Lab 5

First, a reminder!

Pushing tested code to your Git repo

- IMPORTANT:
  - You must compile and test your code before pushing it to your Git repo
  - Why is this important?

Lab 5

Incremental Development

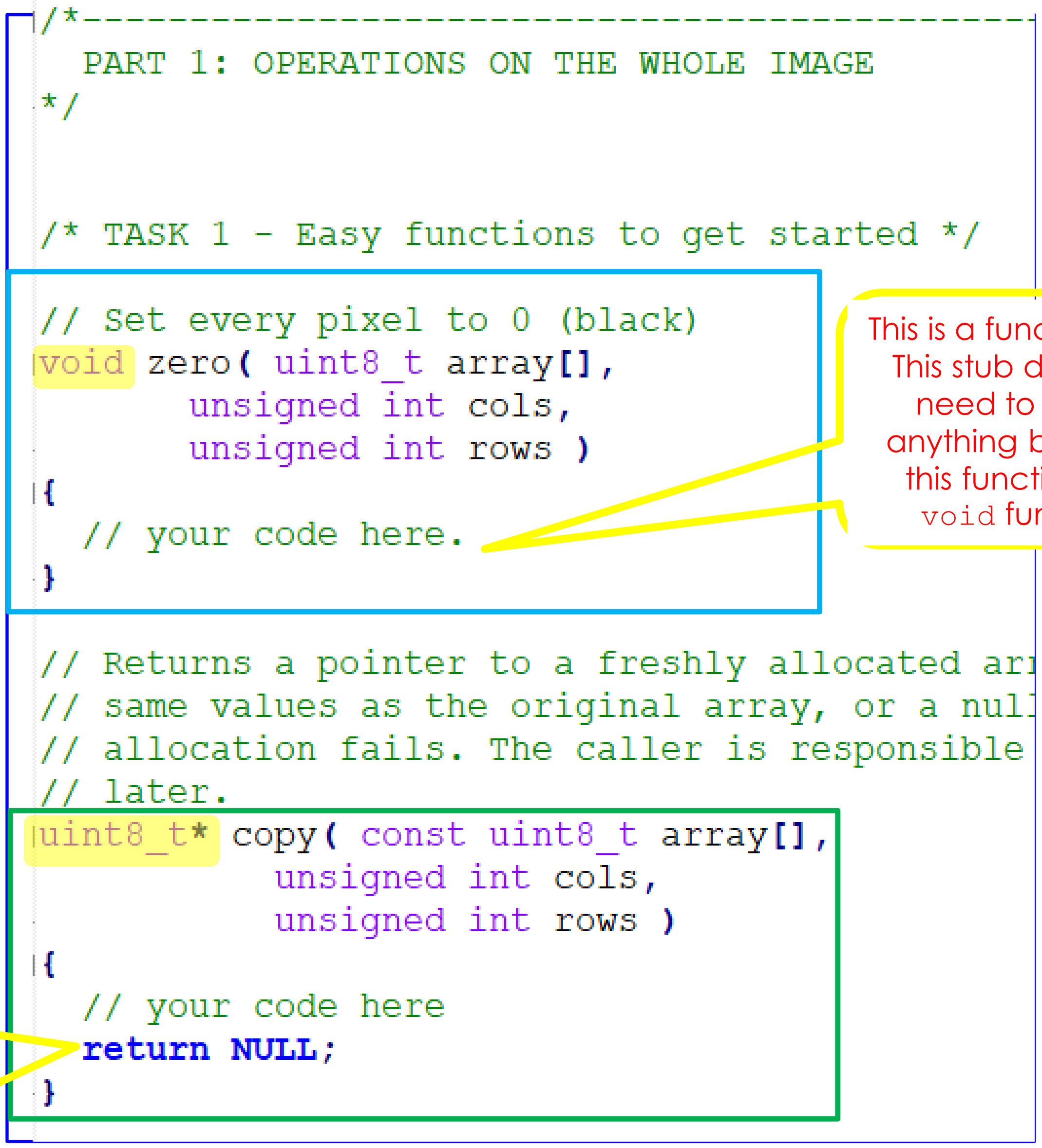
Incremental Development

- Idea: develop your program incrementally, a “chunk” at a time
- Why? So that, if the “chunk” is faulty, you know where to look for the bug(s) -> the “chunk”
- “chunk” can be:
  - Function(s)
  - Class
  - Feature
  - Etc…
- Process:
  - Once you have designed (algorithm) and implemented (code) the “chunk”, you compile it then test it using a test driver containing main( )
  - Only once the chunk works (not only compiles but actually “solves the problem”) one can move on to the next “chunk”

More info: https://en.wikipedia.org/wiki/Method_stub

Example: Incremental Development in Lab 3!

- Lab 3 (`imgops.c`) is well set up for incremental development
- Why?
  - `imgops.c` already has function stubs
  - These function stubs allow `imgops.c` to compile even if we have not added our code yet
  - So, we can design, implement, compile and test each function (or each task) one at a time
  - Grading robot grades one (or a few) function(s) at a time as well => 1 or a few functions -> 1 task!



```
/*---------------------------------------------------
