EECS 22: Assignment 4 Digital Image Processing

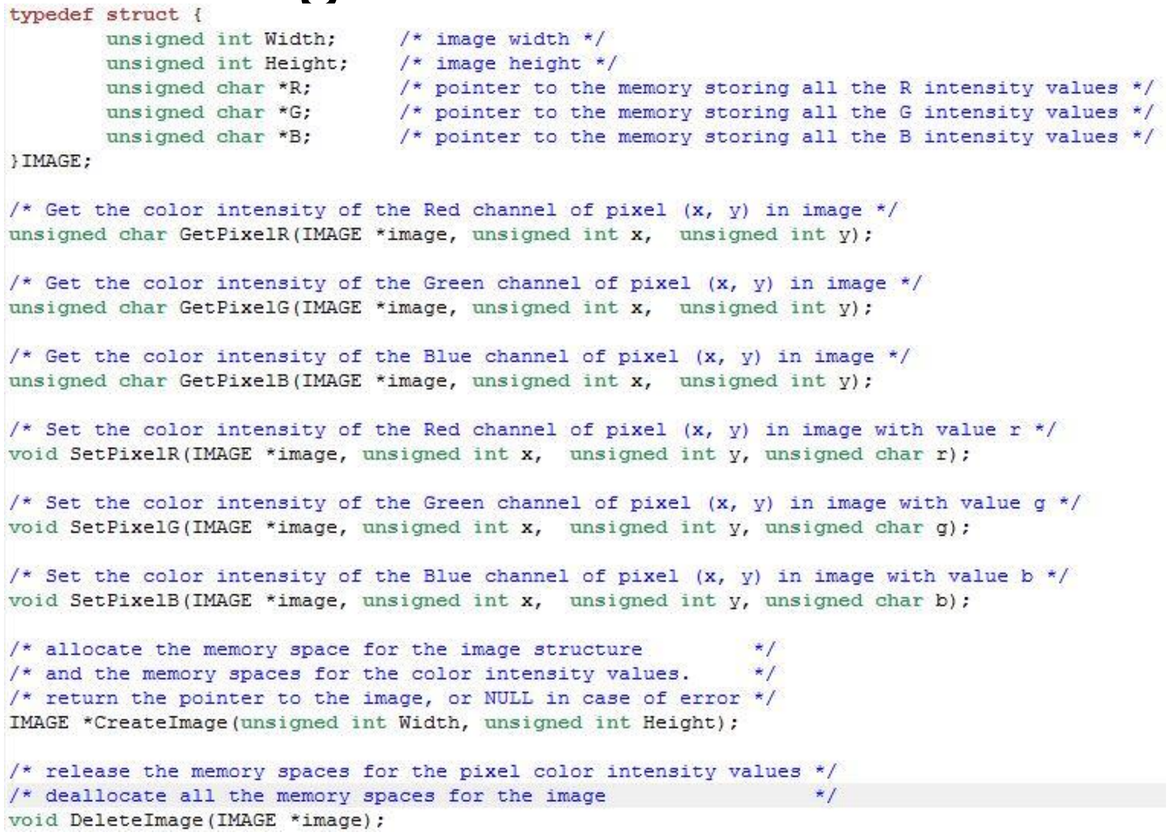
Due on Monday 11/18/2013 11:00pm

Outline

- Change program to accommodate varying input file size
  - struct and malloc + free
- Add 3 new Advanced DIP operations
  - Resize, rotate, Julia set
- Extend the Test all functions
- Extend the Makefile
- Use “Valgrind” tool to find memory leaks

