Image Processing

Lab 8: https://maryash.github.io/135/labs/lab_08.html

- Sadab Hafiz

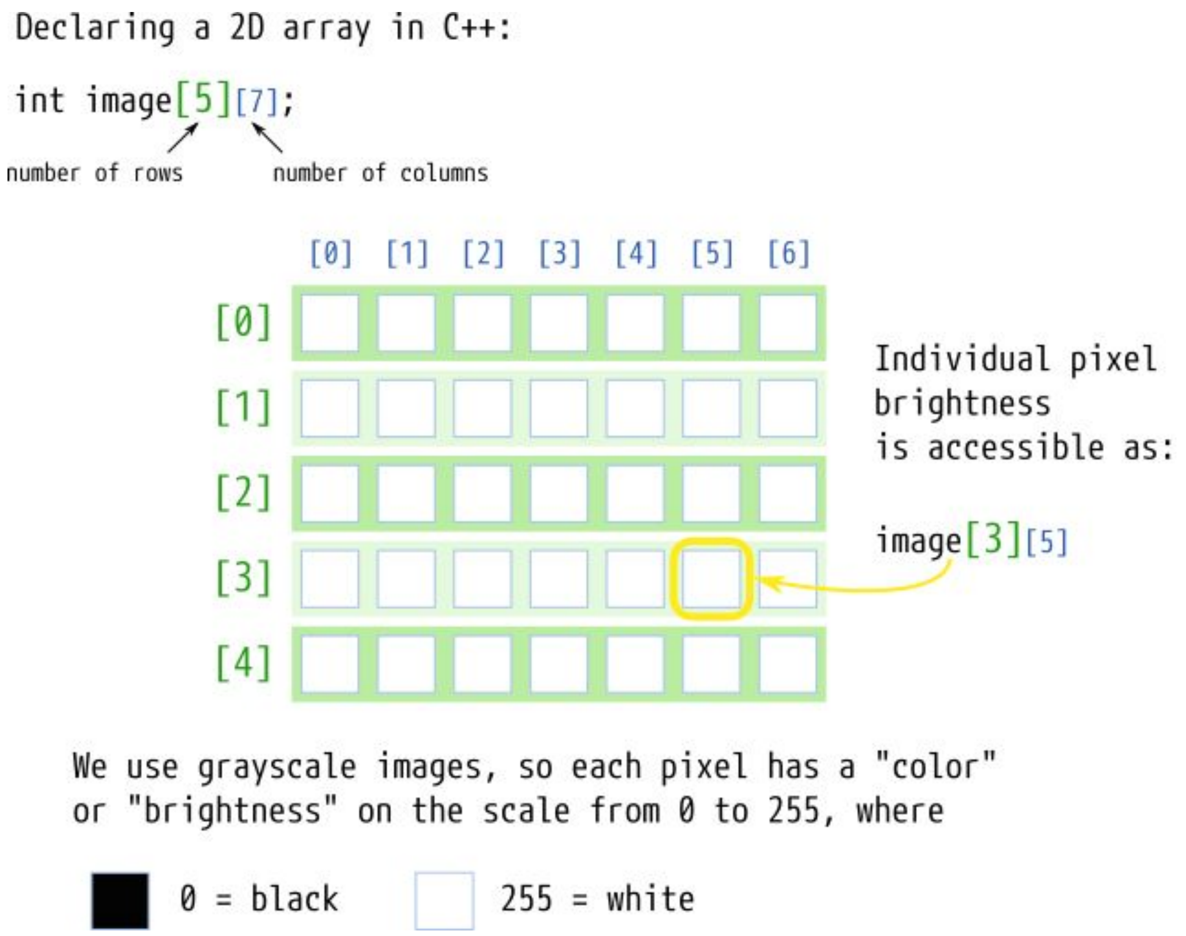
Image in pgm format

In this lab, we will modify image files that are in pgm format. The file contains pixel data that can be represented using a 2D array. The sample image provided in the lab can be found in: inImage.pgm

If you want to use your own image that is in png format, you can use the program called Eye of Gnome (eog). Unfortunately, this is only available in mac/linux. You can find pgm format images online that can also be used.

