EECS 22: Assignment 4

Prepared by: Mark Boyher, Yang Xiang, Mark Boyher, Prof. Halima Bouzidi,
Yutong Wang, Prof. Rainer Dömer

02/12/2025

Due on Wednesday 03/01/2024 at noon, 12:00pm. Note: this is a two-week assignment.

Contents

1 Digital Image Processing (DIP)

In this assignment, you will practice the use of dynamic memory allocation. Based on the program *PhotoLab* for Assignment 3, you will redesign several DIP operations to support varying image sizes. Then, you will add more DIP operations whose resulting images will differ in size compared to the original one. Thus, you can use your *PhotoLab* program to perform the DIP operations on any of your own pictures.

1.1 Initial Setup

We recommend to start from our supplied files.
Make a directory *hw4* and copy the following files from directory ∼*eecs22/public* to it.

```
mkdir hw4
cd hw4
cp ~eecs22/public/Image.h .
cp ~eecs22/public/FileIO.h .
cp ~eecs22/public/FileIO.c .
cp ~eecs22/public/Test.h .
cp ~eecs22/public/Test.c .
cp ~eecs22/public/Constants.h .
cp ~eecs22/public/DIPs.h .
cp ~eecs22/public/Advanced.h . (EDIT)
cp ~eecs22/public/EngPlaza.ppm .
cp ~eecs22/public/watermark_template.ppm .
cp ~eecs22/public/border.ppm .
```

- **Image.h** is the header file for the new structure definition and pixel mapping functions declarations, used in Section 1.2.2.
- **Test.h**, **Test.c** contains `AutoTest(void)`, should be used by `main()`.

1.2 Add Support for Different Image Sizes

To support varying image sizes, we cannot define and pass three fixed size arrays to the DIP operation functions as in previous assignments. Instead, we need to use dynamic memory allocation to claim three blocks of memory whose size will be decided at run-time, then pass the pointers pointing to an image structure to the DIP functions, since the size of the input image cannot be determined at compile time.

1.2.1 Use Pointers in One Dimensional Memory Space instead of Arrays with Two Dimensions

In this assignment, we use three pointers to type *unsigned char* for the color intensity values for each pixel instead of three fixed-size arrays. However, pointers only point to a memory space in one dimension. Therefore, we need to map 2-tuple coordinates of pixels to a single value index corresponding to the pixel color information in memory.

For example, we have an image of size 10×5, and three pixels (0, 0), (9, 4), and (6, 4). 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 = $6+4*10$, as shown in Figure 1.

In general, the index value for the pixel (x, y) in an image of size **Width**×**Height** in the one dimensional storage space will be x **+** y *** Width**.

1.2.2 The Image.c Module

Please implement module **Image.c** (see provided **Image.h**) to handle basic operations on the image. **Image.h** is the header file for the new structure definition and pixel mapping function declarations.

- The **Image** struct: used to aggregate all the information of an image, defined in **Image.h**.

```
typedef struct {
  unsigned int W;  /* image width */
  unsigned int H; /* image height */
  unsigned char *R;    /* pointer to the memory storing all the R intensity */
```

						x						
		0	1	2	3	4	5	6	7	8	9	
	0	0	1	2	3	4	5	6	7	8	9	
	1	10	11	12	13	14	15	16	17	18	19	Height = 5
y	2	20	21	22	23	24	25	26	27	28	29	
	3	30	31	32	33	34	35	36	37	38	39	
	4	40	41	42	43	44	45	46	47	48	49	
						Width = 10						

Figure 1: Unidimensional storage of pixels (0, 0), (9, 4), and (6, 4) in a 10×5 size image.

```
                         /* 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 and set the value of the color intensities of 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 GetPixelR(const Image *image, unsigned int x, unsigned int y);

unsigned char GetPixelG(const Image *image, unsigned int x, unsigned int y);

unsigned char GetPixelB(const Image *image, unsigned int x, unsigned int y);

/* Set the color intensity of the Red channel of pixel (x, y) in image to r */
void SetPixelR(Image *image, unsigned int x, unsigned int y, unsigned char r);

void SetPixelG(Image *image, unsigned int x, unsigned int y, unsigned char g);

void SetPixelB(Image *image, unsigned int x, unsigned int y, unsigned char b);

/* Return the image's width in pixels */
unsigned int ImageWidth(Image *image);

/* Return the image's height in pixels */
unsigned int ImageHeight(Image *image);
```

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 these functions. Please call these functions in your DIP functions in order to set / get the intensity values of the pixels.

