CSE 232 Spring 2016

Programming Project #8

Assignment Overview:

Perform some image processing on existing image files. More experience with classes and 2D vectors

The assignment is worth 50 points (5% of the course grades), and must be completed before 11:59 pm on Monday, March 28th

Background:

Image processing is an interesting but typically complicated process. Even reading in and writing out images can be a chore without library support.

However, there are some simple image representation formats that are text files and with which we can do some simple manipulation.

PNM files

There is a class of files called Netpbm for portable any format (PNM) files that can represent an image as a text file where each individual pixel is coded in the file. See https://en.wikipedia.org/wiki/Netpbm_format for more details. They include:

- pbm files which are used to represent black and white images
- pgm files which are used to represent graymap images
- ppm files which are used to represent color images

The advantages to these kinds of files are their portability and easy access for image manipulation. Their clear downside is their size. These files can get very large and many other image formats work with ways to reduce the size of image files while maintaining their quality

PGM files

We are going to work with PGM files, graymap files. In a graymap file, each pixel is represented by a number from 0 to 255 (allowing the number to be stored in a single byte). The value 0 represents pure black, the value 255 pure white and the integer values in between represent shades of gray. If you ever had a black-and-white television, this was the kind of image it displayed. Most OS support PGM files pretty directly so viewing them is not really very hard.

Be careful that you work with a PGM ascii file. There is a binary file format as well that you cannot easily edit.

A PGM file is a text file with a fairly simple format (see https://en.wikipedia.org/wiki/Netpbm_format#PGM_example for details):

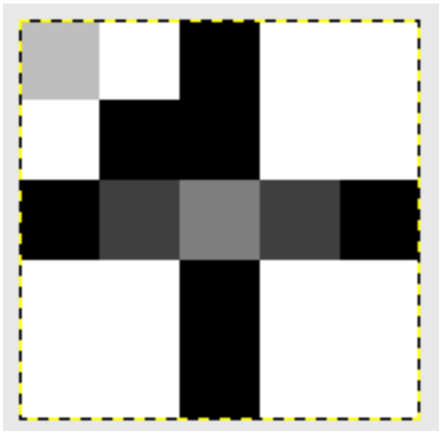
- a "magic number" which is always "P2"
    - a "P5" magic number is the non-ascii version. Don't use one of those!
- two integers, space separated, that represent respectively the `x_size` and `y_size`, the dimensions, of the image
- another integer, also space separated, the `max_gray` that represents the maximum gray value in the image

- what follows is, for each row a series of space separated integers in the range 0 to max_gray.
    - there are `x_size` values in the row
    - there are `y_size` rows
- comments can occur anywhere in the file. If so, then the first character of that line is a '#'.
    - clearly, comments are ignored.
- separators can be white space, line feeds, stuff that is normal for C++ input.
- I've seen recommendations that no line should be more than 70 chars wide, but I've personally written out pgm images that are 512x512 and wrote each row as a single line without any issue. YMMV.

Here is an example.

```
P2
# a test
# another
5 5
245
183  245    0  245 245
245    0    0  245 245
0     61  122   61   0
245  245    0  245 245
245  245    0  245 245
```

It is a very small image (only 5x5 pixels) but if you blow it up it looks like:



On the lab machines and X2Go, you can just double click on these images and see what they look like. The default image viewer you get allows things like zoom etc.

Convolution, see https://en.wikipedia.org/wiki/Kernel_(image_processing) for details
Convolution, at least for images, creates a new pixel based on some combination of values from its near neighbors and itself. This operation is performed for every pixel in the image.

