

CS 306 – Linux Programming – Spring 2012

Lab #1

(Due: Wednesday 2/15/12 by 3:00pm)

The subject of your first programming assignment is writing a C program to duplicate a subset of the functionality of the Linux/UNIX `head` command. In other words, you will create a C program that prints out the first so many lines in a file argument. This first program will involve only basic C language programming—no system calls will be used. All I/O will be done with functions from the *C Standard Library*. Furthermore, your version of `head` will be much simpler than the system version because:

- it need accept only a single option, `-n`;
- it need accept at most a single file argument.

You should assume that the command (executable) that will be created from your program will be named `myhead`. The syntax for your `myhead` command can then be given as:

```
myhead [-nN] [FILE]
```

The `-n` option specifies the number of lines to be printed from the file. If this option is not given, the *default number of lines to print is 10*. The `-n` option takes an integer argument (denoted as `N` above). With `head`, the integer argument may be immediately adjacent to the `-n` or may be separated from it by whitespace. To further simplify your task, your program can *assume the numeric argument will be immediately adjacent to the `-n`*, e.g., `-n25`.

`head` accepts an *arbitrary* number of file arguments (pathnames), including *none*. If one or more files are specified, the program prints the required lines from each file in sequence. Your program need *accept at most a single file (pathname) argument*. If *no file argument* is given with the command, your program is to print lines read from *standard input (stdin)*, just as `head` would do.

Here are examples of syntactically valid calls to `myhead`:

- `myhead test.text`
- `myhead -n15 test.text`
- `myhead`
- `myhead -n15`

The assignment is to be submitted in a single file, named `lab1.c`. This file must be able to be compiled and run on its own, which means it must contain a `main` and any needed functions (plus header includes, etc.). `main` must be set up to accept *command line arguments* using the C `argc` and `argv` mechanism.

Your code must define and use at least *two functions*, with prototypes:

- `void head_stream(FILE *fpntr, int lines)`
 - `fpntr` – an open file stream to read from;
 - `lines` – the number of lines to be printed from the stream;
- `char *get_next_line(FILE *fpntr)`
 - `fpntr` – an open file stream to read from;
 - return value: a valid C string, or else `NULL`.

`head_stream()` is to perform the real work of the command, once a file argument (or `stdin`) has been opened for reading. It should loop, getting each of the required lines from the stream and printing them to *standard output (stdout)*.

`get_next_line()` is to do the job of reading the next line from a stream. It is to return the next line as a valid C string. It should return `NULL` if an I/O error occurs during its execution, or if it immediately encounters the *end of file* (i.e., there are no further characters in the file when the function is called).

The basic logic for `main` should be as follows:

1. examine the command line arguments to determine how many lines are to be printed and whether to read from a file or standard input;
2. if a file is to be processed, open it, else use `stdin` as the stream;
3. call `head_stream()` to process the current stream;
4. when `head_stream()` returns, close the stream if it was for a file;
5. exit with appropriate status.

The basic logic for `head_stream()` should be as follows:

1. iteratively call `get_next_line()` to obtain the next line from the stream;
2. print the line to *standard output*;
3. repeat from step 1 until the required number of lines have been printed or there is a `NULL` return from `get_next_line()`;
4. if `get_next_line()` returned `NULL`, check if there was an error and print an appropriate message before returning, else just return.

The basic logic for `get_next_line()` should be as follows:

1. iteratively call `fgetc()` to obtain the next character from the stream, until the end of the current line, the end-of-file, or a read error occurs;
2. store each read character into a “line buffer” (array);
3. when the line end has been reached, turn the array into a valid C string and return it;
4. return `NULL` if there are no further lines or an error occurs.

A key issue in implementing `get_next_line()` is how to allocate memory to hold the string that is to be returned. Memory for the “buffer” to hold the line string is to be allocated within `get_next_line()`. However, there are two potential pitfalls in implementing this. First, the size of an array in C cannot be changed dynamically (in C89 an array size must be known at compile time, while in C99 it can be a function parameter). Unfortunately, the *maximum length* of the lines to be printed are unknown in general, so we cannot know how large to make the array either at compile time or when calling `get_next_line()`. For simplicity in this lab, we will avoid the issue of unknown line length by simply *assuming that files will not have lines longer than 100 characters*.

The second memory pitfall is that in a function, memory for local variables is allocated on the *stack* by default, and this memory will be automatically reclaimed/reused upon exiting the function. This means that one should never return pointers to local variables from a function (and remember that strings are arrays, and arrays are represented by a pointer to the array start). For simplicity here, you are to use an array as the line buffer, but *declare that array to be static*. Memory for static variables is allocated in a special, permanent part of a program’s address space. Normally they are used to retain values between function calls, but we can use this property to allow our string array to be returned safely.

In Lab #2, we will see how to solve both of the above problems, using *dynamic memory allocation* via `malloc()` and associated functions.

