CSE 232 Spring 2019

Programming Project #8

Assignment Overview:

Perform some image processing on existing image files. First experience with C++ structs/classes and more work with 2D vectors

The assignment is worth 60 points (6% of the course grades), and must be completed before 11:59 pm on Monday, March 25th

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):

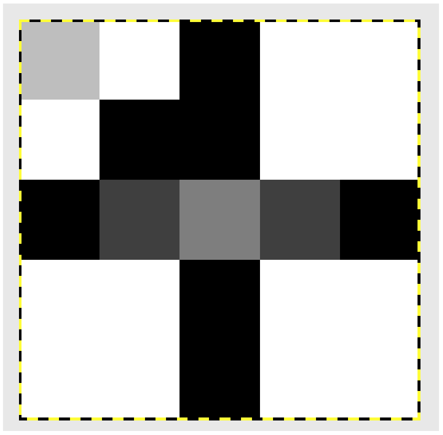
- a "magic number" which is always "P2"
    - a "P5" magic number is the non-ascii version. Don't use one of those!
- two integers, space separated, that represent respectively the `x_size` and `y_size`, the dimensions, of the image

- 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. For this project, each row of the image should be written on a single line.

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:



If you want to view these on the lab machines, go to the start menu, all apps and open **paint.net**. Then us the open menu on paint.net to open the file you wish to look at. Remember, these 5x5 files are very, very small. Some bigger files are provided in the directory for you to look at.

Convolution, see https://en.wikipedia.org/wiki/Kernel_(image_processing) for details

Convolution, at least for images, creates a new single 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 value by its mask value, including the old pixel itself, 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 (red) against the 8 neighbors (green). We want to calculate the new value for the 122 pixel in a modified image. To do so we use the mask matrix values and multiply the mask value with the appropriate neighbor, including the pixel itself. We sum them all up and that is the new value!

```
183   245     0   245 245
245     0     0   245 245        0 -1   0
0      61   122    61   0    *   -1   5 -1 ✉ 488 ✉ 245
245   245     0   245 245        0 -1   0
245   245     0   245 245
```

In row order of the mask, we get the equation:

```
0*0 + -1*0 + 0*245 + -1*61 + 5*122 + -1*61 + 0*245 + -1*0 + 0*245
= 488
```

The new pixel in the new image at the same location of the old pixel will have the value 488. But wait! 488 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 calculated 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 not on the image border
- 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) in row order

```
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
```

Again, this gets reset to 245, the max gray value.

Steganography

Embedding
We are also going to be doing a little bit of cryptography of a kind slightly different than project 5. We are going to hide information in an image. https://en.wikipedia.org/wiki/Steganography
We are going to take 2 PGM images, **plain** and **secret** and modify the original **plain** to hide **secret** within. The rules to modify **plain** are as follows:

1. We assume that **secret** is the same size or smaller than **plain.**
2. We also assume that **secret** (when extracted from **plain**) will be visible as a black and white image. That is, once **secret** is embedded what we can extract is a black and white image.
3. We examine each pixel in **secret** and modify the corresponding pixel (row and column) of **plain** as follows:
    a. If the corresponding pixel from the **secret** image is 0, then the parity of the corresponding **plain** pixel must be set to even
        i. If the **plain** pixel is already even, then it is not modified otherwise the **plain** pixel is updated to its original value – 1 so that it is even
    b. If the **secret** pixel is non-0, then the parity of the corresponding **plain** pixel must be set to odd
        i. If it's already odd, then the corresponding pixel of **plain** is not modified otherwise the **plain** pixel is updated to its original value + 1 so that it is odd.
        ii. Of course, we have to ensure that the `max_value` of the **plain** image is preserved, so if the **plain** pixel is already at `max_value`, then 1 is subtracted.

Here is an example **plain** image.

```
P2
# a plain image
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
```

And an example **secret** image.

```
P2
# a secret image
5 5
245
0      0  112     0    0
0    213    0     32   0
123    0    0     0   31
0    221    0    12    0
0      0  212     0    0
```

The result of embedding the **secret** image in the **encoded** image is

```
P2
# the embedded image
5 5
245
182  244    1  244 244
244    1    0  245 244
1     60  122   60   1
244  245    0  245 244
244  244    1  244 244
```

Extracting
We can extract the **secret** image embedded in the modified **plain** image fairly straightforwardly. We create a new image (let's call it **extracted**) as follows:

1. We create a new image with the column and row values of **plain**
2. For each pixel in **plain** we examine its parity:
    a. if the parity of the **plain** pixel is odd, we write the provided `max_value` as the corresponding pixel of the **extracted** image.
    b. if the parity of the **plain** pixel is even we write 0 as the corresponding pixel of the **extracted** image.

Note that some information is lost in this process. We create **extracted** as essentially a black and white value image (it has only two pixel values, 0 and `max_value` at each pixel) even though **secret** may have a broader set of values (more gray values) . Further, if **plain** was larger than **secret** there will be corresponding noise in the **extracted** image. Not random noise, but pixels set to values that are not relevant to **secret**.

```
P2
# an extracted image, max_val=200
5 5
200
0     0  200    0     0
0   200    0    200   0
200   0    0    0   200
0   200    0   200  0
0     0  200    0   0
```

Problem Statement
You are going to create an `Image struct` that will allow you to read, write, convolve (in various ways), embed and subsequently extract a PGM image.

