EECS 22: Assignment 3

Prepared by: Mina Chookhachizadeh Moghadam, Prof. Dang, Prof. Rainer Domer

October 25, 2017

Due on Wednesday 11/08/2017 6:00pm. Note: this is a two-week assignment.

Contents

1 Digital Image Processing

In this assignment you will learn how to break a program into multiple modules, and compile them into one program. Based on the program *PhotoLab* for Assignment 2, you will be asked to develop some advanced digital image processing (DIP) operations, partition them in separate modules, manipulate images using bit operations, and develop an appropriate **Makefile** to compile your program with DEBUG mode on or off.

1.1 Introduction

In Assignment 2, you were asked to develop an image manipulation program *PhotoLab* by using DIP techniques. 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*. This assignment will be based on Assignment 2.

1.2 Initial Setup

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

```
mkdir hw3
cd hw3
cp  ~eecs22/public/PhotoLab_v2.c .
cp  ~eecs22/public/HSSOE.ppm .
```

We will extend the *PhotoLab* program based on Assignment 2. You must use the provided tmplate file PhotoLab_v2.c file.
Once a DIP operation is done, you can save the modified image as *name*, and it will be automatically converted to a JPEG image and sent to the folder *public_html* in your home directory. Then you are able to see the image with the web browsers Firefox, Safari, Internet Explorer, Edge at: *http://bondi.eecs.uci.edu/∼youruserid*, if required names are used. Chrome is not recommended through caching issues and wrongly displayed index files. If you save images by other names, use the link *http://bondi.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.

1.3 Decompose the program into multiple modules

Please decompose the **PhotoLab_v2.c** file into multiple modules and header files:

- **PhotoLab.c**: the main module contains the *main()* function, and the menu function *PrintMenu()* as well as *AutoTest()*.
- **FileIO.c**: the module for the function definitions of *LoadImage()* and *SaveImage()*.
- **FileIO.h**: the header file for **FileIO.c**, with the function declarations of *LoadImage()* and *SaveImage()*.
- **Constants.h**: the header file in which the constants to be used are defined.
- **DIPs.c**: the module for the DIP function definitions in Assignment 2, i.e. *BlackNWhite*, *Negative*, *ColorFilter*, *Edge*, *Shuffle*, *VFlip*, *VMirror*, *AddBorder*.
- **DIPs.h**: the header file for **DIPs.c**, with the DIP function declarations.
- **Advanced.c**: the module for the function definition of new filters in Assignment 3, *Noise()*, *Sharpen()*, *Posterize()*, and *MotionBlur()*.
- **Advanced.h**: the header file for **Advanced.c**, with the function declarations of *Noise()*, *Sharpen()*, *Posterize()*, and *MotionBlur()*.

HINT: Please refer to the slides of *lectures covering Compiler components, translation units, Make and Makefile* for an example of decomposing programs into different modules.

1.4 Compile the program with multiple modules using static shared library

The *PhotoLab* program is now modularized into different modules: **PhotoLab**, **FileIO**, **DIPs** and **Advanced**. In this assignment we are using shared libraries to group the compiled object code files in to static libraries. Often C functions and methods which can be shared by more than one application are broken out of the application's source code, compiled and bundled into a library.

As shown in lectures, in order to generate the libraries first compile the source code into object files. Use *"-c"* option for **gcc** to generate the object files for each module, e.g.

```
% gcc -c FileIO.c -o FileIO.o -ansi -std=c99 -Wall
```

```
% gcc -c DIPs.c -o DIPs.o -ansi -std=c99 -Wall
...
```

As shown in the Make and Makefile lecture, libraries are typically names with the prefix ”lib”. Here we want to create a librariy named *libfilter*:

```
% ar rc libfilter.a DIPs.o Advanced.o
% ranlib libfilter.a
```

Linking with the library:

```
% gcc PhotoLab.o -lfilter -L. -o PhotoLab
```

Execute the program:

```
% ./PhotoLab
program executes
% _
```

1.5 Using ’make’ and ’Makefile’

On the other hand, we can put the commands above into a **Makefile** and use the *make* utility to automatically build the executable program from source code. Please create your own **Makefile** with at least the following targets:

- *all*: the target to generate the executable programs.
- *clean*: the target to clean all the intermediate files, e.g. object files, autogenerated images, and the executable program(s). Be careful to only delete intermediates files, not any of your true source files.
- *PhotoLabTest*: the target to create and run PhotoLabTest.
- *PhotoLab*: the target to generate the executable program *PhotoLab*.

