

**CS 302 Data Structures**  
**Fall 2009 - Dr. George Bebis**  
**Programming Assignment 2**

**Due date: 10/8/09**

In this assignment, you will template the Image class and the functions you implemented in Assignment 1. Specific objectives include:

- Improve your skills with using templates.
- Learn how to compile your code when using templates.
- Learn to document and describe your programs.

The goal for templating the Image class and the functions associated with it is to make the implementation general enough to handle both gray-scale (PGM) and color (PPM) images. Specifically, your program should be able to read/write/display both gray-level and color images. In addition, it should be able to apply all the image processing transformations you implemented in assignment 1 to both gray-level and color images (e.g., translate a color image, rotate a color image, compute the negative of a color image, etc.).

For demonstration purposes, your driver should read two gray-level and two color images at the beginning of the execution. Then, it should show the results of the image operation chosen by the user on both types of images.

We have already discussed in class the differences between gray-level (i.e., PGM) and color (i.e., PPM) image formats. You need to go over my notes if you do not remember how the two formats are different. You will need two new functions for reading/writing PPM (color) images. These functions are actually very similar to the ones you used in Assignment 1 for reading/writing gray-level images. The code for reading/writing PPM images will be available from the course's webpage.

In color images, each pixel in a color image is represented by three colors: (*Red, Green, Blue*). For example, you can define a new class for color pixels as follows:

```
class RGB {
public:
    functions go here ...
private:
    int r; // red component
    int g; // green component
    int b; // blue component
};
```

Your template Image class will now look as follows:

```
template<class PixelType>
class Image {
public:
    constructor
    destructor
    <rest of the functions ....>
private:
    int N; // no of rows
    int M; // no columns
    int Q; // no gray-levels
    PixelType **pixelVal;
};
```

where *PixelType* will be defined when the user of the class declares an instance (object) of the class, for example:

```
Image<int> grayLevelImage;
Image<RGB> colorImage;
```

What is really **important** to consider when templating the Image class functions is that the implementation of each function should be made as general as possible. In other words, there **should not** be any code in the implementation of a function that is specific to a certain type of images (e.g., color). The idea behind templates is that we should be able to use the **same code**, without making any changes to the implementation, both for gray-level and color images. If you need to make changes to the implementation of the class to support new pixel types, then your code is not efficient and does not hide the implementation details from the user, an important principle of object-oriented programming.

Let us consider, for example, the function *meanGray* which computes the average value of a gray-level image (assume that an integer approximation of the average is enough):

```
int Image::meanGray()
{
    int i, j;
    int avg;

    avg = 0;

    for(i=0; i<N; i++)
        for(j=0; j<M; j++)
            avg = avg + pixelVal[i][j];

    avg = avg / (N*M);
    return avg;
}
```

The templated version of the same function should look like this:

```
template<class PixelType>
PixelType Image<PixelType>::meanValue() // change the name for clarity
{
    int i, j;
    PixelType avg;

    avg = 0;
    for(i=0; i<N; i++)
        for(j=0; j<M; j++)
            avg = avg + pixelVal[i][j];

    avg = avg / (N*M);
    return avg;
}
```

It should be obvious from the example above that we have not changed the implementation of the function. However, certain issues must be addressed now: (1) how does the function know how to initialize a variable of type *PixelType* to 0 ( $avg = 0$ ) (2) how does it know how to add together two variables of type *PixelType* ( $avg + pixelVal[i][j]$ ), (3) how does it know how to assign a value of type *PixelType* to a variable of type *PixelType* ( $avg = avg + pixelVal[i][j]$ ) and, (4) how does it know how to divide a variable of type *PixelType* by an integer ( $avg / (N*M)$ ).

This information must be provided by the user of the class. When declaring an object of type *Image*, then the actual type for *PixelType* must be specified, for example, by passing *RGB* as a parameter to the template. At the same time, the above operations for the type *RGB* must be specified in order to make the implementation of the *Image* class independent from the application.

**Important:** when working with templates, we change the ground rules regarding which file(s)

we put the source code into as we discussed in class (i.e., see my slides on templates).

### **Instructions**

Think carefully about how to template each function in the Image class and provide a discussion for each function in your report. Each function should be discussed in a separate section with the name of the section being the same as the name of the function. The sections should be clearly separated from each other.