Assignment 3 – Preprocessing Pipeline for the Web Crawler

Your team lead has enjoyed the effort you have put in the stop word removal tool, and decided to give you a bit less work for the next step. They mentioned that it is important to learn to work with a pipeline of Unix tools. So, they gave you some smaller tasks related to the preprocessing pipeline.

First, they realized that your code was dealing with too many issues beyond stop word removal, which was making your code less cohesive (cohesion is an important software design quality, and it is related to a piece of code solving one or very few problems). So they decided to have a first step in the pipeline related to cleaning up the text file. This step shall be done by a single program developed by you that will convert every uppercase alphabetic character to lower case and remove every punctuation sign.

Then, they asked you to refactor your stop word removal tool to just dealing with stop words, since it will now read from the output of the previous cleaning step. Your code will still deal with stop word removal but shall have no mentions to punctuation removal or case checking.

They also gave you an existing C program as the third step in the preprocessing pipeline. This step is called stemming, and it works by removing unnecessary parts of words related to plural or verb conjugation, keeping only the word stems. For example, words like *works*, and *worked* are all related to a single stem: *work*. It seems better to have a web search tool that removes unnecessary suffixes and only works with word stems, so the tool treats similar words as only one word carrying the whole meaning of the word group. You will do only minor changes to this code, just in the main function, so that it is able to work with the pipeline.

Finally, they showed you another script by the other team member that may recursively loop over directories with text files, run your pipeline and produce an output directory with subdirectories and similar file contents, except that those files are now totally clean (of punctuation, upper cases, stop words and stem suffixes) and ready for the next phase of word indexing.

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 one or two days before the due date to iron out any bugs in the C programs 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.)

How to deal with multiple assignments in the same git repository.

Typical question: "How can I work on assignment 3 using **`git`** and be sure that graders will correctly grade Assignment 2 since there will be new commits for Assignment 3?

Short answer: tag your commit from assignment 2 as release a2

Long answer:

1) Run those two commands: **`git tag`** and **`git log --pretty=oneline`**

For example, in the instructor's test repo, the results are:

```
roberto@jupiter:~/prog/local/rbittencourt/a0$ git tag
roberto@jupiter:~/prog/local/rbittencourt/a0$ git log --pretty=oneline
8b719553b141b56d0bf23f5230743840c229bf72 (HEAD, origin/master, origin/HEAD, master) test 3
d4f63b5075a179c92df861dd8f9eff9dddd8798c test 2
5bebe5683a9d4e3232f057fc35c585459e9eb21b test 1
577a664c963289da8f68cd88bfbbe071047601ba commit log has text in it now
8bf104d82d0b31930a347909d1fc21a22d07c1ed adding the commit_log.txt file
dec69904074ecd902617668d113ef751b5634d1c initial
```

The command **`git tag`** shows nothing because there are presently no release tags. But, from **`git log`**, look at the list of commits, there are 6 commits there, each one with its hash.

Suppose we want to tag the commit with the "test 2" commit message (`d4f63b5075a179c92df861dd8f9eff9dddd8798c`), that is, we want it to be marked as the release **`a2`**, even though there is already a newer commit after this one, with "test 3" commit message (`8b719553b141b56d0bf23f5230743840c229bf72`).

So we type the following command:

```
roberto@jupiter:~/prog/local/rbittencourt/a0$ git tag a2 d4f63b
```

When we do that, the local repo marks the `d4f63b5075a179c92df861dd8f9eff9dddd8798c` commit as the **`a2`** release version (notice we used the shortened hashcode **`d4f63b`** to make it easier).

If we now type **`git tag`**, we will see that the release version is marked in the local repo.

```
roberto@jupiter:~/prog/local/rbittencourt/a0$ git tag
a2
```

But WATCH OUT, the release version is not at the remote repo yet. And the graders will not see it while you have not sent it. You may think **`git push`** will solve that, but **`git push`** does not send tags to the remote repo by default, you need to be explicit to make it happen. So type **`git push origin a2`**:

```
roberto@jupiter:~/prog/local/rbittencourt/a0$ git push origin a2

--------------------------------------------------------------------------------
Software Engineering Program Computer Support Group                  sengsys@uvic.ca
rbittencourt@git.seng.uvic.ca's password:
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
To ssh://git.seng.uvic.ca/seng265/rbittencourt
 * [new tag]         a2 -> a2
roberto@jupiter:~/prog/local/rbittencourt/a0$
```

After this command, the release version **`a2`** will be at your remote repo, and will be available to the graders. To grade Assignment 2, they will first clone your repo and then they will type **`git checkout a2`**. Doing this, they will be grading the commit you marked as release **`a2`**, and will work with the correct code version for Assignment 2, and not with later commits you will be doing for Assignment 3 or later assignments.

