UNIVERSITY OF CAPE TOWN
DEPARTMENT OF COMPUTER SCIENCE

CS3022H –Tutorial 4

Input

Mask

Output

Write an ***Image class*** which supports simple operations on pairs of *PGM* images. An image is a 2D array of numbers, each of which corresponds to the intensity of a pixel on the screen. So, essentially you will be writing a program that will manipulate 2D arrays of numbers. For any pair of "source" images, your program must support the following image operations, which generate a new output image:

1. Add I1 I2 : add the pixel values of I1 to I2 (i.e. at every corresponding 2D position you add the two values)
2. Subtract I1 I2 : subtract pixel values of I2 from I1
3. Invert I1 : replace each pixel value p with (255 - p) **NOTE: ONE image only**
4. Mask I1 I2 : given I1 and an image I2, copy across values from I1 where I2 has a value of 255. All other output values are set to 0. An example of 'masking' is shown above.
5. Threshold I1 f : for all pixels in I1 > f, set the result to the integer 255, otherwise set the value to 0. You can build a mask using this function.

The program will be driven from the command line, so you will need to use **argv** and **argc** in the main() function to process the parameters the user enters when running the program. The options are of the form (I1 and I2 represent image file names):

- -a I1 I2 (add I1 and I2)
- -s I1 I2 (subtract I2 from I1)

- -i I1 (invert I1)
- -l I1 I2 (mask I1 with I2)
- -t I1 f (threshold I1 with integer value f)

The command line form will thus be

imageops <option> *OutputImageName*

where **imageops** is the name of your program, <option> represents the ***single*** operation to be performed (see above) and *OutputImageName* is the name for the result image. Note that only one option should be specified when you run the program.

When implementing this assignment, you should create a class called **Image** to wrap your image operations and data. The image data will be a block of width x height **unsigned char** values.

Requirements:

The requirements are as follows:

1) Implement an appropriate constructor and destructor for your class. Use a std::unique_ptr<> to manage the space you have allocated. Remember you can use the **get()** method to obtain a raw pointer for data access.

2) Implement appropriate move/copy semantics for your **Image** class along with appropriate constructors and destructor.

3) Implement functions to support the image operations noted above, using operator overloading. Specifically

- + : addition of two images (I1 + I2)
- - : subtraction of two images (I1 − I2)
- ! :Invert an image (!I2)
- / :mask I1 with I2 (I1 / I2)
- *:threshold with f (int) (I1 * f)
- Implement I/O operators for **Image**: << and >>

Make sure you only allow “binary” operations on images with the same size.

4) Implement load()/save() methods for your Image class. More information on this is given in the notes below. You need only support simple PGM (1 byte) images which are a very simple image format. Both these methods will take a string argument which is the file to be opened or written to.

5) Image container: you must create an iterator for your **Image** class. This iterator should be used to gain access to the underlying image data and will support the * and ++/-- operators. Appropriate begin() and end() methods must be written for **Image** to return a reference to an iterator to which these operators can be applied. We have provided a code skeleton showing some features of this “nested class” – you must fill in the remaining functionality. Note

that it requires the use of **friend** to work correctly. **Your algorithms for image operations should be expressed in terms of these iterators.**

If you have implemented this correctly, you should be able to write code like the following:

```
void Image::copy(const Image& rhs)
{
  Image::iterator    beg = this->begin(), end = this->end();
  Image::iterator    inStart = rhs.begin(), inEnd  = rhs.end();


  while ( beg  !=  end) { *beg = *inStart;   ++beg; ++inStart; }
}
```

Notes:

- Image data is of type **unsigned char** (0...255)
- All result data must be clamped to this range i.e. < 0 is set to 0 and > 255 is set to 255.
- You must write a loop to extract the command line parameters first, this will be in main().
- You ***may not*** use a library to handle I/O for PGM files. You must write this code yourself.

This will be used to build the appropriate Image objects. Then you will need to perform the requested operation on the two input images and generate the named output image.

Image Class iterator

The iterator should be a nested class defined within Image. This means it lives within the “namespace” Image, and is thus logically associated. It is however, still separate class and cannot access the private members/data of Image. To get around this Image needs to be declared a “friend” of iterator. The following code snippet provides some of the structure for this class – you must fill in the rest and provide appropriate begin()/end() methods for Image to get the start and end iterators to the contained data. You do not need to provide support for “const_iterator”.

```
class Image
{
  private:
    int width, height;
    std::unique_ptr<unsigned char[]> data;
  public:
  class iterator
   {
     private:
       unsigned char *ptr;

       // construct only via Image class (begin/end)

       iterator(u_char *p) : ptr(p) {}

     public:
```

```
        //copy construct is public
       iterator( const iterator & rhs) : ptr(rhs.ptr) {}
        // define overloaded ops: *, ++, --, =
       iterator & operator=(const iterator & rhs)
       {...}

        // other methods for iterator
   };

   // define begin()/end() to get iterator to start and
   // "one-past" end.

    iterator begin(void) { return iterator(data.get())} // etc
};
```

