Assignment 1

Due: Monday, June 5, 2023 (11:59 pm)

Submission via Git only

Programming environment

For this assignment you must ensure your code executes correctly on the reference platform you configured as part of Assignment #0 and Lab 01. This same environment will also be used by the teaching team when evaluating your submitted work.

All test files and sample code for this assignment are available on your Git repository. Git will be discussed in detail during the week of May 22th. Nevertheless, you can clone your repository by running the following command (only needs to be executed once):

```
git clone ssh://NETLINKID@git.seng.uvic.ca/seng265/NETLINKID
```

An important learning outcome of SENG 265 is to learn the tools on the Unix/Linux platform including shell scripting languages (e.g., Bash), keyboard editors (e.g., vim), dependency analyzers (e.g., make & makefiles), compilers (e.g., gcc), interpreters (e.g., Python 3), file system exploration and manipulation tools (e.g., grep, locate, ls, pwd, cd, tree, cp, rm). Therefore, you will have to make sure that your programs compile, link and execute perfectly on the reference platform.

Individual work

This assignment is to be completed by each individual student (i.e., no group work). You are encouraged to discuss aspects of the problem with your fellow students. **However, sharing of code fragments is strictly forbidden.** Note SENG 265 uses highly effective plagiarism detection tools to discover copied code in your submitted work. Both, using code from others and providing code to others, are considered cheating.

Objectives of this assignment

- Understand a problem description, along with the role of the provided sample input and output.
- Use the programming language C to implement a Unix filter—an event processor named *event_manager.c*—without resorting to dynamic memory allocation.
- Leverage Unix commands, such as diff, in your coding and testing. The lab instructors will explain in the labs how to use the *diff* command.
- Use *git* to manage changes in your source code and annotate the continuous evolution of your solution with “messages” given to commits.
- Test your code against the provided test cases.

This assignment: *event_manager.c*

In this assignment, you will learn C by solving a problem involving a file format used by several calendar programs, such as Google Calendar.

You are to write a C program that inputs lines of data from a provided calendar file, accepts options and arguments from the command line, and then outputs to the console events from the calendar file into a more readable form. To get started, the teaching team has provided a skeleton version of *event_manager.c*.

Once you downloaded the required files for Assignment 1 from your Git repository, you will find the input files with the extension *.ics*. For example, *one.ics* is an extract from the schedule of a fictional UVic student:

```
BEGIN:VCALENDAR
BEGIN:VEVENT
DTSTART:20230214T180000
DTEND:20230214T210000
LOCATION:Burger King
SUMMARY:Romantic dinner with Chris
END:VEVENT
END:VCALENDAR
```

This file contains information for a fictional single event taking place on February 14, 2023. Suppose you type the following arguments into your program (after having copied the *.ics* files from the SENG265 server as directed at the start of this assignment description):

```
./event_manager --start=2023/2/14 --end=2023/2/14 --file=one.ics
```

Then the output to be produced is as follows:

```
February 14, 2023
------------------------
 6:00 PM to  9:00 PM: Romantic dinner with Chris {{Burger king}}
```

Note that the arguments passed to the program were two dates (year/month/day) plus a filename. The output reflects that the script printed out all events within the *.ics* file occurring during the specified range of dates (which, in this case, ends up being a single event).

Note that each line in an *.ics* file has a similar structure:

```
<property>:<value>
```

That is, a property's name and the property's value are separated by a single colon. **In this assignment your program need only pay attention to the following properties:**

- BEGIN
- END
- DTSTART
- DTEND
- RRULE
- LOCATION
- SUMMARY

At the end of this document is a more detailed specification for the input files your program must read and parse, as well as syntax and format expected in the output. Since such specifications frequently are ambiguous, the provided test-output files (i.e., files that begin with the letters *"test"*) represent the required output format. The *TESTS.md* markdown file describes each of the ten tests, with corresponding test outputs listed as *test01.txt, test02.txt,* etc.

In order check for correctness, please use the UNIX command *diff.* For example, assuming that *diana-devops.ics* is in the same directory as your compiled program, you can compare your output against what is expected for the 8th test using the command shown below (assuming the file *test8.txt* is in the same directory as *diana-devops.ics*):

```
./event_manager --start=2023/1/1 --end=2023/12/31 --file=diana-devops.ics | diff test08.txt -
```

The ending dash as an argument to *diff* will compare *test01.txt* with the text stream piped into the *diff* command. If no output is produced by *diff*, then the output of your program matches the file exactly (i.e., the 1st test passes). Note that the arguments you must pass to *event_manager* for each of the tests are shown within the *TESTS.md* file.