NOTE: By using pointers in 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 Assignment 3, we got a pixel's color value by indexing the element from the two-dimensional array: *tmpR = R[x][y]*;
  - Now, we need to get a pixel's color value by calling the get function: *tmpR = GetPixelR(image, x, y)*;
  - In Assignment 3, we set a pixel's color value by indexing the element from the two-dimensional array: *R[x][y] = r*;

   - Now, we need to set a pixel's color value by calling the set function:
     SetPixelR(image, x, y, r);

  By using the set/get functions, we can keep the two-dimensional coordinate system as in Assignment 2 and Assignment 3.

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

- Add **assertions** in these functions to make sure the input image pointer is valid, and the set of pointers to the memory spaces for the color intensity values are valid too. Last but not least, add assertions to ensure that the coordinates are within the valid ranges for the image.

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

1.2.3 Load and Save Image Files

Refer to **FileIO.h** for the defined functions for file I/Os.

- `Image *LoadImage(const char *fname)`
  This function reads the file *fname.ppm* and returns the image pointer if loaded successfully, otherwise returns NULL. The color intensities for channel red, green, and blue are stored in the memory spaces pointed to by member pointers *R*, *G* and *B* of the returned Image pointer, respectively. The memory space of the image is created in this function by a function call to `CreateImage()`, see below.

- `int SaveImage(const char *fname, const Image *image)`
  This function saves the color intensities of the red, green, and blue channel stored in the memory spaces pointed to by member pointers *R*, *G* and *B* of *image* into the file *fname.ppm*. This function returns an error code if something goes wrong. Handle it by letting the user know that the image was not saved.

Please implement the two functions to handle the memory allocation and deallocation in **Image.c**, declared in **Image.h**.

```
/* Allocate the memory space for the image structure         */
/* and the memory spaces for the color intensity values.     */
/* Return the pointer to the image, or NULL in case of error */
Image *CreateImage(unsigned int Width, unsigned int Height);

/* Release the memory spaces for the pixel color intensity values */
/* Deallocate all the memory spaces for the image                 */

void DeleteImage(Image *image);
```

IMPORTANT: The `LoadImage()` function needs the `CreateImage()` function inside to allocate the memory space. Therefore, you should implement the `CreateImage()` and `DeleteImage()` functions correctly before you use the `LoadImage()` and `SaveImage()` functions. `malloc()` and `free()` must be called within `CreateImage()` and `DeleteImage()` only.

1.2.4 Accessing Image Width and Height values

For this task you will need to define the `ImageWidth` and `ImageHeight` functions inside the **Image.c** file. Each of these functions will return the corresponding *width* or *height* values of the input image.

Modify HW2 and other DIP Function Implementations Most of our functions need to be refined by taking the Image structure as a parameter which contains all the information about the image, i.e. your DIP function prototypes should look like below:

```
/* DIPs.h */
Image *BlackNWhite(Image *image);
Image *Negative(Image *image);
Image *ColorFilter(Image *image, int target_r, int target_g, int target_b,
           int threshold, int replace_r, int replace_g, int replace_b);
Image *Edge(Image *image);
Image *VFlip(Image *image);
Image *HMirror(Image *image);
Image *AddBorder(Image *image, char *color, int border_width);
Image *Shuffle(Image *image);

Image *Pixelate(Image *image, int block_size);
Image *Shift(Image *image, int shiftX, int shiftY);

/* Advanced.h */
Image *Crop(Image *image, int x, int y, int W, int H);
Image *Resize(Image *image, double ScaleFactor);
Image *Watermark(Image *image, const Image *watermark_image);
Image *RotateBy90(Image *image, int rotateDirection);
```

IMPORTANT: Note the changes in the return types and the function arguments!
NOTE: Add **assertions** in **ALL** these DIP functions to make sure the input image pointer is valid.

1.2.5 Test.c Module

Include **Test.h** in your **PhotoLab.c**, otherwise `AutoTest(void)` cannot be called in `main()`, you should use the provided `AutoTest(void)`.

