CSC 412 – Operating Systems

Programming Assignment 03 [rev. 1], Spring 2022

Monday, February 28th, 2018

Due date: Tuesday, March 8th, 11:55pm.

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;
- Experiment with process creation (`fork`, `waitpid` and the `exec` family of system calls);
- Use scripting to solve a classical problem of cross-platform development.

This is an individual assignment.

1.2 Handout

There are several handouts for this assignment:

- The code of a small (and incomplete) image processing library, including some "applications" built using this library, and a few images to test with;
- Sets of "job files" to be processed by your script.

2 Part I: TGA Image Files and the Code Handout

2.1 The code supplied

2.1.1 Organization

The code supplied consists of:

- The shell of a small image processing library, stored into the folder `ImageLibrary`, organized into three folders named `include` (for header files), `src` (for source files), and `lib` (for your yet non-existing DLL), respectively. More specifically,
  - `ImageIO.h` and `ImageIO.c` are respectively the header file and the source code for generic (independent from image file format) image reading and writing functions.

  - `ImageIO_tga.h` and `ImageIO_tga.c` are respectively the header file and the source code for the implementation of these functions for color and gray-level images in the uncompressed TGA format.
  - `RasterImage.h` is a header file that define some new data types and functions that are implemented in `RasterImage.c`.
  - `flipV.h`, `flipH.h`, `channel.h`, `gray.h`, and `crop.h` give the prototype of image manipulation functions implemented in `operations.c`.

- The `applications` folder contains source files for three small demo applications that use the library to perform elementary image processing applications.

2.2 About the `RasterImage.h` header file

I want to attract your attention to a few important points regarding this header file.

First, if you are familiar with javadoc style of comments, you probably recognized the characteristic `/**` . For many years, C++ programmers secretly envied their Java-programming colleagues who could produce good-looking code documentation through the javadoc system. They envied them until a brave soul came up with Doxygen, which interprets javadoc-style comments in a multitude of programming languages (C, C++, Python, Perl, etc.). Therefore, coding in C or C++ is not a good excuse to have a lousy documentation anymore. Get familiar with it, and with the Doxygen application, because, starting with the next assignment, you wil be expected write this style of comments and generate the html documentation for all your work.

Second, the two `enum` types, `ImageFileType` and `ImageType`, mention types that we won't encounter in this assignment. We will only deal with `kTGA_COLOR` and `kTGA_GRAY` images that we will store in `RGBA32_RASTER` `GRAY_RASTER` raster arrays. I might as well observe here that I love using `enum` types to represent and group constants that belong together. I already alluded to that in the C code samples that I posted on Sakai at the beginning of the semester. Enumerated types work well with switch statements, are easy to expand, and have all the advantages of an integral type (for calculations and array indices) while offering some—limited—protection regarding range.

Next, the `RasterImage` type stores the 1D raster master pointer as a `void*` pointer. This is a common design decision in C and C++, when the type of the data could be any of multiple possible types (here, besides the 4 bytes of color information that we will be dealing with, we could have a float value or a single byte). Set the pointer to be `void*`, and cast it back to the proper type when the data must be processed. We will encounter this pointer trick several times this semester.

Finally, the `RasterImage` type contains a field for a 2D array (from the name, but it is also cast to a `void*` pointer). This field is currently not used anywhere in the handout's code, because the corresponding 2D array has not been allocated. This will be a task for you to complete for EC when I propose you later in this assignment to "scaffold" a 2D array on top of an existing 1D array (for extra credit).

2.2.1 How to build the handout applications

Assuming that you are in the `Code Handout/ImageLibrary` folder, you build the handout `gray` application with the following command:

```
gcc -Wall -I ./include ./applications/gray.c
       ./src/*.c -o ./executables/gray
```

The `-I` option lets you indicate to `gcc` where to look for header files of your project. You could give multiple locations where to look for header files, but in this project we only need our library's header files.

2.2.2 How to run the handout applications

You can run your new `gray` application by specifying an input image and a destination folder for the output image. Assuming that you are still in the root folder of the library:

```
./executables/gray ../Images/Renoir.tga ../Output
```

(assuming that the destination folder exists).

