CS 215 Spring 2012
Program 5: "Image Mosaic"
Due Thursday 19 April by Midnight

Program 4 established the basics of image manipulation. This assignment will build on those concepts in order to create "photo-mosaics".

In Program 4, input images were broken up into smaller blocks and the average color in each of those blocks was calculated. The algorithm for creating a photo-mosaic will expand on this idea. Once again, the input image will be divided into blocks of an arbitrary size, and it will important to calculate the average color of each pixel block. This metric (the block and its average color) will be the basis for replacing the block with a new image, from a database of possible images, which has approximately the same average color.

Provided with the assignment is a folder of around 900 small 80x60 images. These images are referred to as "filler images". In this assignment you should implement a **`fillerImage`** class that uses inheritance and derives from the **`image`** class (given in the assignment). The **`fillerImage`** class must define methods to find its own average color and store it in a private variable. The starting point in this assignment is to read each of the small "filler images" from their files into a **`fillerImage`** object.

Here is an example definition of the **`fillerImage`** class (note that inheritance from the **`image`** class indicated):

```
class fillerImage : public image {

public:
    fillerImage(); // constructor
    pixel getAverageColor();
    void increaseTimesUsed();
    int getTimesUsed();

private:
    int timesUsed;
    pixel avgPixel;
};
```

The **`getAverageColor()`** method should return a **`pixel`** object that is the average color of the filler image stored in the object. The **`increaseTimesUsed()`** method adds one to the **`timesUsed`** instance variable, allowing the object to keep track of the number of times this **`fillerImage`** object is used in the algorithm. The **`getTimesUsed()`** method returns the value of the **`timesUsed`** instance variable so that client code can access that value when necessary.

The **`fillerImage`** objects will need to be managed as a *data structure*. In this assignment, that data structure will be a standard template library (STL) list. This list, which will contain **`fillerImage`** objects, will need to be managed using STL functions and iterators.

When loading filler images, you will be passed a string to a directory. In this directory are images that follow a particular naming convention: a number followed by an extension. You should open every image from FIRST_FILLER_NUMBER to LAST_FILLER_NUMBER. These are global variables defined in **`globals.h`**. Each file should therefore be of the name:

directory + "/" + number + FILLER_IMAGE_EXT.

Once all of the filler images have been loaded, a base image to modify should be opened in the same manner as in Program 4 (with the **`loadImageFromFile`** function). You will notice that once an image is loaded, like in Program 4, you are able to select a size for the "blocks" that we will be analyzing in the base image. Unlike in program 4, you will notice that the ratio of width to height of these blocks is now locked at 4:3 (the "aspect ratio" of our filler images). Note that now when you define the size of a block, you are saying that for every block of that size, that block in the original image will be replaced with a filler image in the mosaic.

The fun part is to generate the mosaic. When the "Generate Mosaic" button is pressed, a new image should be generated (much like in Program 4) but this time, instead of 1 pixel per block, each block will be represented by a new group of pixels **`FILLER_IMAGE_WIDTH`** by **`FILLER_IMAGE_HEIGHT`**. Once again, these constants are defined in **`globals.h`**.

A *comparison metric* is how to decide which filler image (out of the list of 900 possibilities) is the best to use to represent a particular block in the base image. This metric should return a score with which we can rate the quality of a *candidate* filler image as a potential match. The goal is to select the best match for each block in the input image. The metric will be based on the average overall color of the block of pixels as compared to the same thing in the filler image. You should already be able to generate the average color of a block as well as the average color of a filler image from Program 4. An easy way to score the closeness of a match is to find the Euclidean distance between the two colors by imagining they are 3D points (with red, green, and blue being your axes). Applying the standard distance formula to the colors yields the following distance metric:

```
Distance-1 =   sqrt((blockAvg.red -fillerAvg.red)^2 +
                    (blockAvg.green - fillerAvg.green)^2 +
                    (blockAvg.blue - fillerAvg.blue)^2)
```

Since this is a distance metric, a low score is a better (closer) match. Using this metric, the filler image with the smallest "distance" from the block's average color would be the ideal candidate. However, we also want to factor into our metric how many times the image has already been used in the mosaic. (Having a mosaic consisting of a single repeating image isn't very interesting at all!) To accomplish this and to add a small push towards using a variety of filler images, use this:

```
Distance-2 = sqrt((blockAvg.red - fillerAvg.red)^2 +
                  (blockAvg.green - fillerAvg.green)^2 +
                  (blockAvg.blue – fillerAvg.blue)^2) * (timesUsed+1)
```

Remember: the image with the LOWEST score using this metric is the one that should be used.