View an image file using eog	:	`eog inImage.pgm`
Convert a png file to pgm	:	`convert -compress none dog.png inImage.pgm`
Convert a pgm to png	:	`convert outImage.pgm output-dog.png`

Learn more about EOG: https://help.gnome.org/users/eog/stable/


Declaring a 2D array in C++:

```
int image[5][7];
```

We use grayscale images, so each pixel has a "color" or "brightness" on the scale from 0 to 255, where

■ 0 = black   □ 255 = white

Pixels act like light. 0 is black and 255 is white.

Since we are working with a 2D-array, we have to traverse through the rows and the columns using a nested for-loop.

In our nested for-loop, outer-loop will represent rows and inner-loop will represent columns.

Access individual pixels using `image[row][col]`.

Preliminary Task - Reading and Writing Pictures

For this lab, you are given some starter functions. You can find them here: lab-images.cpp

```
void readImage(int image[MAX_H][MAX_W], int &height, int &width);
```

This function reads the pixel colors into the array, and updates the variables height and width with the actual dimensions of the loaded image.

```
void writeImage(int image[MAX_H][MAX_W], int height, int width);
```

This function saves the given image array of given height and width into an image file that will be called `outImage.pgm`. All the pixels in the array MUST be between 0 and 255(inclusive).

The provided main function creates an array `img`, reads the picture from the `inImage.pgm` into this array, copies this array to a second array of the same dimensions, and writes this second array into the output file `outImage.pgm`.

Things to consider

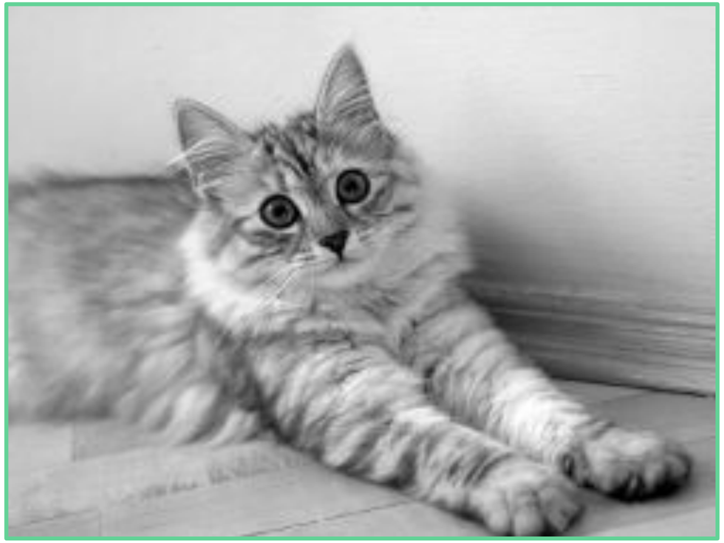
Don’t modify the two provided functions. Only make changes to the main function or write your own functions that can be used in the main function.

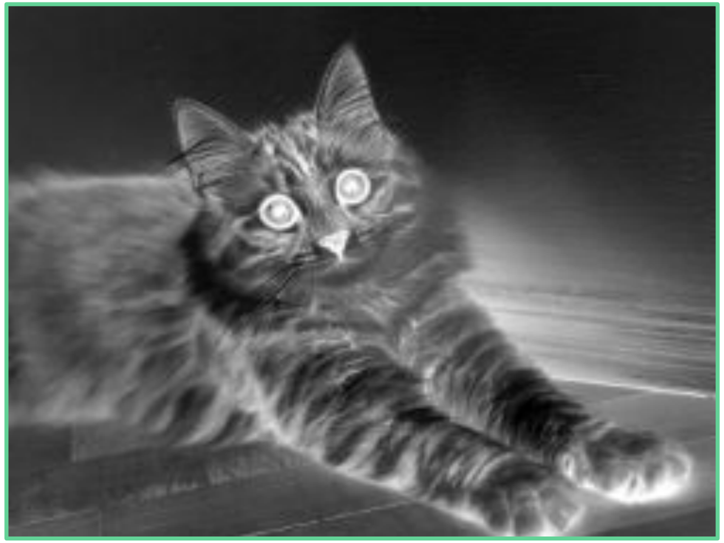
If you look at the main function provided, you will see that instead of modifying the array of the image we are reading, we are creating a new array. For this lab, we will use that new array and never modify the array of the original image.

The size of the array in provided main function is 512 X 512 which is significantly larger than the image that we are using. That is totally fine as the function will keep track of the dimensions. If you are using your own image file, make sure that the height and the width of the image is less than 512.

