EECS 22: Assignment 4

Prepared by: Weiwei Chen, Prof. Rainer Doemer

November 1, 2012

Due on Monday 11/19/2012 11:59pm. Note: this is a two-week assignment.

1 Digital Image Processing [100 points + 10 bonus points]

In this assignment you will learn how to use dynamic memory allocation in your program and how to link against libraries. Based on the program *PhotoLab* for assignment3, you will be first asked to redesign your digital image processing (DIP) operations to accommodate images with different sizes, and then add more DIP operations whose results are images with different sizes from the original one. Thus you can use your *PhotoLab* program to perform the DIP operations on any of your own pictures.

1.1 Introduction

In assignment3, you were asked to decompose your *PhotoLab* program into separate modules and compile them into different programs. The user can load an image from a file, apply a set of DIP operations to the image, and save the processed image in a file by using the *PhotoLab*; or they can using the *Test* program to perform all the DIP operations automatically. This assignment will be an extension of assignment3.

1.2 Initial Setup

Before you start working on this assignment, please do the following steps:

1. Create the subdirectory *hw4* for this assignment, and change your current directory to *hw4*.
2. We will modify and extend the *PhotoLab* program based on assignment3. Please feel free to reuse all your source code (*.c) and header files (*.h) as the starting point for this assignment. You may reuse the solution files to assignment3 which are published on our course website as well. Copy all the source code files (*.c) and the header files (*.h) to *hw4* except **FileIO.h** and **FileIO.c**
3. Copy the new header file for File I/O and the FileIO library file from the *eecs22* account:

   ```
   cp ~eecs22/hw4/FileIO.h ./
   cp ~eecs22/hw4/Image.h ./
   cp ~eecs22/hw4/libfileio.a ./
   ```

   Here,

   - **Image.h** is the header file for the definition of the new structure and declarations of the pixel mapping functions we will use in Section 1.3.2;
   - **FileIO.h** is the new header file for File I/Os, i.e. ReadImage() and SaveImage().
   - **libfileio.a** is the File I/O library.
4. Copy your **Makefile** to *hw4*.

We will still use the PPM image file *ucisailing.ppm* as the test file for this assignment. As before, once a DIP operation is done, you can save the modified image as *name.ppm*, and it will be automatically converted to a JPEG image and sent to the folder *public_html* in your home directory. You will be able to see the images in any web browser at: *http://newport.eecs.uci.edu/∼youruserid*, if required names are used. If you save images by other names, use the link *http://newport.eecs.uci.edu/∼youruserid/imagename.jpg* to access the photo.

Note that whatever you put in the *public_html* directory will be publicly accessible; make sure you don't put files there that you don't want to share, i.e. do not put your source code into that directory.

NOTE: When you finish this assignment, your *PhotoLab* program will be able to manipulate any images (as long as there is enough memory for the program and disk space for the output images). A digital camera usually takes pictures and stores them in the format of JPEG. In order to use the *PhotoLab* program, you need to first convert your JPEG picture (*.jpg, *.JPG, *.jpeg, *.JPEG) into the PPM (*.ppm) raw data format. You can use the following linux command for the conversion:

```
jpegtopnm yourfilename.jpg > yourfilename.ppm
```

1.3 Add support for different image sizes

In this assignment, we will add support for DIP operations on images with different sizes. In the previous two assignments, our programs define two constants **WIDTH** and **HEIGHT** as the fixed size of the input image. At that time, our *PhotoLab* program can only manipulate images with the size of 720x538.

In order to add the support for different image size, we need to redefine size of the arrays that we are using to store the color intensity information for each pixels. The size of the input image cannot be known at program compile time. Thus, we cannot define arrays whose size cannot be determined at the program compile time. Therefore, we need to use dynamic memory allocation to claim three blocks of memory whose size will be decided at the program run time, and these memory spaces will then be used to store the color intensity values for each pixel of the input image.

Instead of defining three arrays and pass them as the arguments to the DIP operation functions, we will now use pointers to point to an image structure and the memory space that will be dynamically allocated by our program at run time.

1.3.1 Use pointers to one dimensional memory space instead of arrays with two dimensions

We need to use dynamic memory allocation since the size of the image will not be known until we run the program. We will use three pointers to type *unsigned char* for the color intensity values for each pixel instead of three fixed sized arrays. However, the pointers only points to the memory space that has only one dimension. Therefore, we need to map the 2-tuple coordinates of the pixels to a single value to index the corresponding pixel color information from the memory space pointed to by the pointers.

For example, we have an image of size 10x5, and three pixels (0, 0), (9, 4), and (6, 4). We assume row major for the image storage in this program. Therefore, the index value for pixel (0, 0) in the one dimensional storage space will be 0; the index value for pixel (9, 4) in the one dimensional storage space will be $49 = 9 + 4 * 10$; and the index value for pixel (6, 4) in the one dimensional storage space will be $46 = 6 + 4 * 10$.
In general, the index value for the pixel (*x*, *y*) in an image of size WIDTHxHEIGHT in the one dimensional storage space will be ***x* + *y* * WIDTH**.

1.3.2 The Image.c module

Please add one module **Image.c** (**Image.h** see the provided one) to handle basic operations on the image.

