Computer Graphics - Homework Assignment 4 - Image Processing

Goals:

- Understand convolution and image processing.
- Gain insight into high performance image processing algorithms.
- Gain more experience with raster images.

Getting Started & Handing In:

- The code will be written in C++. You are encouraged to write helper functions. They can eliminate a lot of redundant code.

- The program is a command line program. It makes use of the open source Qt framework only for its QImage class for image loading and saving and pixel manipulation.

- You should have already successfully installed the open source version of the Qt environment from the last assignment: https://www.qt.io/download-open-source (At the time of writing, version 5.11 is the newest version. Any 5.x version should work. The installer, by default, includes all versions of Qt. Save space by installing only the most recent version and a compiler.) Mac users with Homebrew can alternatively install via: `brew install qt` and `brew cask install qt-creator`.

- Download this assignment. This will create a folder named imageprocessing. Open the file named `imageprocessing.pro`. This should launch the Qt Creator development environment (IDE).

- Build and run the code. The code should compile, but it will complain when running about not having enough arguments. You should see a message like:

  ```
  Usage: ./imageprocessing box radius input_image.png image_out.png
  Usage: ./imageprocessing scale width_percent height_percent input_image.png image_out.png
  Usage: ./imageprocessing convolve filter.png input_image.png image_out.png
  Usage: ./imageprocessing sharpen amount radius input_image.png image_out.png
  Usage: ./imageprocessing edges input_image.png image_out.png
  Usage: ./imageprocessing grey input_image.png image_out.png
  ```

- Add your code to `convolution.cpp`. You may wish to add helper functions at the top. There are some suggested signatures.

- Build and run and test that it is working correctly. Qt Creator has a great debugger.

- Run the following commands on the provided example images (replace `balls` with the name of each example). Copy your output `.png` files into a new `output` subdirectory.

  ```
  ./imageprocessing grey balls.png balls-grey.png
  ./imageprocessing box 0 balls.png balls-box0.png
  ./imageprocessing box 3 balls.png balls-box3.png
  ./imageprocessing box 25 balls.png balls-box25.png
  ./imageprocessing edges balls.png balls-edges.png
  ./imageprocessing sharpen 1 5 balls.png balls-sharpen-1-5.png
  ./imageprocessing sharpen 2 5 balls.png balls-sharpen-2-5.png
  ./imageprocessing sharpen 2 10 balls.png balls-sharpen-2-10.png
  ./imageprocessing scale 100 100 balls.png balls-scale-100.png
  ./imageprocessing scale 50 100 balls.png balls-scale-50w.png
  ./imageprocessing scale 10 100 balls.png balls-scale-10w.png
  ./imageprocessing scale 100 50 balls.png balls-scale-50h.png
  ./imageprocessing scale 100 10 balls.png balls-scale-10h.png
  ./imageprocessing scale 50 50 balls.png balls-scale-50.png
  ./imageprocessing scale 10 10 balls.png balls-scale-10.png
  ./imageprocessing scale 200 200 balls.png balls-scale-200.png
  ./imageprocessing scale 50 200 balls.png balls-scale-50w-200h.png
  ./imageprocessing scale 200 50 balls.png balls-scale-200w-50h.png
  ./imageprocessing convolve filters/identity.png balls.png balls-convolve-identity.png
  ./imageprocessing convolve filters/box3.png balls.png balls-convolve-box3.png
  ./imageprocessing convolve filters/box25.png balls.png balls-convolve-box25.png
  ./imageprocessing convolve filters/linear.png balls.png balls-convolve-linear.png
  ./imageprocessing convolve filters/quadratic.png balls.png balls-convolve-quadratic.png
  ./imageprocessing convolve filters/direction.png balls.png balls-convolve-direction.png
  ./imageprocessing convolve filters/heart.png balls.png balls-convolve-heart.png
  ./imageprocessing convolve filters/heart-flip.png balls.png balls-convolve-heart-flip.png
  ```

