Software Engineering 265
Software Development Methods
Summer 2021

Assignment 1

Due: Monday, June 14, 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 test files and sample code for this assignment is available on the UVic Unix server in `/home/zastre/seng265/a1` and you must use `scp` in a manner similar to what happens in labs in order to copy these files into your `Senjhalla`.

Any programming done outside of `Senjhalla` might not work during evaluation.

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.)

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

- Understand a problem description, along with the role used by sample input and output for providing such a description.
- Use the C programming language to write the first implementation of a concordance-generation program named `concord1` (and do this without using dynamic memory).
- Use Unix commands such as `diff` to support your testing and coding.
- 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: `concord1.c`

In this assignment you will direct your learning of C by solving a problem involving a tool used by many research and scholars that is called a concordance. Specifically we will consider one kind of *concordance* known as *keyword-out-of-context*.

The Oxford English Dictionary defines a concordance is an "alphabetical arrangement of principal words contained in a book, with citations of the passages in which they occur". Someone using a concordance is therefore able to look up a word of interest to them for that book, and find all the locations (page numbers and line numbers) where the word is used.

The concordances your program will create will be for texts much, much smaller than a complete book! Instead the concordance will relate words to the lines in which they occur in some text file being processed.

As an example, consider the following small text file. Line numbers are given for your reference. (Among the test files for this first assignment, this corresponds to the contents of `in04.txt`.)

```
1 the fish a dog cat dog rabbit
2 the fish and cat
3 a rabbit or elephant
```

The concordance for this file is to be generated using the following command (where I assume all of the needed text files are contained in the same directory as the executable).

```
$ ./concord1 -e english.txt in04.txt
```

The program produces this output to the console:

```
CAT       the fish a dog cat dog rabbit (1)
CAT       the fish and cat  (2)
```

```
DOG       the fish a dog cat dog rabbit (1*)
ELEPHANT  a rabbit or elephant (3)
FISH      the fish a dog cat dog rabbit (1)
FISH      the fish and cat  (2)
RABBIT    the fish a dog cat dog rabbit (1)
RABBIT    a rabbit or elephant (3)
```

Notice that each line is made up of a keyword, followed by an input line within which that word appeared, and ends with a number reference. The keywords are printed in alphabetical order. The word `cat` (printed as `CAT` when shown as a keyword) appears on lines 1 and 2 of the input; the word `dog` appears in only one input line – line 1 – but does so more than once, hence the asterisk. And so on and so forth.

You will also notice that some words contained within the input file **do not** appear as keywords in the concordance. For example, `and`, `or`, and `the` are not really of interest to anyone using a concordance – that is, they are excluded. A file with `exclusion` words was given to `concord1` on the command line via the `-e` argument.

How must concord1 determine the keywords? It may do so in effect by (a) reading the contents of the input file for which an index is needed, then (b) determining the unique words in that file, and finally (c) reading the words in the exclusion file and removing those words from the set discovered in the previous step.

With respect to output, keywords must always appear at the start of the line. (If the keyword itself had been capitalized in the output sentence, then that style of concordance would have been called *keyword-in-context*). To aid with formatting, the longest keyword determines how input lines themselves are indented, *i.e.*, length of longest keyword + two spaces. The testfiles for this assignment can provide you with more examples of input and their expected output; see below for more details on the location of test files.

Some simplifying assumptions

In order to reduce the complexity of the program, I have provided the following simplifications and size limits.

1. All input consists of lower-case letters, and all words on a line are separated by a single space. There is no other punctuation such as commas, colons, quotations marks, periods, etc. That is, for the first assignment you need not worry about cases such as that keyword `computer` appearing as `computer` or `"Computer` or `computer-aided` or `computer?` or `computer'`, etc. etc.

2. All provided exclusion-word files will have one word per line, and lines will be in alphabetical order.

3. Each input file (i.e., those for which a concordance is to be created): (a) will have at most 100 lines; (b) has input lines at most 80 characters long (including spaces and newline character); (c) has words no longer than 20 characters; (d) has no more than 500 unique keywords, although a specific keyword may appear many times in the input file. Also: files with exception words will have no more than 100 lines (i.e., no more than 100 words).

I also have several more restrictions on coding which are actually helpful simplifications:

4. Your solution **must not** make use of the dynamic-memory functions `malloc`, `calloc`, `valloc`, `realloc`, etc. That is, all memory needed for your solution can be statically allocated (i.e., program-scope variables) given the size limits described above in item 3.

