Software Engineering 265
Software Development Methods
Fall 2021

Assignment 3

Due: Wednesday, 17th November 2021 @23:55 -- Submission via Brightspace
(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 machines 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 course instructor and the rest of the teaching team when evaluating submitted work from students.

All test files for this assignment are available on the SENG server in /seng265work/A2/ and you must scp in a manner similar to previous assignments, in order to copy these files into your Senjhalla

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

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

- Revisit the C programming language, this time using dynamic memory.
- Use Git to manage changes in your source code and annotate the evolution of your solution with messages provided during commits.
- Test your code against the provided test cases.
- Use *valgrind* to determine how effective your solution is in its management of dynamic memory.

This assignment: process_cal3.c, Using C's heap memory

You are to write an implementation again in C such that:

- **only** dynamic memory is used to store event info, and
- **only** linked-list routines are used (i.e. arrays of events *are not* permitted).

In addition to these requirements, the program itself now consists of several files, some of which are C source code, one of which is for build management:

- emalloc.[ch]: Code for safe calls to malloc, as is described in lectures, is available here.
- ics.h: Type definition for events.
- linky.[ch]: Type definitions, prototypes, and codes for the singly-linked list implementation described in lectures. You are permitted to modify these routines or add to these routines in order to suit your solution. Regardless of whether or not you do so, however, you are fully responsible for any segmentation faults that occur as the result of this code's operation.
- makefile: This automates many of the steps required to build the process_cal3 executable, regardless of what files (.c or .h) are modified. The Unix make utility will be described in lectures.
- process_cal3.c: A starter file.

You must ensure all of these files are submitted in brightspace. DO NOT add any extra files.

A call to process_cal3 will use identical arguments to that from the previous assignment. For example:

```
./process_cal3 --start=2021/12/10 --end=2021/12/12 --file=one.ics
```

A few more observations:

- All allocated heap memory is automatically returned to the operating system upon the termination of a Unix process or program (such as process_cal3). This is true regardless of whether the programmer uses free() to deallocate memory in the program or not. However, it is always a good practice to deallocate memory via free() – that is, one never knows when their code may be re-used in the future, and having to rewrite existing code to properly deal with memory deallocation can be difficult. A program where all memory is properly deallocated by the programmer will produce a report from valgrind stating that all heap blocks were free and that the heap memory in use at exit is "0 bytes in 0 blocks". valgrind will be discussed during labs. **Severe marking penalties** will be applied to solution that do not deallocate memory.
- You must **not use program-scope or file-scope variables**. You must **not use arrays of type struct event_t or event_t.**[1]

[1] This also means you may not have an array or arrays of node_t where each array element is simply a linked list that is one node in length.

- You must **make good use of functional decomposition**. Phrased another way, your submitted work **must not** contain one or two giant functions where all of your program logic is concentrated.
- You are free to use regular expressions in your solution, but are not required to do so.
- Some of the limits from previous assignments that were placed on certain values are no longer needed, e.g., maximum number of events, maximum line length, etc. *Also note that some events have an ending time that is on the day following the starting time.*

What you must submit

- The seven files listed earlier in this assignment description (process_cal3.c, emalloc.c, emalloc.h, ics.h, listy.c, listy.h, , makefile), submitted to the assignment 03 page in Brightspace. Brightspace is the only acceptable way of submission.
- Note that Brightspace will allow multiple submissions of an assignment. However, **only the latest batch of submitted files retained** and earlier submissions will be deleted. It is your responsibility to ensure that **all the correct files are submitted.**
    - If you submit some files first and then return to submit the remaining files, Brightspace will remove the earlier batch of files submitted.
    - For example, suppose you submit ics.h, emalloc.h and emalloc.c first. Then, you return to submit listy.h, listy.c and process_cal3.c. When you do this, brightspace will remove ics.h, emalloc.h and emalloc.c and the only files retained by brightspace will be listy.h, listy.c and process_cal3.c. This will result in an incomplete submission for the assignment.
    - Therefore, **you must submit all the files in each submission.** Otherwise, the submission will be incomplete and a failing grade will result.

Evaluation

Our grading scheme is relatively simple.

- "A" grade: A submission completing the requirements of the assignment which is well-structured and very clearly written. All tests pass and therefore no extraneous output is produced. valgrind produces a report stating that no heap blocks or heap memory is in use at the termination of process_cal3.
- "B" grade: A submission completing the requirements of the assignment. process_cal3 can be used without any problems; that is, all tests pass and therefore no extraneous output is produced. valgrind states that some heap memory is still in use.
- "C" grade: A submission completing most of the requirements of the assignment. process_cal3 runs with some problems.
- "D" grade: A serious attempt at completing requirements for the assignment. process_cal3 runs with quite a few problems; some non-trivial tests pass.
- "F" grade: Either no submission given, or submission represents very little work, or no tests pass.