ATTENTION: if you do not tag the **`a2`** release, the graders will grade from your latest commit. So, in order to keep working with git, add your release tag as soon as possible, and then you may continue working with Assignment 3.

Finally, when you finish Assignment 3, tag its final commit with a release name **`a3`** and send it to the remote repo, so then you will be able to work comfortably with Assignment 4. Follow this simple process for all the assignments in this course.

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 (Roberto).** 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.

Description of the task

You will develop three programs in C that read text from the standard input and produce text sent to the standard output. The idea is simple. Suppose you **`cat`** from a text file and pipe the results into your pipeline and then redirect the pipeline output to another file. Doing this provides general preprocessing functionality and different parts of your code may be reused in other scenarios. Figure 1 indicates the data flow in your preprocessing pipeline.

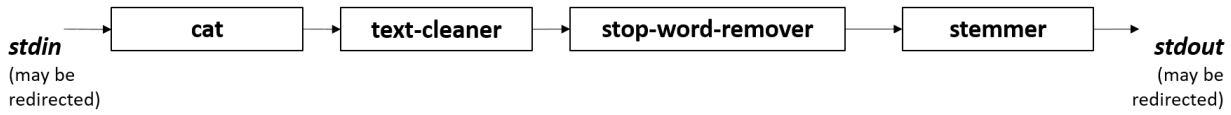

Figure 1: Preprocessing pipeline of the web search tool.

Your programs will be run from the Unix command line. Input is expected from **`stdin`**, and output is expected at **`stdout`**. **You must not provide filenames to the program, nor hardcode input and output file names**.

Cleaning text files from punctuation and dealing with cases

The program you will develop in this step is quite straightforward. It reads character by character from the standard input and sends a cleaner version to the standard output, doing the following actions:

- When the character is a punctuation sign, skip it, i.e., do not reproduce it in the standard output;
- When the character is an uppercase letter, send to standard output the lowercase version of that letter;
- Otherwise, i.e., when the character is a regular character (letters, numbers, spaces, tabs or newlines), just send it unchanged to the standard output.

Assuming your current directory contains your executable version of **`text-cleaner.c`**, (i.e., named **`text-cleaner`**), and a **`tests1/`** directory containing the assignment's test files related to text cleaning is also in the current directory, then the command to remove punctuation signs and change all the words to lower case will be.

```
% cat tests1/in01.txt | ./text-cleaner
```

In the command above, output will appear on the console. You may want to capture the output to a temporary file, and then compare it with the expected output. The **`diff`** command allows comparing two files and showing the differences between them. Use **`man diff`** to learn more about this command.

```
% cat tests1/in01.txt | ./text-cleaner > temp.txt
% diff tests1/out01.txt temp.txt
```

The same thing (i.e., producing output and comparing it with the expected output) can be combined into a one-liner:

```
% cat tests1/in01.txt | ./text-cleaner | diff tests1/out01.txt -
```

The ending `hyphen/dash` informs **`diff`** that it must compare the contents of **`tests/out07.txt`** with the input piped into **`diff`**'s **`stdin`**.

When you are confident your text cleaning code is working correctly, commit it and push it to the remote repository.

Refactoring the stop word removal program

Your stop word removal code was doing too much. Remove any code that is related to either the treatment of punctuation or of upper and lower cases. That will make your code more cohesive. To do so, be careful not to break your previous code. Your changes are simple, but your new code must still be correct, implementing the stop word removal feature and passing all the tests.

Assuming your current directory contains your executable version of **`stop-word-remover.c`**, (i.e., named **`stop-word-remover`**), and a **`tests2/`** directory containing the assignment's test files related to stop word removal is also in the current directory, then the command to remove stop words will be:

```
% cat tests2/in01.txt | ./stop-word-remover
```

In the command above, output will appear on the console. You may want to capture the output to a temporary file, and then compare it with the expected output.

```
% cat tests2/in01.txt | ./stop-word-remover > temp.txt
% diff tests2/out01.txt temp.txt
```

The same thing (i.e., producing output and comparing it with the expected output) can be combined into a one-liner:

```
% cat tests2/in01.txt | ./stop-word-remover | diff tests2/out01.txt -
```

When you are confident your refactored stop word removal code is working correctly, commit it and push it to the remote repository. Commit that separately from the previous step of text cleaning.