Varying image size

- New image.h header file



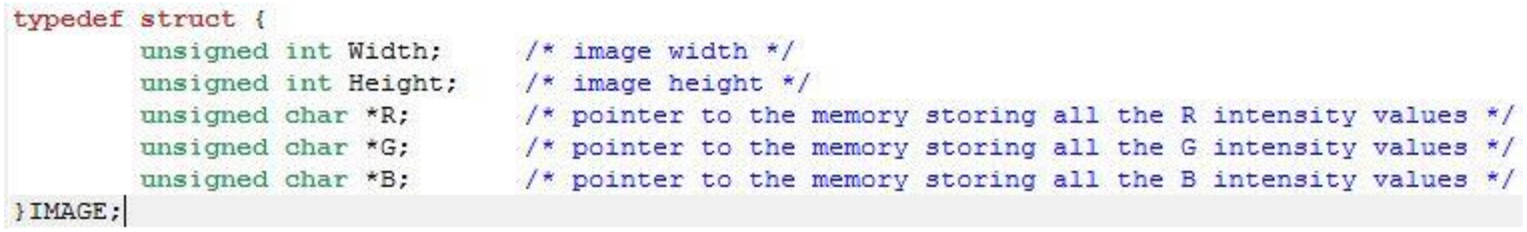
```c
typedef struct {
        unsigned int Width;     /* image width */
        unsigned int Height;    /* image height */
        unsigned char *R;       /* pointer to the memory storing all the R intensity values */
        unsigned char *G;       /* pointer to the memory storing all the G intensity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

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

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

/* Get the color intensity of the Blue channel of pixel (x, y) in image */
unsigned char GetPixelB(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 SetPixelR(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 SetPixelG(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 SetPixelB(IMAGE *image, unsigned int x,  unsigned int y, unsigned char b);

/* 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);
```

Varying image size

- R G B fixed size arrays replaced by IMAGE data structure
- Passing around pointers to an IMAGE variable
  - IMAGE *image;
- IMPORTANT: Define CreateImage function first because it is used in FileIO.c to allocate the memory for the image.
  - Your program will probably not compile or run properly (if compiled successfully) if this function is not defined correctly.

struct

- New data type defined by a struct



```
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;
```

- A data type that can encapsulate other data types, including itself
- As if you're creating a new 'data type'

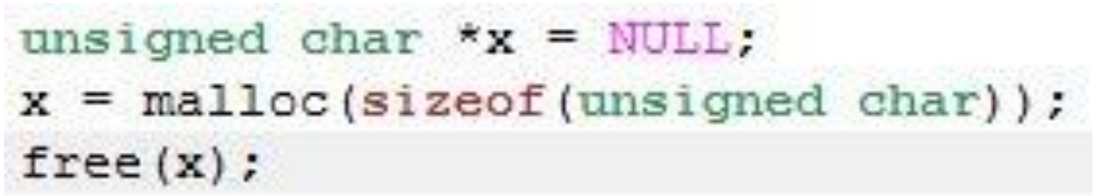
struct

- Can declare variables using the ‘new data type’
  - IMAGE x;
    - creates a new variable x of type IMAGE
    - Can access the variables inside struct x by the following
      - x.Height, x.Width, x.R, x.G, x.B
  - IMAGE *x;
    - Can access the variables inside struct x by the following
      - (*x).Height, (*x).Width, (*x).R, (*x).G, (*x).B
        - » Must do indirection first
      - x->Height, x->Width, x->R, x->g, x->B
        - » ‘->’ is equivalent to ‘(*x).’
- **More details in Lecture 11**

malloc

- Malloc (memory allocation)
  - Telling the program to reserve an amount of memory for variable use
  - Typically involves pointers
  - Dynamically allocated (unlike static fixed size [e.g. array])
  - Should be free()’d after it’s use
- **More details in Lecture 12**

malloc

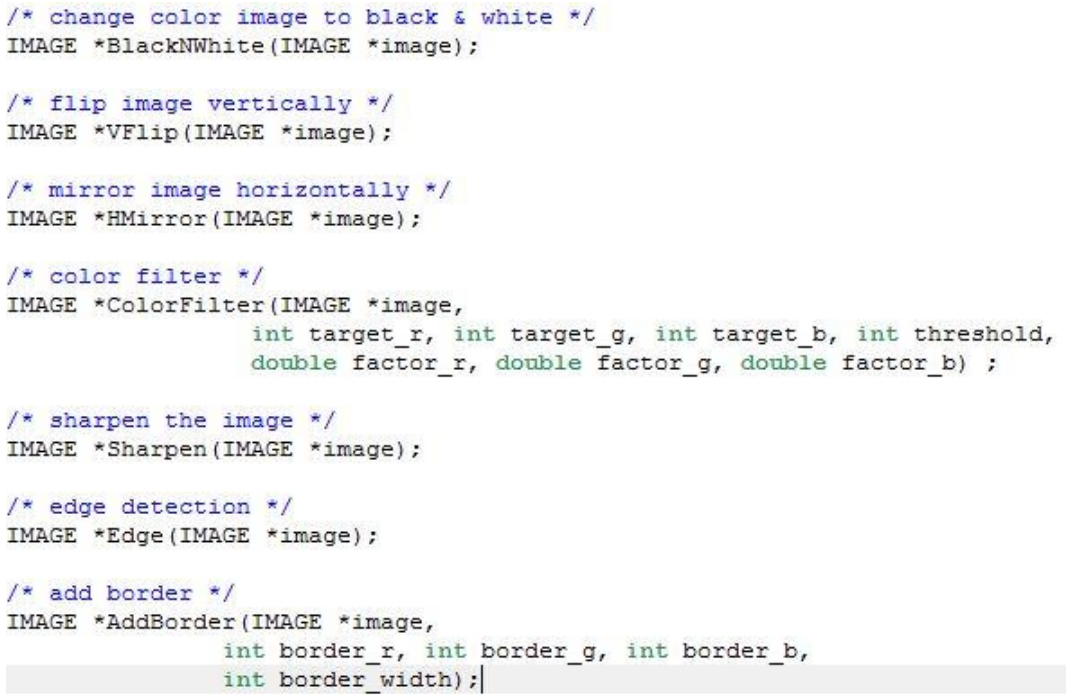


```
unsigned char *x = NULL;
x = malloc(sizeof(unsigned char));
free(x);
```

