# Software Engineering 265
# Software Development Methods
# Fall 2022

*Assignment 3*

Due: Monday, November 21st, 11:55 pm by "git push"
(Late submissions **not** 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 a lab workstation. (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 the course instructor as soon as possible. (Code-similarity analysis tools will be used to examine submitted work.) The URLs of significant code fragments you have found online and used in your solution must be cited in comments just before where such code has been used.

## 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 #2.
- Use `valgrind` to determine how effective your solution is in its management of dynamic memory.

**Using C's heap memory for creating the concordance: `concord3.c`**

In assignment 2 you used Python to implement an extended version of the concordance-generation 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 some combination of 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 example below, the output of `concord3` is also being compared with what is expected for test 14:

```
cat in14.txt | ./concord3 | diff - out14.txt
```

where the assumption here is that `in14.txt` and `out14.txt` are available in the same directory as the compiled-and-built executable `concord3`.

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** for this assignment.

- 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.

**Note that if an assignment #1 test file is provided, your script must produce a single line of output, as shown below (i.e. please use this wording exactly):**

```
% cat some-kind-of-a1-tests-directory/<sometest>.txt | ./concord3

Input is version 1, concord3 expected version 2
```

## Exercises for this assignment

1. Within your `git` repo ensure there is an `a3/` subdirectory. (For testing please use the test-case files provided for assignment #2.) 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.

2. 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* OR write, test, and debug routines for *dynamically-sized arrays* OR *some combination of both linked-lists and dynamically-sized arrays*.

3. 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".

4. For this assignment you can assume all test inputs will be well-formed (i.e., our teaching team will not test your submission for handling of input or for arguments containing errors, other than the trivial "version" error describe up above). 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.**

- **If you have been provided such permission mentioned above**, then any additional source-code files that you introduce for your solution must be in the repository. You yourself will be responsible 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.

Up to ten students may be asked to demonstrate their submitted work for A#2 to a member of the teaching team. Information about this will be communicated to selected students after the due date.