5. All code must appear in a single source-code file named `concord1.c`.

Testing your work

Test data is provided to you on the SENG file system (i.e., the files you are able to copy through the use of `scp` when using your Senjhalla) when logged into a machine in ELW B238 either directly or remotely). These are located in:

`unix.engr.uvic.ca:/home/zastre/seng265/a1`

Please note: When using `scp` to copy files, please construct the path name using what I have given immediately above. The markdown file `TESTING.md` explains what is required for each test (i.e., input file, exclusion-words file, file with expected output).

As you do the initial coding for your solution, you will most likely visually check the output produced by your code in order to compare with what is expected. However, at some point this will not be precise enough as extra spaces in output are impossible to see. More robust testing will therefore require you to use the Unix command `diff`.

Shown below is an execution of `concord1`. I'm assuming that all of the test files are in the same directory as `concord1`. Note the use of the pipe operator and the hyphen/dash as an argument to `diff`.

```
./concord1 in04.txt -e english.txt | diff out04.txt -
```

The ending dash as an argument to `diff` will compare `out04.txt` with the text stream piped into the `diff` command (denoted by the dash argument given to `diff`). If no output is produced by `diff`, then the output of your program identically

matches the file (i.e., the test-case passes). Note that the arguments you must pass to `concord1` for the different tests are shown in the `TESTS.md` file.

Again: Please use `diff` when testing your solution. When the teaching team evaluates your work, the output produced by your program must exactly match the expected test output in order for a test to pass.

A warning to anyone who is neglectfully creative: You are not permitted to invent options or arguments, nor is output from `concord1` to appear in a separate new file. Doing so may result in a failing grade for this assignment.

Exercises for this A#1

1. If you have not already done so, ensure your git project is checked out from the repository. Within your project ensure there is an `a1/` subdirectory. Ensure all directories and program files you create are placed under git control. (You need not add the test directory to git control unless you wish to do so.) Test files are available as described earlier in this document.

2. Write your program. Amongst other tasks you will need to:
   - obtain a filename argument from the command line;
   - read text input from files, line by line, and the text within those lines
   - 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).

3. **Do not use `malloc()`, `calloc()` or any of the dynamic memory functions.**

4. Keep all of your code in one file for this assignment. In later assignments we will use separable compilation available in C.

5. You are welcome to use program-scope (i.e., global) variables for keeping track of such things as the input lines, the keywords, and the exception words. However, there are no other variables for which it makes sense to make them program scope, and failure to observe this may result in a somewhat lower grade.

6. 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 7 and higher until 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.

7. 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). Later assignments might specify error-handling as part of their requirements.

8. Ensure you appropriately use `add`, `commit`, and `push` to submit your work to our course's git server. You will probably want to use other commands during development (such as `log`, `status`, and perhaps `pull` if you have work on different computers). Avoid any use of `branch` or `checkout` as this will cause problems for A#1.

What you must submit

- A single C source file named **`concord1.c`** within your git remote repository (in your repo's `a1/` subdirectory) containing a solution to Assignment #1.
- Note that the use our git server is the only acceptable method of submission. Ensure your submission is properly named (as above) and in the correct directory (as shown above). Neglecting these details may result in significant grade deduction.
- **No dynamic memory-allocation routines are to be used for Assignment #1.**

One last bit of advice

The purpose of this assignment is to present a text-processing problem around which to structure your learning of the C programming language. As a consequence, I will not be overly concerned about run-time or space efficiency, and so you need not try to implement clever algorithms. Use simple implementation approaches. For example, you can store the contents of an input file into a two-dimensional character array. As another example, once you've determined the keywords, you can repeatedly scan stored input lines for the presence of each keyword when generating final output. I will, however, request that any file input is done once per program execution (i.e., once for the input file, once for the exception-word file) – that is, do not re-read the file as this repeated I/O can result in a very slow-running program.

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. Program-scope (i.e., global) variables are kept to a minimum. `concord1` runs without any problems; that is, all tests pass and therefore no extraneous output is produced.

- "B" grade: A submission completing the requirements of the assignment. `concord1` runs without any problems; that is, all tests pass and therefore no extraneous output is produced. The program is clearly written.

- "C" grade: A submission completing most of the requirements of the assignment. `concord1` runs with some problems.

- "D" grade: A serious attempt at completing requirements for the assignment. `concord1` runs with quite a few problems; even through the program runs, it may be that no tests pass.

- "F" grade: Either no submission given, or submission represents very little work.