To use your **Makefile**, please use this command:

```
% make all
```

The executable program *PhotoLab* shall then be automatically generated.

Requirement: There must be a rule fore each object file depending on the corresponding .c file and any other needed dependency. Dependencies which are not needed will reduce the points.

HINT: Please refer to the slides of *Lecture 11* for an example on how to create a **Makefile**.

1.6 Advanced DIP operations

In this assignment, please add one more module named **Advanced**, consisting of **Advanced.c** and **Advanced.h** and implement the advanced DIP operations described below.

Please reuse the menu you designed for Assignment 2 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:  Color filter an image
 6:  Sketch the edge of an image
 7:  Shuffle an image
```

```
 8:  Flip an image vertically
 9:  Mirror an image vertically
 10: Add border to the image
 11: Add noise to an image
 12: Sharpen an image
 13: Posterize an image
 14: Motion Blur
 15: Test all functions
 16: Exit
please make your choice:
```

1.6.1 Add noise to an image

In this operation, you add white noise to an image. You need to define and implement a function to do the job. If the percentage of noise is n, then the number of noise pixels added to the image is given by $n * WIDTH * HEIGHT/100$, where $WIDTH$ and $HEIGHT$ are the image size. The noisy pixels are distributed randomly and they are white. To generate the initial random number, you have to use a random number generator which is provided by the *C* standard function *rand*(). This function generates a random number of type `int` in the range of 0 to $RAND\_MAX$. This function is declared in the header file `stdlib.h`.
In practice, no computer function can produce truly random data; they only produce pseudo-random numbers. These are computed by a formula and the number sequences they produce are repeatable. A seed value is usually used by the random number generator to generate the first number. Therefore, if you use the same seed value all the time, the same sequence of "random" numbers will be generated (i.e. your program will always produce the same "random" number in every program run). To avoid this, we can use the current time of the day to set the random seed, as this will always be changing with every program run. With this trick, your program will produce different numbers every time you run it.
To set the seed value, you have to use the function **srand()**, which is also declared in the header file `stdlib.h`. For the current time of the day, you can use the function time(), which is defined in the header file `time.h` (`stdlib.h` and `time.h` are header files just like the `stdio.h` file that we have been using so far).
In summary, use the following code fragments to generate the random number for the noise:

1. Include the `stdlib.h` and `time.h` header files at the beginning of your program:

   ```
   #include <stdlib.h>
   #include <time.h>
   ```

2. Include the following lines at the beginning of your main function:

   ```
   /* initialize the random number generator with the current time */
   srand(time(NULL));
   ```

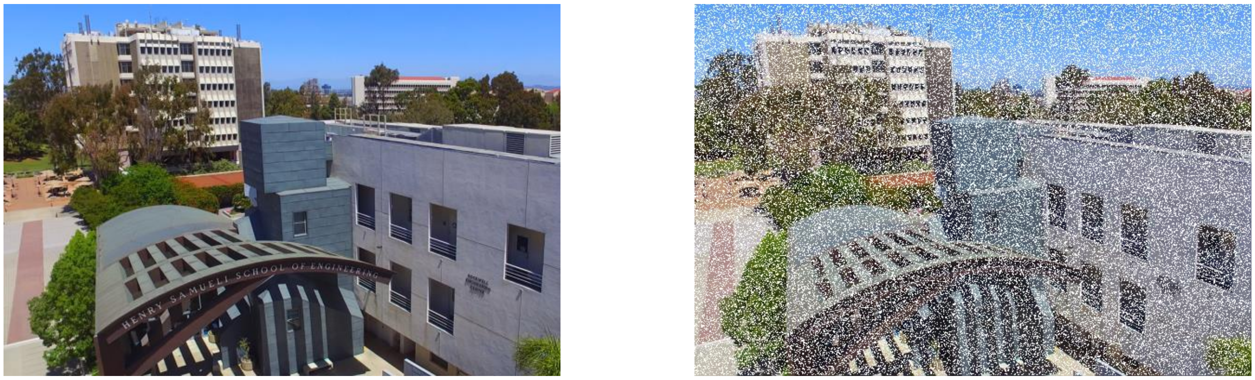
3. To simulate locating a random position, use the following statement:

   ```
   /* generate a random pixel */
   x = rand() % WIDTH;  /* You need to define the variable x. */
   y = rand() % HEIGHT; /* You need to define the variable x. */
   ```

   The integer variables x and y then will have a random values in the range from 0 to $WIDTH$ and 0 to $HEIGHT$ accordingly.

It can happen that by coincidence the same pixel will be selected multiple times. This behavior should not be prevented.
Function Prototype: You need to define and implement the following function to do this DIP.