- malloc allocates a piece of memory the size of 1 unsigned char
- Use a pointer variable to point to that allocated piece of memory by the second line
- Can do any amount of operations on the pointer variable
- free()'s the pointer variable and the piece of memory it is pointing to when it is done using it.

Function Prototypes

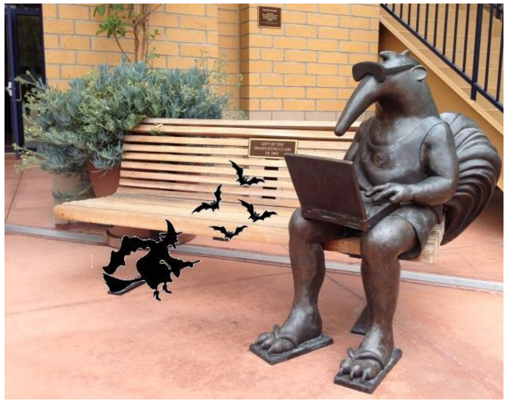
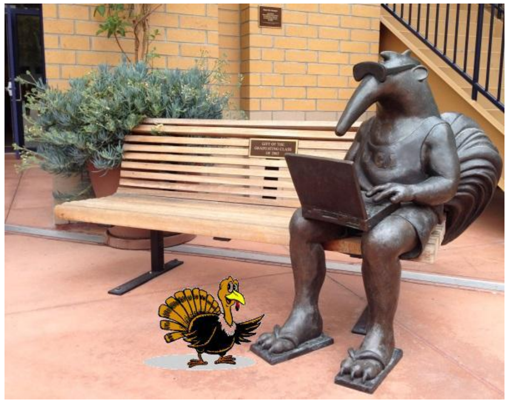
- Changed return types in DIPS.h and Advanced.h
  - Used to void return type, now IMAGE * (pointer to an IMAGE variable)
  - DIPS.h shown below. Something similar in Advanced.h



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

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

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

/* color filter */
IMAGE *ColorFilter(IMAGE *image,
                 int target_r, int target_g, int target_b, int threshold,
                 double factor_r, double factor_g, double factor_b) ;

/* sharpen the image */
IMAGE *Sharpen(IMAGE *image);

/* edge detection */
IMAGE *Edge(IMAGE *image);