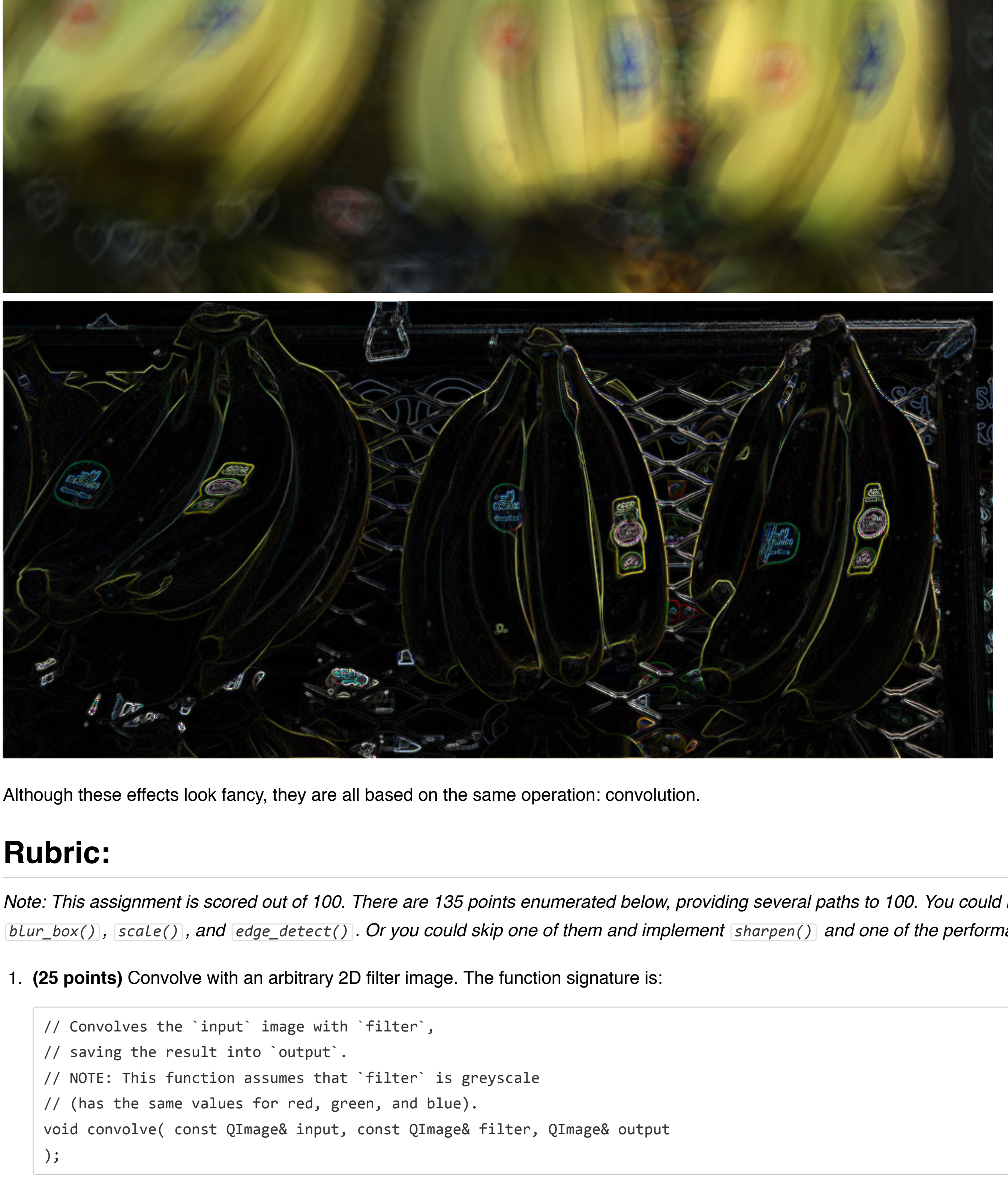
- The example images are:
  - `balls.png`
  - `bananas.png`
  - `puppy.png`
  - `wave.png`
  - `wikipedia.png`
- I have provided a script of commands `run_all.HOW` that will run all of those commands for all the examples. You will likely have to modify it for your environment by changing the path to the convolve program. (I have also provided the script `run_all_gen.py` I used to generate `run_all.HOW`.)
  - On a unix machine (like a Mac), you can type:
    - `sh run_all.HOW`
  - On a Windows machine, I believe that if you change the name of `run_all.HOW` to `run_all.bat` you can then type:
    - `run_all.bat`
- When done, zip your entire `imageprocessing` directory along with the `output` subdirectory and a `Notes.txt` file as `hw04_lastname_firstname.zip` and upload your solution to Blackboard before the deadline. Your `Notes.txt` should describe any known issues or extra features. Your `Notes.txt` should also note the names of people in the class who deserve a star for helping you (not by giving your their code!).

