The image class

As we are dealing with an image processing application, let's start with its main building block: the abstract data type that represents an image. For this assignment, you will work with grayscale images. A grayscale image is represented by a 2D array of pixels, and each pixel can take a value between [0, 255].

The value 0 represents black, 255 is white, and values between represent different gray level intensities (the larger the value, the lighter the pixel is):

An example of how you could define an image class is presented below:

```
class Image
{
public:
   Image();
   Image(unsigned int w, unsigned int h);
   Image(const Image &other);

   ~Image();

   bool load(std::string imagePath);
   bool save(std::string imagePath);
```

Of course, you can choose (and argue your decision) other ways of representing an image, but the class that you write should have at least the methods and operators:

- copy constructor;
- copy assignment operator;
- getters for the height and width of the image. You should also define a class to represent the size (width and height of an image) and have a getter that returns the size of the image in this format;
- two methods that load and save an image from/to a *.pgm* file. These methods should return a Boolean value that indicates if the operation was successful. More details about the *.pgm* file format can be found here: http://netpbm.sourceforge.net/doc/pgm.html ;
- overload for the stream insertion operator: operator<<; this would display the image pixels in a 2D grid. For a pretty print, you could use *setw*[1] to format the way you display these pixels (set *setw*(3) to display the pixels on 3 characters).
- overload for the +, -, and * arithmetic operators between two images. The precondition for this is that both images should have the same size. The operations will be applied element-wise!
- overload for the +, - and * arithmetic operator between an image and a scalar value;
- a method that returns whether the image is empty or not;
- a method that returns a reference to the pixel at a given location in the image. This method should have two overloads: one in which you specify the pixel's position with two values (x and y coordinates) and another one in which you specify the pixel's position with a Point class (that has two fields x and y);
- a method that returns a pointer to a row in the image;
- a method that releases the memory allocated for the image. You could call this method in the destructor of the class;
- in image processing, a region of interest (ROI) is a portion of the image that you want to apply to filter or apply some operation on. This is usually a rectangular region over the image. Write two methods that return a new Image that contains the pixels within a ROI specified as a parameter (you should perform a deep copy of the image pixels). Also, check that the ROI is valid (is within image bounds). You should use only pointer arithmetic for this (so don't use the indexing operator []);
- some static methods that create some "special" types of images: an image filled with zeros and an image filled with ones.

IMAGE PROCESSING TASKS

For these tasks you will first start by writing a base class for image processing: *ImageProcessing*, The class will have a pure virtual method:

void process(const Image& src, Image& dst);

which will be implemented by all its subclasses. Then you will write a subclass for each image processing operation. You will have to implement the following operations:

1. **Brightness and contrast adjustment.**

This processing type operates at each pixel independently and alters the brightness and contrast of the image, as follows:

$$F(x, y) = \alpha \cdot I(x, y) + \beta$$

, where I(x, y) and F(x, y) represent the pixels at coordinates (x, y) from the input image and the filtered image respectively.

The parameters of this operation $\alpha > 0$ (the gain) and β (the bias) can be considered to control the control *contrast* and *brightness* of the image. These parameters will be specified in the constructor of the class. The default constructor initializes $\alpha = 1$ and $\beta = 0$ (so, in this case, this operation is a no-op).
Basically, you should just iterate through each image pixel and apply the transformation to each pixel. Make sure that the result is in the range [0, 255]. If not, clip the result to this interval!

1. **Gamma correction.**

Gamma correction is used to correct the overall brightness of an input image with a non-linear transformation function:

$$F(x, y) = I(x, y)^{\gamma}$$

, where $F(x, y)$ and $I(x, y)$ are the values of the pixels at coordinates (x, y) in the filtered image and input image, respectively, and γ is the gamma encoding factor.
This operation is needed because humans perceive light in a non-linear manner; the human eye is more sensitive to changes in dark tones than to changes in lighter ones. So, gamma correction is used to optimize the usage of bytes when encoding an image, by taking into consideration this perception difference between the camera sensor and our eyes.

IMAGE PROCESSING TASKS

For these tasks you will first start by writing a base class for image processing: *ImageProcessing*, The class will have a pure virtual method:

void process(const Image& src, Image& dst);

which will be implemented by all its subclasses. Then you will write a subclass for each image processing operation. You will have to implement the following operations:

1. **Brightness and contrast adjustment.**

This processing type operates at each pixel independently and alters the brightness and contrast of the image, as follows:

$$F(x, y) = \alpha \cdot I(x, y) + \beta$$

, where I(x, y) and F(x, y) represent the pixels at coordinates (x, y) from the input image and the filtered image respectively.

The parameters of this operation $\alpha > 0$ (the gain) and β (the bias) can be considered to control the control *contrast* and *brightness* of the image. These parameters will be specified in the constructor of the class. The default constructor initializes $\alpha = 1$ and $\beta = 0$ (so, in this case, this operation is a no-op).
Basically, you should just iterate through each image pixel and apply the transformation to each pixel. Make sure that the result is in the range [0, 255]. If not, clip the result to this interval!

