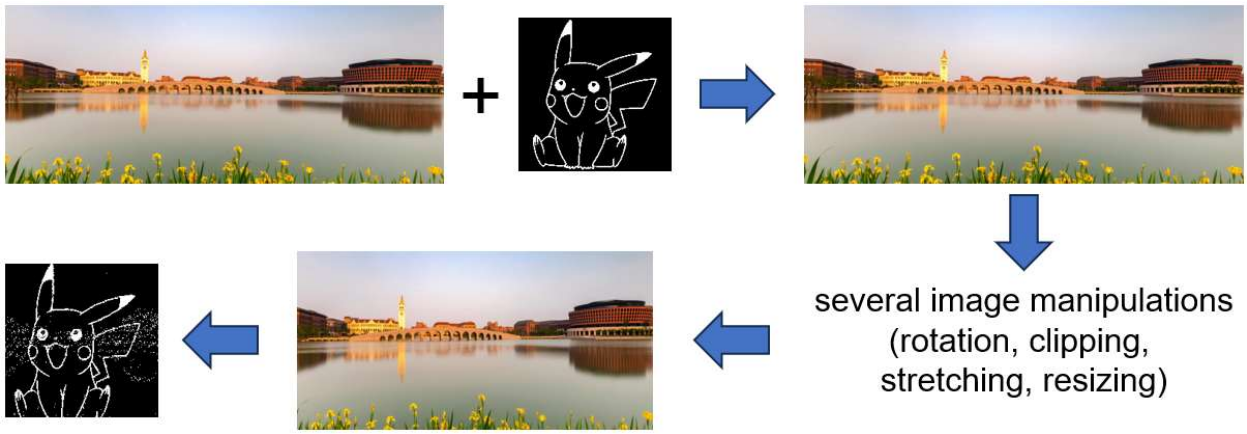
ECE220: Computer Systems & Programming **Spring 2024 ZJUI**
Machine Problem 6 **due: Saturday 6 April, 11:59:59 p.m.**

Using Arrays for Steganography

Your task this week is to implement functions for extracting, adding, and reconstructing watermarks from images. A watermark refers to the imprinting of a creator-specific mark on a piece of paper (originally using water) so as to provide evidence of the origin of the paper. Watermarks are used with many paper currencies, for example, to make counterfeiting the currency more difficult.

In digital data, watermarks are integrated into the data so as to make them difficult to remove without destroying the data, and possibly in a way that a user of the data does not realize that the watermark is present. In this lab, we take the latter approach, applying an effectively invisible watermark to an image in a way that is reasonably robust to manipulations of the image. In the example below, the photograph on the upper left was put on the web by John MacLean, one of the ZJUI Rhetoric professors. Adding a Pikachu watermark produces the image on the upper right—can you see Pikachu? The watermarked image was then rotated, clipped, stretch, and resized to produce the image in the middle of the bottom—look carefully and you'll notice the differences. The watermark was then reconstructed from the image, making it clear that the modified image was taken from our original.

The objective for this week is for you to gain some experience with arrays in C. The task is pretty easy once you understand arrays, requiring only about 30 lines of C code.



Background

An image is composed of an array of pixels. Each pixel has three bytes of information in the RGB format and one byte of transparency referred to as alpha. The data are separated into four arrays called channels, each of which contains either the red, green, blue, or alpha components for each pixel. In this MP, your code need only make use of the red channel, in which each pixel is represented as a value from 0 to 255.

The watermarks in this assignment consist of black-and-white images—you may recognize the characters used in the example file. Since each image requires only one bit per pixel, we have combined eight distinct watermarks into a single file using the 8 bits of each byte in the red channel (actually, we could use the green, blue, and alpha channels to hold distinct images, but refrain from adding that complexity here). In other words, the high bit (bit7, value 128) of each pixel forms one watermark, the next bit (bit 6, value 64) forms a separate watermark, and so forth, for a total of eight.