Task A

Our program needs to invert all the pixels of the input image:





Inverting the pixels

We need to find a way to invert all the pixels. Let's look for a pattern:

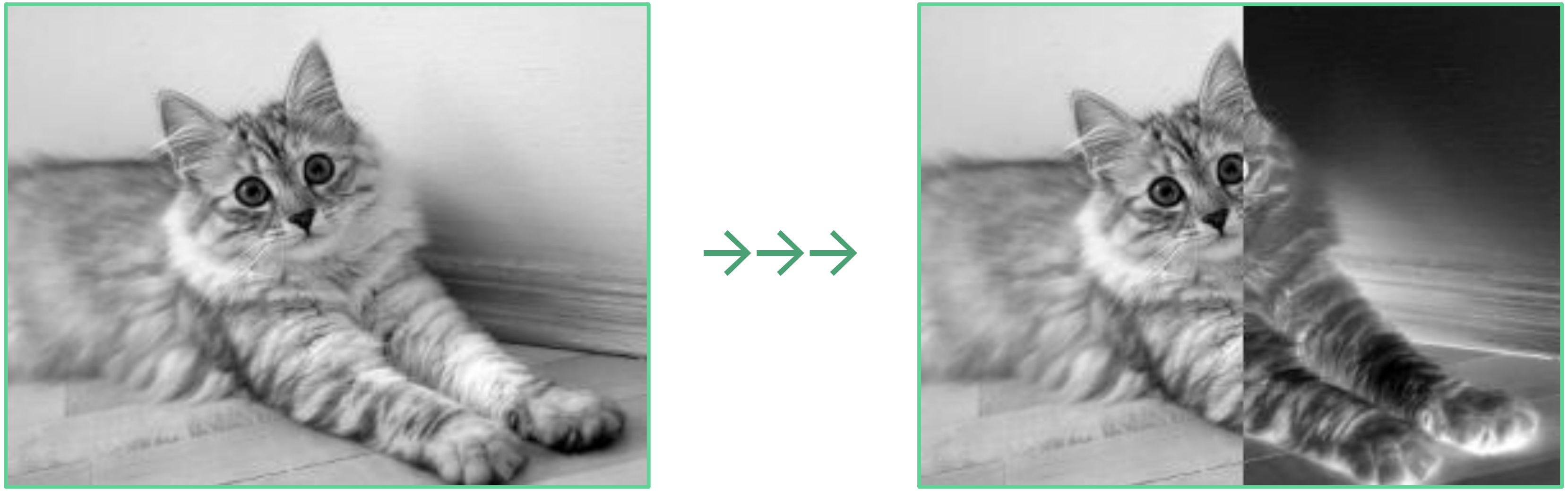
0 → 255	255 - 0 = 255
1 → 254	255 - 1 = 254
2 → 253	255 - 2 = 253
…	…
255 → 0	255 - 255 = 0

Therefore, 255 - pixel = inverted pixel

Within our nested for-loop, the inverted pixel of `image[row][col]` would be:
`255 - image[row][col]`

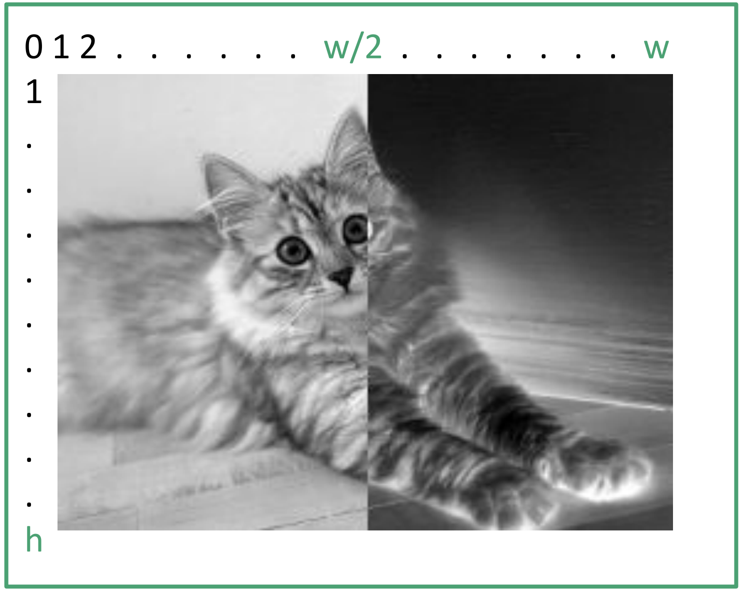
Task B

We will invert the pixels in the right half of the image.



