Homework

22

Images & Structures

Let's write two more "image processing" functions. In the last two assignments, we wrote functions that looked at every pixel and changed it is some way: inverting the colors, changing the transparency or combining pixels from two images into one.

This time, we're going to move the pixels themselves by using structures.

- `flip()` will swap the pixels from left-to-right, or top-to-bottom so that the image is flipped.
- `mirror()` copies the pixels on the right half to the left half, the left-half to the right half, the top to the bottom half, or the bottom half to the top.

These are `void` functions that modify the passed image. The functions are already stubbed out so you can run `make test` and see that both compile and run.

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 `diff` 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.

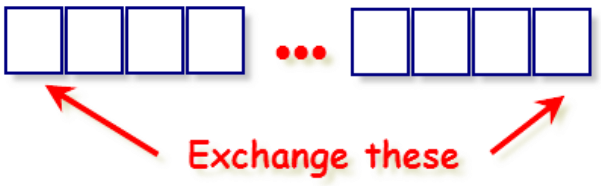
Images & Structures

In H20 and H21, you used pointers manipulate images. Now, suppose you want to **flip an image** horizontally like this:

smiley-face.png

smiley-face-flip-horizontal.png

To horizontally flip an image, you need to take the pixel at the end of each row, and **exchange it** with the pixel at the beginning.



But, you don't have a pixel; you have a portion of a pixel. In the picture above, **p** points to the **red** part of the first pixel, while **end** points to the **alpha** (transparent) component of the last pixel on the row. While we could get around this with some **if** statements and much extra pointer movement, it's really complicated.

Instead, if we could process each element as a pixel, instead of a pixel component, things would be much easier. Fortunately, pointers and structures make that possible.

Type Alias, Structure & Enumeration

Open **h22.h** and you'll find the following definitions:

```
using UC = unsigned char;
struct Pixel {UC red{0}, green{0}, blue{0}, alpha{255};};
enum class Direction {LtoR, RtoL, TtoB, BtoT};
```

- The name **UC** is a type alias; shorthand for **unsigned char**.
- The **struct Pixel** is a user-defined type that has a member for each color.
- The enumeration **Direction** will be used to determine whether you flip or mirror from left-to-right (**LtoR**), right-to-left (**RtoL**), top-to-bottom, (**TtoB**) or bottom-to-top (**BtoT**).

The flip Function

Copy the prototype for **flip()**, from **h22.h** to the implementation file, remove the semicolon, and add a body. Start with this code (where **BPP** is **4**):

```
for (int row = 0; row < height; ++row)
{
    Pixel * front = img + row * width * BPP;
    Pixel * back = front + width - 1; // last pixel
. . .
```

This is similar to the code you've written in the last few assignments. The first byte in of any row is at the address **img + row * width * BPP**. When **row** is **0**, then the beginning

is the starting address of **img**, plus **0**. When **row** is **1**, the beginning of the row is the starting address of **img**, plus the number of bytes in one row. For the third **row** it's twice the width of a row plus the starting address, and so on.

Introducing reinterpret_cast

When you compile your code, you'll get the following error:

```
error: cannot convert 'aka unsigned char*' to 'Pixel*' in
initialization
```

The pointer **front** is a pointer to **Pixel**, while the value calculated on the right is a pointer to **unsigned char**. These types are not the same, so the compiler requires you to explicitly convert them.

You cannot use **static_cast** for this; instead, you use a **reinterpret_cast**, which does not produce a new value, but changes the way that a pointer interprets the address it points to. Change the line that initializes **front** to:

```
Pixel * front = reinterpret_cast<Pixel*>(img) + row * width;
```

Now, since **front** is a **Pixel** pointer (and not a byte pointer, like **img**), when you add **width** to **front** you don't need to multiply **width** times **BPP**, since each unit is automatically the size of a **Pixel**.

Also, unlike the **end** pointer in **negative()**, the **back** pointer in **flip()** points to the last element in the row, not past it.

Now **front** will treat the values it finds as **Pixel** objects instead of **unsigned char**.

Note: using reinterpret_cast is machine-dependent[1] and only allowed under certain circumstances. In this case it works because a pointer to a structure is a pointer to the first member of the structure, and the types of the variables (unsigned char) are the same.

[1] http://en.cppreference.com/w/cpp/language/reinterpret_cast

Horizontal Flip

To complete, use the following algorithm if the **Direction** is either **LtoR** or **RtoL**.

```
For each row from 0 to height-1
   Let front to point the beginning of the row
   Let back to point to the last element in the row
   While front < back Do
      temp <- *front     // "swap" algorithm
      *front <- *back
      *back <- temp
      Increment front
      Decrement back
```

Vertical Flip

For the other two directions (**TtoB** and **BtoT**), the algorithm is similar:

```
For each col from 0 to width-1
   Let top to point the first pixel in the column
   Let bottom to point to the last pixel in the column
   While top < bottom Do
      temp <- *top     // "swap" algorithm
      *top <- *bottom
      *bottom <- temp
      top <- top + width
      bottom <- bottom - width
```

Here are the parts that are different:

1. Initialize **top** with **reinterpret_cast<Pixel *>(img) + col**
2. Initialize **bottom** with **top + width * (height - 1)**
3. Note that the decrement and increment are by **width**, not by **1**.

Try to work out the arithmetic on paper, instead of simply taking my word for it.

Go ahead and finish **flip()** by following the pseudocode from earlier in this section. You should be able to **make test**, and the first set of tests should pass. As always, if you run into problems, bring your questions to Piazza or come to my office hour.

The `mirror()` function

The **`mirror()`** function is almost exactly the same as **`flip()`** with these differences:

1. Copy the loops from **`flip()`**. Don't create **`temp`** and don't swap.
    a. If **`dir`** is **`LtoR`**, copy **`*front`** to **`*back`**.
    b. If **`dir`** is **`RtoL`**, copy **`*back`** to **`*front`**
2. In the second loop, copy **`*top`** to **`*bottom`** if **`TtoB`** and vice-versa otherwise.

Go ahead and finish **`mirror()`**. If you **`make test`**, all of the tests should pass.

Be sure to **`make submit`** to turn in your code for credit before the deadline. As always, if you run into problems, bring your questions to Piazza or come to my office hour.