2.3 The TGA file format

2.3.1 Why TGA?

The TGA (Targa file) image format is one of way-too-many image formats that you may run into. Other formats provide better compression or support for a wider range of colors, so why use this format at all? Because the format for *uncompressed* `.tga` image files is the only truly multi-platform format that is easy to read and write. We could use a more complete library such as `freeimage` or `ImageMagick`, and maybe we will do just that in a future assignment, but, as long as your image file stores data uncompressed and does not contain any comments, the code supplied here will do the job.

2.3.2 Color images and rasters

Pretty much all libraries dealing with image and video data[1] manipulate image data under the form of a 1D "raster," as shown in Figure 1.

So, if our image has m rows of n columns and each pixel is stored on k bytes (for a total of $n \times k$ bytes per row and $m \times n \times k$ bytes for the entire image), then the pixel at row i and column j in the image can be addressed at index $i \times n \times k + j \times k$ of the 1D raster.

In the uncompressed TGA file format, an image file stores each pixel as 3 unsigned bytes (range 0 to 255) encoding the red, green and blue color channels. When we load an image in memory, however, it is more convenient to store each pixel on an even 4 bytes. Besides allowing image processing and other graphic applications to use this additional byte to encode *transparency* (or the "α channel," as it is called), this also means that a pixel occupies the same space as an `int`, so that we can view our 1D raster either as an `int*` pointing to a pixel or as an `unsigned char*` pointing to a color channel of a pixel.

[1] An important exception is Apple's old QuickTime library and the `libquicktime` open-source replication of the API for a tiny bit of QuickTime. I mention this here because we may have an encounter with `libquicktime` before the end of the semester.

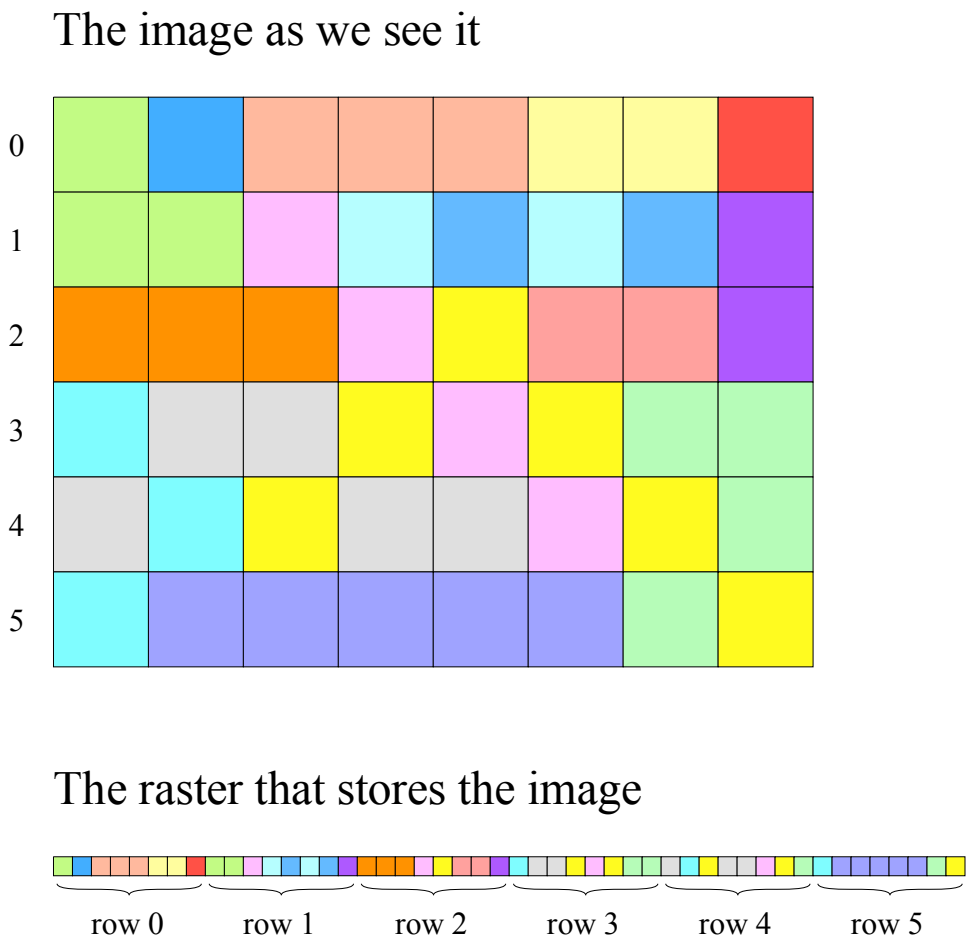

Figure 1: An image stored as a 1D raster.

