CP Lab 5: Functions, pointers, some arrays

Instructions

The purpose of this Lab is to help you get experience in understanding parameter-passing functions in C, and also understanding how arrays and pointers are related. You already have experience of *using* functions from your work with the `descartes` library in Lab 4. In this lab we will explore how to design functions, as well as to use them.

All Labs for CP will take place on the DICE machines in AT5.05 (Thursday and Tuesday) or AT6.06 (Monday). For working alone, you may use any of the DICE labs in AT, as long as they are not booked out by a class. You have 24-hour access to AT levels 3,4,5: if your card/pin does not allow you in, use the ITO contact form on our course webpage to contact the ITO.

Aims

- We will experiment with the details of the `rotate.c` function from Lecture 9.
- Explore the role of pointers as parameters to functions. In particular we will do some experiments with different versions (correct, and incorrect) of `swap` functions.
- We will show you how to print out the memory address of a particular variable using `printf` with the `%p` indicator.
- We will extend the concept of 'program planning' to take in the design of functions.
- We will learn how arrays work when being used as *parameters* to a function.
- We will learn the difference between `arrays` (static pointers) and standard (dynamic) pointers.

Prerequisites

You should have attended your scheduled Lab sessions in weeks 2–5, and completed the Lab sheets during that session. You should be up to date with the material from the CP lectures.

NOTE: If you are still behind with lab 4 (Descartes), then please do Stage A *only* of this lab, then return to working on lab 4. Come back to stages B and C of this lab later.

If you are fully caught up, then do this lab in order.

Getting started

- Log into on of the DICE terminals, bring up a Terminal window, move to your home directory, and create a new subdirectory called `lab5`. You may want to copy the `template.c` file from your `lab2` subdirectory into the `lab5` directory. [1]

[1] If you don't know how to do this, look back at the Lab sheet for Lab 3.

Stage A Arrays as parameters (rotate)

Towards the end of Lecture 9, we discussed how functions can take arrays as parameters, by adding a declaration like `int a[]` (or `double reals[]`, etc) in the header of the function prototype. We discussed how we must *include an* `int` *parameter* for each array, to pass the length of the array (C passes an array into a function as a *pointer*, so the function does not automatically know the length of the array).

On slide 15, we gave code for a function called `Rotate`, whose input parameters are an int array and also an int (interpreted as being the length of the array). The function performs a single clockwise rotation of the elements of the array. In this section of the lab we will experiment with `rotate.c` to understand why the result of the rotation can be seen in the original array in `main` after the function terminates.

The short answer is that `Rotate` is called with an address (the starting address of the array), and therefore `Rotate` will then use this address to access and manipulate the cells of the array starting there (stored in `main`'s environment). We will now create a program `rotate.c` to illuminate this explanation.

- Copy `template.c` to make a new file called `rotate.c` and then add the code for the `Rotate` function (slide 15 of lecture 9) as a function above `main`.
- Please also add a `printf` inside the `Rotate` function, to print the start address of the `b` array, and indicate this output is coming from inside `Rotate`. Use `%p` to output an address with `printf`.
- At the start of `main`, define a local `int` array of length `10` named `primes`, and initialise it to store the first 10 prime numbers (see lecture 9 for how to do this).
- Next, add a `printf` to `main` which prints the start address of the `primes` array, and indicates this output is coming from `main`. Again, use `%p` to output an address with `printf`.
- After this, add the following call in `main`:
  `Rotate(primes, 10);`
- After this, use `printf` (probably in a loop) to output the sequence of entries in `primes`.
- After this, add the following call in `main`:
  `Rotate(primes, 5);`
  and then again output the sequence of entries in `primes`.

Now compile your code as follows:

`gcc -Wall rotate.c` ↲

After DEBUGGING is finished, run your program.

Your working with `rotate.c`

☐ You should be able to observe the following behaviour from running your program:

- The address printed from within `Rotate` should be the same as that printed from `main`.
- After the call `Rotate(primes, 10)`, the array `primes` should store the following sequence:
  `29, 2, 3, 5, 7, 11, 13, 17, 19, 23`
  After the extra call `Rotate(primes, 5)`, now `primes` should store the following sequence:
  `7, 29, 2, 3, 5, 11, 13, 17, 19, 23`

☐ Do you fully understand why the program gave these results? If not, ask the Demonstrator or Lecturer! Now submit your program with `submit cp lab5 rotate.c`

The remainder of this lab takes you through a series of experiments to illustrate pointers and arrays in more detail, and give you a fuller understanding. If you are not confident with the course material so far, you should skip the rest of this lab, continue catching up with the previous labs, and come back to this point later if you have time.