```
/* Add noise to image */
void Noise(int n,
    unsigned char R[WIDTH][HEIGHT],
```



(a) Original image (b) Noisy image with n=30

Figure 1: An image and its noise corrupted counterpart.

```
    unsigned char G[WIDTH][HEIGHT],
    unsigned char B[WIDTH][HEIGHT]);
```

Here, *n* specifies the percentage of noise in the image.
Figure 5 shows an example of this operation where *n* is 30. Once the user chooses this option, the whole menu shuld be printed out and after that your program's output look slike this:

```
Please make enter your choice: 11
Please input noise percentage: 30
"Noise" operation is done!
```

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

1.6.2 Sharpen an image

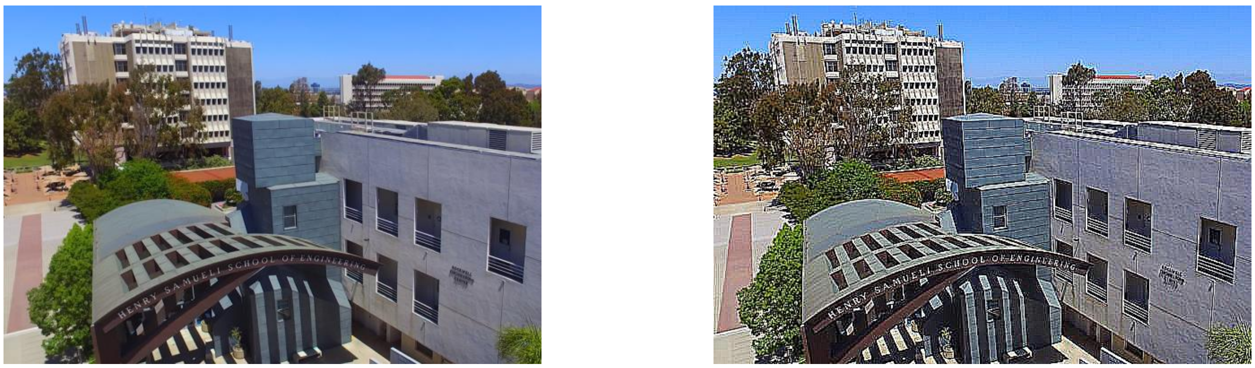
The sharpening works this way: the intensity value at each pixel is mapped to a new value, which is the sum of itself and its 8 neighbours with different parameters. To sharpen the image is very similar to finding edges. Adding the original image to its edge will result in a new image where the edges are enhanced, and make it look sharper. The following shows an example of the filter and the applied pixel:

```
    Filter :             Original Pixels
    X  X  X  X  X         X  X  X  X  X
    X -1 -1 -1  X         X  A  B  C  X
    X -1  9 -1  X         X  D  E  F  X
    X -1 -1 -1  X         X  G  H  I  X
    X  X  X  X  X         X  X  X  X  X
```

To sharpen an edge of the image, the intensity of the center pixel (E) with the value is changed to $(-A-B-C-D+9*E-F-G-H-I)$. Repeat this for every pixel, and for every color channel (red, green, and blue) of the image. You need to define and implement a function to do this DIP. Note that you have to set the boundary for the newly generated pixel value, i.e., the value should be within the range of [0,255]
Note that special care has to be taken for pixels located at the image boundaries. For ease of implementation, you may choose to ignore the pixels at the border of the image where no neighbor pixels exist. It means that they can remain the same as before.
You need to define and implement the following function to do this DIP.

```
/* Sharpen an image */
void Sharpen(unsigned char R[WIDTH][HEIGHT],
             unsigned char G[WIDTH][HEIGHT],
             unsigned char B[WIDTH][HEIGHT]);
```

The sharpen image should look like the figure shown in Figure 2(b):



(a) Original Image

(b) Sharpened Image

Figure 2: An image and its sharpened counterpart.

```
Please enter your choice:12
"Sharpen" operation is done!
```

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

1.6.3 Bit Manipulations: Posterize an image

Posterization of an image entails conversion of a continuous gradation of tone to several regions of fewer tones, with abrupt changes from one tone to another. This was originally done with photographic processes to create posters. It can now be done photographically or with digital image processing, and may be deliberate or may be an unintended artifact of color quantization. (`http://en.wikipedia.org/wiki/Posterization`).

We are going to use bit manipulations to posterize the image. As before, a pixel in the image is represented by a 3-tuple (r, g, b) where r, g, and b are the values for the intensities of the red, green, and blue channels respectively. The range of *r*, *g*, and *b* are from 0 to 255 inclusively. As such, we use *unsigned char* variables to store the values of these three values.

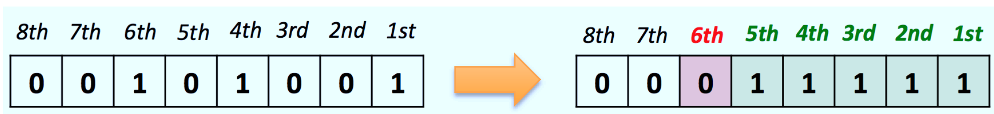
To posterize the image, we are going to change the least $n, n \in \{1, 2, 3, \ldots 8\}$ significant bits of color intensity values so as to change the tone of the pixels. Basically, we will change the *n*th least significant bit of the color intensity value to be 0, and the least $n-1$ bits to be all 1. For example, assume that the color tuple of the pixel at coordinate(0,0) is (41, 84, 163). Therefore,

```
R[0][0] = 41;
G[0][0] = 84;
B[0][0] = 163;
```

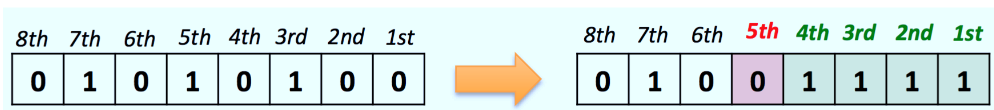
In binary representation, the color tuple will be:

[0][0] = 00101001_2;
[0][0] = 01010100_2;
[0][0] = 10100011_2;

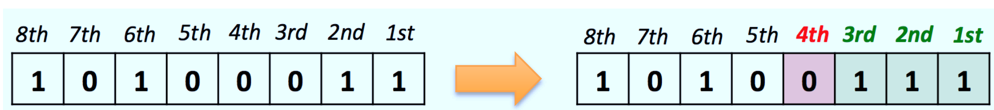
Fig. 3 shows the operation for posterize for different least significant bits of the intensities for the red, green, and blue channels. As illustrated in Fig. 3(a), in order to posterize the least 6 significant bits of the red intensity, we set the 6th bit to be 0, and the 1st to the 5th bits to be 1s. Similarly in Fig. 3(b), to posterize the least 5 significant bits of the green intensity, we set the 5th bit to be 0, and the 1st to the 4th bits to be 1s; and in Fig. 3(c), to posterize the least 4 significant bits of the blue intensity, we set the 4th bit to be 0, and the 1st to the 3th bits to be 1s.



(a) Posterize the least 6 significant bits of the red channel for pixel(0,0)



(b) Posterize the least 5 significant bits of the green channel for pixel(0,0)



(c) Posterize the least 4 significant bits of the blue channel for pixel(0,0)

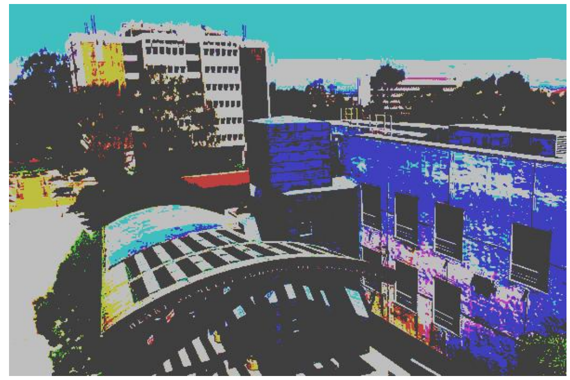
Figure 3: The example of posterizing the color channels.

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