- **The IMAGE struct**: We will use a *struct type* to aggregate all the information of one image. The following struct is defined in **Image.h**:

```
typedef struct {
unsigned int Width; /* image width */
unsigned int Height;  /* image height */
unsigned char *R; /* pointer to the memory storing all the R intensity values */
unsigned char *G; /* pointer to the memory storing all the G intensity values */
unsigned char *B; /* pointer to the memory storing all the B intensity values */
}IMAGE;
```

- Define the functions to get / set the value of the color intensities for each pixel in the image. Please use the following function prototypes (provided in **Image.h**) and define the functions properly (in **Image.c**)

```
/*Get the color intensity of the Red channel of pixel (x, y) in image*/
unsigned char GetRPixel(IMAGE *image, unsigned int x,  unsigned int y);

/*Get the color intensity of the Green channel of pixel (x, y) in image*/
unsigned char GetGPixel(IMAGE *image, unsigned int x,  unsigned int y);

/*Get the color intensity of the Blue channel of pixel (x, y) in image*/
unsigned char GetBPixel(IMAGE *image, unsigned int x,  unsigned int y);

/*Set the color intensity of the Red channel of pixel (x, y) in image with value r*/
void SetRPixel(IMAGE *image, unsigned int x,  unsigned int y, unsigned char r);

/*Set the color intensity of the Green channel of pixel (x, y) in image with value g*/
void SetGPixel(IMAGE *image, unsigned int x,  unsigned int y, unsigned char g);

/*Set the color intensity of the Blue channel of pixel (x, y) in image with value b*/
void SetBPixel(IMAGE *image, unsigned int x,  unsigned int y, unsigned char b);
```

  The mapping from the 2-tuple coordinates (x, y) to the single index value for the one dimensional memory space will be taken care of in this these functions. Please use these functions in your DIP functions for setting / getting the intensity values of the images.

- Please add **assertions** in these functions to make sure the input image point is valid, and the set of pointers to the memory spaces for the color intensity values are valid too.

- Please extend the **Makefile** accordingly: 1) the target to generate **Image.o** 2) add **Image.o** when generating **PhotoLab** and **Test**.

1.3.3 Read and save image files

You may refer to **FileIO.h** for the defined functions for file I/Os.

- **int GetImageSize(const char *fname, unsigned int *Width, unsigned int *Height):** opens the image file with the name *fname.ppm*, and sets the value of the *Width* and *Height* with the value of the width and height of the image respectively.

- **IMAGE *ReadImage(const char *fname):** reads the file with the name *fname.ppm* and return the image pointer. The color intensities for channel red, green, and blue are stored in the memory spaces pointed by the member pointers *R*, *G* and *B* of the returned IMAGE pointer respectively. The memory space of the image is created in this function.

- **int SaveImage(const char *fname, IMAGE *image):** saves the color intensities for channel red, green, and blue stored in the memory spaces pointed by member pointers *R*, *G* and *B* of *image* into the file with the name *fname.ppm*.

In ReadImage() function, the memory spaces to the member pointers *R*, *G* and *B* will be created. At the end of your program, you need to free these memory spaces to avoid memory leakage.
Please write two functions to handle the memory allocations and releases in **Image.c** (Functions declared in **Image.h**).

Please use the following function prototypes.

```
/* allocate the memory spaces for the image              */
/* and the memory spaces for the color intensity values. */
/* return the pointer of the image                       */
IMAGE *CreateImage(unsigned int Width, unsigned int Height);

/*release the memory spaces for the pixel color intensity values */
/*release the memory spaces for the image                        */
void DeleteImage(IMAGE *image);
```

NOTE: the ReadImage() function in the FileIO library needs the CreateImage() function to allocate the memory space.

1.3.4 Modify function prototypes and definitions

All of our functions need to be redefined by taking the IMAGE structure as the parameter which contains all the information about the image.

Your DIP function prototypes may look like below:

- In **DIPs.h**:

```
/* change color image to black & white */
void BlackNWhite(IMAGE *image);

/* reverse image color */
void Negative(IMAGE *image);

/* flip image horizontally */
void HFlip(IMAGE *image);

/* mirror image horizontally */
void HMirror(IMAGE *image);

/* Add a border to the image */
void AddBorder(IMAGE *image, int r, int g, int b, int bwidth);

/* flip image vertically */
void VFlip(IMAGE *image);

/* mirror image vertically */
void VMirror(IMAGE *image);
```

- In **Advanced.h**:

```
/* aging the image */
void Aging(IMAGE *image);

/* Posterization */
void Posterize(IMAGE *image,
   unsigned char rbits,
   unsigned char gbits,
   unsigned char bbits);
```

```
/* blur the image */
void Blur(IMAGE *image);

/* detect the edge of the image */
void EdgeDetection( double K, IMAGE *image);

/* Test all functions */
void AutoTest(const char *fname);
```

NOTE: By using pointers to one dimensional memory space, you need to modify the statements in your functions for array elements' indexing with the pixel setting / getting functions accordingly. For example:

- In assignment3, we get the pixel's color value by indexing the element from the two-dimensional array: *tmpR = R[x][y]*;
- Now, we get the pixel's color value by calling the getting function: *tmpR = GetRPixel(image, x, y)*;
- In assignment3, we set the pixel's color value by indexing the element from the two-dimensional array: *R[x][y] = ...*;
- Now, we set the pixel's color value by calling the setting function: *SetRPixel(image, x, y, ...)*;

By using the setting / getting functions, we can keep the two-dimensional coordinate system as assignment2 and assignment3.