Program Specifications:
You will be provided with a `proj08-struct.h` to start with but you can modify that file as long as the basic types for the listed function members are maintained. You must provide both the resulting `proj08-struct.cpp` and the `proj08-struct.h` file. For testing, to make this easier, we will work on Mimir with small files (like the 5x5 image file above) as well as some larger images. I will also provide some sample PGM images so you can see the results of your work better.

Image struct
- `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**.**
- `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 and view it using paint.net). The output is going to be compared via a **diff** so make sure that the output is **exactly** in the required format. (First line is the magic number, then 2 lines for the dimensions and max_val. After that, each row of pixels is on a newline, and there is a space after every pixel, including the last one.)
- `Image convolve(vector<vector<long>> mask,`
  `long divisor=1, long whiten=0)` : method.
  Note that `div` and `whiten` should have defaults in the header file but **should not** be provided in the class .cpp file (it's a compile error if you do). This is the guts of the whole thing. This creates in a `new_image` (the one that you write into) using the provided `mask` which is the convolution 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. The `new_image` is returned from the function.
    - 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 `divisor` (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 `divisor` but the image tends to be dark so it provides a `whiten` to brighten the resulting `Image`. Use `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 `divisor=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 `divisor` and `whiten.` It returns a new `Image`.

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

- `void embed(const Image & `**`secret`**`)` : method. Takes the **secret** image by const&, and embeds it in the **plain** image (the object on which it is called.)

- `Image extract(long max_val)` : method. This function returns an image which is extracted from the pixels in the calling object (referred to by the `this` pointer). The `max_val` of this image is set to `max_val`, and the dimensions are the same as the called image object. Values of pixels in the resulting image are either 0 or `max_val`

Assignment Deliverables:

There are two changes from normal.

First, We ask you to turn in ***two*** files

- `proj08-struct.h`
- `proj08-struct.cpp`

Note: you do **not** have to submit a folder for this project, (just 2 files).
We give you a `proj08-struct.h` to start with but you might well find that other methods of your own choosing should be included, or you might find that the starter `proj08-struct.h` does not suit your style. You can modify/create a .h file as you wish as long as the elements listed in the specification are provided. Thus when you turn this in you must provide both, even if the `.h` is just a copy of what we gave you to start with.

Second, no hidden test cases. That's because what the tests will be doing is comparing your `Image` (resulting from whatever operation is being tested) to our reference `Image`. We will do this with the Unix `diff` command. The `diff` command looks at two files and lists any differences (element by element) with the two files (The output file needs to match exactly). If everything worked, there should be no differences.

Remember to include your section, the date, project number and comments and you ***do not provide*** main.cpp. If you turn in a main with your code Mimir will not be able to grade you.

1. Please be sure to use the specified file names
2. Always a good idea to save a copy of your file in your H: drive on EGR.
3. Submit to Mimir as always. There will be a mix of visible and not-visible cases.

Assignment notes:

1. Get the reading and writing of Images done first (along with the constructors). Nothing is going to work until you get those two operations working.
2. Do your work locally on really small images that you can calculate by hand to make sure things are working. All the images, (both input and correct output) are provided so you can see what the images should look like.
3. The convolution masks, and potentially alternate values for `div` and `white`, are preset. Use them!
4. The `proj08-main.cpp` uses a different `main` than we are used to. You can look up `argc` and `argv` but the bottom line is this. When you create such a `main` you can provide arguments on the command line, which is very convenient. An example call is:

   ```
   ./a.out 4 sample.pgm
   ```

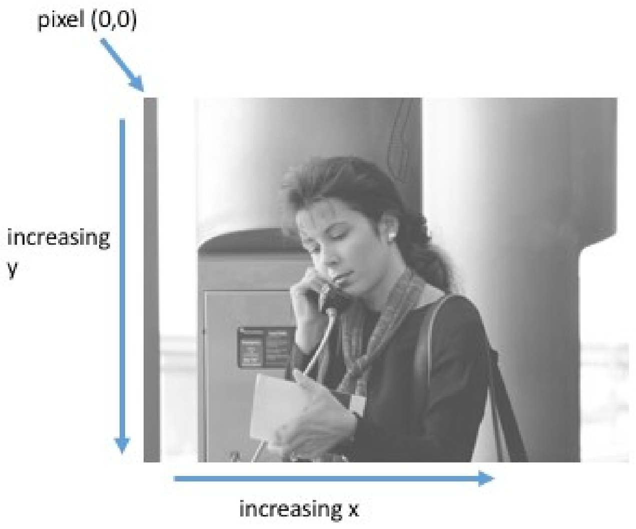
   where 4 is the case number (passed in as `argv[1])` and `sample.pgm` is the file being worked on (passed in as `argv[2])`.

```
./a.out 6 plain.pgm secret.pgm
```

calls the `main` for test 6, using `plain.pgm` as the plain image and `secret.pgm` as the secret image (look at the switch statement).

5. **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. and `v_[y][x]` is a particular column in that 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