Stage B Swap

Can we write a *function* that takes two variables and exchanges their contents? This function does **not** work:

```
void swapwrong(int a, int b) {
  int tmp;
  tmp = a;
  a = b;
  b = tmp;
}

int i = 2, j = 7;
swapwrong(i,j); // does not swap i and j
```

To change variables that don't 'belong to' to a function, we have to use pointers, like this:

```
void swap(int *aptr, int *bptr) {
  int tmp;
  tmp = *aptr;
  *aptr = *bptr;
  *bptr = tmp;
}

int i = 2, j = 7;
swap(&i,&j); // does swap i and j
```

Now in this section of the Lab, we'll explore *why* the first does not work, and *why* the second does.

Step 1

In this part of the Lab, we are going to code up both of the 'swap' functions in the same program for testing. Your program should be named `swap.c` – you can start it off by making a copy of `template.c`.

The `swap.c` program will contain the two swap functions together with a `main` which will declare two *local variables* (local to `main`) `i` and `j` with values 1 and 2 respectively; and which will then ask the user which 'swap' function to use; and then output the values of `i` and `j` after calling that function.

This will allow us to *check* which version of 'swap' worked and which does not. In Step 2, we will add some extra `printf` statements to *understand* why one 'swap' works and the other does not.

We are going to define a program which will have the following features:

- We will declare and implement the non-pointer `swapwrong`.
- We will declare and implement the pointer `swap`.
- Next we must write the initial code for `main`

- The first thing to appear in `main` will be the declaration of two local int variables `i` and `j`, initialised to 1 and 2 respectively.
- Next `main` will use a `printf` to ask the user if they want to use `swapwrong` or `swap`, followed by a `scanf` to read in their response (0 meaning `swapwrong`, 1 meaning `swap`, any other `int` value causing an error message).

  Notice that to do this, we will need an extra variable to read in the option chosen by the user - so we will need to go back and define another local `int` variable - maybe called `option`, say ....

  Also remember we need to use `&x` when using `scanf` with the variable `x`.
- We will use an `if`-statement to *either*
  - Call `swapwrong` with the *values of* `i` and `j`, *or*
  - Call `swap` with the *addresses of* `i` and `j`

    Note that since `swap` takes addresses as its parameters, we must use `&i` and `&j` in the call.
- Finally, we will use a `printf` to output the new values of `i` and of `j` in that order. This will allow the user to test whether the swap succeeded or not.

I have drawn up the 'program plan' for `swap.c`, and it appears over on the next page. Please use this as the basis to code up this first version of `swap.c`

After you have written the program, please compile it using

```
gcc -Wall swap.c
```

Then run it (by typing `./a.out` in your Terminal window), first asking it to use `swapwrong`, and on the next run, to use `swap`. Verify that no swap happens when `swapwrong` is used, but we do get a swap using `swap`.

Step 2

Now in Step 2, we will *understand* why `swap` achieves the swap but `swapwrong` does not. The reason is due to the mechanism of *call by value* which is the paradigm always used by C in substituting for parameters to functions.

For `swapwrong` the following is the situation:

- The declaration (or function prototype) of `swapwrong` asks for two input parameters of type `int`, locally named `a` and `b`
- Therefore when we are *using* `swapwrong` in `main` we must pass it two integers - specifically for us, these will be the values of `i` and `j`.
- The *call by value* mechanism copies the *values* of `i` and `j` and puts these into the 'boxes' for local variables `a` and `b`, and then the *body* of `swapwrong` executes on these local variables.

For `swap` the situation is different:

- The declaration (or function prototype) of `swap` declares two input parameters which are the *memory addresses* of two `int` variables. These input parameters are locally named `a` and `b`. If we want to examine the *values* stored at the address `a`, we will have to write `*a` (for example).