Dealing with potential *errors* is an important part of writing reliable programs. You can find out from the *man page* for a function or system call what errors can occur, and how the function/syscall indicates the error condition. Generally, C library functions and syscalls indicate errors through special return values. Thus, library/syscall calls must typically be wrapped within an `if` (or similar) that tests whether an error return value was produced. You are expected to do this throughout your code, and include code to handle any errors in an appropriate manner (consistent with what the `head` does).

Your program is to produce *exactly the same output* as the `head` command would with the same (limited) arguments. This means that you can test your program by running it and `head`, and comparing the output using a tool like `diff` or `sdiff` (redirect output to files to allow comparisons). Pay particular attention to what `head` does with invalid file arguments and when there are less lines in a file than the number that are supposed to be printed out. Also pay attention to `head`'s *exit status* under these conditions.

Additional instructions/requirements:

- You are to use C library functions (from `<stdio.h>`) to handle *all I/O*. The functions you will need include: `fopen()`, `fgetc()`, `fprintf()`, and `fclose()`.
- You are *not allowed to use* any of the following functions: `fgets()`, `gets()`, `scanf()`, or `fscanf()`.
- All terminal output is to be done with `printf()` and/or `fprintf()`. By convention, *error messages* are output to *standard error* rather than to *standard output*. You are to adhere to this convention.
- `get_next_line()` is *not* to do any output (including error messages), and it must *always return*.
- `get_next_line()` is to return `NULL` both on end-of-file (not an error) and if an I/O error occurs while trying to read from a file. Your program will have to use additional means to distinguish the two cases, so you know whether to print an error message or not. The two situations can be distinguished using `ferror()`, `feof()`, or `errno`.
- Remember that a Linux/UNIX line will be terminated either by a *linefeed* (`'\n'`) or by the end of the file. The linefeed is not considered part of the line and so should *not be included* in the string returned by `get_next_line()`.
- You should use the preprocessor `#define` syntax with the symbol `MAX_LINE_LENGTH` to define the maximum number of characters that your program can support in lines to be printed (not including the newline character). Set this to 100.
- Remember that you must return a *valid C string* containing each line, and these strings must include a null char (`'\0'`) terminator, which adds one byte to the required storage.
- Your program must *error check* the original call (as well as function calls). In the case of an error in the syntax of the original call, print a message similar to what `head` does. (Most commands print a “usage” message giving the command syntax, however.)
- There are two important errors for which you must print messages: (1) invalid file argument, and (2) I/O error when calling `fgetc()`. An invalid file argument is easy to test, so provide a message identical to what `head` prints. Unfortunately, an I/O error is very difficult to force, so you cannot easily find out what `head` prints. In case `fgetc()` encounters an I/O error, print the standard system message (using either `perror()` or `strerror()`). Duplicate the format that most Linux/UNIX programs use, which

is to prefix the message with the name of the program followed by a colon, give an informative message followed by a colon, and end with the system error message. E.g., `myhead: cannot open file: no such file`).

- Your `main` must ultimately return an appropriate status code indicating success or failure. Experiment with `head` to see what it does, as you are to duplicate that behavior. Your code should use the symbols `EXIT_SUCCESS` or `EXIT_FAILURE` rather than 0 or 1.

You must *submit your file electronically from your CS Dept. Linux account*. This will require that you have the `lab1.c` file stored in your Linux account. You submit the file from this account using a command like: `cs306submit lab1.c`. Submissions are timestamped, and no late submissions will be accepted unless the due date/time is extended for the entire class. If you intend to create the program file on other machines, make certain that you figure out how to transfer it to your CS Dept. Linux account prior to submission time. For further information (or in case of problems), see the Electronics Submissions entry on the course webpage.

Development hints:

Students not familiar with C, are strongly urged to develop their program *in stages*. This approach will minimize the number of errors that your code might have at any one time, making it easier to identify and correct these errors. For example, one way that you might proceed is as follows:

1. write `main` to decode the command-line arguments, but just print out what is found to be the file and number of lines to print;
2. add `get_next_line()` and `head_stream()`, but initially implement `get_next_line()` using `fgets()` as done in the sample `firstline.c` program, and have `head_stream()` just get one line and print it;
3. extend `head_stream()` to loop, getting and printing the required number of lines, plus make it deal with a `NULL` return from `get_next_line()`;
4. correctly implement `get_next_line()`, replacing the single call to `fgets()` with a loop calling `fgetc()`, and getting the return of the buffer or `NULL` correct.

Note: `fgets()` does all the work of reading a (known maximum size) line for you, so you are not allowed to use it in your final solution; also the way it handles the newlines separating lines is *not* how you are to handle newlines!

Testing your code:

The most difficult aspect of this program is properly implementing `get_next_line()`. In testing your code, you need to pay particular attention to testing it with files containing lines that are likely to reveal issues with this function. For example, does your function correctly handle lines of exactly 100 characters? What about lines terminated by the end-of-file rather than a newline? Lines of exactly 100 characters terminated by end-of-file?

Another common mistake novice C programmers make is failing to ensure that all strings are proper C strings—i.e., that they are properly null-char terminated. Think about how to best test to find such errors. Hint: having lines of the same or increasing length is less likely to reveal problems of this sort than having lines that increase and then decrease in length.