Please make sure to include the translation unit **Image.h** properly in your source code files and header files.

1.3.5 Modify the AutoTest() function

Please put the File I/O and memory allocations inside the *AutoTest()* function. The only parameter this function will take is the name of the image file that will be used for testing. Please refer to Section 1.5 for more implementation details.

1.3.6 Modify the Makefile to link against the library

We will use the provided library for File I/Os in this assignment. Please adjust your **Makefile** accordingly so as not to compile the object file of FileIO.o, but use the provided **libfileio.a** in stead.

Your own **Makefile** should have at least the following targets:

- *all*: the target to generate all the executable programs.
- *clean*: the target to clean all the intermedia files, e.g. object files, the executable programs, and the generated ppm files.
- **.o*: the target to generate the object file *.o from the C source code file *.c. **FileIO.o is no longer needed**.
- *PhotoLab*: the target to generate the executable program *PhotoLab*.
- *Test*: the target to generate the executable program *Test* which only calls the *AutoTest()* function.

Compile your source code into *PhotoLab* and *Test* by using your **Makefile**:

```
make all
```

HINT: There are two ways to link against a library (.a) file:

1. use it as a normal object file with full name.
2. use the *-l* option of *gcc* to specify which library to use (e.g. *-lfileio* means using the library *libfileio.a*), and the *-L* option of *gcc* to specify the directory of the library.

1.4 Advanced DIP operations

In this assignment, please implement the advanced DIP operations described below in **Advanced.c** (**Advanced.h** as the header file).

Please reuse the menu you designed for assignment3 and extend it with the advanced operations. The user should be able to select DIP operations from a menu as the one shown below:

```
--------------------------------
 1:  Load a PPM image
 2:  Save an image in PPM and JPEG format
 3:  Change a color image to black and white
 4:  Make a negative of an image
 5:  Flip an image horizontally
 6:  Mirror an image horizontally
 7:  Add border to an image
 8:  Flip an image vertically
 9:  Mirror an image vertically
10:  Age the image
11:  Posterize the image
12:  Blur an image
13:  Detect edges
14:  Rotate 90 degrees clockwise
15:  Resize the image
16:  Generate the Mandelbrot image
17:  Overlay an image
18:  Test all functions
19:  Exit
```

Note: option '14: Overlay an image' is a bonus question (10pts). If you decide to skip this option, you still need to implement the option '15: Test all functions'.

1.4.1 Rotate-90-degree

This function rotates the image by 90 degrees clockwise. The size of the image will be the same, but the width (height) of the new image will be the same as its original height (width).

NOTE: As shown in Fig. 2, for pixel indices, the row increases downward, while the column increases to the right. Pixel indices are integer values, and range from 1 to the length of the row or column. The top left pixel's coordinate is (0, 0), and the bottom right pixel's coordinate is (image→Width - 1, image→Height - 1).

You need to first find the coordinates' mapping of the pixels from the original image (x, y) to the rotated image (x', y'). Then, set the color of the pixel at (x', y') in the new image to be the same as the color of the pixel at (x, y) in the original image.

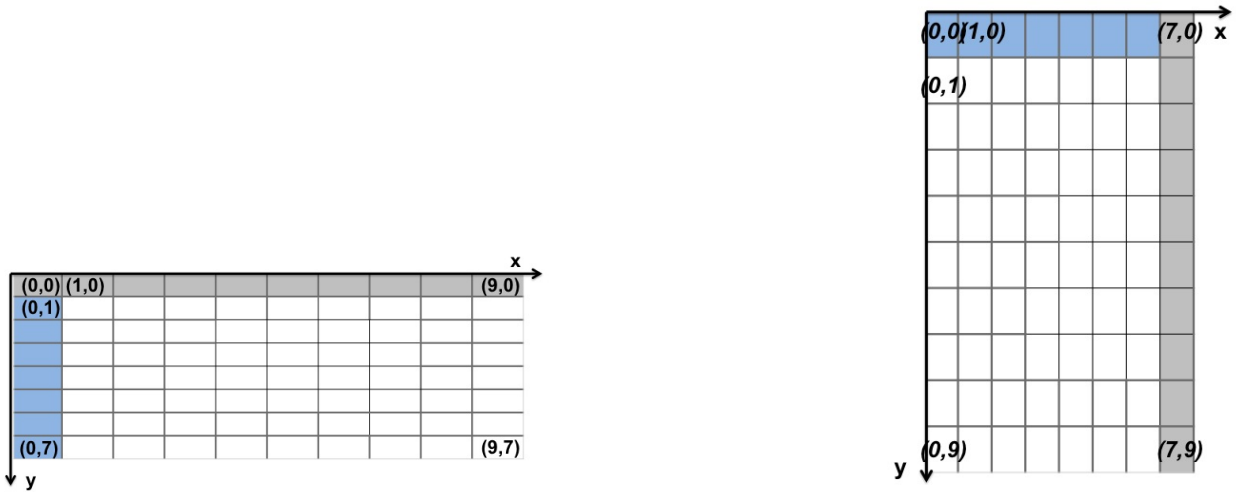
You need to define and implement the following function to do this DIP.

```
/*Rotate 90 degrees clockwise*/
IMAGE *Rotate(IMAGE *image);
```

(a) Original image

(b) Rotated image

Figure 1: An image and its rotatedd counterpart.