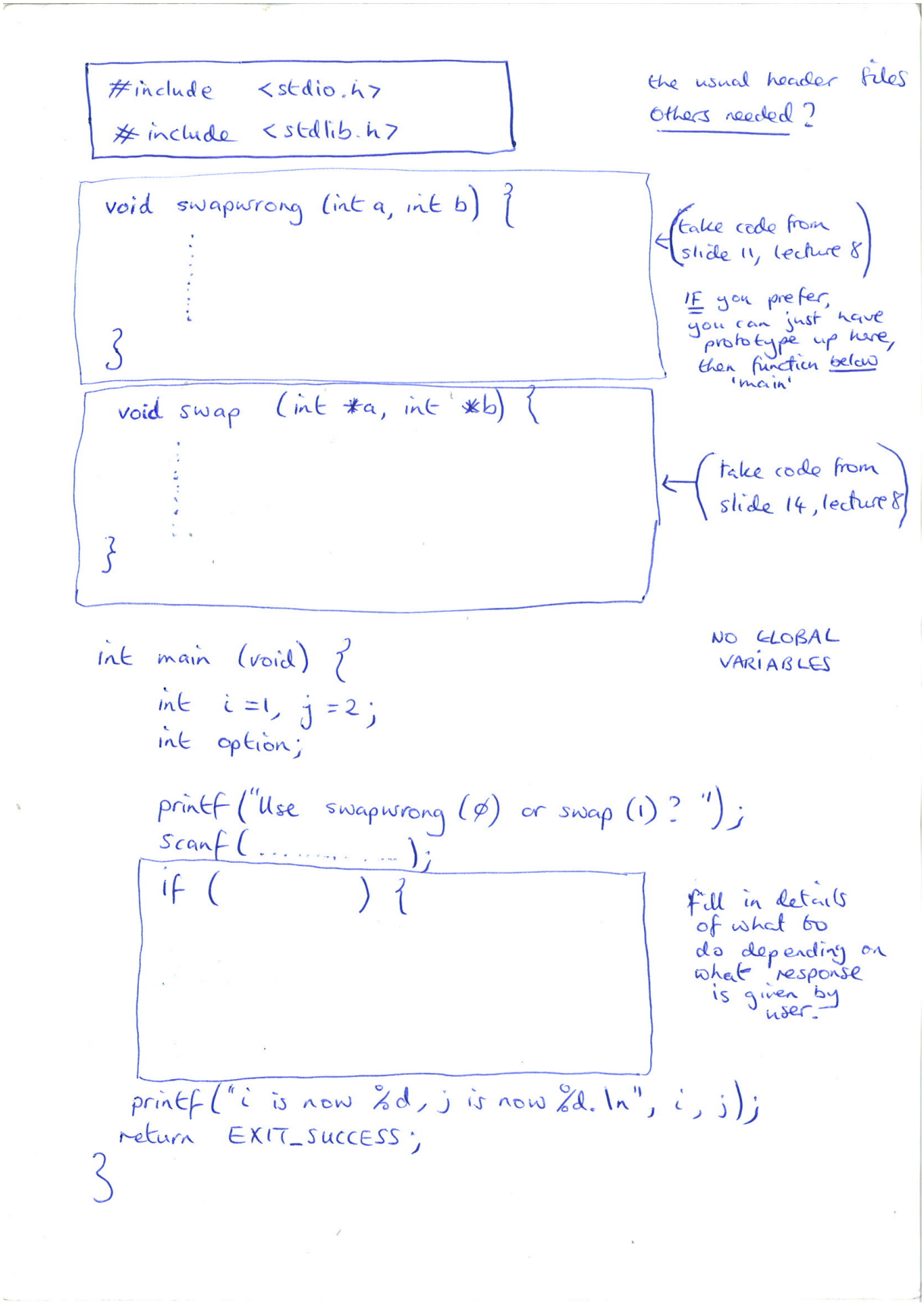

Figure 1: 'Program plan' for `swap.c` for you to complete.

- When we are *using* `swap` in `main` we must pass it *two memory addresses where integers are stored* - specifically for us, this will be `&i` and `&j`.
- The *call by value* mechanism is still used for `swap`, but the values to be copied are the *values* of the address `&i` and the address `&j`. These *address values* get copied into the 'boxes' for the local variables `a` and `b` of `swap`. The *body* of `swap` will execute wrt these 'address values'.

  Because `swap` has been passed the addresses of the original variables of interest from `main`, the swapping gets done on these original locations.

We now show how to add `printfs` to `main`, to `swapwrong` and to `swap` to show us which memory addresses are being worked on.

Variables of `main`

We will need to know the memory locations of the original variables `i` and `j` in main, so that we can check whether `swapwrong` and `swap` work with respect to these locations or not. Therefore in `main`, just underneath where we have our variable declarations, please add the following line of code[2]:

```
printf("i and j are stored at addresses %p and %p.\n\n", &i, &j);
```

Variables of `swapwrong`

To illuminate what is going on with `swapwrong`, we will get it to tell the user the memory locations of the values that it swaps. Add the following lines of code at the start of the body of `swapwrong`.

```
printf("\nWe are in swapwrong.\n");
printf("We are swapping the values at addresses %p and %p.\n\n", &a, &b);
```

The value of adding those lines is that before `swapwrong` does its work, it will tell the user which *memory locations* are being worked on.

