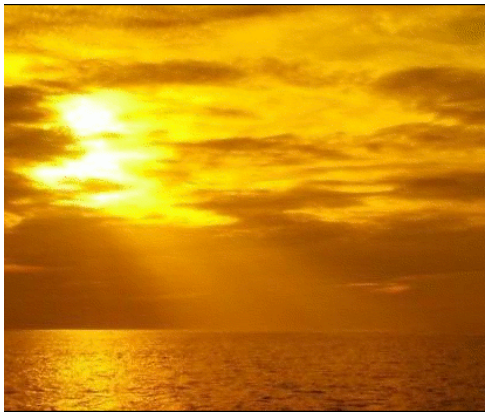
CS 315 Data Structures **Spring 2008**

Project # 2

Due: March 11, 2008

Problem Statement: Write a C++ program to perform the following operations on an image: (a) add a text label to the North-East corner as foreground and (b) apply a recursive transformation that produces several copies of the image and tiles them on an canvas of the same size as the original image. The following figure illustrates the input and the output files for operation (a):



Goals of the project:

- learn to use a C++ library designed for image manipulation
- learn to generate some self-similar images by scaling and recursive tiling.

EasyBMP library:

EasyBMP is a library written in C++ that provides support for opening an image (in the BMP format), and manipulate the individual pixels of the image. You don't have to know the details of how a BMP image is stored in a file. It is enough to know how to make calls to EasyBMP functions. The following are some useful classes and functions from the EasyBMP library:

- *RGBApixel*
  A class that represents a single colored pixel. Each pixel is determined by four color components -- red, green, blue, and alpha. Each of these color components is an unsigned char variable (i.e. an integer between 0 and 255), and is accessed as follows: if v is a pointer to an RGBApixel variable, then the relevant color

components can be accessed using v->Red, v->Green, v->Blue, and v->Alpha, respectively.

- *BMP*
  A class that holds a bitmap image, i.e. a rectangular array of colored pixels (i.e. RGBApixel objects). Once you define a variable of type BMP, you can then manipulate it using the functions listed below.
- bool *BMP::ReadFromFile(const char* FileName)*
  Given the name of an image file, this function loads the image into the BMP class, so that your program can then access the image using member functions listed below. The function returns a bool value indicating whether the read was successful.
- bool *BMP::WriteToFile(const char* FileName)*
  If you want to save the image in your program to disk, this function allows you to do so; you specify the desired filename as the argument to the function. The function returns a bool value indicating whether the write was successful.
- int *BMP::TellWidth()*
  This member function takes no inputs, and returns the number of pixels in the horizontal direction.
- int *BMP::TellHeight()*
  This member function takes no inputs, and returns the number of pixels in the vertical direction.
- RGBApixel* *BMP::operator()(int i, int j)*
  This is an example of an overloaded operator. Here the parentheses have been overloaded, so that given img is an object of type BMP, the expression img(i,j) returns a pointer to the color information stored at the pixel whose *x*-coordinate is i and whose *y*-coordinate is j. The upper-left-hand pixel is given by $i = 0$, $j = 0$ and the positive x-axis is West to East while the positive y-axis is from North to South.

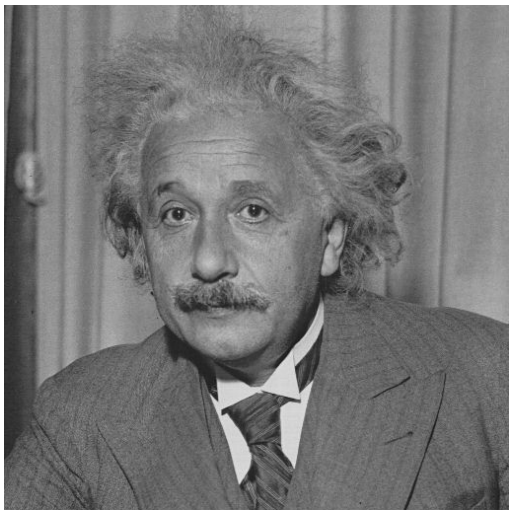
If you are unclear about any of these functions, the manual EasyBMP_UserManual.pdf may be useful and can be found in http://easybmp.sourceforge.net/documentation.html. The manual also contains some examples with source code.