PGM Images

PGM images are *greyscale* - meaning they have no colour, and use only one value to encode an intensity that ranges from (black = 0) through to (white=255). Each value is thus stored as an `unsigned char'. The images I will provide are `raw' PGM - i.e. binary files. However, they have a text header, so you usually open the file as binary (ios::binary), but use the **>>** and **<<** operators to read/write the header information. Getline() is very useful for reading and discarding comment lines. The header is followed by a block of bytes, which represent the image intensities. You can use the **read()** and **write()** methods to manipulate this data. Look at the function prototypes to see what arguments they expect (**www.cplusplus.com** can help here if you have no C++ reference). The PGM images you will receive will have the format:

P5
 # comment line (there can be more than 1 of these comment lines)
 # the string P5 will always be the first item in the file.
Nrows Ncols
255
binary_data_block

where *Nrows* and *Ncols* are integers representing the rows and columns making up the image matrix. There is a newline after each line - uses "ws" to process this correctly. After reading 255 (and using the **ws** manipulator) you will be at the start of the binary data (Nrows*Ncols **unsigned chars**) which you can read in using **read().**

Remember that image data (being a 2D array of **unsigned char**) is indexed from [0][0] and that this is the *top left hand corner* of the image. Generally we read in the image data line by line, starting from the top. ***This is in fact how the binary image data is stored in the PGM format*** – so you can read/write the entire image with one read()/write() statement! This is very convenient.

We will provide some PGM images. You can view the effects of your image operations using an image viewer like **Gimp** on Ubuntu.

If you satisfactorily complete everything requested above you can score a maximum of 90%. An additional 10% of the prac mark can be obtained by adding the following functionality.

Extra credit:

Implement a further “filter” operator (%) for your image which will apply a filter g to the image:

Img % g

This must create a new image which is a filtered version of the input, Img. A filter (in this context) is a N X N array of floating point values, N odd, which are used to transform the input image. You ca create struct/class to represent this data. For example, a 3x3 blurring filter might be

$$\begin{matrix} 1/16 & 1/8 & 1/16 \\ 1/8 & 1/4 & 1/8 \\ 1/16 & 1/8 & 1/16 \end{matrix}$$

The filter will be provided in a text file of format

N val1 val2 val3 ….

where N (an odd integer) is the filter size and the remaining N*N values are real numbers for the entries, in row major order.

When filtering, you turn each input pixel into a new filtered pixel, as follows:

1) Place the centre of the array on the pixel you wish to filter

2) Compute a **weighted sum** which includes all pixels covered by the grid of values (including the pixel in the centre). The weights are the real number at the corresponding positions. Multiply each weight with the corresponding pixel value and add all of this up. This can be done through a simple nested loop – you just need to figure out the indexing.

3) write out the resulting value into the same pixel position in your **output** image.

4) When filtering values close to the boundary of the image, the grid will “fall off”. In these cases, just reflect the values from the inside around the relevant edge e.g. image[row][col] = image[row][2*width-col-1] if col goes off the right-hand edge.

NOTE: you write the filtered output pixels to a new image! You treat the input image as a source for reading only.

Testing and Source Control

You should use **catch.hpp** to unit test your methods. To achieve this it may be easier to write a constructor that takes in width, height and a buffer of values,

instead of reading these from file. Special care should be taken to test the following:

- Move and copy semantics
- Iterator and its operators (including boundary conditions)
- Thresholding, inverting and masking operator overloads
- Addition and subtraction of images. Ensure that the pixels are clamped appropriately.
- [optionally] the filtering operator. Take special care to test the boundary conditions

It may be easiest to compile your unit tests into separate executable(s) and run these tests when the user types “make test”. Be explicit in your readme about how these test cases should be executed.

A simple qualitative test for your program may involve the following sequence of operations:

1. create an image mask M1 (via thresholding)
2. create an inverted mask of M1 called M2 (by inverting M1)
3. use M1 and an Image U1 to create a masked image
4. use M2 and an Image U2 to create a second masked image
5. add the two masked images together to create a new image with pieces of both U1 and U2!

You can use whatever images you want for U1, U2 and M1, but they MUST have the same dimensions.

Please ensure that you make use of version control for your assignment and that you commit small units of work regularly. Your commit messages should be detailed.

PLEASE NOTE:

- *A working Makefile must be submitted. If the tutor cannot compile your program on nightmare.cs by typing **make**, you will receive 50% of your final mark.*
- *Do not submit binary files - submit your source code and any other necessary files to build and test your project.*
- *You must use version control. Failure to submit a local repository will incur a 10% penalty. Bad usage of git will also incur a penalty (e.g submitting binary files, non-descriptive commit messages, irregular/few commits etc.)*
- *You must provide a README file explaining what each file submitted does and how it fits into the program as a whole. The README file should not explain any theory that you have used. These will be used by the tutors if they encounter any problems.*
- *Please ensure that your tarball works and is not corrupt (you can check this by trying to extract the contents of your tarball - make this a habit!). Corrupt or non-working tarballs will not be marked - no exceptions!*

- *A 10% penalty per day will be incurred for all late submissions. No hand-ins will be accepted if later than 5 days. Do not hand in any binary files.*
- *DO NOT COPY. All code submitted must be your own. Copying is punishable by 0 and can cause a blotch on your academic record. Scripts will be used to check that code submitted is unique.*