Lab 8. Image Processing

In this lab, we are going to do image manipulation and image filtering. Since the pixel data of the images is going to be stored in a 2D array, we will also learn how to use 2D arrays.

Preliminaries & PGM graphical format

We will be using simple Portable graymap (PGM) format for our images today, because the code to read and write such images is super simple (much simpler than it is for JPEG or PNG).

Please download inImage.pgm

To view an image file, use the program `eog` (Eye of Gnome):

```
eog inImage.pgm
```

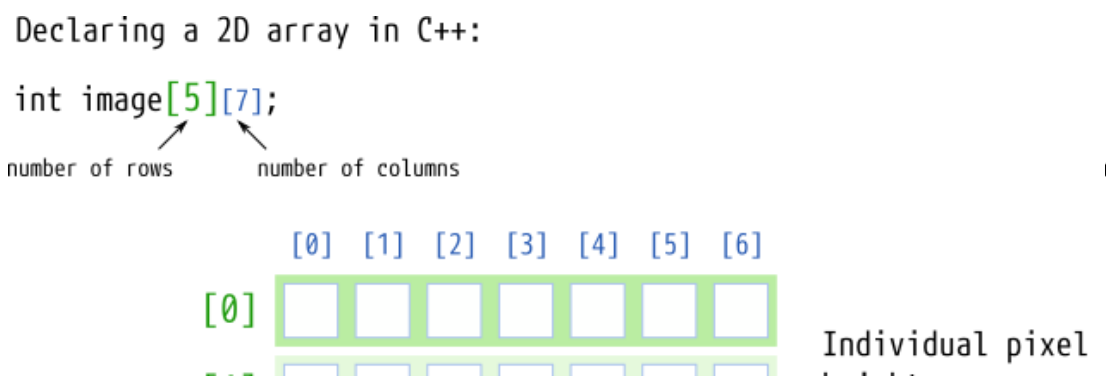
If you wish to use an alternative image other than the one provided, please convert your image file into PGM format using the command `convert` :

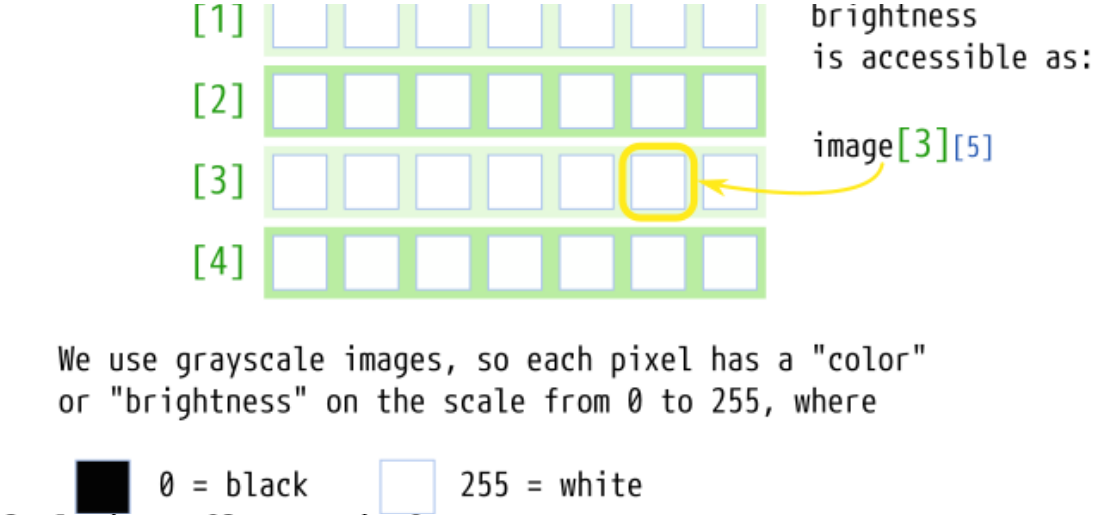
```
convert -compress none dog.png inImage.pgm
```