/* add border */
IMAGE *AddBorder(IMAGE *image,
               int border_r, int border_g, int border_b,
               int border_width);
```

Overlay

- Turn in 2 images
  - 1st image is overlay halloweenBat image at position 100, 150
  - 2nd image is overlay turkey image at position 165, 325

Resize

- Resizing the image based on an user entered percentage between 0 and 500
- More specifically, scale

$$Width_{new} = Width_{old} * (\text{percentage} / 100.00);$$

$$Height_{new} = Height_{old} * (\text{percentage} / 100.00);$$

Resize

- Pixels should be calculated according to

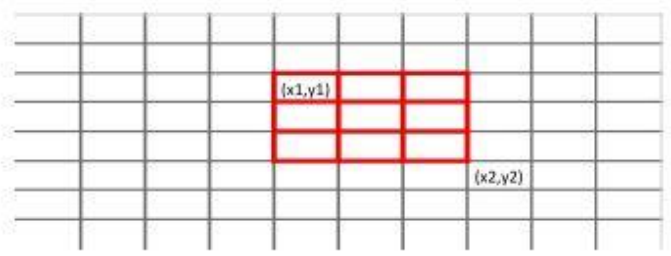


Figure 3: Pixels mapping from the bigger original image to the smaller new image

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

  - Note: x and y are coordinates of new image
  - x1, y1, x2, y2, are coordinates of original image

Resize

(a) Original image

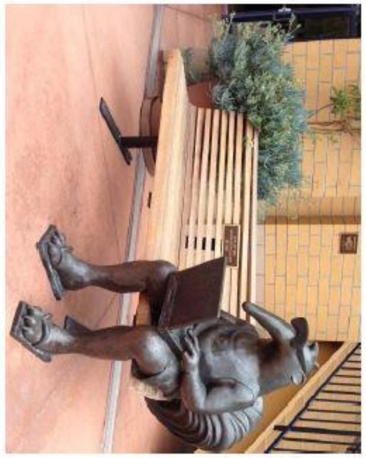
(b) resized to a bigger image (percentage = 175)