1.3 Advanced DIP operations

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

The user should be able to select DIP operations from a menu as the one shown below:

```
Please make your choice:
--------------------------------
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:  Color filter an image
6:  Sketch the edge of an image
7:  Shuffle an image
8:  Flip an image vertically
9:  Mirror an image horizontally
10:  Add border to an image
11:  Pixelate an image
12:  Shift an image
13:  Crop an image
14:  Resize an image
15:  Add Watermark to an image
16:  Rotate an image by 90 degree
17:  Test all functions
18:  Exit
```

NOTE: Your program should:

- Print "`No image to process!`" when menu item 2 - 17 is chosen but the input image pointer is `NULL` (such as loading image failed or no image is loaded).
- Print "`Invalid selection!`" when none of menu item 1 - 19 is chosen.
- Print "`You exit the program.`" and exit properly whenever the user inputs 19.
- Print "`AutoTest failed, error code RC.`" (replace `RC` with return code from `AutoTest`) if `AutoTest` returns a non-zero code, otherwise print "`AutoTest finished successfully.`".

There should be no memory leaks when:

- The user chooses to load an image multiple times then exit menu without doing any DIPs.
- The user chooses to load an image and choose some DIPs then exit menu without saving it to file.

1.3.1 Crop

(a) Original image

(b) Cropped image(x=80 y=160 w=250 h=100

Figure 2: The original image and the cropped Engineering image.

This function crops the image based on a set of user inputs. The user will indicate a starting pixel in the image by entering an x and y offset. Then, the user specifies the size of cropping by entering how many pixels the user wants to crop in the x and y directions. If the crop amount exceeds the image width or height (or both), the returned image will only crop up to the maximum length of the original image.

NOTE: This means that the picture should only allocate the minimum amount of memory needed to store the image.

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

```
Image *Crop(Image *image, int x, int y, int W, int H);
```

Figure 2 shows an example of this operation (the size of the cropped image is 250×100). Once the user chooses this option, your program's output should look like this:

```
Please make your choice: 15
Please enter the X offset value: 80
Please enter the Y offset value: 160
Please input the crop width: 250
Please input the crop height: 100
"Crop" operation is done!
/* ... print menu again and wait for the user's next input */
```

The image is saved with the name 'crop' after this step.

1.3.2 Resize

(a) Original image (512x288)

(b) resized to a bigger image (800x400)

(c) resized to a smaller image (300x200)

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

You need to implement the following function for this DIP.

```
Image *Resize(Image *image, int newWidth, int newHeight);
```

This function resizes the image with new width and height.

```
Please make your choice: 16
Please input the new image width: 1000
Please input the new image height: 1000
"Resizing the image" operation is done!
/* ... print menu again and wait for the user's next input */
```

In this function, we can re-use the rotation matrix method introduced in hw3:
Rotation matrix(`https://en.wikipedia.org/wiki/Rotation_matrix`) is a transformation matrix that is for executing a rotation in Euclidean space.

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

If we set the radian to 0, we get a new matrix to only resize. x' and y' (coordinates in original image) then become:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1/ScaleX & 0 \\ 0 & 1/ScaleY \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

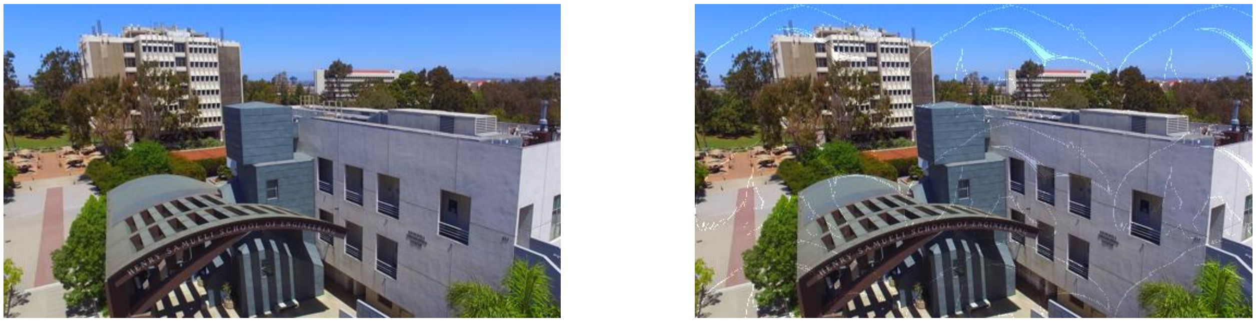
ScaleX, ScaleY in the matrix is calculated from $newWidth/ImageWidth$, $newHeight/ImageHeight$. ImageWidth and ImageHeight should not be predefined constants but from the Image data structure. The major difference between Resize and Rotate is that Resize will produce a new image with different width and height, while Rotate will produce a new image with the same width and height but zoomed in or out.
HINT: For enlarging and shrinking the image, it is easier to iterate over the target image (not over the original image).

