COMP 348: ASSIGNMENT 1

Note that assignments must be submitted on time in order to receive full value. Appropriate late penalties (< 1 hour = 10%, < 24 hours = 25%) will be applied, as appropriate.

THE
C
PROGRAMMING
LANGUAGE

DESCRIPTION: In this assignment, you will gain hands-on experience with the C programming language. While you are not required to be a C expert to complete the work, you will certainly have the opportunity to explore most of the things that we have discussed in class (and a few other things as well).

In terms of your task, it is fairly easy to describe. You will be writing a simple search and replace application. Specifically, your job is to (1) find a user-specified text string within a group of disk files, (2) modify those strings and update the original disk files, and then (3) provide a report that indicates what has been done. It will work as follows:

1. The application will take a single command argument. This argument will be interpreted as a text string. For example, if your program was called *replace.exe*, you might enter *replace.exe apple*

2. You will then open each of the files in the current directory. If you haven't done much programming with files and directories, it isn't especially difficult. That said, the facilities for doing so in C are somewhat more primitive than those in languages such as Java. First, you will want to make use of the functions associated with <stdio.h>. As we discussed in class, these are the basic IO functions, and can be applied not only to the screen and keyboard, but to disk files as well (There are countless online examples of basic C-based I/O). Here, you will use functions for opening and closing files, reading and writing bytes/chars to and from files, and for checking to see to see if you have reached the end of the file.

3. In addition to opening and reading files, you must also be able to read the contents of the directory in order to see what files need to be checked (the current directory = "."). Basically, a directory is just a special type of file that contains a list of the files within it. So the idea is to just loop through the file list, and open and examine each text file (for simplicity, all files used during testing will be text files with a ".txt" suffix, so you don't have to worry about other formats).

In terms of the actual mechanisms for traversing directories on Linux/Unix, it is again not very hard. Essentially, the standard technique is to open the directory and use a WHILE loop to examine the contents (i.e., the individual files). With Linux/Unix systems, the directory functions (e.g., opening and reading directories) are associated with the header file <dirent.h>. You may also want to include <unistd.h> and <sys/types.h>.

4. For each text file found in the directory, you will open each and check for the string that was entered at the command line. The target string may occur multiple times, of course. When you find it, you will modify the text so that the target string is printed in upper case. So, for example, if *apple* is the target, the string "… the Big Apple is fun…" would become "…the Big APPLE is fun…". Please note that while this kind of update can be done with complex regular expression libraries, you will not do that here. Instead, you will just be flipping the lowercase letters to uppercase as required. String related functions are found in <string.h> while character functions are in <ctype.h>.

5. To make things just a little more interesting, directories can be nested. So if the current directory has sub directories, you must search those as well. Note that when you read the directory contents in the main WHILE loop, you can check each entry to determine if it is a regular file or another directory (there are constants defined in <dirent.h> for this purpose.). Again, there are simple examples of this online.

6. Finally, you must keep track of the files that were changed and the number of changes made in each one. At the end of the process (when no more directories can be found from your starting point), you must print out a report. The report will simply list each changed file, along with the count of the changes in each. The files should be ordered/sorted on the number of changes, with the most heavily modified listed first. Note that you do not have to write your own sort algorithm. Instead, <stdlib.h> provides a sorting function called qsort, that can be used to sort arbitrary items. To use qsort, you must simply provide a comparison function that qsort can use (Java uses a similar logic for sorting objects).

So you might have a final report like the one listed on the following page:

```
Target string:  apple

Search begins in current folder: /home/me/testDir


** Search Report **

Updates         File Name
4               dirY\apple.txt
3               bob.txt
2               dirY\sturp.txt
```

Note that the output includes the starting folder. This can be obtained with the `getcwd()` function in <unistd.h>. Moreover, the “File Name” includes the sub-folder element.

EVALUATION: Please note that the marker will be using a standard Linux system – either Window’s WSL/Ubuntu or Docker’s Debian. Your code MUST compile and run from the Linux command line.

To evaluate the submissions, the marker will simply create a small directory structure with a handful of files, some of which will have the target string inside. Every student will be tested on the same folder structure.

The marker will also ensure that you have used good programming practices (e.g., constants, headers, basic error handling). Moreover, there should be NO memory leaks in your program. If you dynamically create any data/variables, you MUST clean them up before the program ends.

DELIVERABLES: Your submission should have multiple source files. To begin it will have a file called `replace.c` that will contain the `main()` function. This file may also contain functions that process the command line argument.

Your submission should include a source file called `traversal.c` that contains the directory traversal logic. A third file called `text.c` will contain any logic associated with the search and replace processing performed on each file. Finally, you will have a file called `report.c` that contains any code related to preparing and printing the final report.

Note that these files will each be relatively small, but this format makes it much easier for the grader to see what is going on.

IMPORTANT: All files (.c and .h) must be prepared in the same folder so that they can be compiled from the command line using the following simple gcc command

```
gcc -Wall traversal.c text.c report.c replace.c
```

Before submitting, you must test your code with this command to make sure that it works (this particular command will produce a compiled executable called `a.out`).

Once you are ready to submit, compress all .c/.h files (and ONLY these files) into a zip file. The name of the zip file will consist of "a1" + last name + first name + student ID + ".zip", using the underscore character "_" as the separator. For example, if your name is John Smith and your ID is "123456", then your zip file would be combined into a file called a1_Smith_John_123456.zip". The final zip file will be submitted through the course web site on Moodle. You simply upload the file using the link on the assignment web page.

Please note that it is your responsibility to check that the proper files have been uploaded. No additional or updated files will be accepted after the deadlines. You can not say that you accidentally submitted an "early version" to Moodle. You are graded only on what you upload.

Good Luck