- The framework and libraries provide all the support code that you need.

- **THIS IS AN INDIVIDUAL, NOT A GROUP ASSIGNMENT. That means all code written for this assignment should be original! Although you are permitted to consult with each other while working on this assignment, code that is substantially the same will be considered cheating.** In your `Notes.txt`, please note who deserves a star (who helped you with the assignment).

Overview:

In this assignment, you will be implementing image processing operations based on convolution. You will be able to scale, blur, and sharpen images, just like Photoshop. You will be able to create effects like this:



Although these effects look fancy, they are all based on the same operation: convolution.

Rubric:

Note: This assignment is scored out of 100. There are 135 points enumerated below, providing several paths to 100. You could implement `convolve()`, `blur_box()`, `scale()`, and `edge_detect()`. Or you could skip one of them and implement `sharpen()` and one of the performance enhancements.

1. **(25 points)** Convolve with an arbitrary 2D filter image. The function signature is:

   ```
   // Convolves the `input` image with `filter`,
   // saving the result into `output`.
   // NOTE: This function assumes that `filter` is greyscale
   // (has the same values for red, green, and blue).
   void convolve( const QImage& input, const QImage& filter, QImage& output
   );
   ```

   This requires a quadruple for loop. There is no easy way around slow $O(n \cdot \text{radius}^{2})$ running time (n is the number of input image pixels). There is a folder of interesting filter images in the handout. The `main()` function ensures that the values in the filter image are greyscale, meaning that the same numbers are stored for red, green, and blue. Because most image formats store their pixel values as 8-bit numbers, assume that the filter images are stored **not** normalized. Normalize them when you apply them by dividing by the sum of all pixel values. Don’t forget that the indices into filter are negated. See the tip below about `.mirrored()`.

2. **(25 points)** Blur with a box filter. Blurring with a box filter is one of the simplest kinds of convolution there is. It simply replaces each pixel with the unweighted average of nearby pixels. For a box, nearby pixels are those whose x or y coordinates are differ by at most radius. The function signature is:

   ```
   // Applies a box blur with `radius` to `input`, saving the result
   // into `output`.
   void blur_box( const QImage& input, int radius, QImage& output );
   ```

   A naive implementation of this takes $O(n \cdot \text{radius}^{2})$ running time (n is the number of input image pixels). You must implement it with faster running time. Because a 2D box filter is separable, you can reduce the running time to $O(n \cdot \text{radius})$ by first blurring horizontally and then blurring vertically (or vice versa).

   1. **(10 additional points)** Because the box filter is unweighted, it is theoretically possible to achieve $O(n)$ running time.
3. **(15 points)** Sharpen the image. Sharpening is the opposite of blurring. Therefore, a simple formula for a sharpened image is: *sharpen(I) = (1 + α)·I - α·blur(I)*, where *α* controls the amount of sharpening. You can use your box blur, and then compute the formula per-pixel. The function signature is:

   ```
   // Sharpens the `input` image by moving `amount` away from a blur with `radius`.
   // Saves the result into `output`.
   void sharpen( const QImage& input, real amount, int radius, QImage& output );
   ```

4. **(25 points)** Scale the image to a new dimension. The function signature is:

   ```
   // Scales the `input` image to the new dimensions, saving the result
   // into `output`.
   void scale( const QImage& input, int new_width, int new_height, QImage& output );
   ```