Here is an example in action. Take the following scenario:

Average Block Color:

```
red:       85
green:    212
blue:      15
```

Filler Image Average:

```
red:      100
green:    128
blue:      37
```

This filler image has already been used twice in the mosaic.

Following our metric, we would score this filler image as:

```
sqrt( (85 – 100)^2 + (212 – 128)^2 + (15 – 37)^2 ) * (2+1)
sqrt(225 + 7056 + 484) * 3
88.12 * 3
264.36
```

The above filler image would get a score of 264.36. If 264.36 is the lowest score after checking all of the filler images, then this filler image should be selected to be used to represent this block in the final mosaic.

There are two remaining steps. First, because the final mosaic should look as much like the original as possible, the filler images must be "tinted" to help their color match the image block they are replacing. Second, the filler image pixels will be copied into the output image in the correct spot – the place that replaces the original image block.

In Program 4 you wrote functions to increase the value of an individual component (red, green, or blue) of the image. Copy all the pixels of the filler image to the appropriate location in the newly created image, and modify each pixel so that the red component is increased or decreased an amount corresponding to (**`blockAvg.red - fillerAvg.red`**). In other words, tint the pixels from the *filler image* toward the average value of the red in the *image block*. This should also be done for the green and blue components. Once you have the average pixel colors for the image block and the filler image, find the difference between each component and add/subtract that difference to each pixel of the copied filler image. (REMEMBER: you MUST make sure the values stay between 0 and 255! If a value goes negative, set it to 0. Greater than 255, set it to 255.)

Continuing the example from above, if we chose that filler image to represent our block in the mosaic, we would need to adjust every pixel of the copied filler image as follows:

```
Red:          85 - 100  =   -15
Green:       212 - 128  =    84
Blue:         15 -  37  =   -22
```

So for every pixel in the filler image, as we are copying them to the final product, we will want to subtract 15 from every red value, add 84 to every green value, and subtract 22 from every blue value.

Clearly this process (select a filler image according to the metric, copy its pixels into an output image at the appropriate place, and tint those pixels according to the average R/G/B values from the metric) must be repeated for every block in the original image. When completed, return a pointer to the new output image so that it will be displayed by the **`displayImage()`** function (as in Program 4).

Supplied for you is a GUI interface written in C++ and available for check-out in the class code repository. Use the same instructions for checking out and building this project as you did for Program 4. You should only need to modify the file: CS215Pgm5.cpp as well as any .h and .cpp files you add to the project (for example, for the **`fillerImage`** class.). You must implement the following functions in CS215Pgm5.cpp:

bool loadImageFromFile(string filename)

INPUTS: a string containing a path to a file to open. This value is returned from the user's selection in the open file dialog.
OUTPUTS: a boolean indicating whether or not the image could be opened correctly.

void saveImageToFile(string filename)

INPUTS: a string containing a path to save the current image out to.
OUTPUTS: NONE

image* displayImage()

INPUTS: NONE
OUTPUTS: This function should return a pointer to the image object that is currently being viewed on the screen. If a user has loaded an image correctly, you should return a pointer to an image object containing the base image. If a user has used the generate mosaic button, you should of course return a pointer to an image object that reflects these changes.

bool loadFillerImagesFromFolder(string folder)

INPUTS: a string containing a path to a directory containing all of the filler images.
OUTPUTS: A boolean indicating if all the images loaded correctly into the linked list or not. Return false, and delete all of your dynamically created variables if an error occurs.

void programExiting()

INPUTS: NONE
OUTPUTS: NONE
This function is called when the program is being closed. Use this to clean up and delete any dynamically allocated memory so that no memory leaks occur!

bool imageLoaded()

INPUTS: NONE
OUTPUTS: A boolean indicating if a base image has been successfully loaded.

void generateMosaic(int blockWidth, int blockHeight)

INPUTS: Integers indicating the width and height of the "blocks" to be replaced with filler images in the final mosaic.
OUTPUTS: NONE
This function should generate a photo mosaic in the process described above.

fillerImage* findBestFillerImage(pixel color)

INPUTS: A pixel object indicating the average color of the image block to be matched by a filler image.

OUTPUTS: A pointer to a filler image object (remember, these objects are going to be stored in an STL list) which is the result of the metric **`Distance-2`** being applied to all the filler image objects in the STL list. The returned pointer is a pointer to the filler image object with the minimum score using the metric.
This function is a *helper* function, and should be invoked by **`generateMosaic()`**.