(a) Coordinates for the original image

(b) Coordinates for the rotated image

Figure 2: An image and its rotatedd counterpart.

(a) Original image

(b) resized to a bigger image (percentage = 150)

(c) resized to a smaller image (percentage = 80)

Figure 3: An image and its resized bigger and resized smaller counterparts.

NOTE: The *Resize()* function will either consume the input image and return a new image with the rotated size, or do in place modifications on the input image and return itself then.

Figure 1 shows an example of this operation. Once the user chooses this option, your program's output should like this:

```
Please make your choice: 14
"Rotate 90 degree clockwise" operation is done!

--------------------------------
 1:  Load a PPM image
 2:  Save an image in PPM and JPEG format
 3:  Change a color image to black and white
 4:  Make a negative of an image
 5:  Flip an image horizontally
 6:  Mirror an image horizontally
 7:  Add border to an image
 8:  Flip an image vertically
 9:  Mirror an image vertically
10:  Age the image
11:  Posterize the image
12:  Blur an image
13:  Detect edges
14:  Rotate 90 degrees clockwise
15:  Resize the image
16:  Generate the Mandelbrot image
17:  Overlay an image
18:  Test all functions
19:  Exit
Please make your choice:
```

Save the image with the name 'rotate' after this step.

1.4.2 Resize

This function resized the image with the scale of *percentage*.

- *percentage* $==$ 100, the size of the new image is the same as the original one.
- *percentage* $<$ 100, the size of the new image is smaller than the original one.

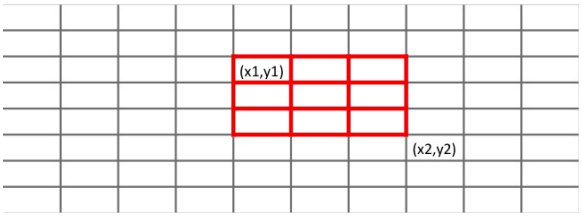


Figure 4: Pixels mapping from the bigger original image to the smaller new image

- *percentage* > 100, the size of the new image is bigger than the original one.

More specifically, with the scale of *percentage*,

- $Width_{new} = Width_{old}$ * (percentage / 100.00);
- $Height_{new} = Height_{old}$ * (percentage / 100.00);

If *percentage* is greater than 100, we need to duplicate some pixels from the original image to the new bigger one. (x', y') is the position coordinates for the pixel in the new image, (x, y) is the coordinates of the pixel in the original image. Copy the color of the pixel(x, y) in the original image, to pixel (x', y') in the new image, and in this case:

```
x = x' / (percentage / 100.00);
y = y' / (percentage / 100.00);
```

If *percentage* is smaller than 100, we will have fewer pixels in the new smaller image than in the original image. In order not to lose too much information from the original image, we get the average value of the color intensities of multiple pixels in the original image and use this average value as the color intensity of one pixel in the smaller image.

More specifically, as shown in Figure 4, each grid representing one pixel in the image, we will get the average value for the color intensities of all the red edged pixels in the original image (from (x_1, y_1) to ($x_2 - 1$, $y_2 - 1$)), and use this average value as the color intensity of the pixel (x, y) in the new image, where:

```
x1 = x / (percentage / 100.00);
y1 = y / (percentage / 100.00);
x2 = (x + 1) / (percentage / 100.00);
y2 = (y + 1) / (percentage / 100.00);
```

You need to define and implement the following function to do this DIP.

```
/*Resize*/
IMAGE *Resize(unsigned int percentage, IMAGE *image);
```

NOTE: The *Resize()* function will consume the input image and return a new image with the new size. Please delete and create the images properly in this function.

Figure 3 shows an example of this operation. Once the user chooses this option, your program's output should like this:

```
Please make your choice: 15
Please input the resizing percentage (integer between 1~500): 150
"Resizing the image" operation is done!

--------------------------------
 1:  Load a PPM image
 2:  Save an image in PPM and JPEG format
```

```
 3:  Change a color image to black and white
 4:  Make a negative of an image
 5:  Flip an image horizontally
 6:  Mirror an image horizontally
 7:  Add border to an image
 8:  Flip an image vertically
 9:  Mirror an image vertically
10:  Age the image
11:  Posterize the image
12:  Blur an image
13:  Detect edges
14:  Rotate 90 degrees clockwise
15:  Resize the image
16:  Generate the Mandelbrot image
17:  Overlay an image
18:  Test all functions
19:  Exit
Please make your choice:
```

Save the two images for this operation:

1. 'resizeb': a bigger image with scale *percentage* = 150.
2. 'resizes': a smaller image with scale *percentage* = 80.