Isolating the right half of the image

Let's identify what the right half of the image is based on the height and width.



h	:	maximum height
w	:	maximum width
w/2	:	half of maximum width

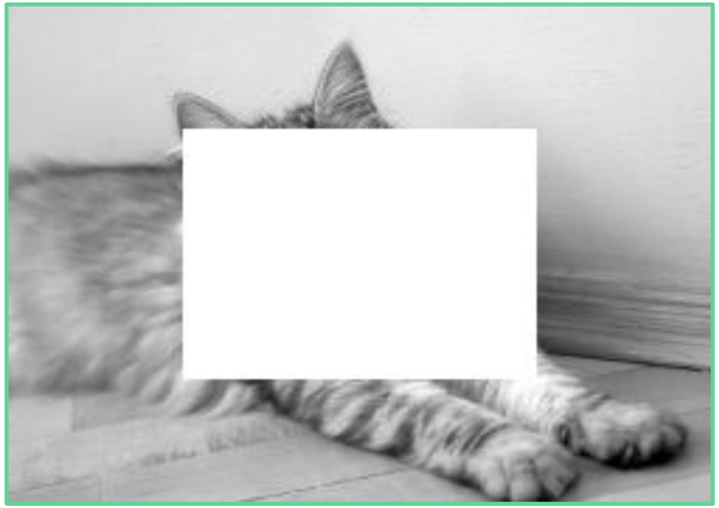
The right half of the image occurs when \`col\` is greater than the half of the maximum width.

Therefore, using an if-statement, check if \`col>=w/2\` and invert the pixels if that is the case. Otherwise, set the pixel equal to the original image pixel.

Task C

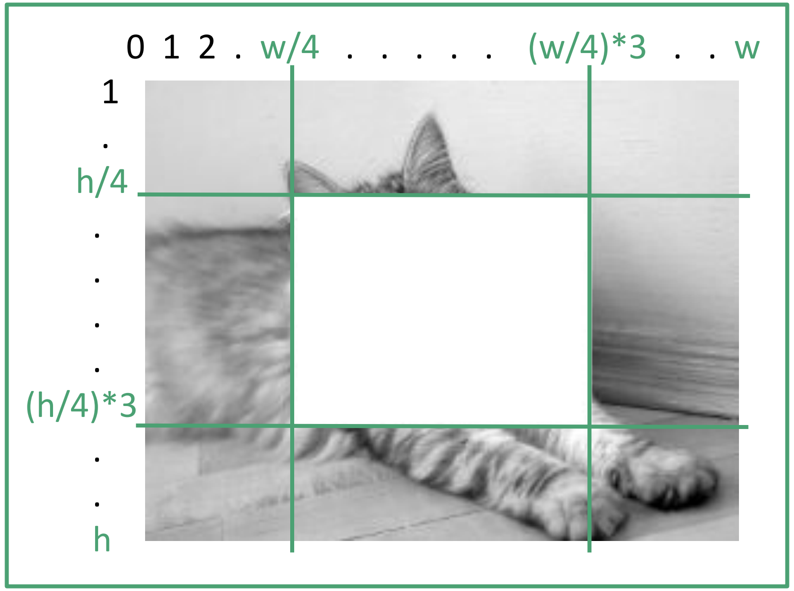
For this task, our program will produce a rectangular box in the middle of the image. The box is 50% by 50% of the original image height and width:

$\rightarrow\rightarrow\rightarrow$



Isolating the center of the image

Let’s identify what the 50% in the center of the image is based on the height and width.