It is very import to use the *diff* command for testing. Your output must exactly match the expected test output for a test to pass. Do not attempt to visually check test cases — the eye is often willing to deceive the brain, especially with respect to the presence (or absence) of horizontal and vertical spaces. Please note that *diff* will fail if there is extra or missing white space. In this case, the test is considered a failed test.

Exercises for this assignment

To facilitate the development of event_*manager*, try to decompose the problem into small and more manageable parts. To promote this, we suggest dividing the assignment into the following parts.

Part 1: Argument processing

- Obtain the arguments from the command line using the char *argv[] parameter of your main function.
- To get the actual value of the argument, use the function strtok().

Part 2: Read file content

- You will need to read line by line the file passed as an argument to *event_manager*. The functions fopen() and fgets() together with for or while loops can readily accomplish this task.

Part 3: Process event lines

- Once your program can read all the lines from the input *.ics* file, you will need to process each line to obtain relevant data to produce the required output.
- Use the function strtok() again to obtain data from the *.ics* properties.
- Another option is to create some representation of the events in the file using a struct (optional but recommended).

Part 4: Print output

- Use the printf() function and format specifiers to generate the required output.

Requirements and recommendations

1. **You MUST use the -std=c99 flag when compiling your program as this will be used during assignment evaluation (i.e., the flag ensures the 1999 C standard is used during compilation).**

2. **DO NOT use malloc(), calloc() or any of the dynamic memory functions.** For this assignment you can assume that the longest input line will have 80 characters, and the total number of events output generated by a *from/to/testfile* combination will never exceed 500.
3. Keep all of your code in one file for this assignment (that is, *event_manager.c*). In later assignments we will use the separable compilation facility available in C.
4. Use the test files to guide your implementation effort. Start with the simple example in test 01 and move onto 02, 03, etc. in order. **Refrain from writing the program all at once, and budget time to anticipate when things go wrong!** Use the Unix command *diff* to compare your output with what is expected.
5. For this assignment you can assume all test inputs will be well-formed (i.e., the teaching team will not evaluate your submission for handling of input or for arguments containing errors). Later assignments might specify error-handling as part of their requirements.
6. Use git when working on your assignment. Remember, that the ONLY acceptable method of submission is through Git.

What you must submit

A single C source file named **event_manager.c**, submitted to the a1 folder **in your Git repository**. Git is the **only** acceptable way of submission.

Evaluation

Assignment 1 grading scheme is as follows.

A grade: A submission completing the requirements of the assignment which is well-structured and clearly written. Global variables are not used. event_manager runs without any problems; that is, all tests pass and therefore no extraneous output is produced. This is a good submission that passes all the tests and does not have any overall quality issues. Outstanding solutions get an A+ (90-100 marks). Solutions that are not considered outstanding by the evaluator will get an A (85-89 marks). A solution with minor issues will be given an A- (80-84 marks).

B grade: A submission completing the requirements of the assignment. event_manager runs without any problems; that is, all tests pass and therefore no extraneous output is produced. The program is clearly written. Although all the tests pass, the solution includes significant quality issues. Depending on the number of qualitative issues, the marker may give a B+ (77-79 marks), B (73-76 marks) or a B- (70-72 marks) grade.

A submission with any one of the following cannot get a grade higher than B:

- Submission compiles with warnings
- Submission has 1 or 2 large functions
- Program or file-scope variables are used

A submission with more than one of the following cannot be given a grade of higher than B-:

- Submission compiles with warnings
- Submission has 1 or 2 large functions.

- Program or file-scope variables

C grade: A submission completing most of the requirements of the assignment. event_manager runs with some problems. This is a submission that present a proper effort but fails some tests. Depending on the number of tests passed, which tests pass and a qualitative assessment, a grade of C (60-64 marks) or C+ (65-69 marks) is given.

D grade: A serious attempt at completing requirements for the assignment (50-59 marks). event_manager runs with quite a few problems. This is a submission that passes only a few of the trivial tests.