(c) resized to a smaller image (percentage = 60)

Rotate

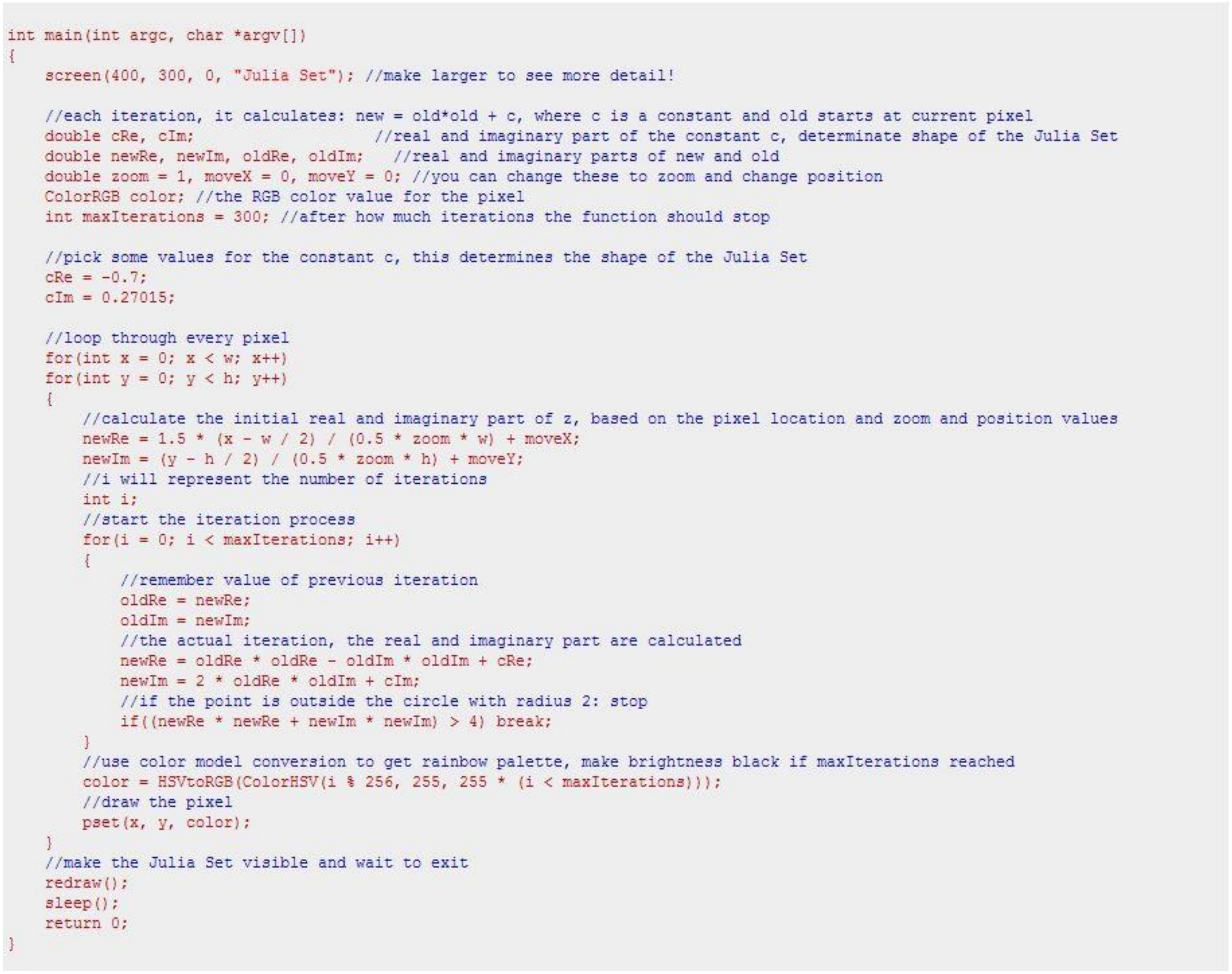
- Rotate 90 degrees
  - Invert width and height
  - Assign appropriate pixel to new coordinate
    - Top left (0, 0) pixel should be new image's top right (Width-1, Height-1) pixel

(a) Original image


(b) Rotated image

Julia Set

- A set of operations that reveals more and more detail through progressive and recursive computation
- Adapting source code from online to fit our DIP program, specifically http://lodev.org/cgtutor/juliamandelbrot.html
- Note that the Julia Set does not operate on a specific file. Rather, it just does computations on an image file size and produces a fine-detailed image
- No input file is needed for the Julia Set
  - Note that we did not have to read in UCI_Peter before calling the Julia Set operation
- The following page contains the algorithm's source code adapted from the link listed above

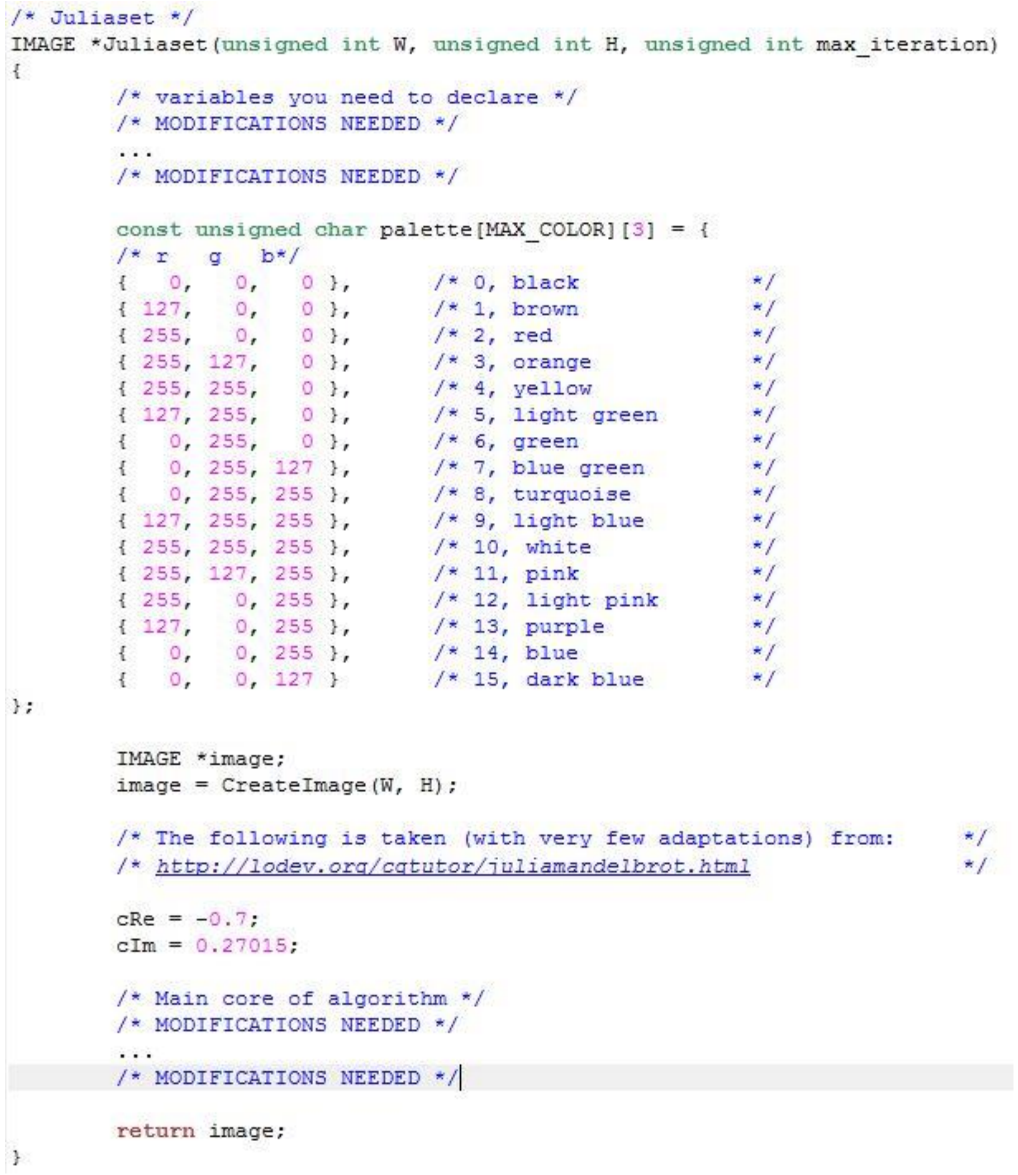


