Homework

19

Pointers & Graphics

You likely have used some sort of image-editing software, whether it is Photoshop, Illustrator, the GIMP or an online image editing tool. This assignment, will show you how to implement editing operations on PNG and JPG graphics files.[1] As a demonstration, convert an image to its negative values.

In addition to getting more practice working with pointers, you'll also get exposure to using external C open-source libraries. Go ahead and get started by using get-starters to upload the starter code to your workspace. Type **`make run`** to run the sample code. You'll find the code that runs at the end of **`h19.cpp`**. Let's take a look at the code.

Loading and Writing Images

C++ itself doesn't have any built-in support for images, graphics or user-interfaces. All of those capabilities are added using libraries. We're going to use the ***`stb_image`*** and ***`stb_image_write`*** libraries, written by Sean T. Barret (the stb) and placed into the public domain. These libraries provide the ability to read and write several different image formats, in several different ways. Both are C libraries instead of C++ libraries.

- I have compiled and linked both libraries into **`libh19.a`** for this assignment
- **`h19.h`** includes prototypes for the library functions we are using. Note that these prototypes are wrapped in an **`extern "C" { };`** block to ensure our C++ programs can link to them. If we left this off, they wouldn't be found.
- If you want to use more of the functions, follow the documentation link.

The **`run()`** function at the bottom of **`h19.cpp`** shows several examples of opening different files and then writing out different versions. Let's look at that before we go on.

---

[1] Adapted from the ImageShop assignment, Stanford CS106A, Spring 2017

Loading a JPEG Image

```
// 1. Load a jpg file using 4 bytes per pixel (bpp RGBA)
int width, height, bpp, channels = 4;
unsigned char * paris =
    stbi_load("images/paris.jpg",       // input file
              &width, &height, &bpp,    // pointers (out)
              channels);                // channels (in)
```

This code loads an image from your disk into memory.

- The ***stbi_load()*** function returns a **pointer** to the first byte of the image in memory. The type of the pointer is an **unsigned char**, which, in C++ speak means "raw" byte. If loading fails, then the function returns **0**.
- The first argument to the function is the **path to the file.** This can be absolute or relative (as used here), but it must be a **C-style string**.
- The next three arguments are the **address of the width**, **height**, and **bytes-per-pixel** of the image. These are **output parameters**; that means that we create the variables, pass **their addresses** as arguments, and the function will **fill them in**. The information flows **out of the function**, not into it.
- The last argument is an **input** parameter telling the ***Load()*** function how to interpret the image. Here we're telling it to provide us with ***4*** channels of input (even though the original image only has ***3-RBG***).

Saving an Image

The image saved does not need to be in the same format as the image read. For instance, **JPEG** doesn't have transparent colors, but I can write the image back out as a **PNG**, which does. The library has different functions for each image type.

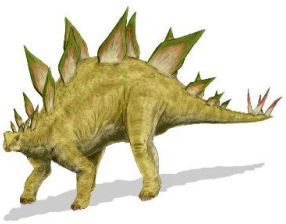
```
// Now write it out in current folder as a 4-bytes-per-pixel PNG
stbi_write_png("paris.png", width, height, channels, paris,
               width * channels);
```

The first five arguments are the same for each type of output (although you call a different function name for each). Notice how we're using the variables that were initialized by the previous function call. The last argument here is unique to **PNG** files. It tells the function at what byte the next row begins.

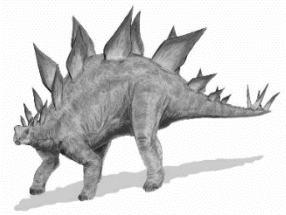
Freeing the Memory

The ***stbi_load()*** function returns a pointer, but inside that function it asks the operating system to allocate enough memory to hold the image that it loads from disk. This memory is **on the heap**, which met in this chapter. In the C programming language, you have to remember to **free** that memory before your program ends. We do that by using the function ***stbi_image_free(paris)***.

Changing the Format



Although the previous example added an alpha channel to the Par picture, it didn't really change how it looked. Our second example loads this 4-color **PNG** image as a 1-channel (that is, grayscale) image. We'll then save it as a **BMP** which is the native format for Windows applications.



