Software Engineering 265
Software Development Methods
Summer 2021

Assignment 3

Due: Monday, July 19, 11:55 pm by submission via git
(no late submissions accepted)

Programming environment

For this assignment you must ensure your work executes correctly on the virtual machines you installed as part of Assignment #0 (which I have taken to calling `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 starter code for this assignment is available on the UVic Unix server in `/home/zastre/seng265/a3` and you must use `scp` in a manner similar to what happens in labs in order to copy these files into your `Senjhalla`. The same tests used for assignment #2 (i.e. those in `/home/zastre/seng265/a2`) will be used for assignment three with the exception of test 18 (i.e. this test will not be used).

Any programming done outside of `Senjhalla` might not work during evaluation. (I know many of you are using VS Code, but you should try to get comfortable with `vim` as you never know when you'll be confronted with an environment without an IDE.)

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 about 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.)

Questions

If you have any questions about the assignment, please post them to our course's channel at `rocket.csc.uvic.ca` (i.e., RocketChat). By using this Slack-like environment, I can not only provide answers to queries but also give each of you a chance to write answers to questions from fellow students.

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 19 provided test cases from assignment #2 (i.e. all tests except for test 18).
- Use `valgrind` to determine how effective your solution is in its management of dynamic memory.

`concord3.c`: Using C's heap memory for CONCORD

In assignment 2 you used Python to implement an extended version of the CONCORD scheme (i.e., words with identical spelling but different lettercase were considered the same keyword). For this assignment you are to write an implementation called `concord3` to provide the same functionality, but this time using C and using dynamic memory.

We will, however, re-introduce two restrictions on input in order to help you with your problem solving for your solution. That is:

- You may assume that keywords will be at most 40 characters long, *and you may use* a compile-time constant to represent this.

- You may assume the length of any input line will be at most 100 characters, *and you may use* a compile-time constant to represent this.

Note that there *is no restriction* or upper limit on the *number* of distinct keywords or exceptions words, nor is there any restriction on the *number* of lines of input. (This is similar to assignment #2.) That is, you are not permitted to use any compile-time constants for these. **Put differently, you are not permitted to *statically* declare arrays large enough to hold all the keywords, exception words, and input lines.**

Therefore in order to store keywords, exception words, and input lines, **you must to use either linked-lists** or **dynamically-sized arrays** or both.

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

- `concord3.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 consisting of words can be constructed, traversed, and destroyed.

- `emalloc.[ch]`: Code for safe calls to `malloc()`, as is described in lectures, is available here.

- `seng265-list.[ch]`: Type definitions, prototypes, and code 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 `concord3` executable, regardless of what files (`.c` or `.h`) are modified. The Unix `make` utility will be described in lectures.

Starter versions of these files are in the `/home/zastre/seng265/a3` directory.

You must ensure all of these files are in the a3/ directory of <u>your</u> repo, and must also ensure that all of these files are properly added, committed, and pushed.

You are not permitted to add source-code files to your submission without first obtaining express written permission from the course instructor.

A call to `concord3` will use identical arguments to that from the first assignment. In the examples below, the output of `concord3` is also being compared with what is expected:

```
./concord3 in14.txt -e english-2.txt | diff out14.txt -
```

```
./concord3 -e latin-2.txt in19.txt | diff out19.txt -
```

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 `concord3`). 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 write our code such that we 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 the deallocation of memory 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 during labs.

- You must **not use program-scope or file-scope variables**.

- 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 `a3/` subdirectory. (For testing please use the files provided for assignment #2, with the exception of test 18 which will not be used for assignment evaluation.) All 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/a3` 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
  - write, test, and debug routines for dynamically-sized arrays.

- 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 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 (`concord3.c`, `emalloc.c`, `emalloc.h`, `seng265-list.c`, `seng265-list.h`, , `makefile`). **You are not permitted to add source-code files to your submission without first obtaining express written permission from the course instructor.**

- Any additional source-code files that you introduce for your solution. You must take responsibility for ensuring such files are in your git project. If these files are missing, then the teaching team will be unable to build and test your solution for A#3.

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 `concord3`.

- "B" grade: A submission completing the requirements of the assignment. `concord3` 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. `concord3` runs with some problems.

- "D" grade: A serious attempt at completing requirements for the assignment. `concord3` 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.