```
int main(int argc, char *argv[])
{
    screen(400, 300, 0, "Julia Set"); //make larger to see more detail!

    //each iteration, it calculates: new = old*old + c, where c is a constant and old starts at current pixel
    double cRe, cIm;                   //real and imaginary part of the constant c, determinate shape of the Julia Set
    double newRe, newIm, oldRe, oldIm;   //real and imaginary parts of new and old
    double zoom = 1, moveX = 0, moveY = 0; //you can change these to zoom and change position
    ColorRGB color; //the RGB color value for the pixel
    int maxIterations = 300; //after how much iterations the function should stop

    //pick some values for the constant c, this determines the shape of the Julia Set
    cRe = -0.7;
    cIm = 0.27015;

    //loop through every pixel
    for(int x = 0; x < w; x++)
    for(int y = 0; y < h; y++)
    {
        //calculate the initial real and imaginary part of z, based on the pixel location and zoom and position values
        newRe = 1.5 * (x - w / 2) / (0.5 * zoom * w) + moveX;
        newIm = (y - h / 2) / (0.5 * zoom * h) + moveY;
        //i will represent the number of iterations
        int i;
        //start the iteration process
        for(i = 0; i < maxIterations; i++)
        {
            //remember value of previous iteration
            oldRe = newRe;
            oldIm = newIm;
            //the actual iteration, the real and imaginary part are calculated
            newRe = oldRe * oldRe - oldIm * oldIm + cRe;
            newIm = 2 * oldRe * oldIm + cIm;
            //if the point is outside the circle with radius 2: stop
            if((newRe * newRe + newIm * newIm) > 4) break;
        }
        //use color model conversion to get rainbow palette, make brightness black if maxIterations reached
        color = HSVtoRGB(ColorHSV(i % 256, 255, 255 * (i < maxIterations)));
        //draw the pixel
        pset(x, y, color);
    }
    //make the Julia Set visible and wait to exit
    redraw();
    sleep();
    return 0;
}
```

Julia Set

- The source code uses 256 colors to draw the Julia set image. The more colors used to draw the image, the more detailed the image becomes. However, this also increases complexity.
- Therefore, we will restrict the amount of colors we can draw to only 16. This allows us to simplify the code and produce decently complex pictures.
- Suggested to follow the setup on the following slide for implementing this function. Insert your code between the sections marked off as /* MODIFICATIONS NEEDED */

Julia Set



```
/* Juliaset */
IMAGE *Juliaset(unsigned int W, unsigned int H, unsigned int max_iteration)
{
        /* variables you need to declare */
        /* MODIFICATIONS NEEDED */
        ...
        /* MODIFICATIONS NEEDED */

        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       */
        { 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      */
};

        IMAGE *image;
        image = CreateImage(W, H);

        /* The following is taken (with very few adaptations) from:     */
        /* http://lodev.org/cgtutor/juliamandelbrot.html                */

        cRe = -0.7;
        cIm = 0.27015;

        /* Main core of algorithm */
        /* MODIFICATIONS NEEDED */
        ...
        /* MODIFICATIONS NEEDED */

        return image;
}
```

Test all functions

- Extend to incorporate new DIP functions
- Note the return type and argument type changes!!

Extend the Makefile

- Be sure to incorporate the new image.c, fileIO.h, and fileIO.c files
- Generate 2 executables
  - Photolab: Interactive menu without DEBUG statements
  - PhotoLabTest: Automatically tests all functions with DEBUG statements

Valgrind

- Use this tool to find improper memory usages
  - Not free()-ing memory after malloc()-ing and using them
  - Trying to use a location beyond the space the user has malloc()-ed
- Compile your code and run the following command
  - valgrind --leak-check=full PhotoLabTest

Important Point

- Almost all functions are passing pointers to IMAGE variables and not the IMAGE struct itself
- Passing pointers ensures that values will change
- Know the difference in how to access variables, how to modify variables, etc.
- You should clearly know the difference and why this matters!