CMSC 216 Project #4 Fall 2024

Calendar **Due: Wed, Oct 23 / Mon, Oct 28, 2024, 11:55 pm**

1 Overview

In this project you will use dynamic memory allocation techniques in order to implement a calendar application. There are two deadlines:

- Wed, Oct 23, 11:55 pm - Your code must pass the first two public tests (public01, public02). That is the only requirement for this deadline. We will not grade the code for style. This first part is worth .5% of your course grade (NOT .5% of this project grade). You can still submit late for this part.
- Mon, Oct 28, 11:55 pm - Final deadline for the project. Notice you can still submit late (as usual).

2 Objectives

To practice dynamic memory allocation, function pointers, and linked lists.

3 Grading Criteria

Your project grade will be determined by the following:

Results of public tests	20 pts
Results of release tests	34 pts
Results of secret tests	24 pts
Student tests	10 pts
Code Style	6 pts
Makefile	6 pts

4 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. Posting code online can lead to an academic case where you will be reported to the Office of Student Conduct.

This project has been used in the past and you may find implementations online. Notice we are aware of the code sources, so you will be part of an academic integrity case if you use any such sources (even if you modify them). If you violate academic integrity rules, we will ask for an XF in the course; no exceptions.

5 Project Files

The files for this project can be found in the project4 directory of the public class directory. Make sure you copy that folder to the 216 folder in your home directory. The files associated with the distribution are:

1. `calendar.h` - This file provides prototypes for the functions your must implement. It also provides structure definitions, typedefs, and symbolic constants. **You may not modify this file.**
2. `event.h` - Defines an Event structure. **You may not modify this file.**
3. Makefile - This is the file where you will define the project's makefile. The name of the makefile must start with capital M.
4. public01.c, public02.c, public03.c, public04.c, public05.c, public06.c, public07.c - The public tests for this project.
5. student_tests.c - You will write student tests in this file.

6 Specifications

6.1 Makefile

Your `Makefile` should be set up so that all programs are built using separate compilation (i.e., source files are turned into object files, which are then linked together in a separate step). All object files should be built using the following options:

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

IMPORTANT: **You must implement the Makefile using explicit rules**; using implicit rules is not allowed. The Makefile examples associated with the lecture

`https://www.cs.umd.edu/class/fall2024/cmsc216-040X/prot/lectures/Week06/Make.pdf`

rely on explicit rules. 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 we have students that by mistake delete their implementation due to a Makefile error.

You will need to have the following targets present in your `Makefile`:

1. `all`: make all executables
2. `public01, public02, public03, public04, public05, public06, public07`: each one building a public test
3. `student_tests`: one of your executables.
4. `student_tests.o`:
5. `calendar.o`:
6. `clean`: delete all object files and executables

While you are welcome to add other targets to your `Makefile`, the ones listed above must be present. Make sure you add all the necessary targets that avoid any unnecessary compilation (hint: use the touch command to check your makefile).

