CSE 560 **Winter 2010, 10:30 a.m. Section**

LAB #1: Warm-Up Assignment

DUE: 10:30 a.m., Friday, January 15

An early hurdle in these laboratory assignments is handling the details of file processing. This warm-up exercise is designed to help your team successfully jump this hurdle. This exercise is worth two percent of your course score.

Let us consider a *byte* to be a nonnegative integer value b where $0 \leq b < 2^8$; hence, b is a byte if and only if b can be represented by 8 bits using the simple binary encoding scheme. Any file can be considered to be a (possibly empty) sequence of bytes.

A file can also be considered to be a sequence of "lines". Each line is a sequence of bytes, where such a sequence could be the empty sequence. The first line of the file (if such a line, by this definition, exists) is the sequence of bytes beginning with the first byte in the file (if it exists), and continuing up to, but not including, the first *next-line marker* in the file. (If the file begins with a next-line marker, then the first line of the file exists and is the empty sequence.) (The sequence of bytes that represents a *next-line marker* depends on the operating system involved. In Unix systems, it is just one byte, 10. In DOS/Windows systems, it is a sequence of two bytes, 13 followed by 10. In Macintosh systems, it is just one byte, 13. You may create your program so that it processes these files according to the definition of next-line marker used by just one operating system; simply specify in your documentation for which operating system this program is intended, and the corresponding definition of next-line marker.) Similarly, the second line of the file (if a second line exists) is the sequence of bytes beginning with the first byte after the file's first next-line marker (if the first byte after the file's first next-line marker exists), and continuing up to, but not including, the second next-line marker in the file. (If there are no bytes between the first and second next-line markers, then the second line of the file exists and is the empty sequence.) The description would be similar for the third, fourth, fifth, and so on, lines of the file, if they exist. The last line of a file always exists, by the definition that follows. If the file contains no next-line markers, the last line is the entire file. In this case, the file's last line is also its first line. Hence, the first line of a file always exists, also. If the file contains at least one next-line marker, the last line is the sequence of characters that follows the last next-line marker. A file containing n next-line markers is separated by those markers into a sequence of $n + 1$ lines.

In accordance with the ASCII scheme for encoding characters, we consider a byte b to be "printable" if and only if $b = 9$ (representing the horizontal tab character) or $32 \leq b \leq 126$.

Your program should process two input files, and produce two output files. One input file is the primary input file, the other is the auxiliary input file. One output file is the result output file, the other is the report output file. A correct auxiliary input file consists of either one or two lines. If it contains two lines, the last line is empty, the first is not. Each byte b of the first line represents, according to the ASCII character encoding, a decimal digit (i.e., $48 \leq b \leq 57$). The first byte of the first line is not 48 (i.e., not '0'). Your program is to consider the sequence of digits in this first

line to represent a positive integer according to the usual decimal representation scheme. Let's call this integer *num_lines*.

Your program should process the primary input file. It should produce one line of output (to the result output file) for each line of the primary input file, but no more than *num_lines* lines should be written to the result output file. A line of the result output file should contain, in order, only printable bytes of the corresponding input line. If the input line contains 80 or fewer printable bytes, the output line should contain all and only those bytes. If the input line contains more than 80 printable bytes, the output line should contain all of the first 80 printable bytes, and only those, from that input line.

Your program should report, in the report output file, how many lines were produced in the result output file, and whether all lines of the primary input file have corresponding lines in the result output file. If the report output file cannot be properly identified or if it cannot be written to the file system, then an error message stating the problem should be sent to the stderr device. Otherwise, any and all error messages produced by your program should be reported in the report output file. You may also choose to provide error messages to the stderr device. You should strive to make your program detect and handle all possible errors. The report output file should also list the line numbers of all lines (among the first *num_lines* lines) in the primary input file that contain more than 80 printable bytes.

Your program should properly handle input files (both primary and auxiliary) of any size, from the empty file to very large graphic image files. What are your program's internal memory requirements if there are many millions of bytes between next-line markers? (The internal memory requirement should be quite small.)