Variables of `swap`

We also want to understand why `swap` does work. We know that it has been passed addresses of memory locations, not values. Therefore we will print out both the addresses and also the values stored there. Add the following code at the start of the body of `swap`.

```
printf("\nWe are in swap.\n");
printf("We are swapping the values at addresses %p and %p.\n\n", a, b);
```

Comparing the two swaps

Now recompile your code after having added these outputs:

```
gcc -Wall swap.c ↲
```

There may be some DEBUGGING to be done.

[2] We use `%p` to indicate an address is to be passed for output. The substituted value must either be a pointer variable (eg, `x`, if `x` was declared as a pointer variable) or alternatively must be the address `&x` of a variable.

Next, do two runs of the code (as usual, do this by typing `./a.out` at the command line)

In each run, you will see the information about `i` and `j`s addresses; then the query about which 'swap' to use.

First ask for `swapwrong` to be used - that means that will be given the result of the three `printf`s you added to `swapwrong`.

- Please *carefully examine* the memory addresses, and compare them to those initially given for `i` and `j`.
- Do they match?
- Does this explain why `swapwrong` does not work?

On the second execution, ask for `swap` - this time you will see the result of the three `printf`s you added to `swap`.

- Please *carefully examine* the memory addresses, and compare them to those initially given for `i` and `j`.
- Do they match?
- Does this explain why `swap` does work?

Your result for `swap.c`

- ☐ From Step 1, you should see that no swap happens if `swapwrong` is used; but a swap does happen if `swap` is used.
- ☐ In Step 2, when working with `swapwrong`, you should notice that the addresses printed out by `swapwrong` are not the addresses of `i` and `j`.
- ☐ In Step 2, when working with `swap`, you should notice that the addresses printed out by `swap` are the same as the addresses of `i` and `j`.
- ☐ Are you happy you understand the reason that `swap` causes `i` and `j`s values to be exchanged back in `main`. If *not*, ask the demonstrator or lecturer to explain.
- ☐ Can you now understand the reason `scanf` requires you to pass the address of the variable where the read-in value will be stored, even though `printf` just wants a variable name?
- ☐ When you are finished, rename `a.out` to `swap` or similar for later use.
- ☐ Submit your program with `submit cp lab5 swap.c`

Stage C Arrays and pointers

This first task involves writing a short simple program to illustrate the difference between *pointers* and *arrays*. As mentioned in the 'Arrays' lecture, it is true that 'arrays are pointers', however there are subtle differences.

You will need to start a new program - called `array.c` or similar - you may want to use your `template.c` file to start off. Within your `main`, please start off by making just *two* declarations:

```
int *p;
int r[10];
```

What do these declarations achieve?

(a) `int *p;` will declare one 'box' (variable) in memory which has enough space to store a memory address (memory address of an `int` variable):

  (i) The initial value which sits in `p` will be just random junk.

  (ii) There is no guarantee that the `0xff...` junk in `p` at the start, actually corresponds to any actual usable address in memory.

  (iii) The address stored in `p` can change: we can initialise it, reassign different addresses later, etc.

(b) `int r[10];` will declare 10 'boxes' in sequential order in memory, each with space to store one int, and will set the *static pointer* `r` to have the address of the *first* (`r[0]`) of these 10 boxes.

  (I) The address that `r` has just after this declaration *does* correspond to a true address (with space for one `int`) in memory.

  (II) `r` corresponds to a real specific address where 10 `int` variables start. It will be associated with this specific address for all its life (`r` is a *STATIC pointer*)

One thing before we start investigating - after the declarations, please add the following assignment statement:

```
r[0] = r[0];
```

This has no effect on values stored; we are only adding it to avoid getting a Warning about the array `r` being set but not used.

1: Demonstration of (a):(i)-(ii) and (b):(I)

To demonstrate (a)(i) and (a)(ii), add the following lines to appear after the two declarations of `array.c` (the second line is only there so that `gcc` will not complain about `r` being an unused variable):

```
*p = 6;
r[6]=4;
```

- Now save this program, and return to the command line, and compile with `gcc`. You will probably see the following warning:

  ```
  array.c:7:  warning:  'p' is used uninitialized in this function
  ```

- Since the above is only a warning, maybe try running the code by typing `./a.out`.

  You will probably get a *segmentation fault* - A segmentation fault means trying to access memory which it is not allowed to, or which doesn't exist.

There is nothing wrong with the *form* (or *syntax*) of `*p =6;`. This is appropriate for any *actual* memory address representing a variable. Problem is that `p` is uninitialized and the address it has is just junk.