Note that different applications may store the bytes of the color channel in different orders. The most common orders are `argb` (alpha, red, green, blue) and `rgba`. Throughout this assignment, and future assignments dealing with images, we will use exclusively the `rgba` format.

What this means is that if I want to access the color information of the pixel at row i and column j of the same $m \times n$ image, then I could get a pointer to this pixel's color channels, each seen as an `unsigned char` by writing

```
unsigned char* rgba = (unsigned char*)raster + 4*(i*n + j);
```

Now, I can view this pointer as an array and access directly my color components:

- `rgba[0]`: red channel,
- `rgba[1]`: green channel,
- `rgba[2]`: blue channel,
- `rgba[3]`: alpha channel.

Alternatively, I could see my pixel as a 4-bye `int` that I access through an `int*` pointer by writing

```
int pixel = *((int*)raster + i*n + j);
```

Now I can access my color channels by extracting the different bytes of the `pixel` variables. There is just a small problem: the byte order. Different CPU architectures have a different way to encode `int`, `float`, etc. Or rather, the encoding is the same, but the byte order is different. Intel processors (and their AMD clones) are "small-endian," which means that their least significant byte is stored first, and their most significant byte is stored last.

3 Part II: C Programs

3.1 Implement cropping

Your first task consists in implementing in the source file `operations.c` the function `crop` as defined in the corresponding header file. Then use (and test) this function by adding a source file for the application `crop` (source file `crop.c`), which takes as arguments the path to a valid TGA image file, the path to an output directory, and the x and y coordinates of the upper-left corner of the cropping rectangle and with and height of the crop rectangle.

The name of the output image should be that of the input image, with the `[cropped]` suffix. For example, if the input image is named `myImage.tga` and the command was

```
./crop ../Images/myImage.tga ../Output 24 120 85 210
```

then the name of the output image should be `myImage [cropped].tga`.

If the two corner points represent a rectangle that is—partially—outside of the image, your program should reject the input.

Note: You are going to observe that the internal format of the images is (or maybe the way the library reads them) is upside-down, so that row 0 is in fact the bottom row of the image. Thse are internal considerations for the library, not for the user. So it's your job to make sure that you properly convert to-down row indices, as users view them, into bottom-up row indices as stored internally.

3.2 Dispatcher process

Now that we have a few working applications, we can write the code of a program that will operate as a "dispatcher." This program will be launched with one argument: The path to a text file containing a list of image operation commands. For example:

```
./executables/crop ../Images/clown.tga ../Output 24 120 85 210
./executables/flipV ../Images/clown.tga ../Output
./executables/crop ../Images/myImage.tga ../Output 24 120 85 210
./executables/crop ../Images/Renoir.tga ../Output 100 20 60 80
./executables/flipH ../Images/clown.tga ../Output
```

For each task in the job file, the dispatcher process should create a child process that will then make a call to a function of the `execXY` family to perform the task.

3.3 Extra Credit

3.3.1 Extra credit 1: Image comparison (5 to 8 points)

This program named `comp` takes as arguments the path to two TGA image files and compares the two images' rasters. If they are identical, then the program should return 0. If they are different, the program should return 1.

You must first implement in `operations.c` a function that performs the operation, then in the `applications` directory a demo application that uses the function. Finally, you must update the build script to produce the executable for this demo application in the `executables` directory.

For the full 8 points, check also for mirrored versions of your images.

3.3.2 Implement the raster2D (8 pts)

The `RasterImage` data type includes a `void` pointer named `raster2D`. This pointer is meant to be used to access the image as a 2D array of pixels rather than as a 1D raster. That is, if we want to set the pixel data at Row i and Column j of the image to black,...

