Software Engineering SENG 265
Software Development Methods
Spring 2022

Assignment 1

Due: February 11 @ 23:55 — submission via repository in Gitlab
(no late submissions accepted due to 200+ students in this course)

Programming environment

For this assignment you must ensure your work executes correctly on the virtual machine you installed as part of Assignment #0 subsequently referred to as *Senjhalla*. This is our "reference platform". This same environment will also be used by the teaching team when evaluating submitted work from students.

All test files and sample code for this assignment are available on the SENG server in /seng265work/2022-spring/a1/ and you must use scp in a manner similar to what happens in labs in order to copy these files into your *Senjhalla*. For example:

```
scp NETLINKID@seng265.seng.uvic.ca:/seng265work/2022-spring/a1/* .
```

Any programming done outside of *Senjhalla* might not work during evaluation, which will result in lost marks or even 0 marks for the assignment. If you are

Individual work

This assignment is to be completed by each individual student (i.e., no group work). Naturally you may want to discuss aspects of the problem with fellow students, and such discussion is encouraged. **However, sharing of code fragments is strictly forbidden.** If you are still unsure regarding what is permitted or have other questions about what constitutes appropriate collaboration, please contact the course instructor. Note **code-similarity analysis tools are used to examine submitted work.**

Objectives of this assignment

- Understand a problem description, along with the role of the sample input and output for provided.
- Use the programming language C to write the first implementation of a file processor named *process_cal.c* without using dynamic memory.
- Leverage Unix commands, such as diff, in your coding and testing.
- Use git to manage changes in your source code and annotate the evolution of your solution with "messages" given to commits.
- Test your code against the provided test cases.

This assignment: *process_cal.c*

In this assignment you will learn C by solving a problem involving a file format used by multiple programs and applications called the Extensible Markup Language (XML). These files usually have a filename suffix of "xml". To simplify your solution, the .xml files provided for this term will be less complex than what is possible using the XML standard.

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

An example of the .xml files that your program needs to process can be found on the SENG server at UVic (*seng265.seng.uvic.ca*) in the /seng265work/2022-spring/a1 subdirectory (*aston-martin-release.xml*):

```
<calendar>
    <event>
        <description>Aston Martin F1 2022 Car Release</description>
        <timezone>GMT-8</timezone>
        <location>Online</location>
        <day>10</day>
        <month>02</month>
        <year>2022</year>
        <dweek>Friday</dweek>
        <start>11:00</start>
        <end>13:00</end>
    </event>
</calendar>
```

This particular file contains information for a single event taking place on February 10, 2022. Suppose you type the following arguments into your program (after having copied the .xml files from the SENG server as directed at the start of this assignment description):

```
./process_cal --start=2022/1/1 --end=2021/2/28 --file=aston-martin-release.xml
```

then the output to be produced is as follows:

```
February 10, 2022 (Friday)
----------------------
 10:00 AM to 01:00 PM: Aston Martin F1 2022 Car Release {{Online}} | GMT-8
```

Note that the three 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 *.xml* file occurring during the specified range of dates (which, in this case, ends up being a single event).

Another possibility is to have multiple events in a single file and events happening on the same date. Consider the contents of the file 2022-season-testing.xml:

```
<calendar>
	<event>
		<description>F1 2022 Pre-Season Testing Day 1 - Session 1</description>
		<timezone>GMT+1</timezone>
		<location>Barcelona, Spain</location>
		<day>23</day>
		<month>02</month>
		<year>2022</year>
		<dweek>Wednesday</dweek>
		<start>08:00</start>
		<end>11:00</end>
	</event>
	<event>
		<description>F1 2022 Pre-Season Testing Day 1 - Session 2</description>
		<timezone>GMT+1</timezone>
		<location>Barcelona, Spain</location>
		<day>23</day>
		<month>02</month>
		<year>2022</year>
		<dweek>Wednesday</dweek>
		<start>12:00</start>
		<end>15:00</end>
	</event>
	<event>
		<description>F1 2022 Pre-Season Testing Day 4 - Session 1</description>
		<timezone>GMT+3</timezone>
		<location>Sakhir, Bahrain</location>
		<day>10</day>
		<month>03</month>
		<year>2022</year>
		<dweek>Thursday</dweek>
		<start>08:00</start>
		<end>11:00</end>
	</event>
</calendar>
```

```
./process_cal --start=2022/2/1 --end=2022/3/15 --file=2022-season-testing.xml
```

```
February 23, 2022 (Wednesday)
----------------------
08:00 AM to 11:00 AM: F1 2022 Pre-Season Testing Day 1 - Session 1 {{Barcelona, Spain}} | GMT+1
12:00 PM to 03:00 PM: F1 2022 Pre-Season Testing Day 1 - Session 2 {{Barcelona, Spain}} | GMT+1

March 10, 2022 (Thursday)
----------------------
08:00 AM to 11:00 AM: F1 2022 Pre-Season Testing Day 4 - Session 1 {{Sakhir, Bahrain}} | GMT+3
```