- Notice the complaint is *only* for `p`. There were no problems with `r[6]=4;`, because space for the variable `r[6]` (and indeed all of `r[0], .., r[9]` had been allocated with that declaration), and could be accessed in terms of `r`.

TO FIX THIS PROBLEM, add the following declaration and assignment, just *AFTER* your initial two declarations and *ABOVE* the assignment statement `*p = 6`:

```
....
int a;
p = &a;
*p = 6;
```

Now compile and run the program a second time. You will have no difficulties. This is because *before* we tried to do `*p` (to look at what is stored at address `p`) we gave it *the address of* a *existing* `int` variable, specifically, the address of `a`. So instead of having a 'junk address' as its value, `p` now is pointing at a true memory address that we have access to.

Experiment with `printf`:

- You can print out memory addresses with `printf`. In the special case of the zero address, which in C is always 'junk', and used to mean 'invalid pointer', `printf` will print it as `(nil)`. To see this, add the following command to your code:

  ```
  printf("p points to the address %p\n", p);
  ```

  First add it immediately below the `int *p;` declaration; and the second time, to be after the assignment `p=&a;`

2: Demonstration of (a)(iii) and (b)(II)

Now we are going to test assignment of addresses between typical pointers, and arrays. Go to the end of your program, just above `return EXIT_SUCCESS;`. Add the following line of code:

```
...
r = p;        /* Goal: give pointer r the address of pointer p */
return EXIT_SUCCESS;
```

Compile with `gcc`, and this time you get an *error*, not just a warning:

```
array.c:10: error: incompatible types when assigning to type ‘int[10]’ from type ‘int *’
```

The problem is that *arrays are static pointers, and they are permanently tied to their initial storage allocation*. You cannot change them to point to a different address in memory, not even to the address of another array (not even to the address of another array of the same length and type).

Now *delete* the most recent command and change it to:

```
...
p=r;        /* Goal: give pointer p the address of pointer r */
return EXIT_SUCCESS;
```

- Compile with `gcc`

  You should see everything is fine.

- Add the following commands

  ```
  printf("r points to the address %p\n", r);
  printf("p points to the address %p\n", p);
  ```

  in two places: just before the assignment `p=r;`, and just after it.

  Compile again and examine the output.

The lesson you can take away is that standard pointers (declared as `int *a` or `float *f`, or whatever type) are re-assignable with memory addresses of any variable of that type. Array pointers are *static* and cannot have their value changed - though *the value of* what they *point to* can be changed.

3: Array indexing and pointers

We all know by now certain relationships between arrays and pointers:

- `r`, the address of the array, is the same as `&r[0]`
- `r[3]` (say) is equivalent to writing `*(r+3)`.

  In this expression '`r+3`' means '3 boxes up' from address `r`.

  Then the `*` means go '3 boxes up' from address `r` and return the (int) value there.

  `r[3]` is just shorthand for `*(r+3)`.
- We know that for a standard (non-array) pointer `p` we can assign `p=r;` to give `p` the address of an array.

Some more interesting things:

- If we want to, we can assign `p` to point to an address in the middle of an array, for example:

  `p = (r+3);`

  Or equivalently, `p = &r[3];`
- After having done this, we can use `p[0]` and `p[1]` ... `p[6]` to access successive locations after `p[0]` (which is `r[3]`).
- Because `gcc` does not do out-of-bounds checking for arrays, we can also do `p[7]`, ..., `p[9]` (these are not part of `r`, as `p` started 3 boxes up), and even `p[10]`, `p[11]` etc.

  Try it! Print out the values at these addresses and see what junk you get (maybe initialise all values of `r` first to see the difference).
- In fact C even lets you use `p[1]` etc *even if the pointer variable* `p` *was only set to a standard non-array address.*

  Try this out by adding the following commands at the end of your program:

  ```
  ...

  p = &a;
  printf("Look two-boxes-up from p: address %p, value %d.\n", &p[2], p[2]);
  return EXIT_SUCCESS;
  ```

Lessons to take away are that *C is very flexible with array indexing, even allowing it to be used with pointers which have never been related to arrays* and (unfortunately) *C does not check array-bounds.*

Your working with `array.c`

- ☐ From part 1, should understand that `int *p;` pointers come initialised to junk values, but that array pointers correspond to real accessible addresses in memory.
- ☐ From part 2, should understand that an array, despite being a pointer, is a *static pointer* tied to a particular location in memory.
- ☐ From part 3, should understand the interesting relationship between arrays, pointers and `*`, and all that can be done with them.

  Also should understand 'pointer arithmetic' (`p+3`)
- ☐ Submit your program with `submit cp lab5 array.c`

Julian Bradfield after *Mary Cryan, October 2016*