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

This function needs to be able to both shrink and enlarge images. The two example images saved after this operation are:

1. 'bigresize': 800x400.
2. 'smallresize': 300x200.

1.3.3 Watermark



(a) Image without watermark

(b) Image with watermark

Figure 4: The image and its watermarked counterpart.

Fig. 4 shows an example of the original and the watermarked image which contains an anteater. You can implement the functionality by loading the image `watermark_template.ppm` first. Next, if the colors R, G, and, B have the value 0 at position x,y in the watermark template image, you have to multiply the R, G, and, B pixel values with the factor 1.45 in the original image at position x,y. At this location take into account an overflow situation and handle it with saturated arithmetic. That is, any value above 255 should stay at the maximum intensity value of 255.

The watermark should also wrap around the original image in case the images are of different sizes. If the watermark template is larger than the image, only the part of the template that fits the image should be applied. Conversely, as presented in the example, if the watermark template is smaller than the image, then it should be tiled to cover the whole image.

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

```
Image *Watermark(Image *image, const Image *watermark_image);
```

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

```
Please make your choice: 16
"Watermark" operation is done!
```

1.3.4 Add borders to an image

This operation will add borders to the current image. The border color and width (in pixels) of the borders are parameters given by the user. Within your implementation, your horizontal and vertical border have the same widths. Figure 5 shows an example of adding borders to an image.

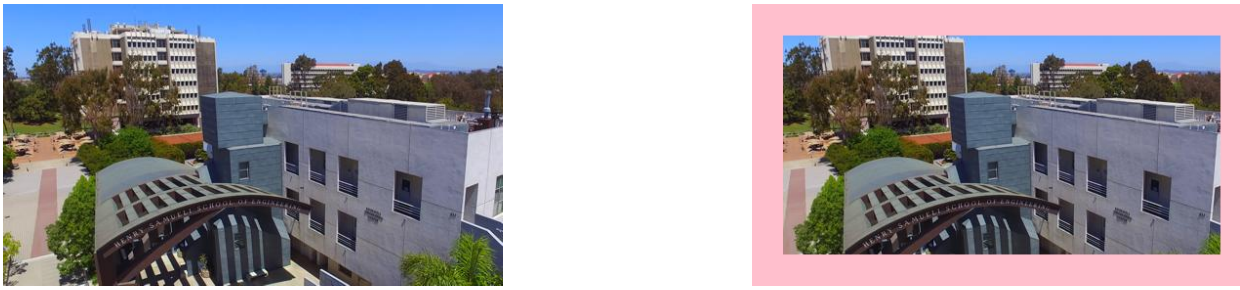
Function Prototype: You need to define and implement the following function to do this DIP.

```
Image *AddBorder(Image *image, char color[SLEN], int border_width);
```

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

```
Please make your choice: 10
Enter border width: 32
Available border colors : black, white, red, green, blue, yellow, cyan, pink, orange
Select border color from the options: pink
"Border" operation is done!
```

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



(a) Original image

(b) Image with borders, border color = pink, width = 32 pixels

Figure 5: An image and its counterpart when borders are added.

1.3.5 Shift an image

(a) Original image

(b) Shifted image, shiftX = 160, shiftY = 40

Figure 6: An image and its counterpart when shifted.

This operation will shift the image in the positive X and Y direction, with positive X being to the right and positive Y being down on the screen.

Please be aware that any part of the image that scrolls off the screen, rolls over onto the opposite end of the screen. The amount of this movement corresponds to the user specified X and Y. Figure 6 shows an example of shifting an image.