1. **Gamma correction.**

Gamma correction is used to correct the overall brightness of an input image with a non-linear transformation function:

$$F(x, y) = I(x, y)^{\gamma}$$

, where $F(x, y)$ and $I(x, y)$ are the values of the pixels at coordinates (x, y) in the filtered image and input image, respectively, and γ is the gamma encoding factor.
This operation is needed because humans perceive light in a non-linear manner; the human eye is more sensitive to changes in dark tones than to changes in lighter ones. So, gamma correction is used to optimize the usage of bytes when encoding an image, by taking into consideration this perception difference between the camera sensor and our eyes.

The value of should be passed as a parameter to the constructor of this class.

For further details about this operation, you can check the following websites: https://www.cambridgeincolour.com/tutorials/gamma-correction.htm
https://en.wikipedia.org/wiki/Gamma_correction.

1. Image convolution

Convolutions are highly used in image processing and machine learning to extract some features from the input image. They involve the usage of a kernel (or filter) that is convolved over the input image as follows:

$$F = K * I$$

, where F is the output (filtered) image, K is the convolutional kernel and I is the input image. The convolutional kernel K has the shape (w, h), where w and h are usually odd numbers: w = 2k + 1.

At each image position (x, y), the convolutional filter is applied as follows:

$$F(x, y) = \sum_{u=0}^{w-1} \sum_{v=0}^{h-1} K(u, v) I(x - u + k, y - v + k)$$

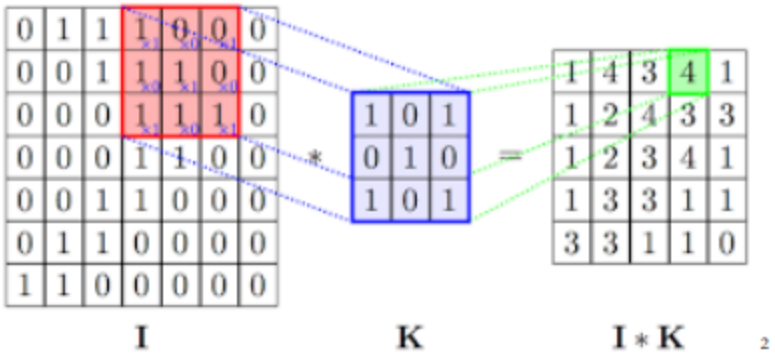

Figure 1. Illustration of image convolution.

A nice animation for image convolution can be found here:
https://upload.wikimedia.org/wikipedia/commons/1/19/2D_Convolution_Animation.gif
When implementing this operation you should pay attention to the image bounds!
The constructor of this function should also take a parameter a pointer to a function that is used to scale the result of applying the convolution kernel to the range [0, 255].

0	0	0	**Identity kernel**. No-op. Applying convolution with this kernel results in the same image.
0	1	0	
0	0	0	

1	1	1	**Mean blur kernel**. (for this kernel, the scaling function should just **multiply** the convolution result to 1/9)
1	1	1	
1	1	1	

1	2	1	**3x3 Gaussian blur kernel**. Still a blurring kernel, but in this case, the central pixel of the kernel has a higher weight in the mean operation; for this kernel, the scaling function should just **multiply** the convolution result to 1/16.0)
2	4	2	
1	2	1	

1	2	1	**Horizontal Sobel kernel.** Used to detect horizontal edges. In this case, the scaling function should be a linear mapping function that converts the range [-4*255, 4*255] to the range [0, 255]).
0	0	0	
-1	-2	-1	

1	2	1	**Horizontal Sobel kernel.** Used to detect horizontal edges. In this case, the scaling function should be a linear mapping function that converts the range [-4*255, 4*255] to the range [0, 255]).
0	0	0	
-1	-2	-1	

-1	0	1	**Vertical Sobel kernel.** Used to detect vertical edges. In this case, the scaling function should be a linear mapping function that converts the range [-4*255, 4*255] to the range [0, 255]).
-2	0	2	
-1	0	1	

Next, you will implement a module to draw different shapes over an image.

Write a class to represent a 2D point (as you noticed, this class is also used in the *Image* class to get a reference to the value of a pixel in the image). This class should contain the x and y coordinates of a point (integer values). Overload the stream insertion and extraction operator for this class. Write a default constructor, that initializes the fields to 0, and a parameterized constructor.

Write a class to represent a Rectangle (as you noticed, this class is also used in the *Image* class to get a region of interest in the image). This class should contain the x and y coordinates of the top-left point, and the width and height of the rectangle. Overload the stream insertion and extraction operator for this class. Write a default constructor that initializes the fields to 0 and a parameterized constructor. Also, write a constructor that takes as parameters two 2D Points (the top left and bottom-right coordinates of the rectangle).

This class should also overload the following operators:

- Addition and subtraction operators: this will be used to translate the rectangle.
- & and | operators. The & operator will compute the intersection between two rectangles, while the | operator will compute the union of two rectangles.

Now that you have classes to represent basic shapes, let's use them in your image processing library. Create a *module* for drawing some shapes over images. You should have at least the following functions in this module[3]:

- drawCircle(Image& img, Point center, int radius, unsigned char color);
- drawLine(Image &img, Point p1, Point p2, unsigned char color);
- drawRectangle(Image &img, Rect r, unsigned char color);

[3] In practice more complex rasterization algorithms are used to convert 2D shapes into a rasterized format, but for this simple application it is ok if you just use the basic formulas for the shapes to determine the position of the points that lie on the shape's boundary.