CSCI 493.66 Unix Tools

with emphasis on string algorithms and Bioinformatics

Homework 1
Due 02/12/09

Saad Mneimneh
Computer Science
Hunter College of CUNY

Problem 1: There is more to more
In this problem, you are asked to write the more program we did in class with 3 additional requirements:

- if the user inputs 'n', the program skips to the next file. If the current file is the last file, 'n' should behave like 'q'.
- if the user inputs 'b', the program goes backward in the current file by an amount equal to the last forward action (either a page, or one line). This should be done only once, like an undo action. In other words, additional 'b's should not have any effect. For this part, learn about the *fseek* function in C, or its equivalent in the C++ stream library, or the *seek* function in Perl.
- the prompt, in addition to displaying 'more?', should also show the percentage of bytes of the current file displayed so far. Use a combination of the *fseek* and *ftell* functions (or what is equivalent in other languages) to solve this problem.

Note 1: make sure to choose a page length equal to the number of rows in your terminal window (24 is the default) minus 1 for the prompt. Also, you might want to set the size of your line buffer to the width of the terminal window. The book uses the following:

```
#define LINELEN 512

. . .

char line[LINELEN]
```

If the line read is longer than the width of the window, this effectively means that more than one line is going to be displayed in the terminal window. To avoid this, set LINELEN to the width of the terminal window. In either case, reading a line may stop before reaching the '\n' character. Depending on whether you are using C or C++, this may set some error flags which you have to clear. Read the online C++ documentation for more information. Also, C and C++ behave differently in terms of keeping/throwing away the '\n' character if encountered.

Note 2: Upon receiving an action from the user, it is possible to clear the screen and display the appropriate page. This will take care of removing the extra

"more?" prompts that would otherwise stay if lines are read one at a time. While acceptable, this is not the preferred method for this exercice. Here's a possible algorithm that you can use (and don't worry about the "more?" prompt):
Think of the forward actions (' ' and '\n') as a sequence of pages and lines respectively, e.g.:

$$p\ l\ l\ l\ p\ p\ l\ p\ l\ l\ l\ l\ l\ p\ l\ p\ l\ \ldots$$

To undo the current action, go back to the seek position corresponding to the last p before the current action. Then read and ignore a number of lines equal to the number of l's between the last p and the current action. Display a page, i.e. make a p action. Here are two examples:

Example 1:

$$p\ l\ l\ l\ p$$

To undo, go back to the seek position of the last p action before the current action, read and ignore 3 lines, make a p action.

$$l\ l\ l\ p$$

Example 2:

$$p\ l\ l\ l\ p\ p\ l\ p\ l\ l\ l\ l\ l$$

To undo, go back to the seek position of the last p action before the current action, read and ignore 4 lines, make a p action.

$$p\ l\ l\ l\ p\ p\ l\ l\ l\ l\ p$$

Note 3: The book uses the getc function to read a character from the tty file. Since the only way to send characters to tty (for now) is to press the return key, any character sent to tty will cause an additional '\n' character to be sent. This additional '\n' character will be read next by the getc function. Effectively, a space will actually correspond to a page followed by an extra line. This means that l is always the last action to be undone. To solve this problem use fgets instead, which retrieves a string (including the '\n' character). Then check the first character of that string.

Problem 2: Sampler
Write a program that accepts its input either from a file for which the name is provided at the prompt, or the standard input (in case nothing is provided at the prompt, a pipe, or a redirection). Note that if nothing is provided at the prompt, standard input is effectively the keyboard.

The program should display a random sample of the input in reverse mode; for instance, only 50 words chosen uniformly at random (instead of 50, this can also be a parameter provided at the prompt if you like). If less than 50 words are seen, all of them are chosen. In case of user input (keyboard), those 50 words must be continuously updated (after each '\n' character) until the user hits the end of file Ctrl D character.

This is not an easy problem, because it requires that you obtain 50 words uniformly at random from a stream for which you don't know the end. We will discuss some possible ways of doing this in class.

Moreover, there might be an added diffculty for obtaining the words, because *fgets* reads lines. In C++, there is a way to change the delimiter from the default '\n'. In Perl, it is easy to split a line into the words. In C, you might want to use the function *strchr* in string.h which can locate the first occurrence of a given character in a string. This can be used to find the spaces.

Note: Since redirections and pipes are handled by the standard input stream, there is no obvious way of distinguishing between an actual redirection/pipe and the keyboard. Observe that all three correspond to a value of 1 for argc in the main function. Since we require the sample to be updated after each '\n' character only when the user enters the text through the keyboard, the above mentioned distinction must be made. Look online for the isatty function and study how you can use it to determine whether input is actually coming from the keyboard or not. A similar function is also available in Perl.