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

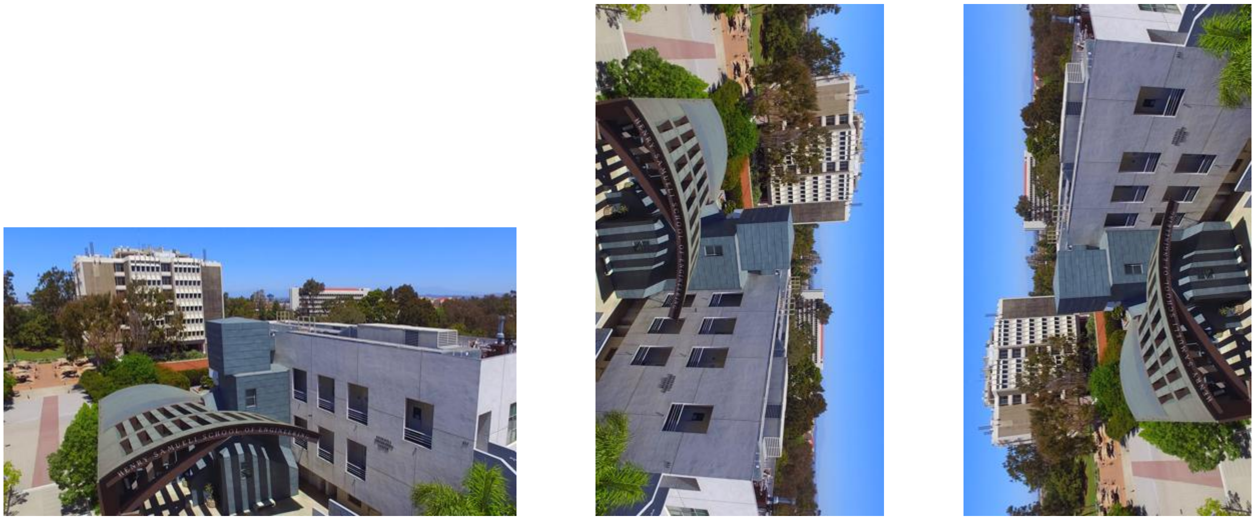
```
Image *Shift(Image *image, shiftX, shiftY);
```

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

```
Please make your choice: 12
"Shift" operation is done!
```

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

1.4 Bonus: RotateBy90



(a) Original image (b) Clockwise Rotated image (c) CounterClockwise Rotated image

Figure 7: An image rotated by 90 degree clockwise and counterclockwise.

You can earn 10 extra points through implementing the Rotateby90 feature.

This DIP will rotate an image by 90 degree. Different from the *rotate* DIP, *RotateBy90* also changes the width and height of the rotated image. *RotateBy90* must rotate the image clockwise or counterclockwise based on the user input, 0: clockwise, 1: counterclockwise.

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

```
Please make your choice: 17
Please input the direction of rotation (0:clockwise, 1:counterclockwise): 0
"RotateBy90" operation is done!
```

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

1.5 Test All Functions

Use the provided `AutoTest(void)` function to test all the DIP operations.

1.6 Extend the Makefile

- Add *Image.h*, *Image.c* to your **Makefile** and adjust it properly.
- Add *Test.h*, *Test.c* to your **Makefile** and adjust it properly.
- Generate 2 executable programs
  1. *PhotoLab* with the user interactive menu and the DEBUG mode off.
  2. *PhotoLabTest*, an executable that just calls *AutoTest(void)* function (with the DEBUG mode on).

Define two targets to generate these 2 programs respectively in addition to *all* and *clean*. You may define other targets as needed.

1.7 Use "Valgrind" Tool to Find Memory Leaks and Invalid Memory Accesses

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 over the end of an array, or forget to free a pointer, *Valgrind* will detect it. You may refer to `http://valgrind.org/` for more details about the *Valgrind* tool.

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

```
valgrind --leak-check=full program_name
```

If there is no problem with the memory usage in your program, you will see information similar to the following upon completion of your program:

```
==xxxxx==
==xxxxx== HEAP SUMMARY:
==xxxxx==     in use at exit: 0 bytes in 0 blocks
==xxxxx==   total heap usage: 129 allocs, 129 frees, 20,476,437 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: 6 from 6)
```

Compile your program with the *"-g"* option in *gcc* to enable detection of memory usage problems in your program.

If there are problems with your program's memory usage, *Valgrind* will provide you with information about the problem and where to fix it.

For your final submission, your program should be free of warnings and free of any errors reported by *Valgrind*.

2 Implementation Details

2.1 Function Prototypes

Please adjust **all** functions in **Advanced.c** and **Image.c**, adjust `main()`:
You can define other help functions as needed. **Do not** modify the provided function declarations in *Image.h* and `AutoTest(void)` in *Test.c*, as well as *Test.h*.

