CSC 412 – Operating Systems

Programming Assignment 03, Fall 2022

Monday, October 2nd, 2023

Due date: Tuesday, October 10th, 11:59pm

1 What This Assignment Is About

1.1 Objectives

The objectives of this assignment are for you to

- Use 1D and 2D arrays in C;
- Perform file I/O in C;
- Search a directory hierarchy in bash;
- Perform a simplified, idealized form of "pattern matching" in "images;"
- Use scripting to solve a classical problem of cross-platform development.

This is an individual assignment.

1.2 Handout

The handouts for this assignment consist in sets of example "image" and "pattern" files that you can use to test your program. The "image" files are given in two versions: One with the incorrect, Windows-style line-ending, and another with the expected Unix-style line ending (see Section 3 for clarification of what this means).

1.3 Project folder organization

Your project folder should simply be named `Prog03` and will contain

- A folder named `Source` containing files for the code of all versions of the C program that you have developed;
- A folder named `Scripts` containing the bash script files;
- Your report.

All scripts and executables are expected to be run from the project's root folder.

2 Part I: Search for Patterns

2.1 Format of the data

As mentioned earlier, the "images" are represented as arrays of text data. The file format we are going to use to represent the "images" is extremely simple:

- the first line gives the width w and height h (number of columns and number of rows) of the "image;"
- the next h lines stores the different rows of the "image."

The following lines of text could encode a small "image"

```
12 10
ABEFGH345TYH
123456789012
ABC3DG123asd
345ASABCFTG0
224CG345D12t
SDFG3224Zd1z
130678901345
098765432101
098765432101
098765432101
```

The "patterns" to search for in the "images" are squares of three rows and three columns. Each pattern is stored in a separate file. We are going to reuse the same image format to store the pattern, so that all pattern files will start with a line `3 3`. This also means that you can use the same code to read the images and the patterns (and that a pattern is simply a small image).

2.2 An important note

You must understand that the "images" and "patterns" we are talking about here are abstractions and simplifications of a real-world problem. Please don't waste time searching online for solutions to "pattern matching in images." The solutions that you are going to find will try to address the **real** problem:

- real color images (encoded on multiple bytes);
- best-fit match rather than exact match;
- handling of noise and illumination variations in images;
- handling of scaling and rotation;
- etc.

In other words, the code you are going to find online will be so complicated and so much more general than what you need to do here that it will be completely worthless. Your solution will involve little more than string comparisons, and your main tools are going to be the `strcmp` function of `<string.h>` or simple character-by-character comparisons.

This being said, if you *are* interested in pattern matching, send me a note or come to my office to discuss your ideas. I approach such problems from the point of view of "classical computer vision" (that is, not machine learning-based). If you want to hear a different, machine learning-based point of view, Profs. Hamel, Alvarez, and Daniels are very good interlocutors.

2.3 Version 1

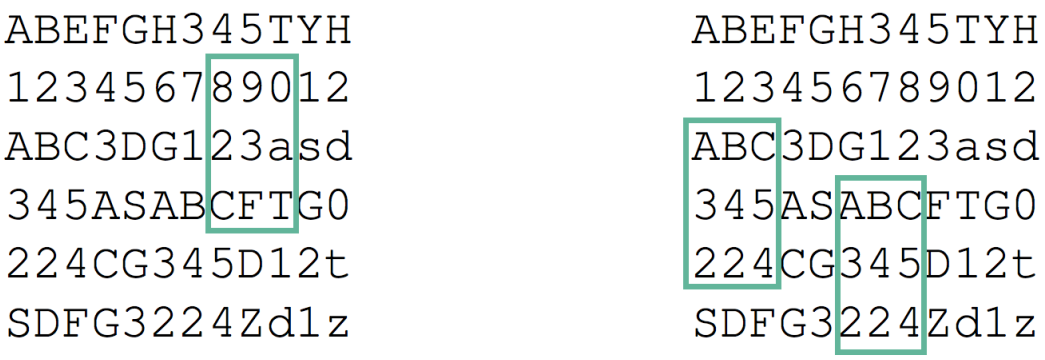
The source file of your program should be named `prog03v1.c`. For a given image and pattern your program should look for every occurrence of the pattern in the image and report in an output file the number of matches found, and their location.
For example, if the pattern is

```
890
23a
CFT
```

then the result for this pattern should be

```
1 2 8
```

because the pattern has been found once in the image, at the location with its upper-left corner at row 2 and column 8 (Figure 1(a)).