If wiew the image as made up of `pixels` stored each on a 4-byte `int`, instead of writing

```
int* raster = (int*) myImage->raster; raster[i*myImage->width +j] =
0xFF000000;
```

write:

```
int** raster2D = (int**) myImage->raster2D;
raster2D[i][j] = 0xFF000000;
```

Similarly, if you prefer to wiew the image as made up of 4 color channels each stored as an `unsigned char`, instead of writing

```
unsigned char* raster = (unsigned char*) myImage->raster; unsigned
char* rgba = raster + i*myImage->bytesPerRow + 4*j; rgba[0] = rgba[1]
= rgba[2] = 0x00; rgba[3] = 0xFF;
```

write:

```
unsigned char** raster2D = (unsigned char**) myImage->raster2D;
unsigned char* rgba = raster2D[i] + 4*j;
rgba[0] = rgba[1] = rgba[2] = 0x00; rgba[3] = 0xFF;
```

What is important here is that the 2D raster cannot be a copy of the 1D raster. Otherwise, changes made to one copy of the raster would not be reflected in the other copy. Rather, the 2D raster should be "scaffolded" on top of the 1D raster. If you think about it, `int** raster2D` can be interpreted as meaning that `raster2D` is a 2-dimensional array of `int`. But it can also be interpreted as meaning that `raster2D` is an array of `int*` pointers pointing to the beginning of a row (and these rows *are* 1D arrays of `int`). So, all you need to do is allocate an array of `int*` to the height) and initialize each element of this array to point to the beginning of a row.

To get the full EC points, you must implement support for the 2D raster in the "constructor" and "destructor" of the `RasterImage` data type, the functions `newImage` and `freeImage` respectively. You must also (re-)implement one of the functions in the `operations.c` source file so that it uses the 2D raster rather than the 1D raster.

4 Part IV: `bash` Scripts

4.1 The end of line headache

This week, the `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.

4.1.1 What the script should do

Your script should be named `script03_01.sh`. It should take as arguments:

[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 path to the directory within which to look for "task files" (the files are identified by the extension `.tsk`);
- the path to a directory where to write the "fixed" task files.

Your script should work over all task file, replacing the Windows end-of-line characters by Unix end-of-line characters and write the "fixed" task files in the output folder. The directory `ScriptData/Windows` contains two task files with Windows end-of-line characters. The desired output files are shown in the directory `ScriptDat/FixedEOL`. Note that these are only "task files" and not "job files" since they don't contain any path information. This will be addressed next.

4.1.2 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.

A note of caution: I have said and written it several times already, but t's worth repeating that whether you are coding in bash or C (or C++, Java, Python, etc.), whenever one of the input arguments is the path to a *folder*, 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 folder 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.

4.2 Dispatcher script

Your script should be named `script03_02.sh`. It should take as arguments:

- the path to the root directory of the image library;
- the path to the directory containing the input images;
- the path to a directory containing task files;
- the path to an output library.

Your script should build the executables for the different applications. Then, for each task file, it should launch a "dispatcher" process with the proper arguments.

Note: Please note that the information in the task files is not complete: It indicates the tasks to performs, e.g. `flipV clown.tga` but it doesn't specify any path. Your script has to fill in this information based on the arguments provided, in order to produce the job files expected by your C dispatcher program.

5 What to Hand In

5.1 Organization of the submission

Create a new folder named `Prog03`. In that folder, place

- Your modified `ImageLibrary` folder (don't submit the executable applications!);
- A folder named `Scripts` where you put your `bash files`;
- your report as a `.pdf` file.

5.2 To verify before you submit the assignment

The graders have to be able to run your code "as is" on their own virtual machine. In particular, they should not have to make changes/fixes to your code for it to build or run.

Important note on the C program: Your C program *must* build with no errors. If there are compiler errors, then you will get a grade of 0 for the code and execution sections of the grading.

Relative paths: Make sure that your C code and `bash` script don't contain any hard-coded paths. To verify this, create a new folder on your home directory, make a copy of the C source file and `bash` script there, rebuild your C program, and verify that the script runs fine in the new location. It is *your* job to make this kind of verifications before you submit an assignment.

5.3 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 program and script. Is it possible to make them either crash or fail?

5.4 Grading

- The C program follows the specifications: 20 points
- The C program performs the task specified (execution): 30 points
- the `bash` scripts performs the task specified (execution): 20 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, etc.): 15 points
- Report: 10 points
- Folder organization: 5 points