CSCI 4460/6460 — Large-Scale Programming and Testing
Homework 1 (document version 1.0) — DUE October 8, 2020
Regular Expressions and Pattern Matching in C

- This homework is due by 11:59PM EDT on the due date shown above
- This homework is to be completed **individually**; do not share your code with anyone else
- You **must** use C for this homework assignment, and your code **must** successfully compile via `gcc` with no warning messages when the `-Wall` (i.e., warn all) compiler option is used; we will also use `-Werror`, which will treat all warnings as critical errors
- Your code **must** successfully compile and run on Submitty, which uses Ubuntu v18.04.5 LTS; note that the `gcc` compiler is version 7.5.0

Homework Specifications

In this first homework, you will use C to implement a rudimentary `grep` tool that supports line-based regular expressions and simple pattern matching. The goals here are to become more comfortable programming in C on Linux and to write **high-quality** and **easily maintainable** code. **Therefore, a portion of your grade will be based on the quality of your code.**

In brief, your program must apply a regular expression to a given input file, displaying all lines of the input file that match the given regular expression. (Play around with `grep` to get an idea of the desired functionality.)

Required Mindset — Functional Decomposition

You've written code for many programming assignments in the past. For this assignment, use as many "best practice" approaches to designing and implementing your code. Adhere to a coding standard. Try to identify which techniques help you produce high-quality bug-free code.

More specifically, **keep careful track** of how many times you compile your code, how many times your code compiles without errors or warnings, and how many times you encountered logic errors (i.e., non-compiler bugs) in your code. Your goal is to minimize these latter two measures to the extent possible.

Also, use **functional decomposition** for this assignment, i.e., break every task down into smaller and smaller functions. Each function should have one key purpose; if not, refactor.

To further encourage you, you will have a cap of only **four** Submitty attempts before you start losing points! And some of the grading criteria will be focused on analyzing your code to determine the level of good commenting, variable naming, program structure, etc. that you achieve.

Specifying Regular Expressions

For this assignment, we will implement a subset of general regular expression syntax that focuses on finding substrings.

A regular expression (regex) is a string used as a pattern for matching. The regex may contain characters that must match exactly; for example, if the pattern is `"printf"`, then it will only match exact substrings (i.e., all lines that contain `"printf"` somewhere).

In addition to this simple substring matching, there are other special characters that enable groups of characters to be matched.

The specific individual character regex syntax that must be supported is:

```
.     match any character
\d    match any digit [0-9]
\D    match any non-digit
\w    match any letter [a-zA-Z]
\W    match any non-letter
\s    match any whitespace character (space or tab '\t')
\\    match a single literal backslash character '\'
```

Further, repetition may be specified by appending the following to any individual character or regex symbol shown above.

```
?     match zero or one instances
+     match one or more instances
*     match zero or more instances
```

Some example regular expressions to support (and test for) are:

```
printf                match "printf"
colou?r               match either "color" or "colour"
\w+                   match a word consisting of one or more alpha characters
<\w+>                 match an XML/XHTML tag, such as "<title>" or "<body>"
\w+\d*@rpi.edu        match an RPI email address by matching one or more alpha
                       characters, followed by zero or more digits, then "@rpi.edu"
                         (though note that the "." here matches any character...)
\d\d\d-\d\d\d\d       match a phone number, such as "276-2819"
if\s*(.+)             match an if statement (to some degree) in C/C++/Java
```

Note that for both `"*"` and `"+"`, use a greedy approach, meaning that you should match as many characters as possible. As an example, the `"if\s*(.+)"` pattern above should match through to the rightmost closing parenthesis (if present).

Ignore the need (or temptation) to add more regex syntax here! While regex matching can be rather complex, especially as the regex syntax is expanded, keep your implementation as simple as possible. Start by focusing on the examples above.

Command-Line Arguments

A text file containing the regular expression is specified on the command-line as the first argument. The input file is then specified as the second argument. As an initial example, you could execute your program and have it apply the regular expression specified in `regex.txt` to input file `hw1-input01.txt` as follows:

```
bash$ ./a.out regex.txt hw1-input01.txt
```

Implementation Details

For your implementation, consider using `fopen()`, `fscanf()`, `fgets()`, `fclose()`, `isalpha()`, `isdigit()`, `isspace()`, and other such string and character functions. Be sure to check out the details of each function by reviewing the corresponding `man` pages.

If improper command-line arguments are given, report an error message to `stderr` and abort further program execution. In general, if an error is encountered, display a meaningful error message on `stderr` by using either `perror()` or `fprintf()`, then aborting further program execution.

Error messages must be one line only and use the following format:

```
ERROR: <error-text-here>
```

For errors involving command-line arguments, display the following two lines to `stderr` exactly as shown below:

```
ERROR: Invalid arguments
USAGE: a.out <regex-file> <input-file>
```

Assumptions

For this initial implementation, you can make the following assumptions:

1. Assume that the maximum line length for any input file is `256` characters.
2. Assume that the regex file contains exactly one non-empty line.
3. Assume that the regex is correct in terms of its syntax.

Required Output

When you execute your program, unless errors occur, your program must display all lines from the given input file that match the given regular expression.

As an example, consider the following `hw1-input01.txt` file:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
  int x = 300-1200;
  printf( "I like Cocoa Puffs\n" );
  printf( "and also Trix %d\n", x );
  printf( "Green is my favorite color\n" );
  printf( "My number is 276-2819, but don't call me\n" );
  return EXIT_SUCCESS;
}
```

Executing your program as follows with the regex `"printf"` would display the output shown below:

```
bash$ ./a.out regex.txt hw1-input01.txt
  printf( "I like Cocoa Puffs\n" );
  printf( "and also Trix %d\n", x );
  printf( "Green is my favorite color\n" );
  printf( "My number is 276-2819, but don't call me\n" );
```

Executing your program as follows with the regex `"colou?r"` would display the output shown below:

```
bash$ ./a.out regex.txt hw1-input01.txt
  printf( "Green is my favorite color\n" );
```

Executing your program as follows with the regex `"\d\d\d-\d\d\d\d"` would display the output shown below:

```
bash$ ./a.out regex.txt hw1-input01.txt
  int x = 300-1200;
  printf( "My number is 276-2819\n" );
```

Submission Instructions

To submit your assignment (and also perform final testing of your code), please use Submitty. Remember that you will only have four submissions before points are deducted. Therefore, be sure to thoroughly review and test your code.

This assignment will be available on Submitty a few days before the due date. To make sure that your program does execute properly everywhere, including Submitty, use the techniques below.

First, use the `DEBUG_MODE` technique to make sure you do not submit any debugging code. Here is an example:

```
#ifdef DEBUG_MODE
    printf( "the value of x is %d\n", x );
    printf( "the value of q is %d\n", q );
    printf( "why is my program crashing here?!\n" );
    printf( "aaaaaaaaaaaaaggggggggghhhh!\n" );
#endif
```

And to compile this code in "debug" mode, use the `-D` flag as follows:

```
bash$ gcc -Wall -Werror -D DEBUG_MODE hw1.c
```

Second, output to standard output (`stdout`) is buffered. To ensure buffered output is properly flushed to a file for grading on Submitty, use `setvbuf()` as follows as your first line in `main()`:

```
setvbuf( stdout, NULL, _IONBF, 0 );
```

You would not generally do this in practice, as this can substantially slow down your program, but to ensure good results on Submitty, this is a good technique to use.