1.4.3 Image for the Mandelbrot Set

The **Mandelbrot set** is a mathematical set of points whose boundary is a distinctive and easily recognizable two-dimensional fractal shape. Images of the Mandelbrot set display an elaborate boundary that reveals progressively ever-finer recursive detail at increasing magnifications. The Mandelbrot set has become popular outside mathematics both for its aesthetic appeal and as an example of a complex structure arising from the application of simple rules, and is one of the best-known examples of mathematical visualization. (`http://en.wikipedia.org/wiki/Mandelbrot_set`)
We are going to write a function to generate an image of the Mandelbrot set in this assignment.

- **The algorithm for drawing a picture of the Mandelbrot set**
  In this DIP operation, we will try to translate the pseudo code of the Mandelbrot set drawing algorithm into C program.

  A lot of real-world programming work requires the programmer to reuse the source code that are written by the others and adjust accordingly to fit into the current project. The reference source code can have different function prototypes, different variable types, and even be written in different programming languages or just pseudo codes. While the reference code provides the base control flow to implement the major behavior of the functions, programmers need to translate the reference code to their working language and build the interfaces for integration.

  As stated on the wikipedia webpage, the pseudo code of the Mandelbrot set drawing algorithm looks like as follows:

```
1: For each pixel on the screen do:
2: {
3:  x0 = scaled x coordinate of pixel within the range of (-2.5, 1)
4: (must be scaled to lie somewhere in the mandelbrot X scale (-2.5, 1)
5: y0 = scaled y coordinate of pixel within the range of (-1, 1)
6: (must be scaled to lie somewhere in the mandelbrot Y scale (-1, 1)
```

```
 7:
 8:  x = 0
 9:  y = 0
10:
11:  iteration = 0
12:  max_iteration = 1000
13:
14: while ( x*x + y*y < 2*2  AND  iteration < max_iteration )
15: {
16:    xtemp = x*x - y*y + x0
17:    y = 2*x*y + y0
18:
19:    x = xtemp
20:
21:   iteration = iteration + 1
22: }
23:
24: color = iteration
25:
26: plot(x0,y0,color)
27: }
```

Here is a brief explanation for this algorithm. The algorithm computes the color for each pixels in the picture based one their coordinates. First, the coordinates of the pixel will be scaled as listed in line 3-6. The x coordinate will be scaled to fall into the range of (-2.5, 1), and the y coordinate will be scaled to fall into the range of (-1, 1). After several initializations, the algorithm starts a while loop to do some computation. The iterations of the loop is decided by the value of x*x+y*y and a constant value stored in *max_iteration* as the maximum boundary for the loops. The color of the pixel is decided by the actual number of the iterations. The plot() function will draw the pixel(x, y) with the color that is determined by the loop iterations.

To implement this algorithm in C, we need to define variables with proper types and design the scaling equations.

Moreover, we need to specific the actual color that is represented by a specific iteration number. We suggest to use 16 colors in the Mandelbrot image. Thus, we first change line 24 into *color = iteration %16*, and then use the variable *color* as the index to get the color intensity values from an array *palette* for different colors.

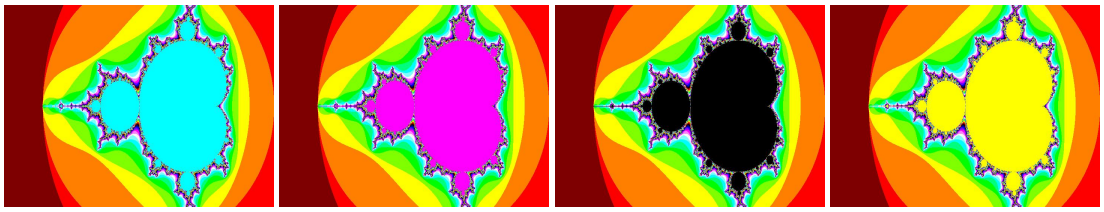
The following definition can be used in the C program:

1. In Advance.h

   ```
   #define MAX_COLOR 16
   ```

2. Use the following array definition for the color palette:

   ```
   const unsigned char palette[MAX_COLOR][3] = {
           /* r   g   b*/
           {   0,   0,   0 },      /* 0, black            */
           { 127,   0,   0 },      /* 1, brown            */
           { 255,   0,   0 },      /* 2, red              */
           { 255, 127,   0 },      /* 3, orange           */
           { 255, 255,   0 },      /* 4, yellow           */
           { 127, 255,   0 },      /* 5, light green      */
           {   0, 255,   0 },      /* 6, green            */
           {   0, 255, 127 },      /* 7, blue green       */
           {   0, 255, 255 },      /* 8, turquoise        */
   ```



(a) max_iteration = 1000 (b) max_iteration = 1500 (c) max_iteration = 2000 (d) max_iteration = 2500

Figure 5: Images for the Mandelbrot set with different number of iterations.

```
            { 127, 255, 255 },      /* 9, light blue        */
            { 255, 255, 255 },      /* 10, white            */
            { 255, 127, 255 },      /* 11, pink             */
            { 255,   0, 255 },      /* 12, light pink       */
            { 127,   0, 255 },      /* 13, purple           */
            {   0,   0, 255 },      /* 14, blue             */
            {   0,   0, 127 }       /* 15, dark blue        */
    };
```

- **Function Prototype**
  You need to define and implement the following function to do this DIP.

  ```
  /*Mandelbrot*/
  IMAGE *Mandelbrot(unsigned int W, unsigned int H, unsigned int max_iteration);
  ```

  Here, **W** is the width of the generated image, **H** is the height of the image, and **max_iteration** is the maximum iterations for the computation of each pixel. This function will create an image, fill in the pixels with different colors according to the Mandelbrot drawing algorithm, and return this image at the end.