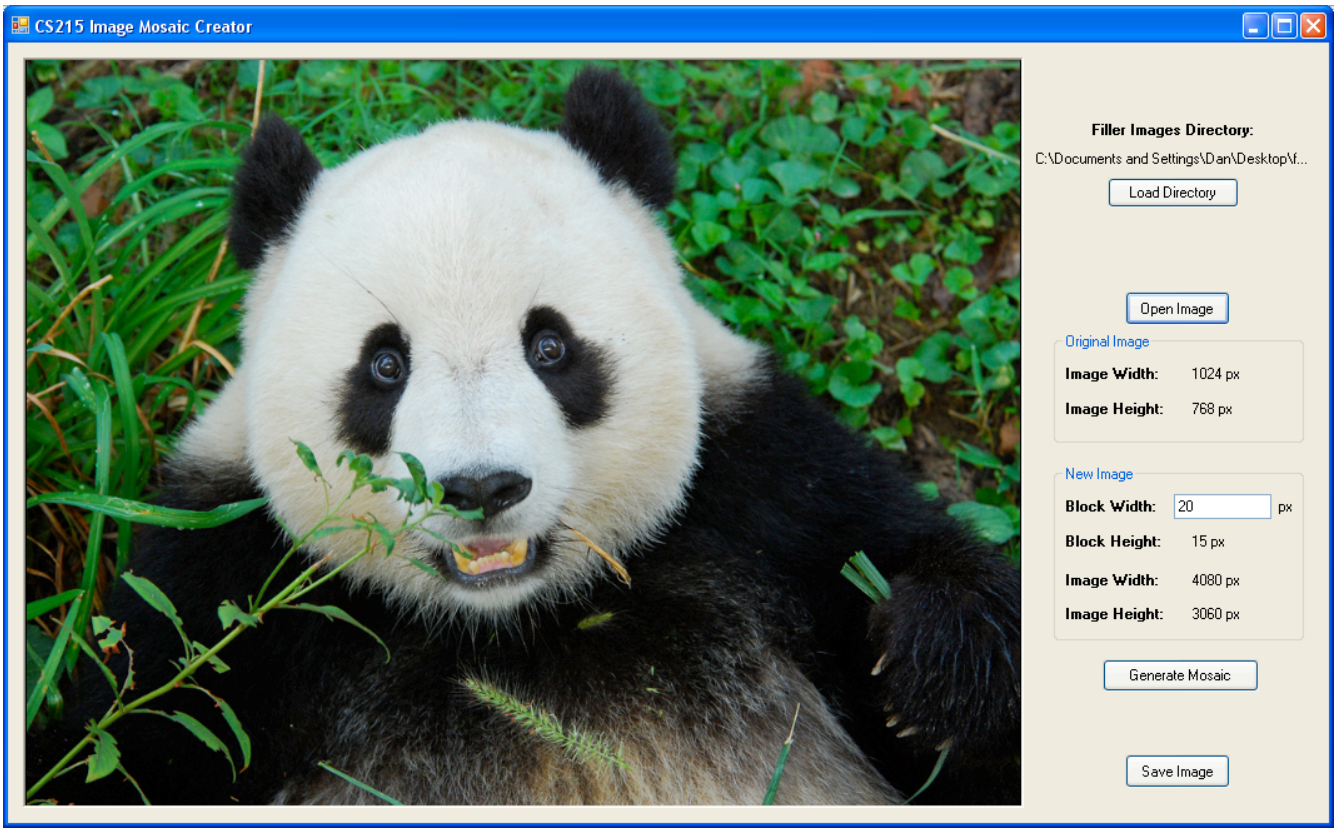
As mentioned above, there is a directory of around 900 filler images that are to be used with the program. A couple of sample images to use as base images have also been included for your convenience. Feel free to test your program with these or any other images. Please note however, that these images can be in jpg, tif, png, bmp, etc. format, but MUST be saved as a 24 bpp (aka 8 bits per RGB value) image. We are limiting our work to this color space. There are many other color spaces that images can be saved in (Grayscale, Palette, CMYK, etc) but these are beyond the scope of this assignment. If an image cannot be loaded for this reason or one similar, a message will be printed out to the console (if it is turned on) and false will be returned from the load function.

Performance Characterization
In order to receive full credit for this assignment, you must also turn in a performance characterization. In particular, use the **`clock()`** function in order to measure two key performance indicators: the total time to create the mosaic (when the user presses "create mosaic"), and the time spent searching the STL list for the filler image that is the best match for a block. In particular, plot each of these times as a function of block size (use pixel area). So the horizontal axis of this performance graph will be block area in pixels, and the vertical axis will be time in whatever units you choose.
In measuring the time spent searching the STL list, measure the average search time of the list. This means that for each block to be replaced, measure the time it takes to find the filler image and then average all these measurements together. Also plot this as a function of block size.
Turn in your data, a graph of the data, and a short discussion of the trends you see in the data (why do the curves look the way they do?). Use a spreadsheet to collect and graph your data.