```
/* Posterize the image */
void Posterize(unsigned char R[WIDTH][HEIGHT],
    unsigned char G[WIDTH][HEIGHT],
    unsigned char B[WIDTH][HEIGHT],
    unsigned int rbits,
    unsigned int gbits,
    unsigned int bbits);
```

Here, *rbits*, *gbits*, and *bbits* specify the number of least significant bits that need to be posterized. Since the size of *unsignedchar* variable is 8 bits, the valid range of *rbits*, *gbits*, and *bbits* will be 1 to 8.
HINT: You will need to use bitwise operators, e.g. '&', '<<', '>>', '|' for this operation.

(a) Image without posterization



(b) Image with posterization, where rbits = 7, gbits = 7, bbits = 7

Figure 4: The image and its posterized counterpart.

Fig. 4 shows an example of our posterized image. Once user chooses this option, your program's output should look like:

```
please make your choice: 13
Enter the number of posterization bits for R channel (1 to 8): 7
Enter the number of posterization bits for G channel (1 to 8): 7
Enter the number of posterization bits for B channel (1 to 8): 7
"Posterize" operation is done!
```

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

1.7 Test all functions

Finally, you are going to complete the *AutoTest()* function to test all the functions. In this function, you are going to call DIP and advanced functions one by one and save the results. The function is for the designer to quickly test the program, so you should supply all necessary parameters when testing.
Please note that AddBorder and MotionBlur should only be included in the test all functions if you actually implement them.
The function should look like:

```
void AutoTest(unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
 unsigned char B[WIDTH][HEIGHT])
{
  char fname[SLEN] = "HSSOE";
  char sname[SLEN];

  LoadImage(fname, R, G, B);
  Negative (R, G, B) ;
  SaveImage("negative", R, G, B) ;
  printf("Negative tested!\n\n");

  LoadImage(fname, R, G, B);
  ColorFilter(R, G, B, 190, 100, 150, 60, 0, 0, 255);
  SaveImage("colorfilter", R, G, B);
  printf("Color Filter tested!\n\n");
```

```
  LoadImage(fname, R, G, B);
  AddBorder(R, G, B, "black", 64) ;
  SaveImage("black", R, G, B) ;
  printf("Border tested!\n\n");

  ...

