CSE 1320 – Homework #8

Assigned: Thursday, March 20, 2014
Due: Tuesday, April 1, 2014 at 10:00am

You must submit your homework through Blackboard (http://www.uta.edu/blackboard/).

Upload the source code in its original format with the extension .c. If there are multiple programs, then each should be in it's own file (just as you have saved it). The source code files should be named using your last name and the assignment number, for example, brezeale-hw01.c. When submitting your source code, do the following:

- At the top of the program, as a comment include at a minimum your name, date, and homework number.

Purpose of this assignment:

1. Gain more experience with dynamic memory allocation.

Assignment: Write a program that performs some of the functionality of the Linux programs `head`, `tail`, and `sort`.

Additional requirements specific to this assignment:

- Your program should accept command-line parameters. It should expect to receive the following, in this order:

  1. one of the following letters: `h`, `t`, `b`, or `s`.
  2. a positive integer, `n`
  3. the name of a file to process

  Example:

  ```
  ./brezeale-hw08 h 5 test.txt
  ```

- The letters indicate the following:

  1. `h`: similar to `head`, this prints the first `n` lines of the file. If `n` exceeds the number of lines in the file, just print the file.
  2. `t`: similar to `tail`, this prints the last `n` lines of the file. If `n` exceeds the number of lines in the file, just print the file.
  3. `b`: this prints both the first `n` and last `n` lines of the file. If `n` is greater than or equal to the number of lines in the file, just print the file.
  4. `s`: prints the file in ascending order as determined by ASCII value. Technically, the integer `n` passed on the command-line serves no purpose here other than to keep the input consistent.

- Don't use structures.
- Read the file exactly once, storing it in a dynamically allocated array of strings. That is, each line of the file will be stored as a null-terminated array of characters (i.e., a string) and you will also have an array to store the addresses of of these strings. The memory allocated for each string should be just large enough to store the particular line of the file that it contains.
- As part of this assignment, we want to learn how to reallocate space to adjust to our needs. Your initial array of addresses of the strings will only be large enough to store 4 addresses (it's this small to virtually guarantee the need to reallocate). When a new line of the file is read that would exceed the size of the array of addresses, then the array should be doubled in size by using the `realloc()` function. Each time that `realloc` is called, print a message indicating that the space is doubling in size and the new size. This will help you with debugging and also make it easier for me to check that you are doing it correctly.
- If you have memory that was allocated using `malloc()`, `calloc`, or `realloc()`, you can expand it using `realloc()`. You can use `realloc()` like this:

```
temp = realloc(oldptr, newsize );
if(temp != NULL)
{
    oldptr = temp;
}
else
{
    printf("unable to reallocate\n");
    exit(1);
}
```

  where `temp` will be the address of the expanded memory. It may actually be the same location pointed to by `oldptr`, but it will be `NULL` if `realloc()` is unable to satisfy your request. `newsize` is the size in bytes that you need. **Be careful:** when you specify `newsize`, make sure it is the total size in bytes that you need. For example, if you had initially stored 10 of some variable type, don't specify `newsize` as 20. It should be `20 * sizeof(object)` where `object` could be `int`, `char *`, `struct node`, etc.
- Free whatever memory is dynamically allocated. This pertains to your calls to `malloc()`. You don't do this separately for calls to `realloc()`.
- You know the following about the data and can use it in your logic:
  - You will still statically allocate an array to buffer a single line read from the file, so you can assume that no line will need more than 1000 bytes to store.

  Nothing else about the lines within the file should be hard-coded to this particular data.

Other requirements:

- **Programs that have completed few of the requirements will receive a grade of zero.**
- Don't use language elements that we have not covered up to this point in the course unless instructed to do so.
- Don't use global variables.
- Your program must compile on omega.uta.edu as a C89 program **without warnings**. It seems that GCC doesn't completely adhere to C89 rules by default, so include the following command-line switches when compiling:

  ```
  -std=c89 -pedantic
  ```

  for example,

  ```
  gcc -std=c89 -pedantic -o filename filename.c
  ```

- I automate part of the grading process; that is, I use a program to help me grade your submissions. Therefore, your output should match mine EXACTLY. In order for you to make sure this happens:

  1. You should download my compiled program from the course website (the specific name will match the homework number). After transferring my program to omega, you will need to make it executable. For example,

     ```
     chmod a+x brezeale-hw01
     ```

  2. Use `diff` to compare my output to yours and it should report no differences; don't try to use visual inspection to determine if they are the same since it is easy to miss spacing differences.

- In order to use diff to compare the output of two programs, you can either run each program and save its output for later comparison, or compare the output of each program directly. For example,

  ```
  ./brezeale-hw01 > brezeale.txt
  ./student-hw01 > student.txt
  diff brezeale.txt student.txt
  ```

  or

  ```
  diff <(./brezeale-hw01) <(./student-hw01)
  ```

  Note that there are no spaces between the less than symbols and the left parentheses in the second example. If there are no differences, `diff` will return nothing. Otherwise, it will tell you where the differences are.

- `diff` will correctly determine if two files differ at the byte level, but it may be difficult to visually identify the differences if it involves hidden characters such as tabs, newlines, and spaces at the end of a line. The `cat` command can be used to display a file, but by default doesn't help you with regard to hidden characters. To see these hidden characters, use the command-line switches `-tev`. For example, if viewing the file `myOutput.txt` use

```
cat -tev myOutput.txt
```

- You should give thought to what the data could be as described in this document and test appropriately. Simply running `diff` to compare my program's output to your program's output may not be sufficient to find mistakes. For example, if strings could have somewhere between 1 and 10 characters and your particular data only has 3 characters, then you should make sure your program works appropriately with strings of other sizes.