Here's the code that that does this.

```
// 2. Load a png file using 1 byte per pixel (Gray scale)
channels = 1;
unsigned char * stego = stbi_load("images/stegosaurus.png",
                                  &width, &height, &bpp, channels);
stbi_write_bmp("stego-bw.bmp", width, height, channels, stego);
stbi_image_free(stego);
```

The first example in ***run()*** changes a **JPEG** into a **PNG**, but the last example goes the other way, removing the alpha channel from a picture and saving it as a **JPEG**. The ***stb_write_jpg()*** function also takes one extra argument, which is the quality of the resulting image. This can go from **0-100**, where **100** has the highest quality.

Your Turn

Open **h19.h** to see the prototype for the **negative()** function. The function converts the color of each pixel in the image to its inverse (like a film negative). This is a **void** function that will modify the image passed as the first parameter. I have already stubbed out the code for you, so you can run **make test** and see it compile.

You'll also see three extra folders: **input** which contains the photos we're going to start with, **expected**, which contains the photos as they should look, and **actual**, which contains the photos after your filters have been applied. If your code fails one of the tests, the **actual** folder will also have a rudimentary "diff" image you can examine. You can look at any of the photos just by double-clicking them in the IDE.

Negative Energy

In this function you should change the image, passed as a **pointer** to **`unsigned char`**, into one whose pixels are the **inverse** of those in the original image. There is nothing to return from the function.

To convert an image to its inverse, for each pixel, set all three of its ***red***, ***green***, and ***blue*** values to be the inverse of their current color value. The inverse of a color value ***k*** is defined as ***255 - k***: the pixel (**110**, **52**, **236**) has an inverse of (**145**, **203**, **19**).

smiley-face.png *smiley-face-negative.png*

Moving Pointers

Look at the declaration for the **`negative()`** function:

```
 */
void negative(unsigned char * const img, int width, int height);
```

Notice the word **`const`** in the declaration of **`img`**. When the word **`const`** comes **after the star**, it means that value inside the pointer cannot be moved; you **cannot make it point to a different location**. Why would the function want to prevent you from changing the value in the pointer? Because after the function is finished, the block of memory containing the image **must be returned to the operating system**. If your function was allowed to change that, **the program would probably crash**.

So, if **`img`** is a **`const`** pointer, then what should you do? Just **create a second pointer**, one that is not constant, that points to the same location, like this:

```
void negative(unsigned char * const img, int width, int height)
{
    unsigned char * p = img;    // can use p to change image
```

Now, you can **invert the first pixel** by using the following algorithm:

```
*p <- 255 - *p       -> Invert the red component
p++                  -> Move p to next component
*p <- 255 - *p       -> Invert the green component
p++                  -> Move p to next component
*p <- 255 - *p       -> Invert the blue component
p++                  -> Move p,
p++                  -> skipping the alpha channel
```

How do you change the rest of the pixels? You need some type of loop. The best kind of loop for this is an **iterator loop**. Create a **second pointer** named **end**, one that points to the byte right past the end of the image, you could use a loop that looks like this:

```
while (p != end)
{
    // Process element p points to
    p++;
}
```

The question is, **how do you get** this pointer variable named **end**?

Pointer Arithmetic

Applying the + and - operators to pointers is pointer arithmetic. Adding an integer to a pointer gives us a new address value. For each unit that is added to a pointer value, the internal numeric value must be increased by the size of the base type of the pointer. In our case, the base type of the pointer, (**unsigned char**) is 1. Here's the pseudocode:

```
Let p point to beginning of the image (p = img)
Let end be img + width * height * BPP (BPP given as 4)
While p != end
   Invert each component and increment as above
```

You can use **auto** (type inference) to declare the variable **end**, so you don't have to manually write out the variable's type. Even though **img** is a **const** pointer, **end** is not be **const** (just like **p** is not **const**). Make sure that **end** cannot be inadvertently moved, by adding an additional **const**.

Once you've added these changes, run **make test** and make submit to turn it in. If you have problems, shout out on Piazza or come by my office hours.