   In theory, scaling reconstructs a continuous function from the input image and then resamples it at evenly spaced locations (the pixels of the output image). The reconstructed continuous function is obtained by convolving a continuous filter with our discrete input image. Note that we only need the values of the reconstructed function at output pixel locations. Therefore, you will iterate over the pixels of the output image and compute the convolution of a continuous filter with the input image. The only difference between convolving with a discrete filter (array) *b* versus a continuous filter *f* is that you will make a function call *f(x)* or *f(x,y)* to access the filter values instead of looking them up in an array *b*[x] or *b*[x,y]. The filter you will use is a triangle function:

   ```
   triangle( radius, x ) = max( 0, 1 - | x/radius | )
   ```

   $$\text{triangle}(radius, x) = \max\left(0, 1 - \left|\frac{x}{\text{radius}}\right|\right)$$

   You will need to normalize this on-the-fly.
   In 2D, the filter is

   ```
    f(x,y) = triangle( radius_x, x ) * triangle( radius_y, y )
   ```

   $$f(x, y) = \text{triangle}(\text{radius}_x, x) \cdot \text{triangle}(\text{radius}_y, y)$$

   By picking the right right radius for x and y, the scaling function will eliminate high frequencies that cause aliasing artifacts. The formula for the radius is:

   ```
   if new_size > old_size: radius = 1
   else: radius = old_size/new_size
   ```

   $$\text{radius} = \begin{cases} 1 & \text{if} \quad \text{new\_size} > \text{old\_size} \\ \frac{\text{old\_size}}{\text{new\_size}} & \text{otherwise} \end{cases}$$

   where size is the width or height. Pseudocode for 1D image resizing can be found on slide 53 of *10 Signal Processing* or in the book (*Fundamentals of Computer Graphics*, Chapter 9 *Signal Processing*).

   1. **(additional 10 points)** The triangle filter is separable, so you can implement scaling in $O(n \cdot \text{radius})$ time by first scaling horizontally and then scaling vertically (or vice versa).
5. **(25 points)** Detect edges. Edge detection can be implemented in various ways. The reference implementation uses 1D convolution with the filter [–1 0 1]. Convolving with the filter horizontally produces a D_x image and vertically produces a D_y image. Note that this filter cannot be normalized, since it sums to 0. Also note that this filter will produce positive and negative values; store the absolute value. The final value for a pixel of the edge detected image is $\sqrt{ D_x^2 + D_y^2 }$. The function signature is:

   ```
   // Performs edge detection on the `input` image.
   // Stores the result into `output`.
   void edge_detect( const QImage& input, QImage& output );
   ```

6. **(additional ? points)** Additional operations. Make suggestions!

Tips

- All the code you write will go into `convolution.cpp`.

- Convolution operates on the red, green, and blue channels of the image independently. Ignore alpha. (Even preserving alpha would produce the wrong result.)

- You will want almost all of the filters you will implement to be normalized, meaning that their values sum to 1. Only edge detection makes use of filters which sum to 0. Don’t normalize edge detection filters. The `convolve()` function takes in unnormalized filters; you can normalize on-the-fly.

- It is easy to normalize on-the-fly by keeping track of the denominator.

- When convolving near the edges of an image, only apply the portion of the filter that lies in the image (ignore out-of-bounds pixels). When you do this, you will be ignoring part of the filter. Therefore, the part of the filter that you do use will no longer sum to one. You will need to renormalize by dividing by the sum of non-ignored filter values.

- Do not perform arithmetic operations on a QRgb. First extract the components to integers or floating point types.

- Don’t store partial sums in a `QRgb`, which have only 8-bit precision for each channel. If your partial sums are real numbers, you would lose a lot of precision if you round after each addition. If you are normalizing on-the-fly, the partial sums may overflow, because the sum of 8-bit numbers often won’t fit into another 8-bit number. When storing the results back into a `QRgb`, make sure values are in the range [0,255]. You can use `min()` and `max()` or the provided `clamp()` helper function.

- You can’t perform convolution in-place, because you will be overwriting values that you still need to read. If a function makes use of separable convolution, don’t forget to create a temporary intermediate image or array (`std::vector<int>`) as necessary.

- You can halve the amount of code you need to write for separable filters by iterating with pointers to pixel data. See the discussion of `image.scanLine()` below. For non-separable functions, working with pointers to pixel data will not reduce the amount of code you write (though the code will run faster).