6.2 Calendar Overview

For this project you will implement a calendar application that allows the scheduling of events at specific days. A calendar is defined by the following structures:

```
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 structure will represent a node in a linked list. The Event structure provides an event's name, start time (military time), duration in minutes, a pointer to data that provides information about the event, and a reference to another Event structure.

The Calendar structure keeps track of events by using an array of linked lists (**events** field). Each array entry is associated with a linked list that represents events scheduled for a particular day. The size of the array corresponds to the number of days the calendar has.

Calendar events can be sorted based on different criteria. The **comp_func** field represents a comparison function (similar to compareTo in Java) that will allow the insertion of events based on the criteria defined by the function. For example, you can have a calendar where events are sorted by duration, and another where events are sorted by name.

Events can be associated with additional information (**info** field). This information can be a structure, a string, anything. Once an event is no longer needed, the calendar can remove this information if a free function (that knows how to free anything associated with the information) is provided. The **free_info_func** field represents this free function.

6.3 Calendar Functions

You must write all your functions in a file called calendar.c. Functions rely on the macros SUCCESS and FAILURE defined in calendar.h.

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 structure based on the parameters provided. The function allocates memory for the following items:

   a. Calendar structure
   b. **name** field will be assigned memory necessary to store a copy of the name parameter. The name represents the calendar's name.
   c. **events** field will be assigned memory necessary to represent an array of pointers to Event structures. The size of this array corresponds to the **days** parameter.

   The out parameter **calendar** will provide access to the new Calendar structure. Notice the total number of events will be set to zero. The function will fail and return 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 will return 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 will not be printed. Information about each event (name, start time, and duration) will be printed regardless the value of **print_all**. See public tests output for format information. Notice that the heading "**** Events ****" will always be printed.

   This function will fail and return FAILURE if calendar and/or output_stream are NULL; otherwise the function will return 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 list associated with the **day** parameter. The event must be added in increasing sorted order using the comparison function (comp_func) that allows comparison of two events. The function must allocate memory for the new event and for the event's name. Other fields of the event structure will be initialized based on the parameter values.

   This function will fail and return FAILURE if calendar and/or name are NULL, start time is invalid (must be a value 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, if the event already exist, or if any memory allocation fails. Otherwise the function will return SUCCESS.

Notice that events are uniquely identified across the calendar by name.

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

   This function will return a pointer (via the out parameter **event**) to the event with the specified **name** (if any). If the **event** parameter is NULL, no pointer will be returned. Notice the out parameter **event** should not be modified unless the event is found. The function will fail and return FAILURE if calendar and/or name are NULL. The function will return SUCCESS if the event was found and FAILURE otherwise.

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

   This function will return a pointer (via the out parameter **event**) to the event with the specified **name** in the specified **day** (if such event exist). If the **event** parameter is NULL, no pointer will be returned. Notice the out parameter **event** should not be modified unless the event is found. The function will fail and return FAILURE if calendar and/or name are NULL, or if the day parameter is less than 1 or greater than the number of calendar days. The function will return SUCCESS if the event was found and FAILURE otherwise.

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

   This function will remove the specified event from the calendar returning any memory allocated for the event. If the event has an **info** field other than NULL and a **free_info_func** exists, the function will be called on the **info** field. The number of calendar events must be adjusted accordingly. This function will fail and return FAILURE if calendar and/or name are NULL or if the event cannot be found; otherwise the function will return SUCCESS.

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. You can assume the **calendar** and **name** parameters are different than NULL.

8. `int clear_calendar(Calendar *calendar);`

   This function will remove all the event lists associated with the calendar and set them to empty lists. Notice that the array holding the event lists will not be removed. The total number of events will be set to 0. If an event has an **info** field other than NULL and a **free_info_func** exists, the function will be called on the **info** field. This function will fail and return FAILURE if calendar is NULL; otherwise the function will return SUCCESS.

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

   This function will remove all the events for the specified **day** setting the event list to empty. The total number of events will be adjusted accordingly. If an event has an **info** field other than NULL and a **free_info_func** exists, the function will be called on the **info** field. This function will fail and return FAILURE if calendar is NULL, or if the day is less than 1 or greater than the calendar days; otherwise the function will return SUCCESS.

10. `int destroy_calendar(Calendar *calendar);`

    This function will return memory that was dynamically-allocated for the calendar. If an event has an **info** field other than NULL and a **free_info_func** exists, the function will be called on the **info** field. This function will fail and return FAILURE if calendar is NULL; otherwise the function will return SUCCESS.

6.4 Calendar Testing

1. You must 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, but 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, though you are welcome to rewrite it as you add tests.

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

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

4. We expect your tests to cover specific cases. For example, your tests should check what takes place when adding the same event twice. If we do not see such a test case you will lose credit. There are other cases we are expecting you to test.

6.5 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. We set the submit server to use valgrind with the following options:

    ```
    valgrind --track-origins=yes --leak-check=full --show-leak-kinds=all -q
    ```

    Make sure you use the same options while testing your code. Remember that replacing the -q option with -s displays information in case of errors.
3. If a function requires multiple dynamic-memory allocations and one of them fails (malloc/callocs 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 failed, just return FAILURE (don't worry about mem1) */
    ```

4. The best way to check you are using dynamic-memory allocation correctly is to test often.

6.6 Other

1. While adding event: When the compare function returns 0 it should not make a difference where you add the new event, but add the new event before the existing one.
2. Your code will work in the submit server without you providing a makefile, but you should write your makefile immediately and not wait until the end. Remember to backup your code before writing a makefile as students have removed their implementation by mistake while writing a makefile.
3. You need to run valgrind in all the public and student tests. If you are failing release tests, chances are that it is because valgrind is reporting errors on public tests.

6.7 Style grading

For this project 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. Make sure you avoid code duplication, otherwise you will lose credit.
4. Although using multiple returns in a function is fine, 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. Just make sure you defined 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/`

7 Submission

7.1 Deliverables

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

7.2 Procedure

The submission procedure is the same as for previous projects (execute "`submit`" in the project directory).