Note that elements in .xml files are defined by tags (i.e., <nameOfTag>). One element can have child elements (i.e., tags can be embedded in other tags) and the end of an element is defined by </nameOfTag>. Therefore, for this assignment in XML, one element can be a list of other elements (e.g., <calendar> or <event>) or can represent a string/numeric property (e.g., <timezone>, <description>, <dweek>). The original XML specification allows to include properties/attributes inside tags, but, for the sake of simplicity, this is not included in this assignment.

In this assignment your program need only pay attention to the following tags:

- <calendar>
- <event>
- <description>
- <timezone>
- <location>
- <day>
- <month>
- <year>
- <dweek>
- <start>
- <end>

At the end of this document is a more detailed specification for the input files your program must accept, and properties expected in the output. However, given that such specifications always have some elements of ambiguity, the test-output files (i.e., all those provided to you beginning with the letters "*test*") can be considered as using 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 called *diff*. For example, assuming that aston-martin-release.xml is in the same directory as your compiled program, you can compare your output against what is expected for the first test using the command shown below (and which assumes the *test01.txt* file is in the same directory as aston-martin-release.xml):

```
./process_cal --start=2022/1/1 --end=2021/2/18 --file=aston-martin-release.xml | diff test01.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 identically matches the file (i.e., the first test passes). Note that the arguments you must pass to *process_cal* for each of the tests is shown within *TESTS.md* (i.e., /seng265work/2022-spring/a1) on the UVic SENG server).

Please use diff. Your output must exactly match the expected test output in order for a test to pass. Do not attempt to visually check all test cases - the eye is often willing to deceive the brain, especially with respect to the presence (or absence) of horizontal and vertical spaces. 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

1. Write your program. Amongst other tasks you will need to:
   - obtain a filename argument from the command line
   - read text input from a file, line by line, and the text within those lines
   - construct some representation of not only the events in the file, but what is needed to print out the events in the range of dates
   - **You should 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 100 characters, and the total number of events output generated by a *from*/*to*/*testfile* combination (including repeats of events) will never exceed 1000.
3. Keep all of your code in one file for this assignment (that is, *process_cal.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. (You may want to avoid tests 6 and 7until you have significant functionality already completed.) **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.
7. Remember, that the ONLY acceptable method of submission is through your Gitlab repository.

What you must submit

- A single C source file named **process_cal.c**, submitted to the Assignment 1 page **in the a1 folder of your Gitlab repository**. Gitlab is the **only** acceptable way of submission. It is your responsibility to ensure that the correct file is submitted.
- **No dynamic memory-allocation routines are to be used for Assignment 1.**

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. *process_cal* 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) and a solution with minor issues will be given an A- (80-84 marks).
- **B grade**: A submission completing the requirements of the assignment. *process_cal* 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. *process_cal* 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). *process_cal*
  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:** 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" are contained within the block defined by the tags <event></event>.
3. Starting time: An event's starting date and time is contained on a line including the tag <start>.
4. Ending time: An event's ending date and time is contained on a including the tag <end>
5. Event location: An event's location is contained on a line including the tag <location>.
6. Event date: An event's date information is contained on the lines including the following tags: <day> (numeric), <month> (numeric), <year> (numeric), <dweek> (string), and <timezone> (string)
7. Event description: An event's description is contained on a line including the tag <description>.
8. Events within the input stream are not necessarily in chronological order.
9. Events may overlap in time.
10. No event will ever cross a day boundary.

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> (<day of week>)

   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 <description> text appears on the same line as the even time. (This text may include parentheses.)
8. The event <location> text appears on after the <description> text and is surrounded by square brackets, followed by the character "|" and the <timezone> of the event.

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.