When you do a convolution, you generate a new image from the old image based on the operations described below

These operations are represented by a small matrix, called the kernel or convolution mask, that represents these neighbor operations. For each pixel, you multiply each old pixel's neighbor by its mask value and sum all those values up for the new pixel value. For example, let's take our simple image and apply a sharpen mask to the middle pixel with the value 122 (bold, italics, underlined). We want to calculate the new value for that pixel in a modified image. We have indicated the mask area in the original image in green.

```
183   245     0   245 245
245     0     0   245 245        0 -1   0
0      61   122    61   0    *  -1   5 -1 → 494 → 245
245   245     0   245 245        0 -1   0
245   245     0   245 245
```

```
0*0 + -1*0 + 0*245 + -1*61 + 5*122 + -1*61 + 0*245 + -1*0 + 0*245
= 494
```

The new pixel in the new image at the same location of the old pixel will have the value 494. But wait! 494 is larger than the `max_gray` we indicated in the file. So that new pixel's value gets reset to the `max_gray`, 245. If the new pixel had been less than 0, we would have set it to 0.

We do this for every pixel, calculating a new pixel value based on the old pixel's neighbors and a convolution mask. The result is a new image. You can imagine sliding the mask along the original image one pixel at a time, doing the calculation and then generating a new pixel in a new image.

A great web site that demonstrates this process is:
http://setosa.io/ev/image-kernels/ Take a look!

The edges
What to do on the edges of an image? Pixels on an edge do not have the required number of neighbors to make the calculation. We have choices on how to handle this:

- shrink the image, so that we only calculate new pixels that have the required number of neighbors.
- wrap the image, so that values on the opposite edge fill in.
- just treat the missing neighbors as 0.

We will do the latter. We will treat missing neighbors as 0. For example, for calculating the new pixel for 183 (where the bolded values of 0 are the image value for missing neighbors)

```
183   245     0   245 245
245     0     0   245 245        0 -1   0
0      61   122    61   0    *  -1   5 -1 → 425 → 245
245   245     0   245 245        0 -1   0
245   245     0   245 245
```

```
0*0 + -1*0 + 0*0 + -1*0 + 5*183 + -1*245 + 0*0 + -1*245 + 0*0
= 425
```

Problem Statement
You are going to create an `Image` class that will allow you to read, write and convolve (in various ways) a PGM image. We are going to write 4 filters and convolve images using those filters.

Program Specifications:
You will be provided with a `main-08.cpp` file and a `class-08.h` file. You must provide the `class-08.cpp` file. I will also provide some sample PGM images, but you can find them on the web yourself. There are applications that will convert from and to PGM. On the lab machines and X2Go server the command line `convert` will do the trick. Give it a try!

Image class

- `Image()` : default constructor is in the header and takes the C++ default
- `Image(string f_name)` : constructor, reads in the PGM file into the class instance. It:
    - o sets the `max_val_`, `height_` and `width_` given in the file
    - o it then reads in every individual pixel value into the `vector<vector<long>> v_` More on that in the notes section.
- `Image(Image& new_img)` : copy constructor. Strictly speaking it really doesn't have to be implemented by you but it's good practice, so go ahead and do it (it's easy).
- `void write_image(string f_name)` : method, writes out the contents of the class instance into the given file as a properly configured PGM file (if you write it, you should be able to read it back in).
- `void convolve(Image& new_image, vector<vector<long>> mask, long w, long h, long div, long whiten)` : method. Note that `div` and `whiten` have defaults in the header file which **should not** be provided in the class cpp file. This is the guts of the whole thing. This takes in a `new_image` (the one to that you write into) and a `mask` which is the convolution mask. `w` and `h` are the dimensions of that mask. You apply the mask to the old image, setting the new pixels in the new image by passing the mask over all the pixels in the old image (the one the `this` pointer points to) and doing the calculation as described. Since the new_image is passed by reference, it is updated in the caller.
    - if any new pixel calculation is greater than `max_val_`, that pixel is set to `max_val_` in the new image.
    - if any new pixel calculation is less than 0, that pixel is set to 0 in the new image
    - for each pixel calculation, the total value of the pixel calculation is divided by the parameter `div` (see the blur method).
    - for each pixel calculation, the parameter `whiten` is added the total value of the pixel calculation (see the edge_detect method).
    - as stated, unavailable neighbors are assumed to be 0 for edge pixels.
- `Image sharpen()` : method. Applies the 3x3 mask below to create the new image. Calls `convolve`. When calling `convolve`, it takes defaults on `div` and `whiten`. It returns a new `Image`.

```
 0  -1   0
-1   5  -1
 0  -1   0
```