and for the reverse transformation:

```
convert outImage.pgm output-dog.png
```

2D arrays for representing grayscale images





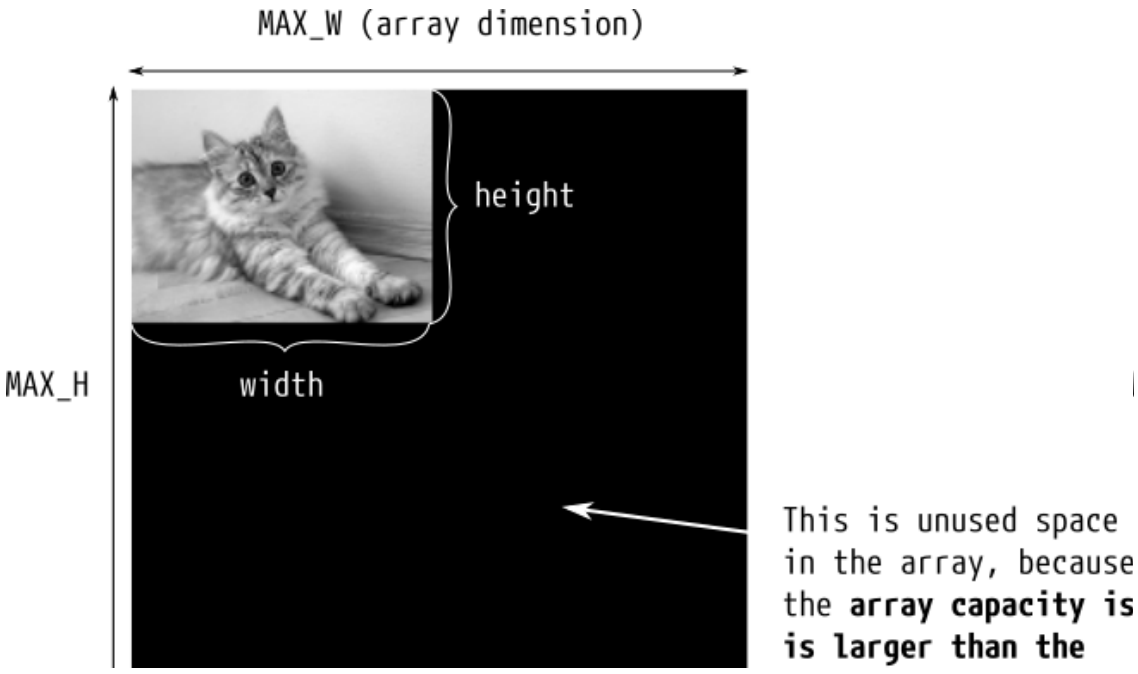
Preliminary Task. Reading and writing pictures. (Don't submit this one)

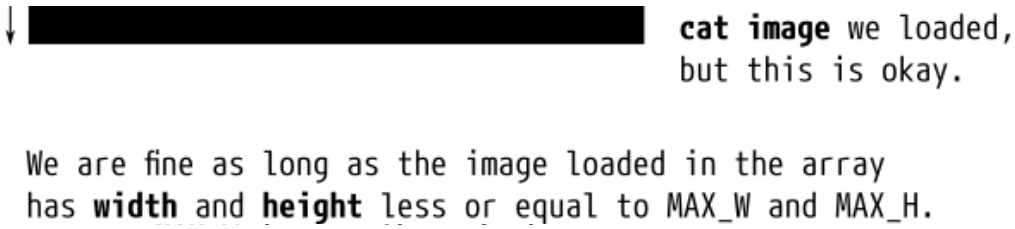
The supplied program `lab-images.cpp` provides two helper functions for reading and writing images:

```
// Reads file 'inImage.pgm' into a 2D array
void readImage(int image[MAX_H][MAX_W], int &height, int &width);
```

When called, the function

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





The second supplied function is doing the opposite operation, given an array and the dimensions of the actual picture in it, it saves the picture into the file `outImage.pgm`:

```
// Writes the array data into file `outImage.pgm`
void writeImage(int image[MAX_H][MAX_W], int height, int width)
```

This function expects that all pixel colors are in the range between 0 and 255 (inclusive), if this is not the case, the program will exit with an error.

The `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`.

- **Test that the program works.**
- **View the resulting image to make sure it's indeed the same as the original one.**

ıing: In this lab, **don't** modify functions `readImage` and `writeImage`, they are already working perfectly fine. Yo is to change the `main` function, and possibly define your own new functions to implement the tasks.

is what good program design is about. Each part of your program should have clear logic, purpose, and meaning.

Task A. Invert colors

Write a new program `invert.cpp` that inverts all colors, so white shades become black, and black become white:

Since black = 0, and white = 255, you should do the following transformation for each pixel color:

`0` → `255`
`1` → `254`
`2` → `253`
…
`254` → `1`

`255` → `0`

Example:

Task B. Invert colors in the right half

Write a program `invert-half.cpp` that inverts the colors **only in the right half** of the picture.

Example:

Task C. Add a white box

Write a program `box.cpp` that draws a white box exactly in the middle of the picture. The dimensions of the box should be 50% by 50% of the original picture's width and height.

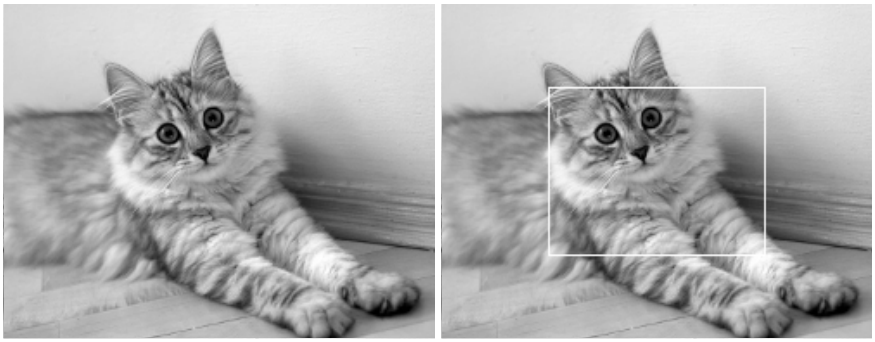
Example:

Since the program should work for all input images that fit into the array, don't "hard-code" the cat picture dimensions into the program, use variables `w` and `h` instead.

