Software Engineering 265
Software Development Methods
Summer 2018

Assignment 1

Due: Tuesday, June 12, 11:55 pm by "git push"
(Late submissions **not** accepted)

Programming environment

For this assignment you must ensure your work executes correctly on the Linux machines in ELW B238. You are welcome to work on your own laptops and desktops; if you do this, give yourself a few days before the due date to iron out any bugs in your code when it executes on a UVic computer. You can use "git push" and "git pull" to move files back and forth between SENG filesystem and your computer. (Bugs in the kind of programming done this term tend to be platform specific. Something that works perfectly on your own machine may end up crashing on a machine in ELW B238, and the fault is very rarely that of the lab machine.)

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 work.)

Objectives of this assignment

- Understand a problem description, along with the role played by sample input and output for providing such a description.
- Use the C programming language to write the first implementation of a SENG 265 text formatter named "uvroff" (and do this without 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 ten provided test cases.

This assignment: "uvroff.c"

You are to write a C program that inputs lines of text files, reads formatting options contained within those text files, and then outputs the text where each line has the appropriate width, indenting, etc. as indicated by the formatting options. For example, here is one such file contained on the SENG file system at */home/zastre/seng265/assign1/tests/in04.txt*:

```
.LW 30
Properly formatting  a   file where
there
is
    a smattering of white space throughout
  really means eliminating that
extra
   white
      space
such that the result
    looks neat
                  and
                        very
            tidy.
```

This particular file contains one formatting option (".LW 30") which indicates that the following text must be formatted such that each line contains at most 30 characters. The formatting program also concatenates the lines of a paragraph in order to eliminate unnecessary white space. The resulting output (in */home/zastre/seng265/assign1/tests/out04.txt*) looks like this:

```
Properly formatting a file
where there is a smattering of
white space throughout really
means eliminating that extra
white space such that the
result looks neat and very
tidy.
```

With your completed *uvroff* program, the input would be transformed into the output via the following command:

```
% ./uvroff /home/zastre/seng265/assign1/tests/in04.txt > ./out04.txt
```

where the file "out04.txt" would be placed in your current directory. To compare the output produced by *uvroff* with what is expected, you can use the Unix "diff" command as shown below (assuming you're still in the same directory as when you executed the command above):

```
% diff /home/zastre/seng265/assign1/tests/out04.txt ./out04.txt
```

Note that "diff" is much (much!) more reliable than using only your eyes.

For this first assignment there are only four formatting commands:

- .LW *width*: Each line following the command will be formatted such that there is never more than *width* characters in each line. The original whitespace in the input text need not necessarily be preserved (i.e., single spaces are used to separate words in the output). If this command does not appear in the input file, then the input text is not transformed in the output.
- .LM *left*: Each line following the command will be indented *left* spaces from the left-hand margin. Note that this indentation must be included in the page width. If this command does not appear in the input file, then the value of *left* is 0 (zero).
- .FT [off | on]: This is used to turn formatting on and off. If the command appears with "off", then all text below the command up to the next ".FT" command is output without being formatted. If the command appears with "on", then all text below the command up to the next ".FT" command is output with as many words as will fill the given page width.
- .LS *linespacing*: If *linespacing* is not zero (0), then between each text line there will appear *linespacing* blank lines. A blank line in the input file is considered to be the same as a text line. If there is no .LS command in the input file, then *linespacing* is zero.

There is some default behavior expected. (Some details from the previous bullet points are repeated below.)

- If no ".LW" command appears, the default mode is ".FT off". In this case all ".LM" commands are ignored. If no ".LS" command appears, then there is no linespacing (i.e., linespacing is zero).
- If a ".LW" command appears, the default mode is ".FT on".
- For this first assignment, there will only ever be one ".LW", ".LM" and ".LS" command in an input file, and these will appear at the top of the file.
- For this first assignment, you can assume that all test files will have page widths that are much greater than left margins (e.g., there will never be a combination such as ".LW 30" and ".LM 40"). Also linespacing specified in test files will never be greater than 2.
- There is no limit to the number of ".FT" commands that can appear within an input file.
- No matter the line spacing, if no blank line exists at the end of the input file, then *there must not be a blank line at the end of the output*. In all other cases input blank lines are treated as regular lines with respect to line spacing.
- The name of the input file is specified as an argument to *uvroff*. There is no output file specified (i.e., output is to *stdout*). ***You must not hardcode filenames in your submitted code!***

Exercises for this assignment

1. Within your local Git repository ensure there a top-level directory named *a1*. (For example, if your Netlink ID is *trumpd*, then your local git repo is named *trumpd*, and you must confirm a directory exists at *trumpd/a1*.)

2. Write your program the *a1* directory described in step 1. Amongst other tasks you will need to:
   - read text input from a file, line by line
   - write output to the terminal
   - extract substrings from lines produced when reading a file
   - create and use arrays in a non-trivial array.
   - use the *-std=c99* flag when compiling to ensure your code is compliant with the 1999 C programming language standard.

3. Do not use *malloc()*, *calloc()* or any of the dynamic memory functions. For this assignment you can assume that the longest input line will have 132 characters, and no input file will have more than 500 lines.

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

5. Use the test files to guide your implementation effort. Start with simple cases (such as those given in this writeup). In general, lower-numbered tests are simpler than higher-numbered tests. Refrain from writing the program all at once, and budget time to anticipate for when "things go wrong".

6. For this assignment you can assume all test inputs will be well-formed (i.e., the teaching team will not test your submission for handling of errors in the input). Later assignments will specify error-handling as part of the assignment.

7. *Use git add and git commit appropriately. While you are not required to use git push during the development of your program, you **must** use git push in order to submit your assignment.*

What you must submit

- A single C source file named *uvroff.c* within your git repository (and in the *a1* directory) containing a solution to Assignment #1. Ensure your work is **committed** to your local repository **and pushed** to the remote **before the due date/time**. (You may keep extra files that you have used during development within the repository.)

- No dynamic memory-allocation routines are permitted for Assignment #1 (i.e., do not use anything in the *malloc()* family of functions).

Evaluation

Our grading scheme is relatively simple and is out of 10 points. We will award grades within the categories below.

- 10/10: *uvroff.c* runs without problems and compiles without warnings. All ten tests pass. The program is clearly written and uses functions appropriately (i.e., is well structured).
- 8/10: *uvroff.c* runs without any problems and compiles without warnings. All ten tests pass.
- 7/10: A submission completing most of the requirements of the assignment. *uvroff.c* runs with some problems; some tests do not pass.
- 5/10: A serious attempt at completing requirements for the assignment. *uvroff.c* runs with quite a few problems; most tests do not pass, although some tests do pass.
- 4/10 or lower: No submission given, submission represents very little work, or no tests pass.