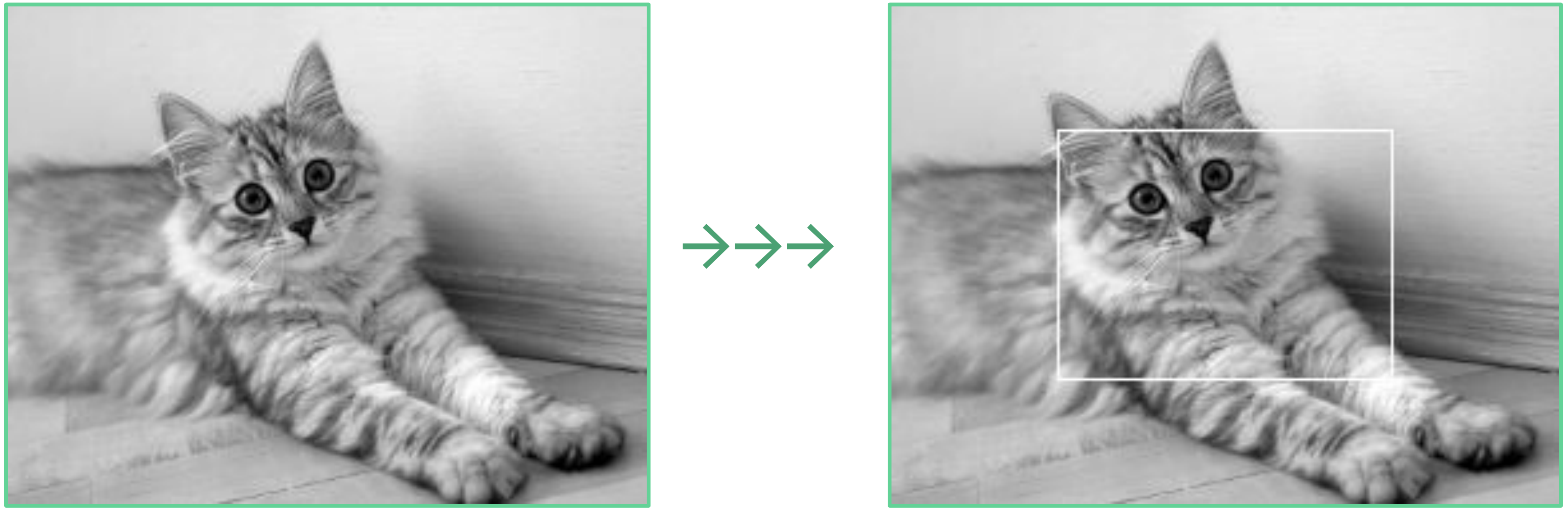


h	:	maximum height
w	:	maximum width
w/4	:	first quarter of the width
h/4	:	first quarter of the height
(w/4)*3	:	third quarter of the width
(h/4)*3	:	third quarter of the height

Pixels that are greater than the first quarter of the height and width but smaller than the third quarter of the height and width are all within the white box in the center. Using an if statement, set those pixels to white.

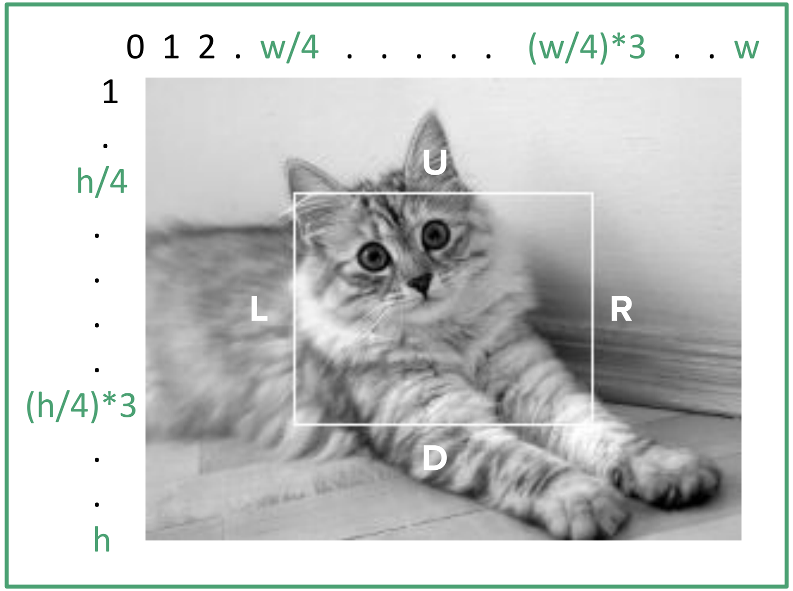
Task D

Produce a rectangular frame in the middle of the image:



Isolating the center of the image

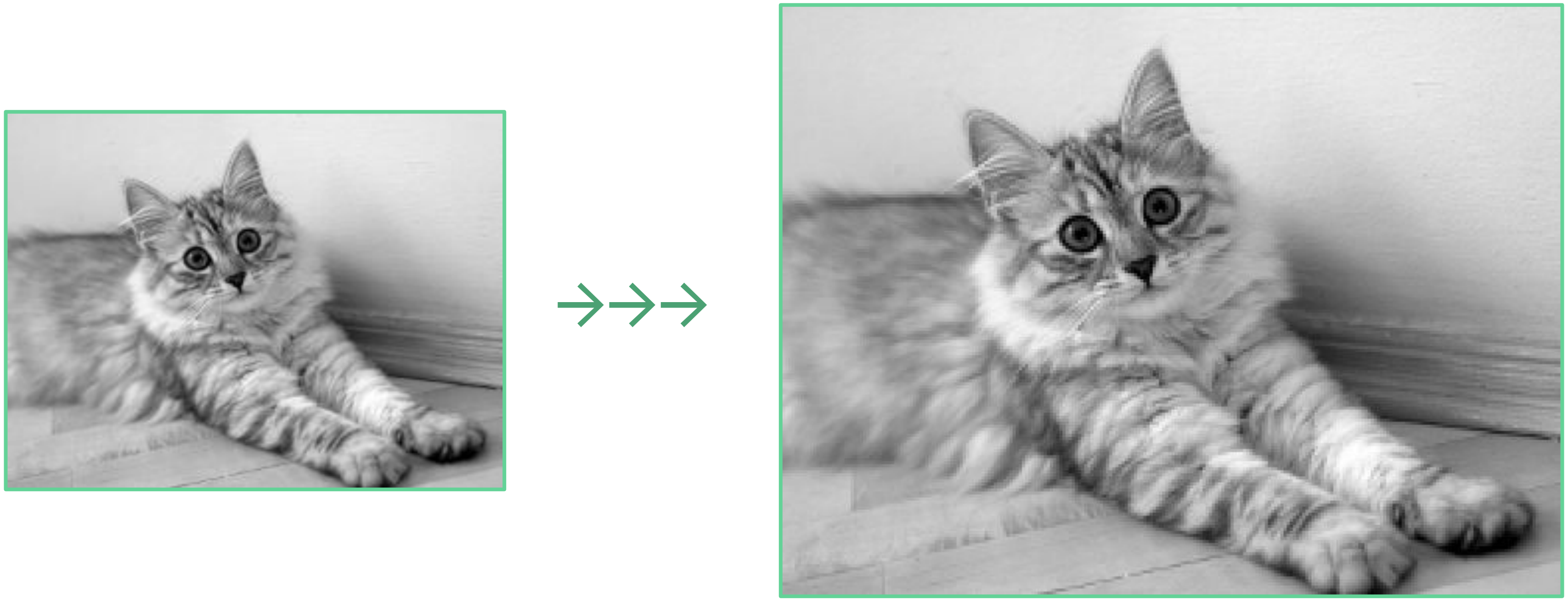
Isolate the lines individually using an if-statement and set the pixels to white.



U: row==h/4 && (col>=w/4 && col<=(w/4)*3)
D: row==(h/4)*3 && (col>=w/4 && col<=(w/4)*3)
L: col==(w/4) && (row>=h/4 && row<=(h/4)*3)
R: col==(w/4)*3 && (row>=h/4 && row<=(h/4)*3)

Task E

Scale the image upto 200% of its original size:



Small Image Pixels vs Large Image Pixels

Make the assumption that the image even when doubled in size, the image will be smaller than the array size limit of 512.

```
100%  →   200%

11 22 →  11 11 22 22
33 44    11 11 22 22
         33 33 44 44
         33 33 44 44

Each number as row, col:
0,0 0,1 →0,0 0,1 0,2 0,3
1,0 1,1   1,0 1,1 1,2 1,3
          2,0 2,1 2,2 2,3
          3,0 3,1 3,2 3,3
```

Since the output image is 2x the size of the original image,
output [row*2] [col*2] = original [row] [col]

Let's see how the rows and the columns are related:

1. output [2] [2] = original [1] [1]
2. output [2] [3] = original [1] [1]
3. output [3] [2] = original [1] [1]
4. output [3] [3] = original [1] [1]

Based on this, we can derive that:

1. output [row*2] [col*2] = original[row] [col]
2. output [row*2] [(col*2)+1] = original[row] [col]
3. output [(row*2)+1] [col*2] = original[row] [col]
4. output [(row*2)+1] [(col*2)+1] = original[row] [col]