2.2 Pass in the Pointer of the struct Image

In the main function, define the struct variable *image* of type Image. It will contain the following 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 are called in the main function, the address of this *image* variable is passed into the DIP functions. Thus, the DIP functions can access and modify the contents of this variable.

In your DIP function implementation, there are two ways to save the target image information. Choose the better and easier one.

Option 1: Using Local Variables Define local variables of type Image to save the target image information. For example:

```
Image *DIP_function_name(Image *inputImage)
{
  Image *outputImage = NULL;
  outputImage = CreateImage(...);
  ...
  DeleteImage(inputImage);
  inputImage = NULL;
  return outputImage;
}
```

Make sure that you properly create and delete the image space.

Then, at the end of each DIP function implementation, you can copy the data in *outputImage* over to *inputImage*, or delete the incoming image and return the new one.

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 the pixels directly. For example, in the implementation of the Negative() function, you can assign the result pixel value directly back to the pixel entry.

NOTE: Please always call SetPixelR (SetPixelG, SetPixelB) function to set the pixel color value and GetPixelR (GetPixelG, GetPixelB) function to read the pixel color value. Also, please always use the ImageWidth and ImageHeight functions to access the values for the image's width and height respectively.

3 Budgeting Your Time

This assignment's workload is heavier than the previous one. Suggested steps:

- Week 1:
  1. Preview C pointer and dynamic memory allocation.
  2. Design pseudo-code for the new DIPs Crop, Resize, Watermark, and RotateBy90.
- Week 2:
  1. Change the implementations of HW2 and HW3's DIP functions to fit this assignment.
  2. Implement the rest of advanced DIP functions.
  3. Adjust the **Makefile** with the targets for the new module.
  4. Use *Valgrind* to check memory usages. Fix any errors and warnings if any complained by *Valgrind*.
  5. 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. Type `script` to start your script.
2. Type `make` to generate *PhotoLabTest* and *PhotoLab*.
3. Type `./PhotoLab`, input 19 to run `AutoTest` then input 20 to exit.
4. Type `valgrind --leak-check=full PhotoLabTest` to run *PhotoLabTest* under *Valgrind*.

5. Type `make clean` to clean all the object files, generated .ppm files and executable programs.
6. Type `exit` to stop the script.
7. Type `mv typescript PhotoLab.script` to rename the script file as required.

NOTE: make sure you 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

Go to the parent directory of your *hw4* directory, turn in your homework by running:

`~eecs22/bin/turnin.sh`

Your *hw4* directory should contain *PhotoLab.script*, *PhotoLab.txt* (used to **briefly** describe your implementations), *PhotoLab.c*, *Image.c*, *Image.h*, *Constants.h*, *DIPs.c*, *DIPs.h*, *FileIO.c*, *FileIO.h*, *Advanced.c*, *Advanced.h*, *Makefile*, *Test.h*, *Test.c*.

6 Grading (100 pts + 10 pts Bonus)

Scores breakdown

- Makefile (10 pts)
- CreateImage, DeleteImage (5 pts each, 10 pts, points will be deducted if no proper error handling in CreateImage or failing to set R/G/B pointers to NULL in DeleteImage)
- GetPixelR, GetPixelG, GetPixelB, SetPixelR, SetPixelG, SetPixelB (1 pts each, 6 pts)
- HW2 functions reimplementation (2 pts each, 16 pts) (BlackNWhite, Negative, ColorFIlter, Edge, VFlip, HMirror, Shuffle, and Pixelate)
- Crop, Watermark Shift, Add Border (7 pts each, 28 pts)
- Resize (5 pts for smallresize, 5 pts for bigresize) (10 pts)
- Menu (5 pts)
- Valgrind 0 errors for *PhotoLabTest* (15 pts)
- RotateBy90 (Bonus: 10 pts)

NOTE: Partial credit based on quick code review will be given if:

- `make` failed, or target *PhotoLabTest* or *PhotoLab* cannot be generated, or any errors or warnings arise
- the output image (ppm or jpg) is incorrect
- the menu is incorrect (such as cannot exit properly, or `AutoTest` function failed, or memory leaks under the test cases described in Section 1.3)
- Valgrind generates errors

In these cases, grading will rely solely on a brief source code review with a rough estimate of how much usable code exists.