Once user chooses this option, your program's output should be like:

```
Please make your choice: 16
Please input the width of the mandelbrot image: 720
Please input the height of the mandelbrot image: 538
Please input the max iteration for the mandelbrot calculation: 2000
"Mandelbrot" operation is done!

--------------------------------
 1:  Load a PPM image
 2:  Save an image in PPM and JPEG format
 3:  Change a color image to black and white
 4:  Make a negative of an image
 5:  Flip an image horizontally
 6:  Mirror an image horizontally
 7:  Add border to an image
 8:  Flip an image vertically
 9:  Mirror an image vertically
10:  Age the image
11:  Posterize the image
12:  Blur an image
13:  Detect edges
14:  Rotate 90 degrees clockwise
```

(a) A second image (170x150)

(b) A third image (100x54)

(c) Overlay the pirate image at position (220, 300), and the solider image at position (600, 340)

Figure 6: An image and the overlayed image.

```
15:  Resize the image
16:  Generate the Mandelbrot image
17:  Overlay an image
18:  Test all functions
19:  Exit
Please make your choice:
```

Fig. 5 shows four Mdandelbrot images with different computation iterations.

Save the image with the name "mandelbrot" after this step.

1.4.4 Image Overlay (bonus: 10 pts)

This function overlies the current image with a second image. In our program, we will put an image of a lego pirate boat and an image of a dingy with two lego soldiers on the original image.

To start the implementation, you need to prompt the user to enter the file name of the second image first, and then you read that image in the beginning of the overlay function (define the second IMAGE image2, and use *CreateImage()* and *ReadImage()* for loading). In this assignment, the second image is **pirate.ppm**(170x150 pixels). Since the image is much smaller than the original one, the user also need to enter the position of overlay with coordinates (x, y).

Take a look at the second image at Figure 6. Note that it has a white background and the blue water part which are inconsistent with our original image. To achieve the overlay effect, we will treat the background and the blue part in the second image as transparent color. That is, each of the non-background/non-blue pixels in rowing.ppm will be overlaid to a position in the original image, whereas background/blue pixels will stay as in the original image. Whether or not a pixel in rowing.ppm is a background/blue pixel can be decided by the RGB values of this pixel. More specifically, if a pixel has RGB value greater than (250, 250 ,250) ($r \geq 250$, $g \geq 250$, $b \geq 250$) which represents white background, this pixel should not be put onto original image.

You need to define and implement the following function to do this DIP.

```
/*Overaly a small image onto the original big image*/
void Overlay(const char *f2name,IMAGE *image,
             unsigned int x_offset, unsigned int y_offset);
```

Use the following command to get the **pirate.ppm** and the **soldier.ppm** file.

```
cp ~eecs22/hw4/pirate.ppm ./
cp ~eecs22/hw4/soldier.ppm ./
```

Once user chooses this option, your program's output should be like:

```
Please make your choice: 17
Please input the file name for the second image: pirate
Please input x coordinate of the overlay image: 220
Please input y coordinate of the overlay image: 300
pirate.ppm was read successfully!
"Image Overlay" operation is done!

--------------------------------
 1:  Load a PPM image
 2:  Save an image in PPM and JPEG format
 3:  Change a color image to black and white
 4:  Make a negative of an image
 5:  Flip an image horizontally
 6:  Mirror an image horizontally
 7:  Add border to an image
 8:  Flip an image vertically
 9:  Mirror an image vertically
10:  Age the image
11:  Posterize the image
12:  Blur an image
13:  Detect edges
14:  Rotate 90 degrees clockwise
15:  Resize the image
16:  Generate the Mandelbrot image
17:  Overlay an image
18:  Test all functions
19:  Exit
Please make your choice: 17
Please input the file name for the second image: soldier
Please input x coordinate of the overlay image: 600
Please input y coordinate of the overlay image: 340
soldier.ppm was read successfully!
"Image Overlay" operation is done!

--------------------------------
 1:  Load a PPM image
 2:  Save an image in PPM and JPEG format
 3:  Change a color image to black and white
 4:  Make a negative of an image
 5:  Flip an image horizontally
 6:  Mirror an image horizontally
 7:  Add border to an image
 8:  Flip an image vertically
 9:  Mirror an image vertically
10:  Age the image
11:  Posterize the image
12:  Blur an image
```

```
13:  Detect edges
14:  Rotate 90 degrees clockwise
15:  Resize the image
16:  Generate the Mandelbrot image
17:  Overlay an image
18:  Test all functions
19:  Exit
Please make your choice:
```

The effect can be seen in Figure 6 when position is chosen as (220, 300) for the pirate boat and (600, 240) for the soldier's dingy.