(a) (b)

Figure 1: Localization of "patterns" in an "image."

Note that we are using 0-based indices for rows and columns in our file output, with row index 1 corresponding to the top of the "image."

If the pattern is

```
ABC
345
224
```

then the result for this pattern should be

```
2 3 1 4 5
```

because the pattern has been found twice in the image, once at the location with its upper-left corner at row 3 and column 1, once at the location with its upper-left corner at row 4 and column 5 (Figure 1(b)).
If the pattern is

```
xxx
yyy
zzz
```

then there is no match and the program should output

```
0
```

because the pattern is not found anywhere in the image.

Note that the file extensions `.img` and `.pat` are just there to help you identify files of the types we are interested in. If you change the extension to `.txt` you can see that they are simply text files.

2.3.1 Input to the program

Your program will be launched (typically by your bash script, but of course also directly from a bash terminal if your script is missing or deficient) with the following arguments:

1. The path to a file storing an "image" that you want to search for pattern within;
2. a list of paths to "pattern" files;

For example, assuming that your executable was built with the name `prog03v1` in the project's root folder, it could be launched with a command such as

```
./prog03v1 ./Data/Images/image1.img \
./Data/Patterns/pattern1.pat ./Data/OtherPatterns/pattern2.pat \
./Data/Patterns/Sub/pattern3.pat
```

The characters `'\'` are part of the input. They are there to allow to enter a very long command over multiple line. In practice, they tell bash to "eat" the next character (which should be an end-of-line character). This is why there *cannot* be a space character after the `'\'`, only an end of line. On the other hand there should be a space character either before `'\'` or at the beginning of the next line (otherwise consecutive items of your command would be stuck together). If you do it well, then the `'\'` essentially means "the command continues on the next line."

2.3.2 Format of the output

Each process[1] will deal with *one* image file, specified by a file path. It will print out its results to the standard output (terminal). An example of output would be the following (these are completely made up output values, just chosen to illustrate the format):

```
pattern1.pat
    2 3 1 4 5
pattern2.pat
    1 5 11
noMatchPattern.pat
    0
pattern31.pat
    3 8 10 12 5 15 1
pattern3.pat
    1 4 125
```

Note that the results for a given pattern are preceded by a tab `'\t'` character. It is worth pointing out that the output only gives the *name* of the pattern file, not the full path, which means that there could be more than one image file with the same name. For example, we can see in the above output example that there were two pattern files named `pattern1.pat` searched by the program (presumably with a different path), and the program printed out separately the results for these two pattern files.

2.4 Version 2: output to a file

The program's source file named should be named `prog03v2.c`. Your program will be launched (typically by your bash script, but of course also directly from a bash terminal if your script is missing or deficient) with the following arguments:

1. The path to a file storing an "image" (the files are identified by a `.img` extension) that you want to search for pattern occurrences in;
2. The path to an "output" folder where to print out the results of the search for the input "pattern";
3. A list of paths to "pattern" files.

For example, assuming that your executable was built with the name `prog03v2`, it could be launched with a command such as

```
./prog03v2 ./Data/Images/image1.img Output/ \
./Data/Patterns/pattern1.pat ./Data/OtherPatterns/pattern2.pat \
./Data/Patterns/Sub/pattern3.pat
```

[1] In the case of this assignment, we will only run one instance of the program at a time, but I want you to get used to the term "process," so that—hopefully—you start thinking that there will be several, possibly many, instances of your program running concurrently, something that will happen soon enough.