  LoadImage(fname, R, G, B);
  Noise(30, R, G, B) ;
  SaveImage("noise", R, G, B) ;
  printf("Noise tested!\n\n");

  LoadImage(fname, R, G, B);
  Sharpen(R, G, B) ;
  SaveImage("sharpen", R, G, B) ;
  printf("Sharpen tested!\n\n");

  LoadImage(fname, R, G, B);
  Posterize(R, G, B, 7, 7, 7) ;
  SaveImage("posterize", R, G, B) ;
  printf("Posterize tested!\n\n");

  LoadImage(fname, R, G, B);
  MotionBlur(R, G, B) ;
  SaveImage("blur", R, G, B) ;
  printf("MotionBlur tested!\n\n");
}
```

Please implement the *AutoTest()* function in **Photolab.c**. Since the *AutoTest()* function will call the functions in the **DIPs.c** and **Advanced.c** modules, please include the header files properly. Also, be sure to adjust your **Makefile** for proper dependencies.

1.7.1 Motion Blur: Bonus

Any kind of blur is essentially making every pixel more similar to those around it. In a horizontal blur, we can average each pixel with those in a specific direction, which gives the illusion of motion.
For this program, we will calculate each new pixel's value as half of its original value. The other half is averaged from a fixed number of pixels to the right. This fixed number can be called the `bluramount`. The larger the value, the more blurring that will occur.

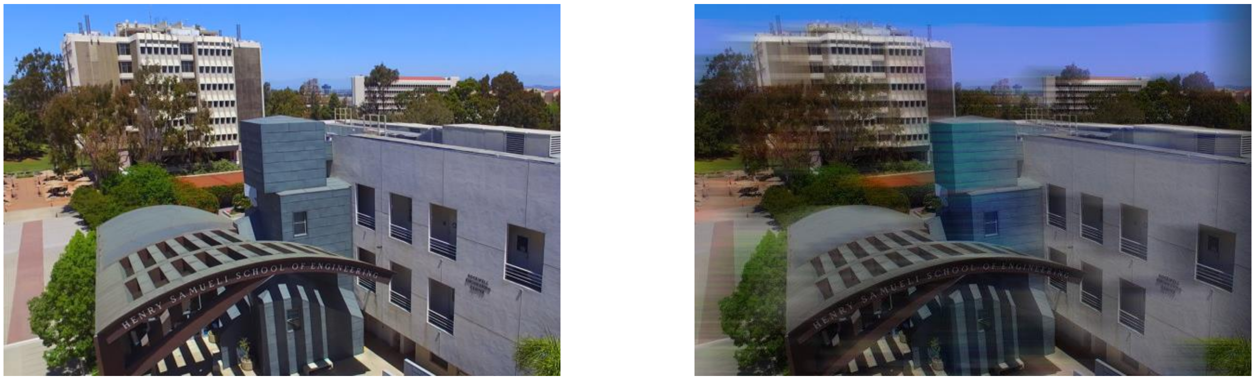
This is applied for the red, green, and blue intensity of each pixel.

You must also ensure you dont access pixels off the bounds of the image. For example, the third pixel from the right should only average itself (half weight), and the next two (at a quarter weight each).

For this program, set the `bluramount` to 50.

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

```
/* Make a blurred image*/
void MotionBlur(int BlurAmount,
    unsigned char R[WIDTH][HEIGHT],
    unsigned char G[WIDTH][HEIGHT],
    unsigned char B[WIDTH][HEIGHT]);
```



(a) Original image (b) blurred image with motion blur=50

Figure 5: An image and its motion blur.

Here, *BlurAmount* specifies the percentage of blur in the image.
Figure 5 shows an example of this operation where *blouramount* is 50. Once the user chooses this option, your program's output should look like this:

```
Please make your choice: 14
Please input blur amount: 50
"motion blur" operation is done!
```

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

1.8 Support for the DEBUG mode

In C programs, *macros* can be defined as preprocessing directives. Please define a macro named **"DEBUG"** in your source code to enable / disable the messages shown in the *AutoTest()* function.

When the macro is defined, the main menu will not appear, your program executes only the function *AutoTest()* and finishes afterwards. The messages in the *AutoTest()* show up. If the macro is not defined, the program will execute in its regular fashion and the main menu will appear. The messages in the function *AutoTest()* will not show up. The printf statements in the `LoadImage()` and `SaveImage()` function will stay.
Please decide in which function and in which module this **"DEBUG"** macro needs to be added.

1.9 Extend the Makefile

For the **Makefile**, please

- extend it properly with the targets for your program with the new module: **Advanced.c**.
- generate two executable programs
  1. *PhotoLab* with the user interactive menu and the **DEBUG** mode off.
  2. *PhotoLabTest* without the user menu, but with only the *AutoTest()* function for testing, and turn the **DEBUG** mode on. Note that we can thus use the same source files to generate two different programs.

  Define two targets to generate these two programs. Please use the *"-D"* option for gcc to enable / disable the DEBUG mode instead of defining the "DEBUG" macro in the source code. You may need to define more targets to generate the object files with different DEBUG modes.

2 Implementation Details

2.1 Function Prototypes

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

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

/* Add noise to image */
void Noise(int n,
    unsigned char R[WIDTH][HEIGHT],
    unsigned char G[WIDTH][HEIGHT],
    unsigned char B[WIDTH][HEIGHT]);

/* Sharpen an image */
void  Sharpen(unsigned char R[WIDTH][HEIGHT],
unsigned char G[WIDTH][HEIGHT],
        unsigned char B[WIDTH][HEIGHT]);

/* Posterize the image */
void Posterize(unsigned char R[WIDTH][HEIGHT],
    unsigned char G[WIDTH][HEIGHT],
    unsigned char B[WIDTH][HEIGHT],
    unsigned int rbits,
    unsigned int gbits,
    unsigned int bbits);

/* Motion blur */
void MotionBlur(int BlurAmount, unsigned char R[WIDTH][HEIGHT],
    unsigned char G[WIDTH][HEIGHT],
    unsigned char B[WIDTH][HEIGHT]);
```

You may want to define other functions as needed.

2.2 Global constants

The following global constants should be defined in **Constants.h** (please don't change their names):

```
#define WIDTH       600 /* image width */
#define HEIGHT      400 /* image height */
#define SLEN         80 /* maximum length of file names */
```

Please make sure that you properly include this header file when necessary.

3 Budgeting your time

You have two weeks to complete this assignment, but we encourage you to get started early as there is a little more work than for Assignment 2. We suggest you budget your time as follows:

- Week 1:
    1. Decompose the program into different modules, i.e. **PhotoLab.c**, **FileIO.c**, **FileIO.h**, **Constants.h**, **DIPs.c**, **DIPs.h**.

2. Create your own **Makefile** and use it to compile the program.
3. Create module **Advanced.c**, **Advanced.h**, and implement an initial advanced DIP function.

- Week 2:
  1. Implement all the advanced DIP functions.
  2. Implement the *AutoTest()* function.
  3. Figure out how to enable/disable the **DEBUG** mode in the source code and add targets to the **Makefile** accordingly.
  4. Script the result of your programs and submit your work.
  5. Bonus part

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**: type 'make clean', then 'make', then './PhotoLab'.
3. Choose 'Test all functions' (The file names must be 'BlackNWhite', 'Negative', 'ColorFilter', 'Edge', 'Shuffle', 'Vflip', 'Vmirror', 'Noise', 'Sharpen', 'Posterize', and 'MotionBlur' for the corresponding function).
4. Exit the PhotoLab.
5. Compile and run *PhotoLabTest*: type 'make PhotoLabTest'.
6. Test the dependencies in your Makefile: type 'touch Advanced.c', then 'make PhotoLab'.
7. Stop the script by typing the command: *exit*.
8. Rename the script file to *PhotoLab.script*.

NOTE: make sure to 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*
- *FileIO.c*
- *FileIO.h*
- *Constants.h*
- *DIPs.c*
- *DIPs.h*
- *Advanced.c*
- *Advanced.h*
- *Makefile*
- *PhotoLab.txt*

6 Grading

- *DIP zoom: 15 points*
- *DIP sharpen: 15 points*
- *DIP posterize: 15 points*
- *Menu: 5 points*
- *Autotest: 5 points*
- *Decomposition 15 points*
- *Makefile: 30 points*
- *Bonus: 10 points*