Applying a watermark to an image means scaling the watermark vertically and horizontally to cover most of the image, then replacing the low bit of the image's red channel pixels with the one-bit watermark. Reconstructing a watermark can then be done by counting the number of 1s in bit 0 in the region corresponding to a pixel in the scaled watermark, converting to 0 or 1 based on whether half or more of the bits are 1, and producing the reconstructed watermark as a black-and-white image.

The Image Arrays

Your functions will be given a single array corresponding to the red channel in the input image and a second array corresponding to the red channel in the output image. These arrays will be passed as pointers to 8-bit pixels (that is, as `uint8_t*`). However, they represent two-dimensional arrays of specified `height` and `width` (these values are also provided to your functions). To make use of the arrays, you must calculate the appropriate offset into the one-dimensional C arrays based on the two indices of the pixel that you want to access. Recall from class that you should make use of the expression `x + y * width` to access the pixel at (x,y). For example, given an array `uint8_t* data` for the red channel, you can access the (x,y) pixel in the array with the expression `data[x + y * width]`.

Pieces

data + x + y * width

In addition to code, we have provided a file containing eight watermarks and a few sample images in the **`Images`** subdirectory. The **`Examples`** subdirectory contains a few images processed by the "gold" version of the program (the version that I write before asking you to do the assignment). The images' names tell you the arguments to pass when you run the program.

There is more code in the package this time, but you still need to look at only three files:

`mp6.h`	This header file provides function declarations and descriptions of the functions that you must write for this assignment.
`mp6.c`	The source file for your functions. Function headers for all functions are provided to help you get started.

Other files provided to you include

`mp6main.c`	A source file that interprets commands and calls your functions.
`Makefile`	A file that describes how to build your program, allowing you to type "`make`" instead of re-typing the full compiler command every time.
`imageData.h` `imageData.c`	Some utility functions written by a former TA for the class.
`lodepng.h` `lodepng.c`	A package for loading and storing PNG images.

You need not read any of these, although you are welcome to do so.

Details

You should read the descriptions of the functions in the header file and peruse the function headers in the source file before you begin coding. Here are the tasks that you need to complete for this assignment:

1. Write a function to extract one watermark from an image containing 8 watermarks (see **`Images/watermarks.png`**).

2. Write a function to apply a watermark to another image by scaling the watermark and copying the watermark's red channel bit 0 into the image's red channel.
3. Write a function to reconstruct a watermark of a given size by counting the number of 1s in the bit 0 position over the region corresponding to each scaled watermark pixel.

All three functions must be written in **`mp6.c`**. Function signatures for each appear in **`mp6.h`** so that **`mp6main.c`** can call your functions as necessary.

Start by completing the first function in the **`mp6.c`** file, **`extract_watermark`**. The function header in the code provides a complete description of the arguments. For each pixel in the original image, you must examine the bit numbered **`bit`** (another parameter) and write either a 0 or a 255 (depending on whether the data bit is 0 or 1, respectively) into the same pixel in the output image. Once your function is complete, you can split the **`Images/watermarks.png`** image into eight separate watermark images using MP6 command 1 (see "The Interface" below). The results should be recognizable images, even if you do not know the source of the art.

Next, you must implement **`add_watermark`**, which applies a watermark to a given image. Two images are passed to this function: the red channel for an image to which you must apply the watermark, and a watermark image. The latter is the output from your **`extract_watermark`** function, so until you have finished that function, you should not expect **`add_watermark`** to receive reasonable inputs.

Generally, in this assignment, **the image being watermarked must be both wider and taller than the watermark image**. We will not test with smaller images, and the behavior of the functions under such conditions need not be well-defined. Preferably, the image should be several times the watermark size in each dimension.

