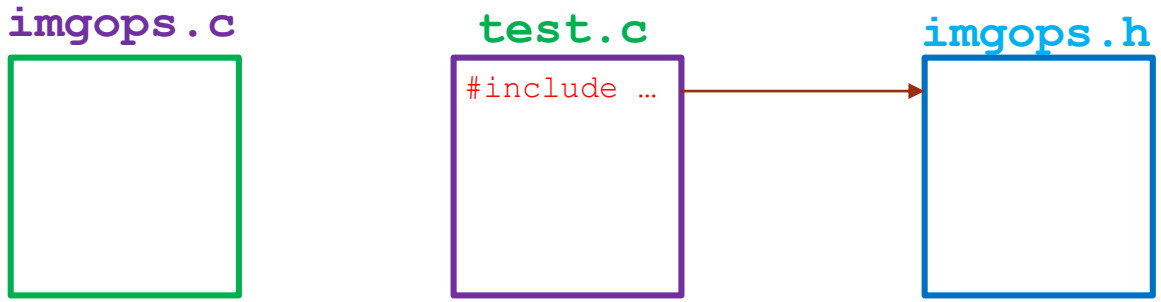
Lab 3

Helpful Tips

Helpful Tips about Lab3

- Read the entire Lab 3 first, especially the Guide section, and do the `get_name(), etc…` exercises before diving into `imgops.c`
- Then start implementing the functions in `imgops.c` and using `test.c` to test them
  - In Lab 2, we used the **multi source file model**: **`funcs.c`**, **`main.c`** and **`funcs.h`**
  - In this Lab 3, we shall use the same model: **`imgops.c`**, **`test.c`** and **`imgops.h`**



  - **`test.c`** contains the **`main`** function!

How to proceed with Lab 3 Tasks!

- Just like Lab 2, download a zip file, unzip it -> directory called **3**
  - `git add` this directory to your Git repo
  - As a consequence, your Git repo will have many files

    -> This is not a problem!!! Do not delete them from your Git repo.

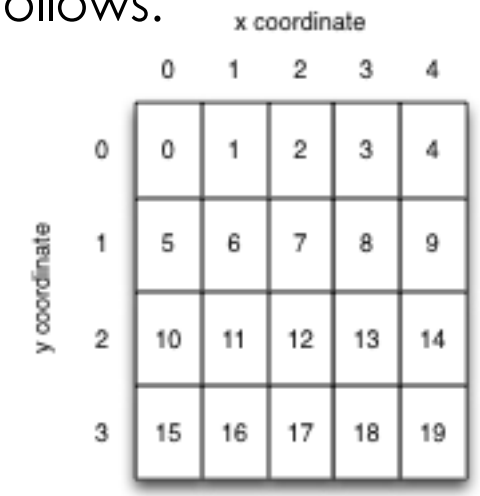
- Unlike Lab 2, all tasks in 1 file -> `imgops.c`
- Task descriptions and requirements are in both files: `imgops.h` and `imgops.c`
- Do one task at a time: compile + test it using `test.c` then submit `imgops.c` to Git repo (you will submit `imgops.c` many times)
- So you'll get 1 successful report (passed) at a time while the other not-yet-implemented tasks will receive unsuccessful report (failed)! -> this is not a problem!!!

Helpful Tips about Lab 3 in general

- When implementing a task, try to make use of as many of the other functions in `imgops.c` as possible
  - This means: calling the functions already implemented from the one you are currently implementing
- Helpful C library functions, i.e., look them up on the Internet:
  - `malloc(…)` and `free(…)`
  - `memcpy(…)` and `memset(…)`
  - `fmin(…)`
- Memory management
  - You must always release the memory you dynamically obtained using system functions such as `malloc(…)`
  - This means: always match a call to `malloc(…)` with a call to `free(…)`

Helpful Tips about Lab 3 in general

- Representation of a raster (or bitmap) image
- For example, an image of width = 5 (columns), height = 4 (rows) would be represented as follows:



- In our program, we represent a raster image as an one-dimensional array of unsigned chars of size (image_width * image_height)
  - In which each row of pixels is stored consecutively in the array
- By convention, image coordinates have the origin in the top left, and y values increase downwards