Adapting the stemmer program to the pipeline

The program you will develop in this step is quite straightforward. You will use the **`stemmer.c`** (which came from **`https://tartarus.org/martin/PorterStemmer/c.txt`**) as a basis for your stemmer. The problem with this code is that its **`main()`** function is reading multiple files, instead of reading from the standard input. It is also not checking the number of parameters correctly (it should receive no parameters from the command line). You shall read the main function and fix it to read instead from the standard input and to receive no parameters. Reusing code from other developers is an important skill for software developers, and minor changes (sometimes called glue code) may be needed to adapt the external code to your requirements.

Assuming your current directory contains your executable version of **`stemmer.c`**, (i.e., named **`stemmer`**), and a **`tests3/`** directory containing the assignment's test files related to stemming is also in the current directory, then the command to transform your input into the required output will be:

```
% cat tests3/in01.txt | ./stemmer
```

In the command above, output will appear on the console. You may want to capture the output to a temporary file, and then compare it with the expected output.

```
% cat tests3/in01.txt | ./stemmer > temp.txt
% diff tests3/out01.txt temp.txt
```

The same thing (i.e., producing output and comparing it with the expected output) can be combined into a one-liner:

```
% cat tests3/in01.txt | ./stemmer | diff tests3/out01.txt -
```

When you are confident your stemming code is working correctly, commit it and push it to the remote repository. Commit that separately from the previous steps of text cleaning and stop word removal.

Testing the full pipeline

Now it is time to test the full pipeline. Your team wrote a shell script that exercises the full pipeline. It is located in **`/home/rbittencourt/seng265/a3/`** and is named **`preprocessor.sh`**. Take a look at this script to review a bit of what you learned about Unix shell scripts. Notice that it sweeps over a directory and all its subdirectories, applying the full pipeline to each of the existing files, and produces as a result a new subdirectory with preprocessed files.

Assuming your current directory contains a shell script named **`preprocessor.sh`**. Also assume that you have a **`files/`** directory containing some text files that were crawled by the crawling tool and cleaned from their HTML markup code by a converter (a tool such as html2text or the lynx –dump command, which convert an HTML file into a similar text file with no HTML tags). Assume as well that you want to dump the result of your pipeline in the new **`newfiles/`** directory, which will be created in your current directory. Then the command to fully step the files through your whole pipeline is:

```
% ./preprocessor.sh files newfiles
```

Test your code with the pipeline to have a better feeling how your tool will work in practice by the preprocessing team at your company.

Exercises for this assignment

You may develop your code the way that suits you best. Our suggestions here are more for facilitating your learning than as a requirement for your work. You may not need to do the exercises in the items 1, 2 and 3 below if you want to practice deeper problem solving skills. But, in case you get stuck, you may look at them as a reference. On the other hand, if C programming seems difficult to you, you may use them as a script to learn and practice.