Save the image with the name "overlay" after this step.

1.5 Test all functions

Finally, you are going to complete the *AutoTest()* function to test all previous functions as in assignment3. In this function, you are going to call DIP functions one by one and observe the results. The function is for the designer to quickly test the program, so you should supply all necessary parameters when testing. We will change the function signature for *AutoTest()* so that the creation and deletion of the image will be taken care of inside this function.

The function should look like:

```
/* auto test*/
void
AutoTest(const char *fname)
{
IMAGE *image;

image = ReadImage(fname);
BlackNWhite(image);
SaveImage("bw", image);
printf("Black & White tested!\n\n");
DeleteImage(image);

...


image = ReadImage(fname);
Rotate(image);
SaveImage("rotate", image);
printf("Rotate 90 degrees clockwise tested!\n\n");
DeleteImage(image);

image = ReadImage(fname);
image = Resize(150, image);
SaveImage("resizeb", image);
printf("Resizing big tested!\n\n");
DeleteImage(image);

image = ReadImage(fname);
image = Resize(80, image);
SaveImage("resizes", image);
printf("Resizing small tested!\n\n");
```

```
DeleteImage(image);

image = Mandelbrot(720, 538, 2000);
SaveImage("mandelbrot", image);
printf("Generate the mandelbrot image tested!\n\n");
DeleteImage(image);

image = ReadImage(fname);
Overlay("pirate", image, 220, 300);
Overlay("soldier", image, 600, 340);
SaveImage("overlay", image);
printf("Overlay tested!\n\n");
DeleteImage(image);
image = NULL;
}
```

Please hard-coded "ucisailing" as the function argument when *AutoTest()* is called in *PhotoLab*. Please allow the user to input the file name to be tested when *AutoTest()* is called in *Test*.

Once user chooses this option, your program's output should be like:

```
Please make your choice: 19
ucisailing.ppm was read successfully!
bw.ppm was saved successfully.
bw.jpg was stored for viewing.
Black & White tested!

...


ucisailing.ppm was read successfully!
rotate.ppm was saved successfully.
rotate.jpg was stored for viewing.
Rotate 90 degrees clockwise tested!

ucisailing.ppm was read successfully!
resizeb.ppm was saved successfully.
resizeb.jpg was stored for viewing.
Resizing big tested!

ucisailing.ppm was read successfully!
resizes.ppm was saved successfully.
resizes.jpg was stored for viewing.
Resizing small tested!

mandelbrot.ppm was saved successfully.
mandelbrot.jpg was stored for viewing.
Generate the mandelbrot image tested!

ucisailing.ppm was read successfully!
pirate.ppm was read successfully!
soldier.ppm was read successfully!
overlay.ppm was saved successfully.
overlay.jpg was stored for viewing.
Overlay tested!
```

1.6 Extend the Makefile

For the **Makefile**, please

- extend it properly with the targets for your program with the new module: **Image.c**.
- generate two executable programs
  1. *PhotoLab* with the user interactive menu.
  2. *Test* without the user menu, but just call the *AutoTest()* function for testing. The needed file **Test.c** will be very simple (only the *main()* function, getting the test file name from the user, and calling *AutoTest()*).

  Define two targets to generate these two programs respectively. There's no need to worry about the "DEBUG" mode for this assignment. You may either take those statements for supporting the "DEBUG" mode out of your source code, or just compile your object files with the "-DDEBUG" option.

1.7 Use the "valgrind" tool to Find Memory Leaks and Invalid Memory Use

Valgrind is a multipurpose code profiling and memory debugging tool for Linux. It allows you to run your program in *Valgrind*'s own environment that monitors memory usage such as calls to malloc and free. If you use uninitialized memory, write off the end of an array, or forget to free a pointer, *Valgrind* can detect it. You may refer to `http://valgrind.org/` for more details about the *Valgrind* tool.

In this assignment, please use the follow command to check the correctness of your memory usages:

```
valgrind --leak-check=full programname
```

If there's no problem with the memory usage in your program, you will see some information similar as below when finishing executing your program:

```
==xxxxx==
==xxxxx==
==xxxxx==  HEAP SUMMARY:
==xxxxx==      in use at exit: 0 bytes in 0 blocks
==xxxxx==   total heap usage: 142 allocs, 142 frees, 25,559,076 bytes allocated
==xxxxx==
==xxxxx==  All heap blocks were freed -- no leaks are possible
==xxxxx==
==xxxxx==  For counts of detected and suppressed errors, rerun with: -v
==xxxxx==  ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 4)
```

You need to turn on the *"-g"* option while compiling your program by *gcc* so as to get more detailed information if there's any memory usage problem with your program.
If there are some problems with the memory usage in your program, *Valgrind* will provide you the information about where to fix in your program.

