CSE410 - Advanced Programming In UNIX

Assignment 3

Assignment goal

The purpose of this assignment is to gain some experience with C programming by implementing a utility program that is similar to grep, but without the ability to process regular expressions. In particular, in this assignment, you will:

- Gain experience creating and running C programs,
- Become familiar with some of the basic C libraries, including those for file and string handling,
- Get a better understanding of how Unix utilities are implemented, and
- Gain some basic experience with the unix debugger, gdb.

This assignment does not include any particularly complicated logic or algorithms, but it will require you to organize your code well and make effective use of the C language and libraries. It is meant as an orientation to the Unix/Linux C programming environment.

There should be **NO NEED** to do any dynamic memory allocation (malloc/free) in this assignment. Just use the stack.

Documentation

In the book, you will find pages 470 through 478 useful for string and file operations.

Remember that we looked at an example of a C program in class that read data from a file.

Google is your friend!

Getting ready

No additional files provided for this assignment.

Synopsis

In this assignment, you will implement in C a Unix utility program `gasp`. The command

```
gasp [options] STRING FILE...
```

should read the listed files (FILE...) and copy each line from an input file to stdout if it contains STRING. Each output line should be preceded by the name of the file that contains it. The argument STRING may be any sequence of characters (as expanded, of course, by the shell depending on how it is quoted). There are two available options, which may appear in any order if both are present:

- `-i` Ignore case when searching for lines that contain STRING. If the -i option is used, the strings "this", "This", "THIS", and "thiS" all match; if -i is not used, they are all considered different.
- `-n` Number lines in output. Each line copied to stdout should include the line number in the file where it was found in addition to the file name. The lines in each file are numbered from 1.

Your program does not need to be able to handle combinations of option letters written as a single multi-character option like -in or -ni. But it does need to be able to handle any combination of either or both (or neither) option.

You can assume that all options precede STRING

At the end of this asignment, your program should have at least three functions: `main`, `match`, and `lowercase`.

Technical Requirements

Besides the general specification given above, your program should meet the following requirements to receive full credit.

- Be able to handle input lines containing up to 500 characters (including the terminating \0). This number should not be hard-wired in the code, but should be specified with an appropriate `#define` preprocessor command so it can be changed easily. Your program is allowed to produce incorrect results, fail, or crash if presented with input data containing lines longer than this limit.

- You can assume that STRING will be no longer than 100 characters (including the terminating \0), although you may not need this assumption.
- Use standard C library functions where possible; do not reimplement operations available in the basic libraries. For instance, strcpy in <string.h> can be used to copy \0-terminated strings; you should not be writing loops to copy such strings one character at a time.
  Exception: there is a `getopt` function in the Linux library that provides simplified handling of command line options. For this assignment, only, you may not use this function. You should implement the processing of command line options yourself, of course using the string library functions when these are helpful.
- For the -i option, two characters are considered to be equal ignoring case if they are the same when translated by the`tolower(c)` function (or, alternatively, `toupper(c))` in `<ctype.h>`.
- If an error occurs when opening or reading a file, the program should write an appropriate error message to stderr and continue processing any remaining files on the command line.
- Your code must compile and run without errors or warnings when compiled with gcc -Wall on attu.

Code Quality Requirements

As with any program you write, your code should be readable and understandable to anyone who knows C. In particular, for full credit your code must observe the following requirements.

- Divide your program into suitable functions, each of which does a single well-defined task. For example, there should almost certainly be a function that processes a single input file, which is called as many times as needed to process the list of files on the command line (and which, in turn, might call other functions to perform identifiable subtasks). Your program most definitely may not consist of one huge main function that does everything. If you wish, you may include all of your functions in a single C source file, since the total size of this program will be fairly small. Be sure to include appropriate function prototypes near the beginning of the file. As stated above, there should be at least 3 functions in your program.

- Comment sensibly, but not excessively. You should not use comments to repeat the obvious or explain how the C language works -- assume that the reader knows C at least as well as you do. Your code should, however, include the following minimum comments:
    - Every function must include a heading comment that explains what the function does (not how it does it), including the significance of all parameters and any effects on or use of global variables. It must not be necessary to read the function code to determine how to call it or what happens when it is called. (But these comments do not need to be nearly as verbose as, for example JavaDoc comments.)
    - Every significant variable must include a comment that is sufficient to understand what information is stored in the variable and how it is stored. It must not be necessary to read code that initializes or uses a variable to understand this.
    - In addition, there should be a comment at the top of the file giving basic identifying information, including your name, the date, and the purpose of the file.
- Use appropriate names for variables and functions: nouns or noun phrases suggesting the contents of variables or the results of value-returning functions; verbs or verb phrases for void functions that perform an action without returning a value. Variables of local significance like loop counters, indices, or pointers should be given simple names like i, n, or p, and do not require further comments.
- **No global variables**. Use parameters (particularly pointers) appropriately.
- No unnecessary computation. For example, if you need to translate the STRING argument to lower- or upper-case, do it once; don't do this repeatedly for each input line or even for each input file. Don't make unnecessary copies of large data structures; use pointers. (Copies of ints, pointers, and similar things are cheap; copies of arrays and large structs are expensive.) Don't read the input by calling a library function to read each individual character. Read the input a line at a time (it costs just about the same to call a I/O to read an entire line into a char array as it does to read a single character). But don't overdo it. Your code should be simple and clear, not complex containing lots of micro-optimizations that don't matter.

What follows are step-by-step instructions for this assignment

0. Hello World (not graded)

In a file called `gasp.c`, write a program that writes "Hello World" to stdout. **Test your program**.

1. Command-line Arguments

Modify `gasp.c` so that it prints back to standard output a confirmation of the command-line arguments that it receives:

If the user provides option `-i`, print "`Case sensitivity disabled`". For option `-n`, print "`Line numbers requested`".

Additionally, print the pattern and the names of the files to standard output.

If the user provides either the wrong options or fewer than one argument following the last option, print an informative "usage" message.

Test your program.

2. File I/O

Modify `gasp.c` such that it reads the content of the files provided as input and writes it back to standard output.

Test your program

3. Basic string manipulation

Implement the grep feature (as described above) without options.

Feel free to comment the debug messages that you implemented in parts 0 and 1 above. But make sure to leave them in the file!

Test your program

4. Additional string manipulation

Implement the options.

Test your program

Submission

- You only need to turn in `gasp.c` .

In Piazza Submission process:

Type: note

To: instructors

Folder: assignment N (N is your assignment number)

Summary: Your name and ID, Assignment N

Details: small description and attachment s.