- `Image edge_detect()`: method. Applies the 3x3 mask below to create the new image. Calls `convolve`. When calling `convolve`, it takes defaults on `div` but the image tends to be dark so it provides a `whiten` to brighten the resulting `Image`. The examples used `whiten=50.` It returns a new `Image`.

```
0   1  0
1  -4  1
0   1  0
```

- `Image blur()` : method. Applies the 3x3 mask below to create the new image. Calls `convolve`. When calling `convolve`, it takes defaults on `whiten` but mask always overshoots the `max_val_.` To compensate we provide a `div=9` (to normalize the result, averaging the pixel across its 8 neighbors).. It returns a new Image.

```
1   1  1
1   1  1
1   1  1
```

- `Image emboss()` : method. Applies the 3x3 mask below to create the new image. Calls convolve. When calling `convolve`, it takes defaults on `div` and `whiten.` It returns a new `Image`.

```
-2  -1   0
-1   1   1
 0   1   2
```

Assignment Deliverables:

`class-08.cpp` is what is required. None of the other provided files should be modified.

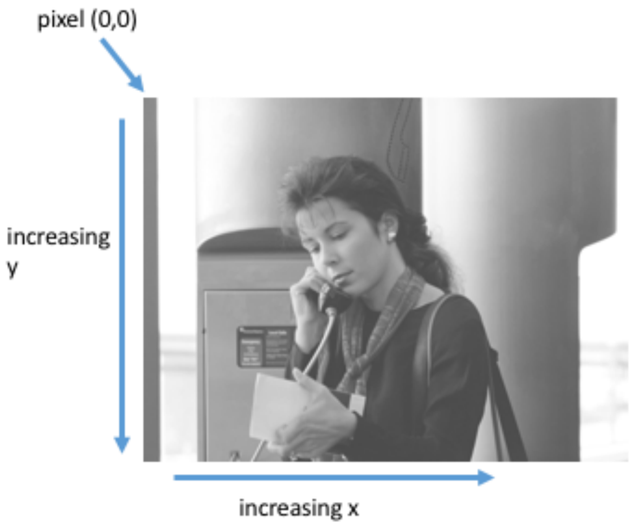
Please be sure to use the specified file name, and to preserve the file in your account until your assignment has been graded. You will electronically submit a copy of the file using the "handin" program.

Assignment notes:

1 **Calling convolve** When a filter calls convolve it:
- creates a vector<vector<long>> mask and you just assign the values as shown
- creates the new image to pass in. It is probably convenient to use the copy constructor to make that new image.
- calls convolve(new_im, mask, 3,3) could also have div and whiten

2 **2D Vectors and Images** This is discussed in the slides, but the coordinate system can get a little wonky and you have to keep track. Let's use the `vector<vector<long>> v_` , which contains the grayscale image pixels. You would:

- create a `vector<long> temp`, make sure you `temp.clear()` before each use, and `push_back` each long from a row onto `temp`. You know how big a row is so you know how many to read.
- `temp` now has one row of values, you `push_back` the whole row onto `v_`.
- you do that for all the rows, and you know how many rows there are.
- you can now talk about a particular pixel at v_[0][0] (or any other legal index), but where is that in the image? That location is **top,left** of the image. Furthermore, the indexing is actually `v_[y][x]` (not [x][y]) as you might expect) because `v_[y]` is actually a whole row. Furthermore, as y **grows you go down** the image and as x grows you go to the right.


pixel (0,0)
increasing y
increasing x