- Convolution, correctly defined, says that you iterate over the filter with flipped (negated) coordinates. This only matters for unsymmetrical filters. The only ones you will encounter are `heart.png` and `direction.png`. You should use `filter.mirrored( true, true )` instead of `filter` to get the correct results.

- You can compare your output in a few ways:

  - Open both images in a viewer which lets you flip back and forth in-place with, for example, the right and left arrow keys. You could, for example, open them in browser tabs and switch tabs back-and-forth. Rapidly switching back and forth in-place is a good technique to visually understand the differences.

  - With the built-in function `difference()`, accessible from the command line via:

    ```
    ./imageprocessing difference input_image1.png input_image2.png image_out.png
    ```

  - With the Python script provided in the `examples` directory:

    ```
    python imgdiff.py input_image1.png input_image2.png image_out.png
    ```

  - Do not use a program which returns true or false based on whether all the bits match. Slightly different implementations can round to slightly different answers, which is fine. Our spec is not bit-exact (and arguably should not be).

Qt functions you need for this assignment

QImage:

- `image.pixel(x,y)` returns the `QRgb` color for pixel x,y of a `QImage image`.

- `image.setPixel(x,y,c)` sets the pixel to a `QRgb` color `c`.

- `image.width()` and `image.height()` return the width and height of the image.

- `image.scanLine(y)` returns a pointer to the array of `QRgb` pixel data for row y. That pointer points to the pixel (0,y). If you have a pointer to a pixel `QRgb* pix`, the next pixel in the row is `pix+1` and the next pixel in the column is `pix+image.bytesPerLine()/4`. Therefore, the pointer to the first pixel in column x is `(QRgb*)image.scanLine(0)+x`. By keeping track of the *stride* between pixels, you can write general functions that iterate over either rows or columns. Such a function would take a pointer to the first pixel, the stride between pixels, and the number of pixels. This can substantially reduce the amount of code you need to write when you only need to iterate over an image’s rows or columns, as opposed to iterating over a square region. The code will also run faster. For an example of how to use these methods, see the `greyscale()` function.

- `QImage( width, height, QImage::Format_ARGB32 )` creates a blank image full of `QRgb` pixels.

- `image.mirrored( true, true )` returns a copy of the image mirrored horizontally and vertically.

`sqrt(x)`, `std::min(a,b)`, `std::max(a,b)`. These are not Qt functions. They are part of C’s `math.h` and C++’s . Nevertheless, you will find them useful. Note that `std::min` and `std::max` require both parameters to have the exact same type. If not, you will get a very long compiler error since they are generic functions written using C++ templates.

QRgb. To get the red, green, blue, and alpha components of a QRgb color `c` as 8-bit values, use `qRed(c)`, `qGreen(c)`, `qBlue(c)`, and `qAlpha(c)`. In this assignment, we are ignoring alpha. To create an RGB `QRgb` color, use `qRgb( red, green, blue )` with 8-bit parameters. Note that `QRgb` is not a class or a struct. It is a `typedef` for an `unsigned int`, and those functions are just getting and setting the appropriate bytes. The header `qrgb.h` is very short and readable. Here is most of it:

```
typedef unsigned int QRgb;                        // RGB triplet

inline int qRed(QRgb rgb)                // get red part of RGB
{ return ((rgb >> 16) & 0xff); }

inline int qGreen(QRgb rgb)                // get green part of RGB
{ return ((rgb >> 8) & 0xff); }

inline int qBlue(QRgb rgb)                // get blue part of RGB
{ return (rgb & 0xff); }

inline int qAlpha(QRgb rgb)                // get alpha part of RGBA
{ return rgb >> 24; }

inline QRgb qRgb(int r, int g, int b)// set RGB value
{ return (0xffu << 24) | ((r & 0xffu) << 16) | ((g & 0xffu) << 8) | (b & 0xffu); }

inline QRgb qRgba(int r, int g, int b, int a)// set RGBA value
{ return ((a & 0xffu) << 24) | ((r & 0xffu) << 16) | ((g & 0xffu) << 8) | (b & 0xffu); }
```