F grade: Either no submission given, or submission represents little work or none of the tests pass (0-49 marks). No submission, 0 marks**.** Submissions that do not compile, 0 marks. Submissions that do not run, 0 marks. Submissions that fail all tests and show a very poor to no effort (as evaluated by the marker) are given 0 marks. Submissions that fail all tests, but represent a sincere effort (as evaluated by the marker) may be given a few marks.

In general, straying from the assignment requirements will be penalized severely.

Additional Criteria for Qualitative Assessment

Documentation and commenting: the purpose of documentation and commenting is to write information so that anyone other than yourself (with knowledge of coding) can review your program and quickly understand how it works. In terms of marking, documentation is not a large mark, but it will be part of the overall quality assessment.

Functional decomposition: quality coding requires the good use of functions. Code that relies on few large functions to accomplish its goals is very poor-quality code. Typically, a good program has a main function that does some basic tasks and calls other functions, that do most of the work. A solution that passes all tests, but contains all code in one or two large functions will not be given a grade better than a B. You must not use program-scope or file-scope variables.

Proper naming conventions: You must use proper names for functions and variables. Using random or single character variables is considered improper coding and significantly reduces code readability.

Debugging / Comment artifacts: You must submit a clean file with no residual commented lines of code or unintended text.

Quality of solution: the marker will access the submission for logical and functional quality of the solution. Some examples that would result in a reduction of marks: solutions that read the input files several times, solutions which represent the data in inappropriate data structures, solutions which scale unreasonably with the size of the input.

ONLY solutions that comply with the criteria mentioned above and with other relevant aspects at discretion of the evaluator will be considered as A+ submissions.

Input specification

1. All input is from ASCII-data test files.

2. Data lines for an “event” begin with a line “BEGIN:VEVENT” and end with a line “END:VEVENT”.
3. Starting time: An event’s starting date and time is contained on a line of the format “DTSTART:<icalendardate>” where the characters following the colon comprise the date/time in icalendar format.
4. Ending time: An event’s ending date and time is contained on a line of the format “DTEND:<icalendardate>” where the characters following the colon comprise the date/time in icalendar format.
5. Event location: An event’s location is contained on a line of the format “LOCATION:<string>” where the characters following the colon comprise the string describing the event location. These strings will never contain the “:” character.
6. Event description: An event’s description is contained on a line of the format “SUMMARY:<string>” where the characters following the colon comprise the string describing the event’s nature. These strings will never contain the “:” character.
7. Repeat specification: If an event repeats, this will be indicated by a line of the format “RRULE:FREQ=<frequency>;UNTIL=<icalendardate>”. The only frequencies you must account for are weekly frequencies. The date indicated by UNTIL is the last date on which the event will occur (i.e., is inclusive and for the sake of simplicity, repetitions will only happen within the same month and year). Note that this line contains a colon (“:”) and semicolon (“;”) and equal signs (“=”).
8. **Events within the input stream ARE ALWAYS in chronological order.** This reduces the complexity of the assignment significantly.
9. Events may overlap in time.
10. No event will ever cross a day boundary.
11. All times are local time (i.e., no timezones will appear in a date/time string). This semester we will ignore the effect of switching to daylight savings.

Output specification

1. All output is to stdout.
2. All events which occur from 12:00 am on the --start date and to 11:59 pm on the --end date must appear in chronological order based on the event’s starting time that day.
3. If events occur on a particular date, then that date must be printed only once in the following format:

   ```
   <month text> <day>, <year>
   ------------------------------------
   ```

   Note that the line of dashes below the date must match the length of the date. You may use function such strftime() from the C library in order to create the calendar-date line.
4. Days are separated by a single blank line. There is no blank line at the start or at the end of the program’s output.
5. Starting and ending times given in 12-hour format with “am” and “pm” as appropriate. For example, five minutes after midnight is represented as “12:05 am”.

6. A colon is used to separate the start/end times from the event description
7. The event SUMMARY text appears on the same line as the even time. (This text may include parentheses.)
8. The event LOCATION text appears on after the SUMMARY text and is surrounded by square brackets.

Events from the same day are printed on successive lines in chronological order by starting time. Do not use blank lines to separate the event lines within the same day.

In the case of tests provided by the instructor, the Unix “diff” utility will be used to compare your program’s output with what is expected for that test. Significant differences reported by “diff” may result in grade reductions.