ECE2036 **Fall Semester, 2012**

The QDisplay API

Introduction. The next assignments for ECE2036 require a graphical interface for you to use to display and manipulate images. There are many different "application programming interfaces" (API's) that have been created for exactly this. The Microsoft WIN32 API is a good example, as well as Apple's "Carbon" interface, and MIT's "X-Windows". However, we want to focus on the algorithms for managing the images, rather than learning complex API's. To that end, we will use a simple interface called *QDisplay*. The *QDisplay* API provides only very basic functionality, but is sufficient to suit our needs.

The QDisplay API The *QDisplay* interface is derived from the more complicated *Qt* graphics library. This library is common across Linux and Windows platforms, so the same interface should work on both systems. Our interface uses a `QDisplay` object to interface your code to the *Qt* code, simplifying many of the details of creating and modifying graphics images. The object class definition is provided in file `qdisplay.h`. Details of the API are below.

1. All *Qt* applications require a single instance of an object of class `QApp`. This class requires the `argc` and `argv` parameters from the main program as arguments to the constructor. The simplest thing to do is create this object as the first line of your `main` program. This is provided for you in the skeleton `filter.cc` program.
2. A new display window can be created by creating an object of class `QDisplay`. The constructor requires a reference to the single `QApp` object as a parameter. The window is initially empty, and does not display. You can load images or create a blank image with the member functions discussed below.
3. Member function `Load` is used to load an existing image into the display window.
4. Member function `BlankImage` creates an all white image of the specified width, height, and depth. For our purposes, we will only use a depth of 8 (8-bit gray scale) or 32 (32-bit RGB).
5. Member function `Save` will save your image to a file.
6. Member functions `Width`, `Height`, and `Depth` return the width, height, and depth of the image. The depth value is the number of bits per pixel.
7. Member function `ImageData` returns a one dimensional array of bytes that represent the pixel data. The size of the array is `Width() * Height()`. The return type is a pointer, either to an array of `unsigned char` (for 8-bit pixels) or to an array of type `QRgb`, which is a 32–bit value with the red, green, and blue components (plus an *alpha* value that we are ignoring).
8. Member function `Update()` re-draws the image with the updated pixel values. Presumably, your program has modified the image data (obtained using the `ImageData` method) in some way and we want the new values displayed. This methods updates the entire image, and can be slow.
9. Member function `Update(int x, int y, int w, int h)` updates only a portion of the display, starting at the specified x and y coordinates and extending for the specified width and height. This is more efficient than simply updating the entire screen, and should be used for this assignment whenever possible.
10. Member `UpdateRate` specifies the maximum allowable update rate for the window, specified in frames per second. Specifying zero indicates infinite update rate, other values will delay the updating to be no more frequently than the frames-per-second update rate specified.
11. Member `Show` specified that the window should be visible and displayed on the screen. The default is to *not* display the window, so this must be called if the window should be visible.
12. Member function `Run` of the `QApp` object will simply process *Qt* events until the last window is closed. This should be called when your program is complete so you can visually inspect your results on the display.