Helpful Tips about Lab 3 in general

- Helpful code fragments
  - We can map the coordinates `(x,y)` of a pixel in a raster image of `rows` and `cols` to an array `index` as follows:
    - `index = (y * cols) + x`
    - `imageArray[index] = imageArray[(y * cols) + x]`
  - If we want the coordinates `(x,y)` of a pixel in a raster image of `rows` and `cols` and we know its `index` in an array, we an use:
    - `y = index / cols`
    - `x = index % cols`

Helpful Tips about Lab 3 in general

- Always check the validity of the memory address returned by `malloc(…)` as follows:

```
// allocate memory for an array of len int's
int* arrayInt = malloc( len * sizeof(int) );
if ( arrayInt != NULL )    // OR if (!arrayInt)
// Now, I can safely use arrayInt
// otherwise, my call to malloc failed
...
// I am definitely finished with arrayInt so let's free it
free( arrayInt );
// This code will cause a segmentation fault (segfault)
// if I use the pointer again by mistake: (GPS!)
arrayInt = NULL;
```

arrayInt is either a pointer to an array of len integers on the heap OR zero (i.e., a null pointer) if the allocation failed!

Helpful Tips about Lab 3 in general

- Always check the validity of function parameters

```
/*-----------------------------------------------
  PART 0: OPERATIONS ON A PIXEL
*/

// get the value in the array at coordinate (x,y)
uint8_t get_pixel( const uint8_t array[],
           unsigned int cols,
           unsigned int rows,
           unsigned int x,
           unsigned int y )
{
    assert( x < cols );
    assert( y < rows );
    return array[ y*cols + x ];
}
```

- Can also use conditional statements:

```
if ( ( x < cols ) && ( y < rows ) )
    // Then it is safe to use the parameters x and y
```

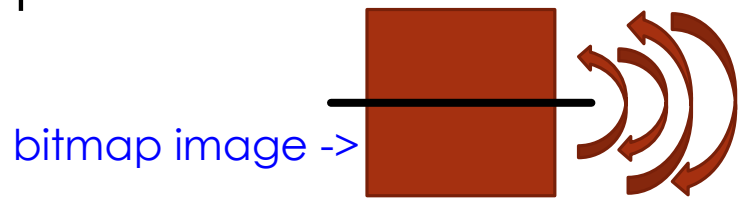
Helpful tips for Task 1

- Can you implement the function `zero(…)` of Task 1 without using a loop?

Helpful tips for Task 3

- Flip horizontal:

Flip vertical:



- Challenge:
  - Try to swap without using extra memory (i.e. without using a third variable called **`temp`**)
  - How? Here is how: search for "Bit twiddling Hacks" (https://graphics.stanford.edu/~seander/bithacks.html)
    - Look for "Swapping values"

Helpful tips for Task 6

- C functions and/or bit of code you may find useful:
  - `scale_brightness(…)`
  - `round( … * scale_factor)`
  - `fmin(…)`

see `imgops.h`

Helpful tips for Task 7 – `normalize(...)`

- Our goal is to normalize all pixel values in `image`, i.e., to adjust all pixel values from their current range to a normalized range
  - for example, pixel values ranging from [10..178] are adjusted (normalized) to the range [0..255] <- normalized range

Step 1. Shift: If min pixel value in `image` is 10 and max is 178, adjust the min so it becomes 0, then adjust max the same way:

Step 2. Scale: Scale all these shifted values (in [0..168] range) so that they are now in the normalized range [0..255]. This is done by multiplying each of them by a factor. For example:

0 x factor = 0

168 x factor = 255

You need to find this factor!