2 Implementation Details

2.1 Function Prototypes

For this assignment, you need to define the following functions in **Advanced.h**:

```
/*** function declarations ***/

/* Rotate 90 degrees clockwise */
IMAGE *Rotate(IMAGE *image);

/* Resize */
IMAGE *Resize( unsigned int percentage, IMAGE *image);

/* Mandelbrot */
IMAGE *Mandelbrot(unsigned int W, unsigned int H, unsigned int max_iteration);

/* Overaly a small image onto the original big image */
void Overlay(const char *f2name,IMAGE *image,
 unsigned int x_offset, unsigned int y_offset);

/* Test all functions */
void AutoTest(const char *fname);
```

The following functions in **Image.h**:

```
/*** function declarations ***/

/*Get the color intensity of the Red channel of pixel (x, y) in image */
unsigned char GetRPixel(IMAGE *image, unsigned int x,  unsigned int y);

/*Get the color intensity of the Green channel of pixel (x, y) in image */
unsigned char GetGPixel(IMAGE *image, unsigned int x,  unsigned int y);

/*Get the color intensity of the Blue channel of pixel (x, y) in image */
unsigned char GetBPixel(IMAGE *image, unsigned int x,  unsigned int y);

/*Set the color intensity of the Red channel of pixel (x, y) in image with value r*/
void SetRPixel(IMAGE *image, unsigned int x,  unsigned int y, unsigned char r);

/*Set the color intensity of the Green channel of pixel (x, y) in image with value g */
void SetGPixel(IMAGE *image, unsigned int x,  unsigned int y, unsigned char g);

/*Set the color intensity of the Blue channel of pixel (x, y) in image with value b */
void SetBPixel(IMAGE *image, unsigned int x,  unsigned int y, unsigned char b);

/*allocate the memory spaces for the image*/
IMAGE *CreateImage(unsigned int Width, unsigned int Height);

/*release the memory spaces for the image*/
void DeleteImage(IMAGE *image);
```

You may want to define other functions as needed.

2.2 Pass in the pointer of the struct IMAGE

In the main function, define the struct variable *image* of type IMAGE. It will be used as the aggregation of the image information: *Width*, *Height*, pointers to the memory spaces for all the color intensity values of the *R*, *G*, *B* channels. When any of the DIP operations is called in the main function, the address of this *image* variable is passed into the DIP functions so that the content of this variable can be accessed and modified in the DIP functions.

In your DIP function implementation, there are two ways to save the target image information. Both options work and you should decide which option is better based on the specific DIP manipulation function at hand.

Option 1: using local variables You can define local variables of type IMAGE to save the target image information. For example:

```
void DIP_function_name(IMAGE *image)
{
    IMAGE *image_tmp;

    image_tmp = CreateImage(image->Width, image->Height);

    ...

    DeleteImage(image_tmp);
    image_tmp = NULL;
}
```

Make sure you will create and delete the image space properly.
Then, at the end of each DIP function implementation, you should copy the data in *image_tmp* over to *image*.

Option 2: in place manipulation Sometimes you do not have to create new local array variables to save the target image information. Instead, you can just manipulate on *image.R*, *image.G*, *image.B* directly. For example, in the implementation of Negative() function, you can assign the result of 255 minus each pixel value directly back to this pixel entry.

NOTE: Please call SetRPixel (SetGPixel, SetBPixel) function for pixel color value setting, and GetRPixel (GetGPixel, GetBPixel) function for pixel color value achieving.

3 Budgeting your time

You have two weeks to complete this assignment, but we encourage you to get started early as there are more work than assignment2. We suggest you budget your time as follows:

- Week 1:
    1. Design the **Image.c** (**Image.h** as the header file) module.
    2. Change the signature and definitions of the existing functions.
    3. Modify the *AutoTest()* function.
    4. Adjust the **Makefile** with the targets for the new module, and compile programs by linking against the **libfileio.a** library.
    5. Implement one DIP function if possible.
- Week 2:
    1. Implement all the advanced DIP functions.
    2. Complete the *AutoTest()* function.
    3. Use *Valgrind* to check memory usages. Fix the code if *Valgrind* complains about any errors.
    4. Script the result of your programs and submit your work.

4 Script File

To demonstrate that your program works correctly, perform the following steps and submit the log as your script file:

1. Start the script by typing the command: *script*.
2. Compile and run *PhotoLab* by using your **Makefile**.
3. Choose 'Test all functions' (The file names must be 'bw', ..., 'rotate', 'resizeb', 'resizes', 'mandelbrot', 'overlay' for the corresponding function).
4. Exit the program.
5. Compile *Test* by using your **Makefile**.
6. Run *Test* under the monitor of *Valgrind*.
7. Clean all the object files, generated .ppm files and executable programs by using your **Makefile**.
8. Stop the script by typing the command: *exit*.
9. Rename the script file to *PhotoLab.script*.

NOTE: make sure use exactly the same names as shown in the above steps when saving modified images! The script file is important, and will be checked in grading; you must follow the above steps to create the script file. ***Please don't open any text editor while scripting !!!***

5 Submission

Use the standard submission procedure to submit the following files as the whole package of your program:

- *PhotoLab.c*
- *PhotoLab.script*
- *Image.c*
- *Image.h*
- *Constants.h*
- *DIPs.c*
- *DIPs.h*
- *Advanced.c*
- *Advanced.h*
- *Test.c*
- *Makefile*

Please leave the images generated by your program in your *public_html* directory. Don't delete them as we may consider them when grading! You don't have to submit any images.