  PART 1: OPERATIONS ON THE WHOLE IMAGE
*/


/* TASK 1 - Easy functions to get started */

// Set every pixel to 0 (black)
void zero( uint8_t array[],
       unsigned int cols,
       unsigned int rows )
{
  // your code here.
}

// Returns a pointer to a freshly allocated ar
// same values as the original array, or a nul
// allocation fails. The caller is responsible
// later.
uint8_t* copy( const uint8_t array[],
           unsigned int cols,
           unsigned int rows )
{
  // your code here
  return NULL;
}
```

This is a function **stub**. This stub does not need to return anything because this function is a `void` function.

This is also a function **stub**. However, this stub **does** need to return something to satisfy the function declaration.

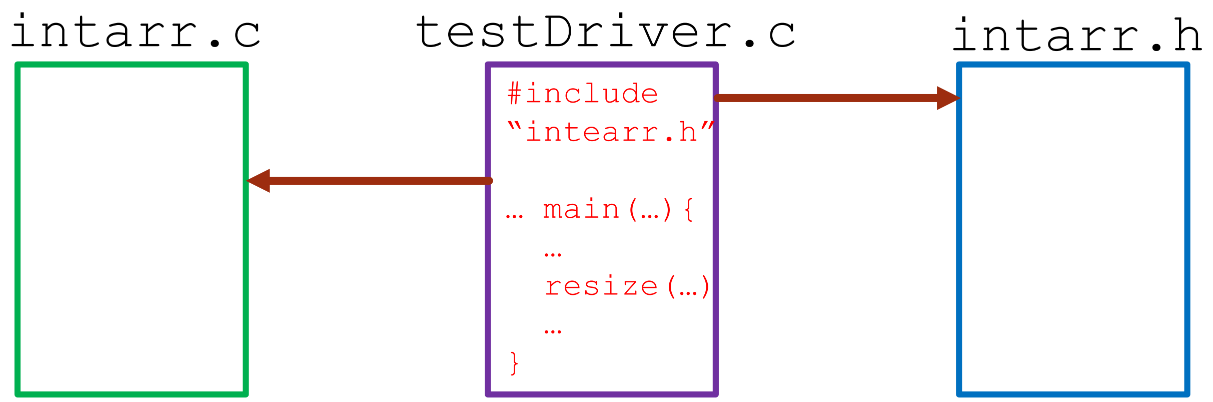
Incrementally Developing Lab 5

- Lab 5 (`intarr.h`) needs to be set up for incremental development
- How?
  - By first creating `intarr.c` (copy `intarr.h` to `intarr.c` and remove the `struct`'s, put some header files, …)
  - By initially implementing each function as function stubs
    - This allows `intarr.c to` compile without our code
  - Then, we can design, implement, compile and test each function one at a time

Lab 5

Helpful Tips

Using the "Multi Source File" model



.c -> C code

This file contains definition (body) of some functions.

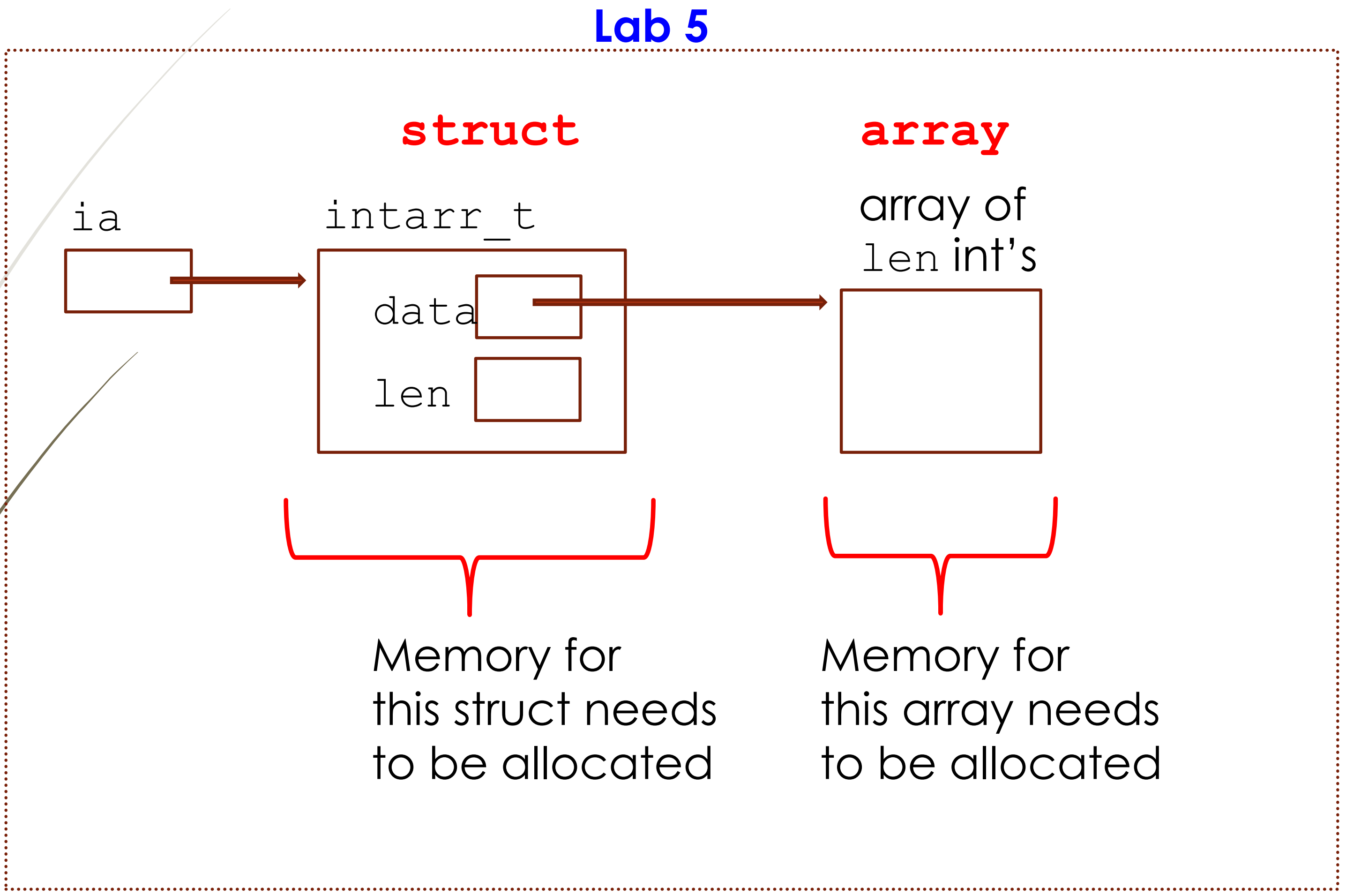
.c -> C code

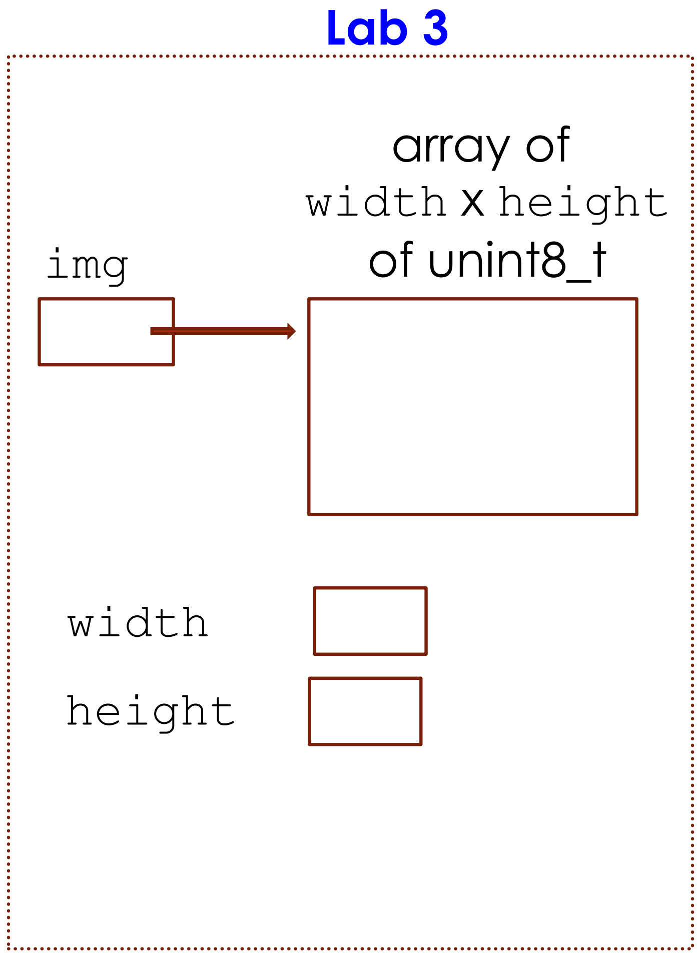
This file contains the main function and possibly others functions (optional). It may call functions defined in other files.

.h -> header

This file contains function headers (also called function declarations, or function prototypes).

Introducing `struct`





NOTE: `len` is the size of the array `data` and the number of elements in `data` so it means two different things!

Helpful Tips about Lab 5

- Always validate the parameters to functions
- Call functions already implemented (either yours in `intarr.c` or C Library functions)
- Useful functions:
  - `malloc( )   +   free( )`
  - `memcpy( )`
  - `realloc( )` (may be useful in `resize( )`)
- Do not forget to modify `len` after a successful call to `realloc( )`
- **`free( aPtr );`** should be followed by **`aPtr = NULL;`**

Helpful Tips about Lab 5

You may want to investigate …

- the function `assert( )`
  - How it works
  - What it returns
- `enum`

Task 6 and Task 7

- Do Task 7 before Task 6
- In Task 6
  - In `intarr_push(…)` -> it makes total sense to call `intarr_resize(…)`
  - In `intarr_pop(…)` -> Careful: do we really have to call `intarr_resize(…)?`
    - Why is calling `intarr_resize(…)` problematic? What happens when you are pop'ing the last element?
      - Calling `realloc` using size 0 -> problematic -> unpredictable
      - Check it out: https://en.cppreference.com/w/c/memory/realloc
  - Solutions:
    1. In `intarr_pop`(…), do not readjust the memory allocated for the array, simply readjust `len`
    2. You may add an "`if`" statement in `intarr_resize(…)` to avoid calling `realloc` when `len` is 0