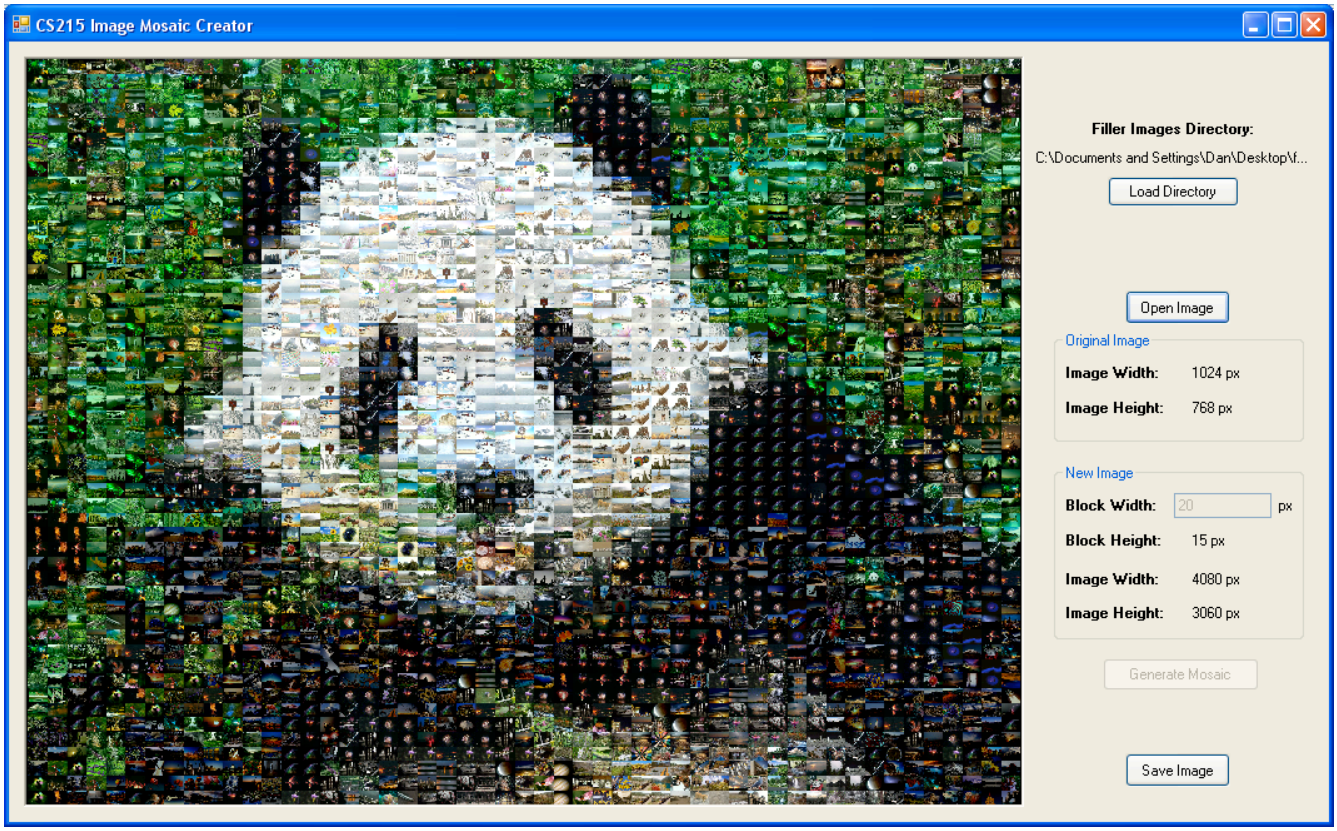
Special Note for Final Project: Submit with your solution a custom mosaic created with your program. Label this image "Contest-Submission". We will judge your creation to select winners/runners-up for special recognition. Do not miss this special opportunity to show off your work and your creativity! If you would like to use a different filler image database you may. But it will be up to you to create it.

Submit the following files to the class portal as one zip file:
fillerImage.h
fillerImage.cpp
CS215Pgm5.cpp

Example Screens from the program:
After loading an image:



After generating a photo-mosaic from 20 by 15 px blocks:



Zoomed in on the image: