`: build the executable of student tests
4. `student_tests.o`:
5. `calendar.o`:
6. `my_memory_checker_216.o`:
7. `clean`: delete all object files and executables

You can add additional targets. Indeed, you will have to in order to achieve separate compilation.

IMPORTANT: Build all object files using the following gcc options:

-ansi -Wall -g -O0 -Wwrite-strings -Wshadow -pedantic-errors -fstack-protector-all -Wextra

IMPORTANT: Use only explicit rules in your Makefile. Do not use implicit rules. The Makefile examples in the (week06) lecture slides Make.pdf rely on explicit rules.

Note: You should define your Makefile from the start. If you decide to write your Makefile at the end, make sure you back up your code before you write the Makefile. Every semester there are students who mistakenly delete their implementation due to a Makefile error.

5.2 Calendar Overview

The calendar application allows events to be scheduled on specific days. A calendar is defined by the following structs:

```
typedef struct event {
   char *name;
   int start_time, duration_minutes;
   void *info;
   struct event *next;
} Event;

typedef struct calendar {
   char *name;
   Event **events;
   int days, total_events;
   int (*comp_func) (const void *ptr1, const void *ptr2);
   void (*free_info_func) (void *ptr);
} Calendar;
```

An Event struct represents an event. The fields of the Event struct maintain a pointer to the event's name, the start time (military time), the duration in minutes, a pointer to information about the event, and a pointer to another Event struct. The last field is used to place the Event struct in a linked list.

The Calendar struct keeps track of events by using an array of linked lists of Event structs. Field **name** points to the calendar's name. Field **events** points to an array, each entry of which points to a linked list of events scheduled for a particular day. The array has an entry for every day of the calender. Field **days** indicates the number of days in the calendar. Field **total_events** indicates the number of events in the calendar.

Field **comp_func** points to a comparison function (similar to compareTo in Java) for events. The events for each day are kept in increasing sorted order according to this function. Different comparison functions enable different sorting criteria: for example, events sorted by duration or by name.

Field **free_info_func** points to a function used to free the information pointed to by the **info** field of an event. This allows the information associated with an event to have arbitrary structure, for example, a struct, a string,

an array, anything. When an event is removed, this function is called to free the memory that stores the event's information.

The events in a calendar are uniquely identified by their names. An attempt to add an event fails if the event's name equals the name of an event already in the calendar.

5.3 Calendar Functions

Write all your functions in a file called calendar.c. The functions rely on the macros SUCCESS and FAILURE defined in calendar.h. If a function returns FAILURE, then it does not modify the calendar. Assume that if a char pointer in-parameter is not NULL, then it points to a nul-terminated string.

1. `int init_calendar(const char *name, int days,`
   `int (*comp_func) (const void *ptr1, const void *ptr2),`
   `void (*free_info_func) (void *ptr), Calendar ** calendar)`

   This function initializes a Calendar struct based on the parameters. The function allocates memory for the following items:

   a. Calendar struct
   b. Chunk of memory (pointed to by the **name** field) to hold a copy of parameter name's string.
   c. Chunk of memory (pointed to by the **events** field) to hold an array of pointers to Event structs. The length of the array is given by parameter days.

   The out-parameter **calendar** provides access to the new Calendar struct. The total number of events (field **total_events**) is set to zero.

   The function returns FAILURE if calendar and/or name are NULL, if the number of days is less than 1, or if any memory allocation fails. Otherwise the function returns SUCCESS.

2. `int print_calendar(Calendar *calendar, FILE *output_stream, int print_all)`

   This function prints, to the designated output stream, the calendar's name, days, and total number of events if **print_all** is true; otherwise this information is not printed. Information about each event (name, start time, and duration) is printed regardless of the value of **print_all**. See public tests output for format information. Notice that the heading "**** Events ****" is always printed.

   The function returns FAILURE if calendar and/or output_stream is NULL; otherwise the function returns SUCCESS.

3. `int add_event(Calendar *calendar, const char *name, int start_time,`
   `int duration_minutes, void *info, int day)`

   This function adds the specified event to the event list associated with parameter **day**, ensuring that the resulting event list is in increasing sorted order according to the comparison function (field **comp_func**). If the new event and an existing event in the event list have the same comp_func value, the new event is added **before** the existing event. The function allocates memory for the new event and for the event's name. The other fields of the event struct are initialized based on the parameter values.

   The function returns FAILURE if calendar and/or name are NULL, start time is not between 0 and 2400 (inclusive), duration_minutes is less than or equal to 0, day is less than 1 or greater than the number of calendar days, an event with the same name already exists in the calendar, there is no comparison function (comp_func is NULL), or if any memory allocation fails. Otherwise the function returns SUCCESS.

