Reading Material
Lab Exercise 3: Pointers and arrays

Pointers

Here is a short video on introduction to pointers. It's about 10 minutes long and gives a good overview of pointers in relation to memory layout etc.

A pointer is a special kind of variable that contains the address of a memory location. It can be helpful to think of it as a box in which we write the address of another variable i.e. this address implicitly *points* to somewhere else. That is to say, it *references* a location in memory, and accessing the value at that location is called *dereferencing* the pointer. As with all variables, we can change what we store in a pointer to point to another place in memory.

There are two very important (and sometimes confusing) operators associated with pointers.

***** Has two distinct uses:

1. When an asterisk is used in a **declaration**, it declares the variable to be a pointer. For example, in the declaration of a_ptr in Figure 1 (line 3).
2. When an asterisk is used on the right-hand side of an assignment it **dereferences** a variable that is already a pointer, enabling you to access the variable to which the pointer is pointing. For example, when we ask for the value stored at the memory location pointed to by a_ptr in Figure 1 (line 13).

& is used to access the address of a variable. For example, when we ask for the address of a and store it in a_ptr in Figure 1 (line 6).

struct

Structures are the collections of one or more variables, possibly of different types. The main application is in organising complicated data where all related variables can be put under one name to form a unit. They can look like classes in Java but they **are not the same** for various reasons including:

- They do not have member functions
- They can be allocated on the stack and passed by value
- There is no notion of inheritance

```c
int a;
//this declares a_ptr as a pointer that points to an object of
    type int
int* a_ptr;
a=100;
// let a_ptr take the address of the variable a as its value
a_ptr=&a;
// Only now does a_ptr become a pointer to the particular variable
     a -- before this, it was merely a pointer that could point to
     any integer variable. More importantly, before assigning it a
     value it will be pointing to a random bit of memory.

// print out the location pointed to by a_ptr, which would be the
    address of variable a
printf("%d \n", a_ptr);

// print out the contents of the location pointed to by a_ptr,
    which is the value of variable a
printf("%d \n", *a_ptr);
```

Figure 1: Example of declaring and initializing pointers

Interestingly, we get few guarantees about where things are placed in memory when they are created but we are guaranteed that structures are allocated contiguously. As a thought experiment - how will the compiler achieve this in memory whilst handling the *alignment issue*? This is a related interesting read but well out-of-scope for this course.

malloc

Basic data types in C have a fixed size. You can find it out by using the `sizeof` operator as seen used in Figure 2. Note that the sizes of these datatypes is not necessarily fixed – it can vary by machine.

However, sometimes we want to define our own data structure, or to have variables whose size can vary (e.g. read from input). This is where we need memory allocation to assign a block of memory to a variable, in order to create dynamic data.

The memory allocation is performed by the function `malloc` (which comes from "memory allocation"). The argument of `malloc` is an integer argument which is the number of *bytes* of memory that we want. The memory is allocated from the heap and a pointer to the beginning of that block of memory is returned.

If there is insufficient storage in the heap, `malloc` returns NULL.

The pointer returned when `malloc` is successful is 'untyped' (it is a pointer of type `void*`). It needs to be casted to a particular type before its use.

```c
char *new_memory;
int memory_size = 20 * sizeof(char);
new_memory = (char *)malloc(memory_size);
if(new_memory == NULL)
  printf("No memory allocated!\n");
else
  printf("Memory for 20 characters allocated\n"');
```

Figure 2: Example of using malloc

Figure 2 gives a typical usage of `malloc` where a pointer is declared, some memory is allocated (big enough for 20 characters) and this pointer is made to point to this memory. Importantly, this code checks whether the returned memory is NULL. Not doing this can lead to lots of problems.

free

Good programming practice is to free the allocated memory when we no longer need it (C does not have automatic garbage collection as we have in Java!). Not doing so leads to memory leaks, which can mean that you run out of memory.

Memory is freed using the `free` function. The argument of `free` is a pointer to the beginning of a block that you have been allocated using the `malloc` function. This means that you need to remember this address between the calls to `malloc` and `free`. It is important that you (i) do not try to free memory that was not allocated, and (ii) do not use memory that you have deallocated.

After we have finished using the memory allocated in Figure 2 we should run free(new_memory) to release the allocated block. Is there anything odd here? When I allocated the memory I told it how much to allocate but I didn't do this when freeing, what's going on? (answer here).

arrays

Remember, there are no such thing as arrays in C. There is just some syntax that makes things look like arrays[1]. The notion expands to pointer arithmetic as follows:

$$a[i] \quad \equiv \quad *(a+i)$$

That is a is a interpreted as a pointer and i defines some offset from that pointer of where to look. This should explain why the following code doesn't do what we want it to do:

```c
int length_of_array(int array[]){ return sizeof(array); }
```

[1]Also remember that there are also no such thing as strings in C, just null-terminated arrays of characters, which we know do not exist.

There are a number of other things that you need to be careful with when using arrays in C. Whilst you can use something like `int a[10] = {};` to nicely initialise an array of size 10 to 0, one cannot do this for dynamicly sized arrays. For example, trying to compile

```
void foo(int size){
  int array [size] = {};
  // do stuff
}
```

will lead to something like `error: variable-sized object may not be initialized`. And note, if you want to return an array it needs to be dynamically allocated, one cannot do the following:

```
int* get_array(){
  int array [10];
  return array;
}
```