The name of the output file produced by your process should be the "root name" of the image with the `_Matches` suffix and the `.txt` extension. In the case here, the output file name should be `image1_Matches.txt` and its path should be `./Output/image1_Matches.txt`. The content of the file should be what was going out to the terminal in Version 1.

A note of caution: Whether you are coding in C or bash (or C++, Java, Python, etc.), whenever one of the input arguments is the path to a *directory*, be careful that the forms of a path ending with a slash or without a slash are valid (e.g. `./Output/` and `./Output` are both valid directory path entries), and you cannot just assume that your user will use one form and not the other. It is **your job** as a programmer to make sure that your program can handle both forms properly.

This means that you should check if the last character of the path is a `/` and then either decide to "erase" that character in the string that end with one, or add one to all the strings that lack one. Pick one style and stick to it.

2.5 Extra credit: general pattern (3 points)

This version of the program should be named `prog03EC1.c` and should generalize your pattern matching code so that the patterns to search for can now be of any size (still specified in the first line of the pattern file). This version can be developed on top of either Version 1 or Version 2.

Note: I understand that some of you may have coded directly their `prog03.c` so that it already handles patterns that are not 3×3. In that case the code of `prog03.c` and `prog03EC1.c` will be identical, and there is no problem with that. Still submit two source files so that we know what to grade.

2.6 Extra credit: Rotation and flip-invariant pattern search (up to 10 points)

If you implement this EC, provide a separate source file named `prog03_EC2.c`. That version of the program should be able to find matches in an image after vertical mirroring, horizontal mirroring, or rotations. This version can be developed on top of either Version 1 or Version 2.

3 Part II: `bash` Scripts

3.1 The end-of-line headache

This week, the first `bash` script we are asking you to write is not only going to launch a C/C++ program. It is also going to perform some simple string manipulation to address a very old, vexing problem that all of us who have to use different platforms must run into, sooner or later.

The issue as stake here is that of the characters used to indicate a new line in a text file, that is, the end of the current line. This can be referred to as the "newline," "end of line," or line ending problem. It carries over from the days of mechanical typewriters. When typists arrived to the end of a line, they had to execute a "linefeed" (LF) to make the paper sheet move up by one line, and then a "carriage return" (CR) to reposition the head at the beginning of the new line.

When the different operating systems were developed in the 70s and 80s, they came with different scheme to represent the end of line, most based on the linefeed (LF, `'\n'`, `0x0A`, 10 in decimal) or the carriage return (CR, `'\r'`, `0x0D`, 13 in decimal).

Most notably:

- Early on, Unix systems (and later on, Amiga, BeOS, and Linux) opted for LF;
- The TRS 80, the Apple II, and the original Macintosh System (then Mac OS until Mac OS 9 up to the early 2000's), chose CR;
- CP/M, and therefore DOS, Windows, and OS/2 (along with Atari TOS and others) picked CR+LF.

As you can imagine, this posed several problems. For a start, typically text editors on Windows (for cause of being the dominant OS, with 95% of the installed base) and Unix (for "true OS" religious reasons) only recognized the native endline format. You can see this at work if you try to view in Notepad[2] one of your Linux programs freshly unzipped from the archive. Other common problems are that the "same" text file has a different length on Windows and Unix or that the same C code is not guaranteed to give the same results on the different platforms. Finally, developers of file transfer clients had to offer different transfer mode: one for text file, in which end of line characters had to be translated, and one for "binary" files in which inserting a character `0x0A` before every occurrence of a `0x0D` could corrupt images, videos, etc.

Since then, Mac OS X (and now macOS and iOS) being a Unix system, Apple has switched to LF as its supported end of line format. So we are pretty much down to two formats: LF vs. CR+LF. This is the problem that you are going to address in this assignment. We are going to provide some text files (representing "images" and "patterns" to search for in the images) to be used as input for the C section of this assignment. All files will be in the Windows format, and your script will "translate" the files into the Unix LF end of line format.