4. `int find_event(Calendar *calendar, const char *name, Event **event)`

   This function returns SUCCESS if calendar has an event with the specified **name**. In this case, it returns a pointer to that event via parameter **event** iff the parameter is not NULL (no pointer is returned if parameter **event** is NULL).

The function returns FAILURE if calendar and/or name are NULL, or if calendar does not have an event with the specified name.

5. `int find_event_in_day(Calendar *calendar, const char *name, int day, Event **event)`

   This function returns SUCCESS if calendar has an event with the specified **name** in the specified **day**. In this case, it returns a pointer to that event via parameter **event** iff the parameter is not NULL.

   The function returns FAILURE if calendar and/or name are NULL, if the day parameter is less than 1 or greater than the number of calendar days, or if the event is not found in the specified day.

6. `int remove_event(Calendar *calendar, const char *name)`

   This function returns SUCCESS if the calendar has the specified event. In this case, it removes the event from the calendar, updates the number of events, and frees any memory allocated for the event. It calls the **free_info_func** on the event's **info** field iff both of these are non-NULL.

   This function returns FAILURE if calendar and/or name are NULL, or if the event is not in the calendar.

7. `void *get_event_info(Calendar *calendar, const char *name)`

   This function returns the **info** pointer associated with the specified event. The function returns NULL if the event is not found. Assume the **calendar** and **name** parameters are not NULL.

8. `int clear_calendar(Calendar *calendar)`

   This function returns SUCCESS if calendar is not NULL. In this case, it removes every event (processing the event as described in function remove_event) and sets every event list to empty. The array of pointers to event lists is not removed.

   The function returns FAILURE if calendar is NULL.

9. `int clear_day(Calendar *calendar, int day)`

   This function returns FAILURE if calendar is NULL or if day is less than 1 or greater than the number of calendar days.

   Otherwise it removes all the events for the specified **day** (processing each event as described in function remove_event), sets the day's event list to empty, and returns SUCCESS.

10. `int destroy_calendar(Calendar *calendar)`

    This function returns FAILURE if calendar is NULL.

    Otherwise it removes every event (processing the event as described in function remove_event), frees all memory that was dynamically allocated for the calendar, and returns SUCCESS.

5.4 Calendar Testing

1. Define tests for your calendar implementation as described in the file student_tests.c. You will be graded on the quality of these tests. The main function in student_tests should run all tests, exit with the EXIT_FAILURE code if any of your tests fails, and EXIT_SUCCESS otherwise. The student_tests.c file in the distribution performs this task; you are welcome to rewrite it as you add tests.

2. Write student tests as you develop your code. If you need assistance during office hours, we may ask for these tests.

3. We expect at least one test for each function you need to implement.

4. We expect your tests to cover specific typical and atypical cases, for example, adding the same event twice, adding an event when comp_func is NULL, and so on. You will lose credit if we do not see such tests.

5.5 Dynamic Memory Allocation

1. As with all programs that use dynamic memory allocation, you must avoid memory leaks in your code in all circumstances.
2. Keep in mind that when you run valgrind on code that uses the memory tool we have provided, the tool may detect leaks for memory that valgrind allocates. To test your code using valgrind, disable our memory tool (by commenting out the function calls associated with the tool) and run valgrind. While running valgrind use the suggested options (e.g., valgrind --leak-check=full).
3. If a function requires multiple dynamic-memory allocations and one of them fails (malloc/calloc returns NULL), you are not responsible for freeing the memory of those allocations that were successful. For example:

```
mem1 = malloc(...)  /* This one is successful */
mem2 = malloc(...)  /* This one fails; just return FAILURE (don't worry about mem1) */
```

4. Frequent testing is the best way to check whether you are using dynamic-memory allocation correctly.

5.6 Style grading

Your code is expected to conform to the following style guidelines:

1. Your code (only calendar.c and student_tests.c) must have a comment at the beginning with your name, university ID number, and UMD Directory ID (i.e., your username on Grace).
2. Do not use global variables.
3. Avoid code duplication, otherwise you will lose credit.
4. Using multiple returns in a function is fine, but you should try to keep them to a minimum.
5. You should avoid **continue**, but if you need it, use at most one per function. We may penalize if you use more than one.
6. You should try to write code that is efficient, clear, and brief (if possible).
7. Adding auxiliary functions that support the functions you need to implement is recommended. It helps you to focus on different parts you need to implement. The more functions, the better (within reason). Just make sure you define them as static. Remember you cannot modify calendar.h.
8. As you are writing your code ask TAs for feedback regarding style.
9. Follow the C style guidelines available at:

   http://www.cs.umd.edu/~nelson/classes/resources/cstyleguide/

6 Submission

As usual, run **submit** in the project directory to submit.

The only files we grade are calendar.c, student_tests.c and Makefile. We use our versions of the header files to build our tests, so if you make changes to these files your code *may not compile*.