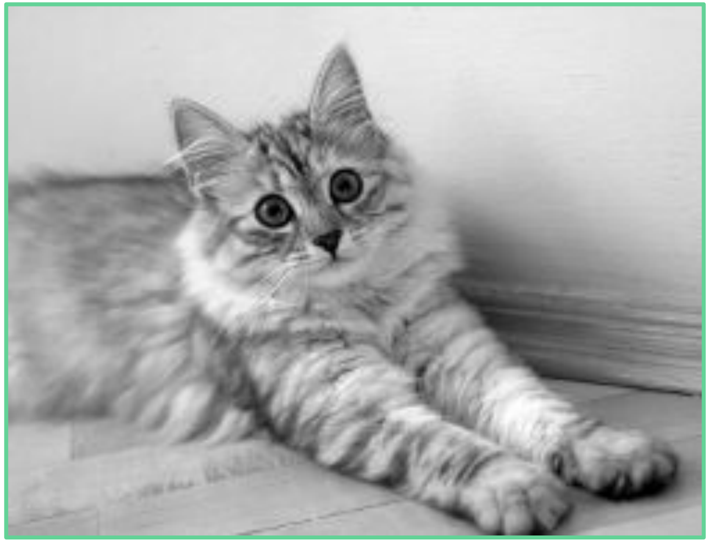
Modifying the main function (Task E)

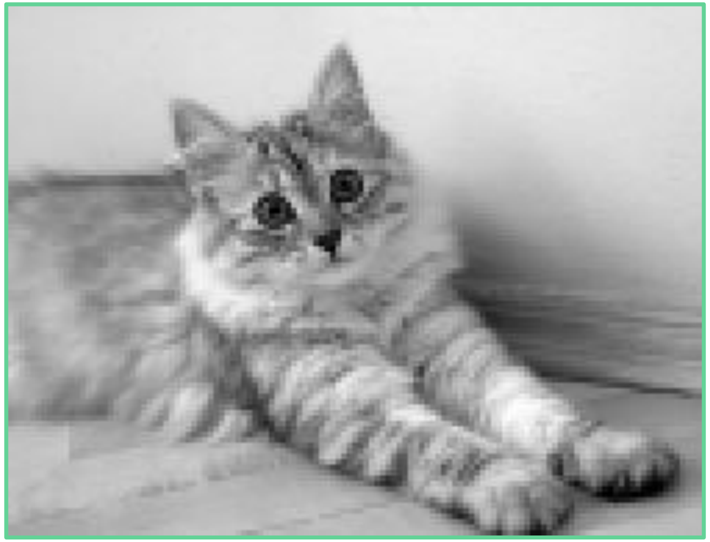
```
int main() {

      … we don’t need to change anything before the nested loops

      for(int row = 0; row < h; row++) {

            for(int col = 0; col < w; col++) {

                  out[row*2][col*2] = img[row][col];

                  … other three translations (2,3,4 from the last slide)

            }

      }

      … save the image with 2x dimensions of the original image

}
```

The image won’t be visible completely if it is not saved with twice the dimensions of the original image.

Task F

Write a program to pixelate the input image:





Pixelating the image

Pixelating an image means replacing every four adjacent pixels to the average of the pixels.

original → pixelated

```
10 20 30 40  →  16 16 36 36
11 21 31 41     16 16 36 36
12 22 32 42     18 18 37 37
13 23 33 43     18 18 37 37
```

(10+20+11+21)/4 = 16
(30+40+31+41)/4 = 36
(12+22+13+23)/4 = 18
(32+42+33+43)/4 = 37

The adjacent pixels are:

A. img[row][col]
B. img[row+1][col]
C. img[row][col+1]
D. img[row+1][col+1]

Average = (A+B+C+D)/4

Set ALL the adjacent pixels of the output image to the average. For example: out[row][col] = average;

Modifying the main function (Task F)

```
int average = 0;

for(int row = 0; row < h; row+=2) {                     // skip every other row

     for(int col = 0; col < w; col+=2) {                // skip every other col

          average = (A+B+C+D)/4;                        // refer to previous slide

          out[row][col] = average;

          …set the other three adjacent pixels of out to average

     }

}

writeImage(out, h, w);
```

Since four pixels of the output image are set in one iteration of the for-loop, it is necessary to skip odd rows and cols. Therefore, instead of row++ and col++ we can do row+=2 and col+=2.