1. Write your program **`text-cleaner.c`** program in the **`a1/`** directory within your git working copy (Recall that you are continuing your work from Assignment 2 here in Assignment 3, and the software repository must be the same).
    a. Practice with **`stdin`** by reading it character by character with a loop and **`fgetc()`** and printing the output with **`printf()`** or **`fputc()`**.
    b. Play with standard input redirection, by redirecting input to read from a file instead of reading from the keyboard.
    c. Learn how to use **`ispunct()`**, including this function with the **`ctype.h`** header file to be able to use the **`ctype`** library to deal with punctuation. Do the same with the **`islower()`** and **`tolower()`** functions to help you deal with letter cases.
    d. Play with the standard output, using **`printf()`** or **`fputc()`** to print on your terminal screen the output as it is required.
    e. Play with standard output redirection, by redirecting **`stdout`** to save data in a file instead of sending them to the console.
    f. Test your program using pipes and the **`diff`** tool with the first test file as explained in the section on text cleaning.
    g. Have you thought about modularizing and commenting your code during the development? If not, now it would be a good time to separate parts of your code into different functions and document your code as well, in case you want an "A" grade.
2. Rewrite your program **`stop-word-remover.c`** program in the **`a1/`** directory within your git working copy.
    a. Remove any code that is related to treatment of punctuation. To do so, be careful not to break your previous code. Your changes are simple, but your new code must still be correct, implementing the stop word removal feature and passing all the tests.
    b. Remove any code that is related to the treatment of upper and lower cases. To do so, be careful not to break your previous code. Your changes are simple, but your new code must still be correct, implementing the stop word removal feature and passing all the tests.
    c. Test your program using pipes and the **`diff`** tool with the first test file as explained in the section on stop word removal.
    d. Have you thought about modularizing and commenting your code during the development? If not, now it would be a good time to separate parts of your code into different functions and document your code as well, in case you want an "A" grade.
3. Rewrite your program **`stemmer.c`** program in the **`a1/`** directory within your git working copy.
    a. Save the **`stemmer.c`** file provided in **`/home/rbittencourt/seng265/a3/`** in your **`a1/`** directory in your working copy of your project.
    b. Briefly browse the **`stemmer.c`** file just to have an idea how it is organized.
    c. The problem with this code is that its **`main()`** function is reading multiple files. Read the main function thoroughly, and try to think of a strategy to change it to read from the standard input instead of reading from multiple files. Do not remove the global variable in the **`main()`** function that is used by other functions in this file.
    d. If you have struggled to find an strategy, recall that you can replace the file pointer variable in the **`main()`** function by the **`stdin`** variable, and that you need no loops since you do not have to deal with multiple files. You may as well recall that you do neither **`fopen`** nor **`fclose`** the **`stdin`**, since C gives it ready to you.
    e. Fix your error message for incorrect number of parameters the same way you did for the other programs.
    f. Test your program using pipes and the **`diff`** tool with the first test file as explained in the section on stemming.
4. Use the **`-Wall -std=c11`** flags when compiling to ensure your code meets the ISO/IEC 9899:2011 standard for the C language (you do not need to do so for the **`stemmer.c`** code, since it was developed for a different C standard, even though it was still ANSI C).
5. Commit your code frequently (**`git add`** and **`git commit`**), so you do not lose your work.

6. When you are done with your commits, do not forget to **`git push`** them into your repo. We will look at the code commits you did in this assignment. We require at least three different commits, one for each step of the pipeline. Our final grading will take that into account.
7. Use the test files in the lab-workstation filesystem located in **`/home/rbittencourt/seng265/a3/`** (i.e., the **`tests1/`**, **`tests2/`** and **`tests3/`** subdirectories of **`a3/`**, all inside lab-workstation filesystem) to guide your implementation effort. Start with simple cases (for example, the ones described in this write-up). 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". There are two pairs of test files in each directory.

What you must submit

- You must submit 3 (three) single C source files named **`text-cleaner.c`**, **`stop-word-remover.c`** , and **`stemmer.c`**, within your **`git`** repository (and within its **`a1/`** subdirectory) containing a solution to this assignment (We are using the **`a1/`** subdirectory both for assignments #2 and #3. So do not start from scratch, start from the last commit you did for the previous assignment). Ensure your work is committed to your local repository and pushed to the remote before the due date/time. (You may keep extra files used during development within the repository, there is no problem doing that. But notice that the graders will only analyze your **`text-cleaner.c`**, **`stop-word-remover.c`** , and **`stemmer.c`** files.)
- No dynamic memory-allocation routines are permitted for this assignment (i.e., do not use anything in the **`malloc`** family of functions).
- You are permitted to declare arrays having a program scope, although you are encouraged to keep this to as small a number as is practicable given this may be your initial steps at programming in C.

Evaluation

Our grading scheme is relatively simple and is out of 100 points. Assignment 3 grading rubric is split into seven parts.

1) Modularization - 10 points - the code should have appropriate modularization, dividing the larger task into simpler tasks (and subtasks, if needed);
2) Documentation - 10 points - code comments (enough comments explaining the hardest parts (loops, for instance), no need to comment each line), function comments (explain function purpose, parameters and return value if existent), adequate indentation;
3) Compiling - 5 points - Code compiling with no warnings when using **`gcc -Wall -std=c11 ...`**, except for the **`stemmer.c`** code;
4) Version control - 15 points – Appropriate committing practices will be evaluated, with at least one separate commit for each of the three programs, as well as continuing commits from assignment 2;
5) Tests: Part 1 - 20 points - passing tests in **`tests1/`** directory: removing punctuation and converting uppercase letters to lowercase;
6) Tests: Part 2 - 20 points - passing tests in **`tests2/`** directory: general tokenizing and removing stop words;
7) Tests: Part 3 - 20 points - passing tests in **`tests3/`** directory: general stemming.

We will only assess your final submission sent up to the due date (previous submissions will be ignored). On the other hand, late submissions after the due date will not be assessed.