Your program should check for file existence and permissions. It should properly handle input files that turn out to be directories and not regular files, by providing an appropriate error message; it should also properly handle input files that are not readable. Your program and its documentation should be clear about what happens when an output file already exists, and, if it does already exist, if it either turns out to be a directory and not a regular file or it is not writeable. It should properly handle an attempt to create a new file in a directory that is not writeable.

To make life a lot easier on some of your users (in particular, your graders), your program must allow a command-line interface. Furthermore, it must allow the four file path names to be specified as arguments on the command line. On a command line, a user can use the "file name completion" feature of the shell (usually the Tab key invokes this feature). This helps the user know whether the file name is spelled correctly (informing the user whether the file exists), and the user can then often accomplish the task with much less typing. Of course, your program should provide an informative error message if it is invoked with the wrong number of command line arguments. Such a message often includes or consists of instructions for proper use of the program, sometimes preceded by the terse noun phrase "Usage:". Your program must interpret the following command-line format, in the order specified, as being correct:

```
program-name primary-input auxiliary-input result-output report-output
```

You are to submit an on-line writeup that contains:

- a table of contents, in HTML

- a user's guide which, in addition to a brief description of what the program is all about, tells how the program is envisioned to be run, including any operating system and/or terminal shell commands and files needed to access the program, the input and output requirements and conventions, and error conditions and their associated error messages
- a programmer's guide describing the overall design of your solution
- a description of the testing you did attempting to reveal defects in your program, including the test data inputs, their associated expected outputs, and some rationale for your choice of tests
- a summary of each of your group's meetings, including decisions made by the group and responsibilities assigned to individual group members

You are also to supply a peer review "grade" for each of the other members in your group. Instructions for supplying this peer review are available on Carmen. This review is due by 5:00 P.M. on the day the lab is due.

In your peer evaluation, rate each person in your group (including yourself) on a scale of 0 to 10. If everyone pitched in reasonably, then everyone should get an 8 (reserve a 10 for truly exceptional work). **The total of the individual ratings must not exceed $8n$, where n is the number of people in your group**. Include comments to justify and clarify the ratings you have assigned.

You should produce "professional-looking" on-line documentation: documentation that you would be proud to show your boss. At least the top level, the table of contents, should be in HTML, viewable by a web browser. (This does **not** mean that **any** on your documentation should be available publicly on the World Wide Web. No, your source code and documentation should remain a trade secret, available only to your team, and, eventually, to your graders and your instructor.) Use a good word processor (such as Microsoft FrontPage, HTML in emacs, Framemaker, LaTeX or Microsoft Word) to facilitate document preparation. You may use whatever other tools you wish to facilitate diagramming and communication among the team.

Your documents should be well written, logically structured, and pleasingly formatted. Hint: leave time for other members of your group to proof read what you write, and vice versa since your grade hinges partially on the rest of your group. This applies to both native and non-native English speakers. Give each other advice on their writing style. Leave time to "smooth over" the writing: the documentation should flow as if all of it was written by the same person, even though not all of the documentation is tailored to the same audience.

You are to work in teams of 4 if at all possible (otherwise 5 if absolutely necessary), with partners of your own choosing.

Divide up the workload conveniently to try to avoid overloading any group member. **Schedule meetings** to discuss problems and progress, and to ensure that each team member does his/her own fair share. It is important that all members of the group have access to the up-to-date status of the design of your program. Keep records ("minutes") of the proceedings of these meetings, since they will be turned in as part of your lab writeup. Use these meetings, email, and your group accounts to facilitate this. Talk about aspects of your solutions among one another, so that each of you understands them. Having someone else look at your work may help uncover flaws in your logic.

Do not make any changes to the lab requirements without authorization from me! Also avoid the temptation to make changes to the decisions of the group without consulting the group for agreement.

Let me know as soon as possible if the team seems to be having trouble (e.g., if one member drops the course or isn't coming to meetings, or if the group isn't scheduling meetings frequently enough).

Breakdown for the lab 1 grade

Group part:

User's guide: 20%
Programmer's guide: 20%
Coding: 10%
Your testing: 10%
Our testing: 20%
Meeting minutes: 10%

Individual part:

Peer review: 10%

The documents will be graded based on both their technical quality and their English.