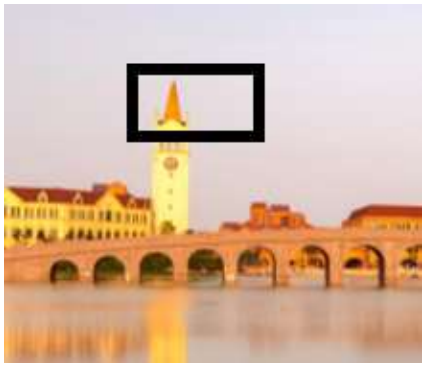
When applying the watermark, you should first calculate the horizontal and vertical scaling factors for the watermark using integer arithmetic:

```
h_scale = width / markwidth
```
and
```
v_scale = height / markheight.
```

Each bit (pixel) in the watermark should then be used for a rectangular group of pixels in the image being watermarked. The rectangle should be **`h_scale`** pixels wide and **`v_scale`** pixels high. Unless the image being watermarked happens to have a width/height that is a multiple of the watermark's width/height, **some pixels at the right and bottom edges of the image will not correspond to a watermark pixel**. **Set the watermark value for these pixels to 0.**

Once your function is complete, you can try applying watermarks to the sample images in the Images directory using MP6 command 2 (see "The Interface" below). Your output should match the examples in the **`Examples`** directory exactly (using **`diff`**). The watermarks will not generally be easy to spot, but you could do an image difference (find a tool on your own for that purpose) with the original to highlight the differences.

Finally, you must write the **`reconstruct_watermark`** function that rebuilds a watermark from a watermarked image. Again, you should first compute the horizontal and vertical scale factors of the image relative to the watermark. Then imagine covering the image with rectangles of the size given by the two scaling factors. Consider the piece of the image to the right, in which a rectangle corresponding to a particular watermark pixel has been highlighted (just as an example illustration—not precise). Your code must find the number of 1s in bit 0 over all pixels in each such rectangle. If the number is equal or greater than half of the total number of pixels



in the rectangle (the product of the scale factors), you should write a 255 into the corresponding watermark pixel. If not, write a 0 into the corresponding watermark pixel.

Again, once your function is complete, you can try reconstructing watermarks from watermarked images using MP6 command 3 (see "The Interface" below). Several files are provided in the **`Examples`** directory—these should match those in **`Images/watermarks.png`** exactly, and a reformatted copy of a watermarked campus image is in the **`Images`** directory, with the somewhat mangled watermark output that should be produced in the **`Examples`** directory).

The Interface

When you have completed one or more of the functions, you can type "**`make`**" (no quotes) to compile your code into an "**`mp6`**" executable program. The make program shows you the compiler commands that it executes on your behalf, so you can see what is happening and will receive any syntax errors or other errors or warnings from the compiler. Fix any warnings!

Once you have compiled your program, you can use the interface built into **`mp6main.c`** to execute your functions. A set of images has been provided to you in the **`Images`** subdirectory along with several copies processed by a correct version of the program in the **`Examples`** subdirectory (take a look).

The **`mp6`** program takes five or six command line arguments:

```
./mp6 <input file> <output file> <command #> …
```

The commands are the same as the task numbers above, and detail on additional arguments for each command can be obtained by running "**`./mp6`**" with no arguments.

Specifics

- Your code must be written in C and must be contained in the **`mp6.c`** file provided to you. We will NOT grade files with any other names. If your code does not work without modifications to other files, you are likely to receive no credit.
- You must implement the **`extract_watermark`**, **`add_watermark`**, and **`reconstruct_watermark`** functions correctly.
- Your code must be well-commented. Follow the commenting style of the code examples provided in class and in the textbook.

Grading Rubric

We put a fair amount of emphasis on style and clarity in this class, as reflected in the rubric below.

Functionality (85%)

- 15% - **`extract_watermark`** works correctly.
- 35% - **`add_watermark`** works correctly.
- 35% - **`reconstruct_watermark`** works correctly.

Comments, Clarity, and Write-up (15%)

- 5% - introductory paragraph explaining what you did (even if it's just the required work)
- 10% - code is clear and well-commented, and compilation generates no warnings
  (note: any warning means 0 points here)

Note that some categories in the rubric may depend on other categories and/or criteria. For example, if you code does not compile, you will receive no functionality points. If your **`extract_watermark`** function fails, **`add_watermark`** may also fail.