Software Engineering 265
Software Development Methods
Summer 2019

Assignment 4

Due: Friday, August 2nd, 11:55 pm by submission via git
(no late submissions accepted)

Programming environment

For this assignment please ensure your work executes correctly on the Linux machines in ELW B238. You are welcome to use your own laptops and desktops for much of your programming; if you do this, give yourself a few days before the due date to iron out any bugs in the C program you have uploaded to the BSEng machines. (Bugs in this kind of programming tend to be platform specific, and something that works perfectly at home may end up crashing on a different hardware configuration.)

Individual work

This assignment is to be completed by each individual student (i.e., no group work). Naturally you will want to discuss aspects of the problem with fellow students, and such discussion is encouraged. **However, sharing of code fragments is strictly forbidden without the express written permission of the course instructor (Zastre).** If you are still unsure regarding what is permitted or have other questions about what constitutes appropriate collaboration, please contact me as soon as possible. (Code-similarity analysis tools will be used to examine submitted programs.)

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 15 provided test cases from assignment #1.
- Use `valgrind` to determine how effective your solution is in its management of dynamic memory.

`calprint4.c`: Using C's heap memory for the last implementation of *calprint*

You are to write an implementation called `calprint4`, and you are required to write it 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:

- `calprint4.c`: The majority of your solution will most likely appear in this file. Some demo code (protected with an `#ifdef DEBUG` conditional-compilation directive) shows how a simple list with two nodes of events can be allocated, assigned values, printed, and deallocated.
- `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 `calprint4` executable, regardless of what files (`.c` or `.h`) are modified. The Unix `make` utility will be described in lectures.

You must ensure all of these files are in the `a4/` directory of your repo, and must also ensure that all of these files are added, committed, and pushed. Do not add any extra files.

A call to `calprint4` will use identical arguments to that from the previous assignment. For example:

```
./calprint4 --start=18/6/2019 --end=18/6/2019 --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 `calprint4`). 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 the last week of labs of the semester.

- You are free to use regular expressions in your solution, but are not required to do so.

- You must **not use program-scope or file-scope variables**. You must **not use arrays `struct event_t` or `event_t`.**[1]

- 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.

Exercises for this assignment

- Within your `git` repo ensure there is an "`a4/`" subdirectory. (For testing please use the files provided for assignment #1.) All seven files described earlier in this document must be in that subdirectory. Note that starter versions of all these files are available for you in the `/home/zastre/seng265/a4` directory.

- Write your program. Amongst other tasks you will need to:
  - read text input from a file, line by line.
  - implement required methods for the solution, along with any other methods you believe are necessary.
  - extract substrings from lines produced when reading a file
  - write, test, and debug linked-list routines.

- Use the test files and listed test cases to guide your implementation effort. Refrain from writing the program all at once, and budget time to anticipate when "things go wrong".

- For this assignment you can assume all test inputs will be well-formed (i.e., our teaching assistant will not test your submission for handling of input or for arguments containing errors). I had wanted to throw some error handling

[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.

at you, but I think you have your hands full enough with wrangling C into behaving well with dynamic memory.

What you must submit

- The seven files listed earlier in this assignment description (`calprint4.c`, `emalloc.c`, `emalloc.h`, `ics.h`, `listy.c`, `listy.h`, , `makefile`).

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 `calprint4`.

- "B" grade: A submission completing the requirements of the assignment. `calprint4` 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. `calprint4` runs with some problems.

- "D" grade: A serious attempt at completing requirements for the assignment. `calprint4` 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.