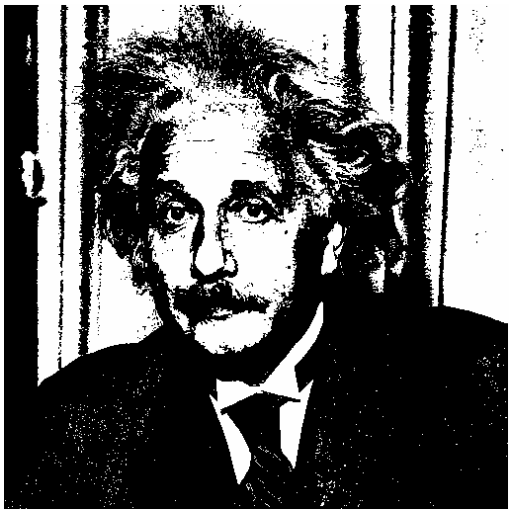
In the following, a simple example is presented to illustrate the use of *EasyBMP*.

Suppose we want to convert a color image or a gray scale image into black and white. This is a tricky process, but a very important one – for example when we want to print a color photo on a black and white printer, we need such an algorithm. We will use a very simple-minded algorithm called *thresholding*. Recall that each color pixel is defined by three color components (R, G, B), each taking a value between 0 and 255. R = G = B = 255 is white, R = G = B = 0 is black. In thresholding, the average the color components of a pixel (i.e., compute (R + G + B)/3) is computed and if this

average exceeds 127, this pixel to mapped to white, otherwise it is mapped to black. In the code below, we take a weighted average by weighting the colors by 0.3, 0.6 and 0.1 (since green is the most sensitive and blue is the least sensitive.) We present the code to implement this algorithm below.

An example of input/output is shown below.





The code is as follows:

```
#include "EasyBMP.h"
using namespace std;

int main( int argc, char* argv[] )
{
 BMP Background;
 Background.ReadFromFile(argv[1]);
 BMP Output;
 int picWidth = Background.TellWidth();
 int picHeight = Background.TellHeight();
 Output.SetSize(picWidth, picHeight);
 Output.SetBitDepth(1);

 for (int i = 1; i < picWidth-1; ++i)
    for (int j = 1; j < picHeight-1; ++j) {
          Output(i,j)->Red = 0;
          Output(i,j)->Blue = 0;
          Output(i,j)->Green = 0;
    }

 for (int i = 1; i < picWidth-1; ++i)
   for (int j = 1; j < picHeight-1; ++j) {
         int col = 0.1* Background(i, j)->Blue +
           0.6*Background(i,j)->Green +0.3* Background(i,j)->Red;
         if (col > 127) {
          Output(i,j)->Red = 255;
          Output(i,j)->Blue = 255;
```

```
            Output(i,j)->Green = 255;
            }
          }
    Output.WriteToFile(argv[2]);
    return 0;
  }
```

The resulting b/w image is much smaller in size.

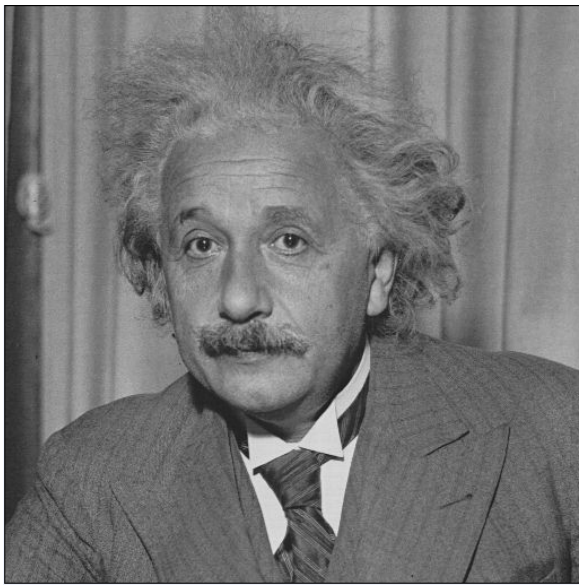
Solution to the two problems:

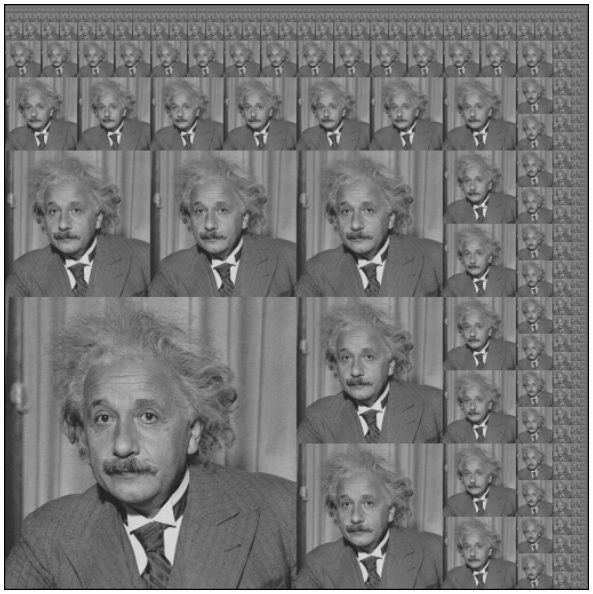
Problem (1)

To label the given image with a given text, you need access to a bit-map image of each letter. For the purpose of this lab, you can assume that the font size of the letters in the label is fixed. (i.e., you will not be required to rescale the font files.) We will assume that the label consists of an alphabetical symbol or a blank space. Also assume that the font images are black in color with black background. The font images will be provided to you. The basic idea is to first copy the background image onto a buffer. Then, your program should overwrite the pixels on the upper-right corner by copying the bit-map images corresponding to the label. For example, if the label is the word "text", your program will calculate the position in which each of the letters t, e, x and t should be written. It should then copy the image `t.bmp, e.bmp etc.` in the appropriate position. Note that when your write a letter, only the letter (not the background) should be copied. You can assume that the heights of all the font images are the same. However, they have varying widths so your program should use the width information to ensure a uniform spacing between letters. If the space needed to typeset the label exceeds the width of the image, your program should display an error message and terminate without adding any text to the image.

Problem (2)

The goal of this problem is to make varying size copies of an image and place them as tiles in a square of the same size as the original image. The precise task is described below formally, but the illustration shown below informally conveys the task to be accomplished. The image on the left is the input and the one on the right is the output.





Let I be the input image and O the output image. Also let Q(O) denote a quadrant of O. (Q may be NE = North East, SW etc.) The South-West (SW) quadrant of O is a scaled down (by a factor of 2) copy of I. The NE(O) = NE quadrant of O is a scaled copy of O itself. NW(O) is such that its lower two quadrants are scaled down (by a factor of 4) copies of I, and the other two quadrants are scaled down (by a factor of 2) versions of NW(O). Similarly, SE(O) has two quadrants that are scaled down versions of I while the other two are scaled down versions of itself. The recursion continues until a 1 x 1 image is reached. In this case, we set O = I. We will assume that the input image is a square image and its height and width are powers of 2 so that we can apply the transformations described above without distortions.

Scaling down an image: To scale down an image of size 2k x 2k to an image of size k x k, you will implement the following algorithm: average the pixels [0,0], [0,1], [1,0] and [1, 1] and store it in [0,0]. More generally, the average of the pixels [2k,2k], [2k, 2k+1], [2k+1, 2k] and [2k+1, 2k+1] will be stored in pixel [k,k].

The size of the complete program is about 200 lines. The recursive procedure to implement problem 2 is about 30 lines long.

What should be submitted?

Your program can contain individual files that implement the solutions to problems 1 and 2. When your main program is run (after compiling it through the makefile you submit), it should take as input several image files, and produce as output three image files. The names of the main image files are `in1.bmp` and `in2.bmp`; in addition, your program will use `a.bmp, b.bmp, … , z.bmp and space.bmp` that are stored in a

directory. This directory path will be specified later. The file `in1.bmp` is the file to which you should add the label (problem 1). The file `in2.bmp` is the one to which you should apply the transformation described in problem 2. Also assume that a file named `label.txt` will contain the label to be added. This file is also in the same directory. The output files should be named `out1.bmp, out2.bmp` and `out3.bmp`.

Grading:

Each problem will be weighted 50 points.

Problem 1: getting the fonts displayed - 35 points, uniform spacing - 10 points, error check - 5 points

Problem 2: correct scaling – 20 points, correct implementation of recursion – 30 points