3.2 What the script should do

Your script should be named `script03_01.sh` and reside in the `Scripts` folders (and will be run from the project's root folder). It should take as arguments:

- the path to the directory containing a number of "image" files;
- the path to a "scrap folder" where to store any temporary files that your script may need to create;
- the path to the directory where to write the "images" with the proper line ending characters. Be careful that the user may choose the same folder as the input folder (this is the main purpose of the "scrap folder").

[2] Of course, you all have installed on your PC a decent text editor, Notepad++ or better, so you would never open a text file by default in Notepad, I am sure.

The image files in the input directory files are encoded using Windows line endings. The script must replace these line endings with Unix line endings. If your script creates temporary files, then it should create them in the indicated "scrap folder" (and create the scrap folder first in that case). At the end of execution, the script should the scrap folder and its contents if it created it.

3.3 What the script should *not* do

There exist utilities that allow you to perform the end-of-line fix on an entire file with a single-line command. You are not allowed to use these commands. Again, I want to force you to write a loop in `bash` to iterate through every line of a text file, because this is an fundamental building block of `bash` programming. If you also happen to know a few extra convenient commands, this is great, but you won't always find the "magic one-line command that solves exactly the problem." And in that case, you will have to be able to roll your own loop.

3.4 Recursive search and execution script

This script should be named `script03_02.sh` and reside in the `Scripts` folders (and will be run from the project's root folder). It should take as arguments:

- the path to the directory within which to look recursively for "patterns";
- the path to a directory containing a number of image files (the files are identified by the extension `.img`);
- the path to an output folder where the results of the searches should be written.

Your script should first build the executable for Version 2. Next, it should search recursively the pattern folder to look for all the pattern files (identified by the extension `.pat`) that it contains. This means that the root search folder may contain a hierarchy of subfolders that you must search on order to find all files with the proper extension. After all the "pattern" files have been located, the the script should call the executable of Version 2 for all image files, each time passing along the paths to all pattern files found and the path to the output folder as destination.

Partial credit:
If you didn't complete Version 2, then you can still call Version 1 with this script instead, but you will only get half the points for this part of the assignment.

4 What to submit

4.1 The pieces

All your work should be organized inside a folder named `Prog03`. Inside this folder you should place:

- Your report;
- A folder named `Programs` containing all the source (and possibly header) files required to build the different programs that you completed;
- A folder named `Scripts` containing the different scripts you developed;
- Your report, as a pdf file;
- Nothing else (binaries, data files, etc.). If we have problems with your project and need to see what kind of data you ran it with, we will contact you.

Submit on Brightspace a `zip` archive of the project folder.

4.2 Grading

4.3 What's on the report

The main sections I want to see in this report are:

- A presentation and discussion of the choices you made for the implementation of the pattern search. This is an algorithm and data structure question, with considerations on style, modularity, and performance.
- A discussion of possible difficulties encountered in this assignment[3].
- A discussion of possible current limitations of your C programs and scripts. Is it possible to make them either crash or fail?

4.4 Grading

- The C program performs the task specified (execution): 45 points
  - Version 1: 25 points
  - Version 2: 20 points
- the `bash` script performs the task specified (execution): 25 points
  - Script 1 (end of line): 10 points
  - Script 2 (recursive search): 15 points

[3] Here, I don't mean "my laptop's battery is dying" or "I couldn't get a pizza delivered to my dorm room."

- C and `bash` code quality (readability, indentation, identifiers, comments, etc.): 20 points
- Report: 10 points

Various penalties:

- Archive other than `.zip`•: 0 for assignment
- Report is not a pdf file: 5 points
- Folder organization (names, junk files, etc.): 10 points
- Does not handle final slash in folder path: 5 points