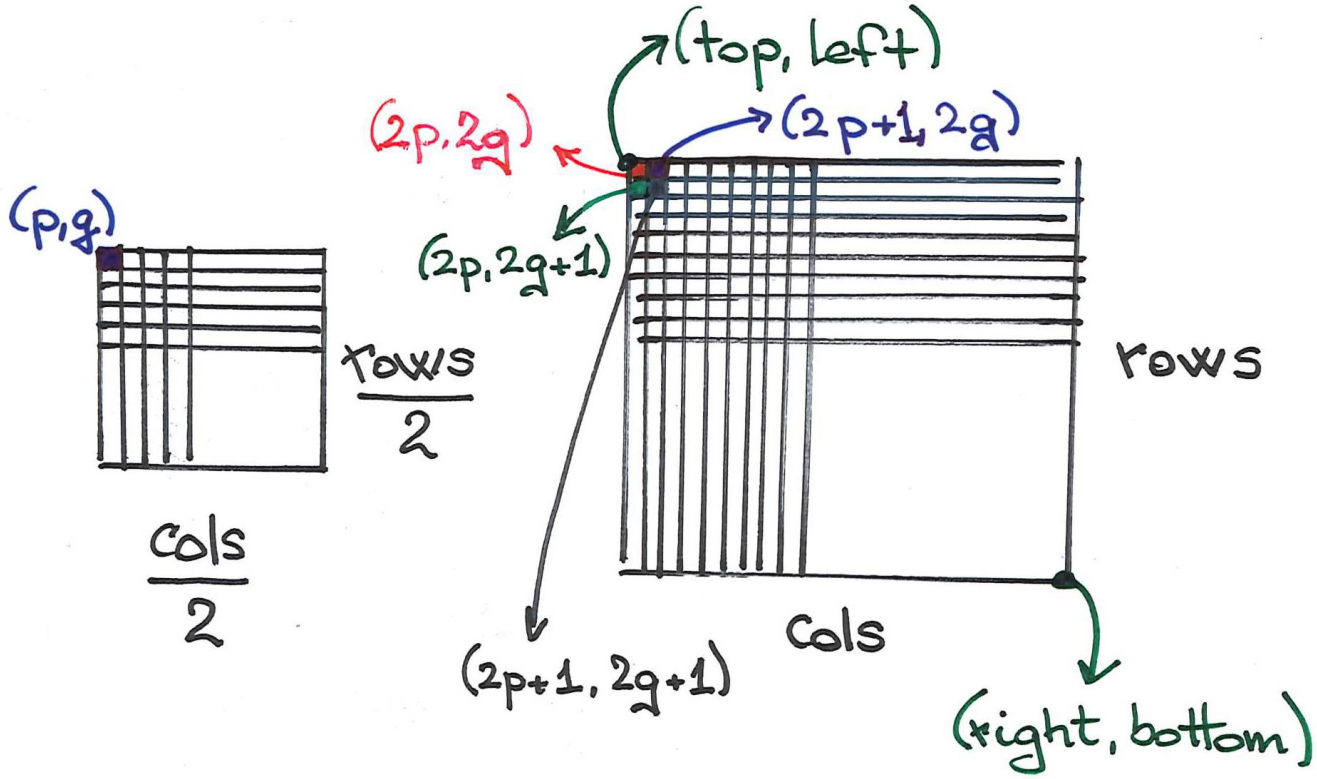
Helpful tips for Task 7 – `normalize(...)`

- When computing factor (`scalingFactor` – a variable of data type `double`), you may wish to initialise it to `255.0` as opposed to `255`!

  -> Why?

  See the difference between `255.0` and `255`?

Helpful tips for Task 8 – `half(...)`



Lab 3 - Task 8

To help you along with Lab 3, we are offering you the solution to Task 8.
You can use it as you wish:

- To verify your understanding of Task 8 and of image manipulation
- To verify your own solution to Task 8
- To submit as your solution to Task 8

Make sure you understand what the code does as it may be part of our Lab Quizzes.

Enjoy!

```
/* TASK 8 */

// Return a new image of size rows/2 by cols/2 If the original image
// has an odd number of columns, ignore its rightmost column. If the
// original image has an odd number of rows, ignore its bottom row.
// The value of a pixel at (p,q) in the new image is the average of
// the four pixels at (2p,2q), (2p+1,2q), (2p+1,2q+1), (2p,2q+1) in
// the original image.
uint8_t* half( const uint8_t array[],
               unsigned int cols,
               unsigned int rows )
{
  //  return array;

  // allocate an image half the original size.
  // note that integer division rounds by truncation towards zero,
  // e.g. 7/2 = 3
  uint8_t *ret = malloc((rows/2)*(cols/2)*sizeof(uint8_t));

  if (ret != NULL)
  {
    // for all pixels in the new, smaller image
    for (unsigned int y = 0; y < rows/2; y++)
      for (unsigned int x = 0; x < cols/2; x++)
      {
        // sum the values of the four pixels in the original image
        // that correspond to this pixel in the new image
        unsigned int total = 0;
        for (unsigned int i = 2*y; i < 2*y+2; i++)
          for (unsigned int j = 2*x; j < 2*x+2; j++)
            total += array[i*cols + j];

        // set the new image pixel to the average color
        // ret[y*(cols/2) + x] = (total + 2)/4;
        ret[y*(cols/2) + x] = round(total/4.0);
      }
  }

  return ret;
}
```