Task D. One-pixel-thick frame

Program `frame.cpp`. Same as the previous task, but it should be a frame exactly one pixel thick.

Example:



Task E. Scale 200%

Program `scale.cpp`. Scale the original picture to 200% of its size. It can be done by increasing the size of the picture by the factor of 2, and copying each pixel of the input as a small 2x2 square in the output. (We don't do any interpolation of colors as more advanced scaling procedures would do.)

```
11 22     ->    11 11 22 22
33 44           11 11 22 22
                33 33 44 44
                33 33 44 44
```

Example:

Task F. Pixelate

Program `pixelate.cpp` will be pixelating the input image.

Example:

One way to pixelate an image is to effectively make every 2x2 *non-overlapping* window contain the same value (averaged over all the pixels in that window of the input). For example, the following

image:

```
10 20 30 40
11 21 31 41
12 22 32 42
13 23 33 43
```

should be transformed to:

```
16 16 36 36
16 16 36 36
18 18 37 37
18 18 37 37
```

where the 16 was computed by averaging 10, 20, 11, and 21 (after rounding), and so on.

For simplicity, assume that the width and the height of the image are even numbers, so the picture is divisible into small 2x2 squares, like in the example above.

Task G (Bonus). Kernel method image filtering

A *sliding* window operator replaces each pixel with some function of its 8 neighbors (and itself). Consider pixel `e` and its 8 neighbors (labeled `a` - `i`) that form a 3x3 window around it:

```
. . . . . .
. . . . . .
. a b c . .
. d e f . .
. g h i . .
. . . . . .
```

The operation replaces pixel `e` (in the middle of the 3x3 window) with some function of its neighbors `f(a,b,c,d,e,f,g,h,i)`. It is possible to implement blur, edge detection, and many other image processing operations using this technique.

References:

- Lode's Computer Graphics Tutorial - Image Filtering
- Interactive demo for different functions (kernels)

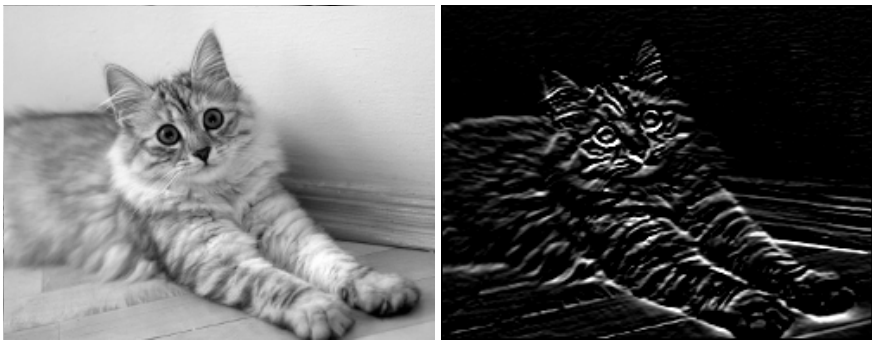
For this task, write a program `kernel.cpp`, which implements a horizontal edge detection

operation. One way to **detect horizontal edges** is to use the function

```
f(a,b,c,d,e,f,g,h,i) = (g+2h+i)-(a+2b+c)
```

(This is one component of the so called Sobel operator, if you want to read more about it.)

Example:



Remark 1: Note that this is a *sliding window* operator unlike the non-overlapping window pixelization operator in the previous task. That is, the considered window is always a window *around the pixel whose value is being computed*.

Remark 2: You shouldn't overwrite the original array. Make a new array for the output, and write the resulting pixel color into the new array.

Remark 3: There are several ways to handle the pixels on the borders, which don't have all 8 neighbors available. Come up with any reasonable way to assign their colors (you can assume that the non-existing neighbors are black, or make the boundary wrap around, or even simply assign black color to the boundary pixels in the output).

Remark 4: If the resulting color is less than 0 or greater than 255, make them 0 and 255 respectively, otherwise `writeImage` function will complain that the colors are out of range.

How to submit your programs

Each program should be submitted through Gradescope

Write separate programs for each part of the assignment.
Submit only the source code (.cpp) files, not the compiled executables.
Each program should start with a comment that contains your name and a short program description, for example:

```
/*
```

```
Author: your name
Course: CSCI-136
Instructor: their name
Assignment: title, e.g., Lab1A

Here, briefly, at least in one or a few sentences
describe what the program does.
*/
```